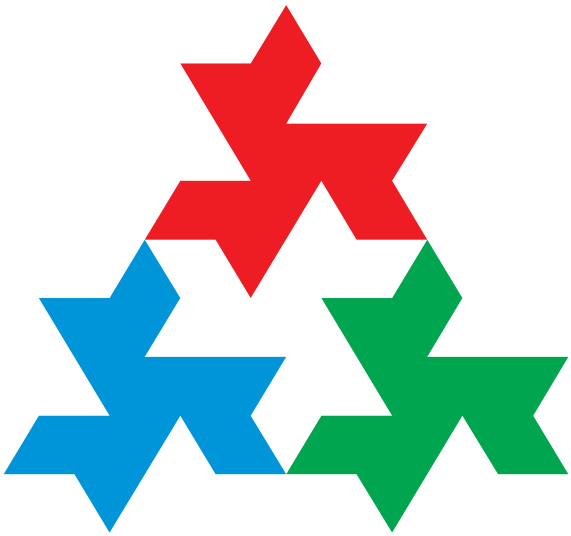
MINISTERSTWO EDUKACJI NARODOWEJ
FUNDACJA ROZWOJU INFORMATYKI
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ



XVII OLIMPIADA INFORMATYCZNA
2009/2010

Olimpiada Informatyczna jest organizowana przy współudziale

WARSZAWA, 2010

MINISTERSTWO EDUKACJI NARODOWEJ
FUNDACJA ROZWOJU INFORMATYKI
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

XVII OLIMPIADA INFORMATYCZNA
2009/2010

WARSZAWA, 2010

Autorzy tekstów:

mgr Szymon Acedański
dr Piotr Chrząstowski
prof. dr hab. Krzysztof Diks
Marian M. Kędzierski
Piotr Niedźwiedź
mgr Marcin Pilipczuk
Michał Pilipczuk
mgr Jakub Radoszewski
prof. dr hab. Wojciech Rytter
Jacek Tomasiewicz
mgr Bartosz Walczak
dr Tomasz Waleń
Michał Włodarczyk
dr Jakub Wojtaszczyk

Autorzy programów:

mgr Łukasz Bieniasz-Krzywiec
inż. Adam Gawarkiewicz
Bartosz Górski
Przemysław Horban
Tomasz Kulczyński
mgr Jakub Łącki
Mirosław Michalski
Piotr Niedźwiedź
Michał Pilipczuk
mgr Jakub Radoszewski
Wojciech Śmietanka
Wojciech Tyczyński
Bartłomiej Wołowiec

Opracowanie i redakcja:

Tomasz Kociumaka
dr Marcin Kubica
mgr Jakub Radoszewski

Skład:

mgr Tomasz Idziaszek
Tomasz Kociumaka

Tłumaczenie treści zadań:

dr Marcin Kubica
Tomasz Kulczyński
mgr Jakub Łącki
dr Jakub Pawlewicz
mgr Jakub Radoszewski

Pozycja dotowana przez Ministerstwo Edukacji Narodowej.

Druk książki został sfinansowany przez

ISBN 978-83-922946-9-6

Spis treści

Sprawozdanie z przebiegu XVII Olimpiady Informatycznej 7

Regulamin Olimpiady Informatycznej 33

Zasady organizacji zawodów 41

Zawody I stopnia — opracowania zadań **49**

Gildie 51

Kolej 59

Korale 73

Najdzielniejszy dzielnik 79

Test na inteligencję 89

Zawody II stopnia — opracowania zadań **97**

Antysymetria 99

Chomiki 109

Klocki 117

Owce 127

Teleporty 135

Zawody III stopnia — opracowania zadań **145**

Monotoniczność 147

Gra w minima 155

Latarnia 161

Żabka 173

Jedynki 187

Mosty 193

Piloci *205*

XXII Międzynarodowa Olimpiada Informatyczna — treści zadań **217**

Detektyw *219*

Ciepło-Zimno *222*

Jakość życia *225*

Języki *228*

Gra w pary *230*

Korek drogowy *233*

Labirynt *236*

Oszczędny kod *240*

XVI Bałtycka Olimpiada Informatyczna — treści zadań **243**

Tabelka *245*

Lego *247*

Misie *249*

Obwód drukowany *251*

Cukierki *253*

Kubły *254*

Miny *255*

XVII Olimpiada Informatyczna Krajów Europy Środkowej — treści zadań **257**

Ochroniarze *259*

Prostokąt arytmetyczny *261*

Sojusze *263*

Odtwarzacz MP3 ..... *266*

Ogromna wieża *268*

PIN *270*

Literatura **273**

Krzysztof Diks

Wstęp

Drogi Czytelniku!

XVII Olimpiada Informatyczna za nami. Wzięło w niej udział ponad 1100 uczniów. Biorąc pod uwagę trudność Olimpiady, jest to liczba niemała, ale naszym pragnieniem jest jej zwiększenie. Zachęcam wszystkich interesujących się informatyką do uczestnictwa w Olimpiadzie. Nawet jeśli startując pierwszy raz, nie potrafimy rozwiązać wszystkich zadań, spróbujmy zrobić tylko niektóre z nich. Startując w Olimpiadzie, stajemy się natychmiast członkami prestiżowego klubu olimpijczyków-informatyków. Korzystając z serwisu Olimpiady i portalu `main.edu.pl`, mamy szansę uczyć się od najlepszych i doskonalić swoje umiejętności tak, żeby w przyszłości osiągać olimpijskie sukcesy. Awans do finału Olimpiady to możliwość wyboru najlepszych uczelni w Polsce do studiowania informatyki. To, w dobie kierunków zamawianych, wysokie stypendia od początku studiów. Sukcesy olimpijskie są też brane pod uwagę przez liczne firmy podczas rekrutacji na atrakcyjne staże wakacyjne.

Młodzi polscy olimpijczycy to ścisła czołówka światowa. Potwierdzają to wyniki laureatów XVII Olimpiady Informatycznej na zawodach międzynarodowych. W Olimpiadzie Informatycznej Krajów Bałtyckich, która odbyła się w maju w Estonii, nasi olimpijczycy zdobyli cztery medale: Paweł Lipski — złoto, Łukasz Jocz i Krzysztof Leszczyński — medale srebrne, a Stanisław Barzowski — medal brązowy. Ten znakomity wynik osiągnęli uczniowie młodsi, którzy w tym roku mają szansę wystartować raz jeszcze w Olimpiadzie. Nasza pierwsza reprezentacja wspaniale spisała się podczas Olimpiady Informatycznej Krajów Europy Środkowej (Słowacja, lipiec) oraz podczas Międzynarodowej Olimpiady Informatycznej (Kanada, sierpień). Zwyciężczynią tegorocznej Olimpiady Informatycznej Krajów Europy Środkowej została Anna Piekarska. Jan Kanty Milczek zdobył srebrny medal, a Adrian Jaskółka i Igor Adamski medale brązowe. W Kanadzie najlepiej z naszych reprezentantów wypadł Adrian Jaskółka, zajmując trzecie miejsce w świecie i zdobywając złoty medal, medale srebrne zdobyli Anna Piekarska i Jan Kanty Milczek, a brąz przypadł w udziale Igorowi Adamskiemu.

Sukcesy naszych olimpijczyków to wynik ich talentu i ogromnej pracy. Nie byłoby ich jednak bez wsparcia ich rodziców oraz nauczycieli ze szkół, w których się uczą. Sukcesy to też owoc wszystkich zaangażowanych w pracę Olimpiady: członków komitetów, głównego i okręgowych, współpracujących z Olimpiadą uczelni, jurorów i autorów zadań. Nie mniejsze słowa uznania należą się wypróbowanym przyjaciołom Olimpiady: Fundacji Rozwoju Informatyki — formalnemu organizatorowi, współorganizatorowi — firmie Asseco Poland SA, firmie Combidata Poland, Ogólnopolskiej Fundacji Edukacji Komputerowej, wydawnictwom — PWN i WNT. Wszystkim gorąco dziękuję za pracę dla Olimpiady.

Czytelnikom życzę przyjemnej lektury i udanych startów w kolejnych edycjach Olimpiady i innych konkursach algorytmicznych.

Krzysztof Diks

Sprawozdanie z przebiegu XVII Olimpiady Informatycznej w roku szkolnym 2009/2010

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 XVII Olimpiady Informatycznej jest Fundacja Rozwoju Informatyki.

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.

5 października 2009 roku rozesłano do 3716 szkół i zespołów szkół młodzieżowych ponadgimnazjalnych plakaty informujące o rozpoczęciu XVII Olimpiady oraz promujące sukcesy młodych polskich informatyków. Zawody I stopnia rozpoczęły się 19 października 2009 roku. Ostatecznym terminem nadsyłania prac konkursowych był 16 listopada 2009 roku.

Zawody II i III stopnia były dwudniowymi sesjami stacjonarnymi, poprzedzonymi jednodniowymi sesjami próbnymi. Zawody II stopnia odbyły się w sześciu okręgach: Krakowie, Poznaniu, Sopocie, Toruniu, Warszawie i Wrocławiu w dniach 9–11.02.2010 roku, natomiast zawody III stopnia odbyły się w ośrodku firmy Combidata Poland w Sopocie, w dniach 13–16.04.2010 roku.

Uroczystość zakończenia XVII Olimpiady Informatycznej odbyła się 16.04.2010 roku w siedzibie firm Combidata Poland i Asseco Poland SA 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:
mgr Szymon Acedański (Uniwersytet Warszawski)

kierownik organizacyjny:
Tadeusz Kuran

członkowie:
dr Piotr Chrząstowski (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)
dr 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 Jakub Pawlewicz (Uniwersytet Warszawski)
zastępca przewodniczącego:
dr Adam Malinowski (Uniwersytet Warszawski)
sekretarz:
Monika Kozłowska-Zając (OEIiZK)
członkowie:
mgr Szymon Acedański (Uniwersytet Warszawski)
prof. dr hab. Krzysztof Diks (Uniwersytet Warszawski)
mgr Jakub Radoszewski (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 hab. 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. inż. 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. Robert Wrembel (Politechnika Poznańska)
zastępca przewodniczącego:
mgr inż. Szymon Wąsik (Politechnika Poznańska)
sekretarz:
Michał Połetek (Politechnika Poznańska)
członkowie:
Mariola Galas (Politechnika Poznańska)
mgr inż. Piotr Gawron (Politechnika Poznańska)
dr Maciej Machowiak (Politechnika Poznańska)
dr Jacek Marciniak (Uniwersytet Adama Mickiewicza w Poznaniu)
dr inż. Maciej Miłostan (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 RP w Gdyni)
dr Paweł Żyliński (Uniwersytet Gdański)

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

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:

Maciej Andrejczuk
Łukasz Bieniasz-Krzywiec
inż. Adam Gawarkiewicz
Konrad Gołuchowski
Bartosz Górski
Przemysław Horban
mgr Adam Iwanicki
Tomasz Kulczyński
mgr Jakub Łącki
Marek Marczykowski
Mirosław Michalski
Jacek Migdał
Piotr Niedźwiedź
mgr Paweł Parys
Michał Pilipczuk
Juliusz Sompolski
Wojciech Śmietanka
Wojciech Tyczyński
Bartłomiej Wołowiec

ZAWODY I STOPNIA

W zawodach I stopnia XVII Olimpiady Informatycznej wzięło udział 1126 zawodników. Decyzją Komitetu Głównego zdyskwalifikowano 39 zawodników. Powodem dys-

kwalifikacji była niesamodzielność rozwiązań zadań konkursowych. Sklasyfikowano 1087 zawodników.

Decyzją Komitetu Głównego do zawodów zostało dopuszczonych 36 uczniów gimnazjów. Pochodzili oni z następujących szkół:

Gimnazjum nr 24 przy III LO	Gdynia	7 uczniów
Gimnazjum nr 86 im. płk. Ignacego A. Boernera	Warszawa	3
Gimnazjum nr 9 im. cc. majora Hieronima Dekutowskiego ps. „Zapora”	Lublin	2
Gimnazjum nr 2 im. Żołnierzy Armii Krajowej	Brwinów	1 uczeń
Gimnazjum nr 50	Bydgoszcz	1
Gimnazjum nr 9 im. Powstańców Wielkopolskich	Bydgoszcz	1
Gimnazjum Katolickie im. św. Królowej Jadwigi	Częstochowa	1
Gimnazjum nr 1 im. św. Jadwigi Królowej	Jasło	1
Gimnazjum nr 1	Jawor	1
Gimnazjum nr 1 im. Antoniego Abrahama	Kartuzy	1
Prywatne Gimnazjum „CUBE”	Konstancin-Jez.	1
Gimnazjum nr 9	Legnica	1
Publiczne Gimnazjum	Nowa Ruda	1
Publiczne Gimnazjum	Przysucha	1
Gimnazjum nr 1 im. Alberta Einsteina	Puszczykowo	1
Publ. Gimnazjum nr 13 im. Polskich Noblistów	Radom	1
Publ. Gimnazjum nr 23 im. Jana Kochanowskiego	Radom	1
Gimnazjum im. Jana Pawła II Sióstr Prezentek	Rzeszów	1
ZS UMK, Gimnazjum Akademickie	Toruń	1
Gimnazjum nr 11 im. Jana III Sobieskiego	Tychy	1
Gimnazjum nr 13 im. Stanisława Staszica	Warszawa	1
Społeczne Gimnazjum nr 16 STO	Warszawa	1
Gimnazjum z Oddziałami Dwujęzycznymi nr 42	Warszawa	1
Gimnazjum nr 77 im. Ignacego Domeyki	Warszawa	1
Gimnazjum Społeczne nr 333	Warszawa	1
Gimnazjum im. Jerzego Andrzeja Helwinga	Węgorzewo	1
Gimnazjum nr 1 im. Hugona Steinhausa	Wrocław	1
Gimnazjum nr 49 z Oddziałami Dwujęzycznymi	Wrocław	1

Kolejność województw pod względem liczby uczestników była następująca:

małopolskie	177 zawodników	zachodniopomorskie	42
mazowieckie	153	łódzkie	34
dolnośląskie	118	podkarpackie	34
śląskie	111	lubelskie	30
pomorskie	110	świętokrzyskie	24
kujawsko-pomorskie	76	warmińsko-mazurskie	19
podlaskie	73	lubuskie	16
wielkopolskie	54	opolskie	16

W zawodach I stopnia najliczniej reprezentowane były szkoły:

V LO im. Augusta Witkowskiego	Kraków	98 uczniów
III LO im. Marynarki Wojennej RP	Gdynia	54
XIV LO im. Stanisława Staszica	Warszawa	54
Zespół Szkół nr 14	Wrocław	52
I LO im. Adama Mickiewicza	Białystok	43
Zespół Szkół Ogólnokształcących nr 6	Bydgoszcz	29
XIII Liceum Ogólnokształcące	Szczecin	25
I LO im. Tadeusza Kościuszki	Legnica	23
V Liceum Ogólnokształcące	Bielsko-Biała	22
VIII LO im. Adama Mickiewicza	Poznań	22
ZS UMK, Gimnazjum i Liceum Akademickie	Toruń	18
VIII LO im. Marii Skłodowskiej-Curie	Katowice	16
II LO im. Króla Jana III Sobieskiego	Kraków	12
Zespół Szkół Zawodowych	Brodnica	11
I LO im. Bartłomieja Nowodworskiego	Kraków	11
I LO im. Edwarda Dembowskiego	Zielona Góra	11
III LO im. Adama Mickiewicza	Wrocław	10
V LO im. Stefana Żeromskiego	Gdańsk	9
LO Politechniki Łódzkiej	Łódź	9
VI LO im. Jana Kochanowskiego	Radom	8
III LO im. Adama Mickiewicza	Tarnów	8
VIII LO im. Władysława IV	Warszawa	8
X Liceum Ogólnokształcące	Wrocław	8
Gimnazjum nr 24 przy III LO	Gdynia	7
I LO im. Mikołaja Kopernika	Krosno	7
VI LO im. Wacława Sierpińskiego	Gdynia	6
ZSO nr 1 im. Jana Długosza	Nowy Sącz	6
IV LO im. Cypriana Kamila Norwida	Białystok	5
II LO im. Mikołaja Kopernika	Kędzierzyn-Koźle	5
IV LO im. Hanki Sawickiej	Kielce	5
II LO im. Hetmana Jana Zamoyskiego	Lublin	5
I LO im. Tadeusza Kościuszki	Łomża	5
Publiczne LO nr II z Oddziałami Dwujęzycznymi im. Marii Konopnickiej	Opole	5
LO im. Mikołaja Kopernika	Ostrów Mazow.	5
III LO im. św. Jana Kantego	Poznań	5
II LO im. Hetmana Jana Tarnowskiego	Tarnów	5
Akademickie LO przy PJWSTK	Warszawa	5
Zespół Szkół Technicznych – Technikum nr 3	Wodzisław Śl.	5
LO im. Bogusława X	Białogard	4
VI LO im. Króla Zygmunta Augusta	Białystok	4
Gdańskie Liceum Autonomiczne	Gdańsk	4
II LO im. Króla Jana III Sobieskiego	Grudziądz	4
I LO im. Stefana Żeromskiego	Jelenia Góra	4

II LO im. Jana Śniadeckiego	Kielce	4
I LO im. Mikołaja Kopernika	Łódź	4
Zespół Szkół Zawodowych nr 3 im. Kardynała Stefana Wyszyńskiego	Ostrołęka	4
Zespół Szkół Elektronicznych	Rzeszów	4
LO Wyższej Szkoły Informatyki i Zarządzania	Rzeszów	4
III LO im. Krzysztofa Kamila Baczyńskiego	Białystok	3
III LO im. Stefana Żeromskiego	Bielsko-Biała	3
Zespół Szkół Elektronicznych	Bydgoszcz	3
I LO im. Cypriana Kamila Norwida	Bydgoszcz	3
Zespół Szkół Łączności im. Obrońców Poczty Polskiej	Gdańsk	3
Zespół Szkół Elektronicznych i Informatycznych im. Komisji Edukacji Narodowej	Giżycko	3
Zespół Szkół Techniczno-Informatycznych	Gliwice	3
II LO im. Czesława Miłosza	Jaworzno	3
I LO im. Mikołaja Kopernika	Katowice	3
Zespół Szkół Informatycznych im. gen. Józefa Hauke Bosaka	Kielce	3
II LO im. Marii Skłodowskiej-Curie	Końskie	3
LO Zakonu Pijarów im. ks. Stanisława Konarskiego	Kraków	3
VIII LO im. Stanisława Wyspiańskiego	Kraków	3
LO im. Władysława Broniewskiego	Krzepice	3
II LO im. Mikołaja Kopernika	Mielec	3
IV LO im. Marii Skłodowskiej-Curie	Olsztyn	3
II LO im. Konstantego Ildefonsa Gałczyńskiego	Olsztyn	3
Zespół Szkół Technicznych	Ostrów Wlkp.	3
LO im. Marszałka Stanisława Małachowskiego	Płock	3
Zespół Szkół Komunikacji im. Hipolita Cegielskiego	Poznań	3
I LO im. Generałowej Zamoyskiej i Heleny Modrzejewskiej	Poznań	3
II LO im. Andrzeja Frycza Modrzewskiego	Rybnik	3
I LO im. Bolesława Prusa	Siedlce	3
I LO im. Kazimierza Jagiellończyka	Sieradz	3
II LO z Oddziałami Dwujęzycznymi im. Adama Mickiewicza	Słupsk	3
IV LO im. Stanisława Staszica	Sosnowiec	3
LO im. Komisji Edukacji Narodowej	Stalowa Wola	3
Publiczne LO im. Władysława Broniewskiego	Strzelce Opolskie	3
I LO im. Marii Konopnickiej	Suwałki	3
IX LO im. Bohaterów Monte Cassino	Szczecin	3
Gimnazjum nr 86 im. płk. Ignacego A. Boernera	Warszawa	3
VI LO im. Tadeusza Rejtana	Warszawa	3
VII LO im. Kamila Konrada Baczyńskiego	Wrocław	3

Najliczniej reprezentowane były miasta:

Kraków	132 uczestników	Grudziądz	5
Warszawa	102	Kędzierzyn-Koźle	5
Wrocław	79	Łomża	5
Gdynia	71	Ostrołęka	5
Białystok	59	Ostrów Mazowiecka	5
Poznań	40	Biała Podlaska	4
Bydgoszcz	37	Białogard	4
Szczecin	35	Jelenia Góra	4
Bielsko-Biała	29	Ostrów Wielkopolski	4
Legnica	26	Płock	4
Katowice	23	Rybnik	4
Toruń	20	Siedlce	4
Gdańsk	19	Wadowice	4
Łódź	18	Chełm	3
Kielce	17	Elbląg	3
Lublin	15	Giżycko	3
Tarnów	15	Jaworzno	3
Rzeszów	12	Kartuzy	3
Zielona Góra	12	Konin	3
Brodnica	11	Krzepice	3
Nowy Sącz	11	Lubliniec	3
Radom	11	Mielec	3
Wodzisław Śląski	9	Piaseczno	3
Częstochowa	7	Radomsko	3
Krosno	7	Sieradz	3
Olsztyn	7	Stalowa Wola	3
Słupsk	7	Strzelce Opolskie	3
Gliwice	6	Suwałki	3
Opole	6	Zabrze	3
Sosnowiec	6		

Zawodnicy uczęszczali do następujących klas:

do klasy I gimnazjum	2
do klasy II gimnazjum	10
do klasy III gimnazjum	24
do klasy I szkoły ponadgimnazjalnej	223
do klasy II szkoły ponadgimnazjalnej	431
do klasy III szkoły ponadgimnazjalnej	358
do klasy IV szkoły ponadgimnazjalnej	39

W zawodach I stopnia zawodnicy mieli do rozwiązania pięć zadań:

- „Gildie” autorstwa Marcina Pilipczuka

- „Kolej” autorstwa Bartosza Walczaka
- „Korale” autorstwa Tomasza Walenia
- „Najdzielniejszy dzielnik” autorstwa Jakuba Radoszewskiego
- „Test na inteligencję” autorstwa Jakuba Radoszewskiego

każde oceniane maksymalnie po 100 punktów.

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

- **GIL** — Gildie

	GIL	
	liczba zawodników	czyli
100 pkt.	488	44,9%
75–99 pkt.	109	10,0%
50–74 pkt.	26	2,4%
1–49 pkt.	72	6,6%
0 pkt.	174	16,0%
brak rozwiązania	218	20,1%

- **KOL** — Kolej

	KOL	
	liczba zawodników	czyli
100 pkt.	37	3,4%
75–99 pkt.	1	0,1%
50–74 pkt.	43	4,0%
1–49 pkt.	185	17,0%
0 pkt.	583	53,6%
brak rozwiązania	238	21,9%

- **KOR** — Korale

	KOR	
	liczba zawodników	czyli
100 pkt.	98	9,0%
75–99 pkt.	38	3,5%
50–74 pkt.	58	5,3%
1–49 pkt.	269	24,8%
0 pkt.	110	10,1%
brak rozwiązania	514	47,3%

- **NAJ** — Najdzielniejszy dzielnik

	NAJ	
	liczba zawodników	czyli
100 pkt.	32	2,9%
75–99 pkt.	21	1,9%
50–74 pkt.	49	4,5%
1–49 pkt.	368	33,9%
0 pkt.	169	15,6%
brak rozwiązania	448	41,2%

- **TES** — Test na inteligencję

	TES	
	liczba zawodników	czyli
100 pkt.	318	29,3%
75–99 pkt.	40	3,7%
50–74 pkt.	22	2,0%
1–49 pkt.	498	45,8%
0 pkt.	161	14,8%
brak rozwiązania	48	4,4%

W sumie za wszystkie 5 zadań konkursowych:

SUMA	liczba zawodników	czyli
500 pkt.	4	0,4%
375–499 pkt.	59	5,4%
250–374 pkt.	154	14,2%
125–249 pkt.	325	29,9%
1–124 pkt.	439	40,4%
0 pkt.	106	9,8%

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 11–13 lutego 2010 roku w sześciu okręgach, zakwalifikowano 315 zawodników, którzy osiągnęli w zawodach I stopnia wynik nie mniejszy niż 202 pkt.

Zawodnicy uczęszczali do szkół w następujących województwach:

małopolskie	70 uczestników	zachodniopomorskie	9
mazowieckie	40	podkarpackie	8
pomorskie	40	lubuskie	5
dolnośląskie	37	świętokrzyskie	5
podlaskie	31	lubelskie	3
kujawsko-pomorskie	24	łódzkie	3
śląskie	19	opolskie	3
wielkopolskie	10	warmińsko-mazurskie	3

Zawodnicy zostali przydzieleni do następujących okręgów:

Kraków	102 zawodników
Poznań	15
Sopot	51
Toruń	25
Warszawa	75
Wrocław	42

W zawodach II stopnia najliczniej reprezentowane były szkoły:

V LO im. Augusta Witkowskiego	Kraków	56 uczniów
III LO im. Marynarki Wojennej RP	Gdynia	34
I LO im. Adama Mickiewicza	Białystok	24
Zespół Szkół nr 14	Wrocław	23
XIV LO im. Stanisława Staszica	Warszawa	22
Zespół Szkół Ogólnokształcących nr 6	Bydgoszcz	12
XIII Liceum Ogólnokształcące	Szczecin	7

ZS UMK, Gimnazjum i Liceum Akademickie	Toruń	7
V Liceum Ogólnokształcące	Bielsko-Biała	5
II LO im. Króla Jana III Sobieskiego	Kraków	5
I LO im. Tadeusza Kościuszki	Legnica	5
VIII LO im. Adama Mickiewicza	Poznań	5
VIII LO im. Władysława IV	Warszawa	5
III LO im. Adama Mickiewicza	Wrocław	4
I LO im. Edwarda Dembowskiego	Zielona Góra	4
VI LO im. Jana Kochanowskiego	Radom	3
LO Wyższej Szkoły Informatyki i Zarządzania	Rzeszów	3
I LO im. Marii Konopnickiej	Suwałki	3

Najliczniej reprezentowane były miasta:

Kraków	63 uczestników	Bielsko-Biała	5
Gdynia	35	Katowice	5
Warszawa	32	Legnica	5
Wrocław	30	Rzeszów	5
Białystok	26	Zielona Góra	5
Bydgoszcz	16	Kielce	4
Poznań	10	Nowy Sącz	4
Szczecin	9	Radom	4
Toruń	7	Suwałki	3

W dniu 11 lutego 2010 roku odbyła się sesja próbna, na której zawodnicy rozwiązywali nieliczące się do ogólnej klasyfikacji zadanie „Antysymetria" autorstwa Jakuba Radoszewskiego i Wojciecha Ryttera. W dniach konkursowych (12–13 lutego) zawodnicy rozwiązywali następujące zadania:

- w pierwszym dniu zawodów:
  - „Chomiki" autorstwa Jakuba Radoszewskiego i Wojciecha Ryttera
  - „Klocki" autorstwa Jacka Tomasiewicza
- w drugim dniu zawodów:
  - „Owce" autorstwa Michała Włodarczyka
  - „Teleporty" autorstwa Jakuba Wojtaszczyka

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:

- **ANT — próbne** — Antysymetria

	ANT — próbne	
	liczba zawodników	czyli
100 pkt.	64	20,6%
75–99 pkt.	1	0,3%
50–74 pkt.	7	2,3%
1–49 pkt.	207	66,8%
0 pkt.	16	5,2%
brak rozwiązania	15	4,8%

- **CHO** — Chomiki

	CHO	
	liczba zawodników	czyli
100 pkt.	21	6,8%
75–99 pkt.	6	1,9%
50–74 pkt.	9	2,9%
1–49 pkt.	30	9,7%
0 pkt.	118	38,1%
brak rozwiązania	126	40,6%

- **KLO** — Klocki

	KLO	
	liczba zawodników	czyli
100 pkt.	17	5,5%
75–99 pkt.	19	6,1%
50–74 pkt.	28	9,0%
1–49 pkt.	98	31,6%
0 pkt.	91	29,4%
brak rozwiązania	57	19,4%

- **OWC** — Owce

	OWC	
	liczba zawodników	czyli
100 pkt.	11	3,5%
75–99 pkt.	5	1,6%
50–74 pkt.	4	1,3%
1–49 pkt.	7	2,3%
0 pkt.	134	43,2%
brak rozwiązania	149	48,1%

- **TEL** — Teleporty

	TEL	
	liczba zawodników	czyli
100 pkt.	25	8,1%
75–99 pkt.	0	0,0%
50–74 pkt.	1	0,3%
1–49 pkt.	68	21,9%
0 pkt.	108	34,8%
brak rozwiązania	108	34,8%

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

SUMA	liczba zawodników	czyli
400 pkt.	2	0,6%
300–399 pkt.	7	2,3%
200–299 pkt.	14	4,5%
100–199 pkt.	42	13,5%
1–99 pkt.	141	43,5%
0 pkt.	104	33,5%

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

ZAWODY III STOPNIA

Zawody III stopnia odbyły się w ośrodku firmy Combidata Poland w Sopocie w dniach od 13 do 16 kwietnia 2010 roku. Do zawodów III stopnia zakwalifikowano 87 najlepszych uczestników zawodów II stopnia, którzy uzyskali wynik nie mniejszy niż 62 pkt.

Zawodnicy uczęszczali do szkół w następujących województwach:

małopolskie	17 uczestników	zachodniopomorskie	4
dolnośląskie	12	lubuskie	2
pomorskie	12	opolskie	2
mazowieckie	10	podkarpackie	2
kujawsko-pomorskie	8	wielkopolskie	2
podlaskie	7	łódzkie	1
śląskie	7	świętokrzyskie	1

Niżej wymienione szkoły miały w finale więcej niż jednego zawodnika:

V LO im. Augusta Witkowskiego	Kraków	13 uczniów
III LO im. Marynarki Wojennej RP	Gdynia	9
Zespół Szkół nr 14	Wrocław	9
I LO im. Adama Mickiewicza	Białystok	5
XIV LO im. Stanisława Staszica	Warszawa	5
XIII Liceum Ogólnokształcące	Szczecin	4
Zespół Szkół Ogólnokształcących nr 6	Bydgoszcz	3
ZS UMK, Gimnazjum i Liceum Akademickie	Toruń	3
Gimnazjum nr 24 przy III LO	Gdynia	2
VIII LO im. Marii Skłodowskiej-Curie	Katowice	2
II LO im. Króla Jana III Sobieskiego	Kraków	2

13 kwietnia odbyła się sesja próbna, na której zawodnicy rozwiązywali nieliczące się do ogólnej klasyfikacji zadanie „Monotoniczność” autorstwa Mariana M. Kędzierskiego. W dniach konkursowych (14–15 kwietnia) zawodnicy rozwiązywali następujące zadania:

- w pierwszym dniu zawodów:
  - „Gra w minima” autorstwa Jakuba Wojtaszczyka
  - „Żabka” autorstwa Jakuba Radoszewskiego
  - „Latarnia” autorstwa Jakuba Wojtaszczyka
- w drugim dniu zawodów:
  - „Piloci” autorstwa Piotra Chrząstowskiego
  - „Jedynki” autorstwa Wojciecha Ryttera
  - „Mosty” autorstwa Szymona Acedańskiego i Jakuba Radoszewskiego

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:

- **MON — próbne** — Monotoniczność

	MON — próbne	
	liczba zawodników	czyli
100 pkt.	19	21,8%
75–99 pkt.	13	14,9%
50–74 pkt.	33	37,9%
1–49 pkt.	7	8,0%
0 pkt.	15	17,2%
brak rozwiązania	0	0,0%

- **GRA** — Gra w minima

	GRA	
	liczba zawodników	czyli
100 pkt.	56	64,4%
75–99 pkt.	0	0,0%
50–74 pkt.	1	1,1%
1–49 pkt.	10	11,5%
0 pkt.	10	11,5%
brak rozwiązania	10	11,5%

- **ZAB** — Żabka

	ZAB	
	liczba zawodników	czyli
100 pkt.	32	36,8%
75–99 pkt.	5	5,7%
50–74 pkt.	6	6,9%
1–49 pkt.	19	21,8%
0 pkt.	20	23,0%
brak rozwiązania	5	5,7%

- **LAT** — Latarnia

	LAT	
	liczba zawodników	czyli
100 pkt.	0	0,0%
75–99 pkt.	0	0,0%
50–74 pkt.	0	0,0%
1–49 pkt.	19	21,8%
0 pkt.	25	28,7%
brak rozwiązania	43	49,4%

- **PIL** — Piloci

	PIL	
	liczba zawodników	czyli
100 pkt.	10	11,5%
75–99 pkt.	5	5,7%
50–74 pkt.	9	10,3%
1–49 pkt.	38	43,7%
0 pkt.	6	6,9%
brak rozwiązania	19	21,8%

- **JED** — Jedynki

	JED	
	liczba zawodników	czyli
100 pkt.	0	0,0%
75–99 pkt.	0	0,0%
50–74 pkt.	2	2,3%
1–49 pkt.	16	18,4%
0 pkt.	16	18,4%
brak rozwiązania	53	60,9%

- **MOS** — Mosty

	MOS	
	liczba zawodników	czyli
100 pkt.	4	4,6%
75–99 pkt.	0	0,0%
50–74 pkt.	1	1,1%
1–49 pkt.	18	20,7%
0 pkt.	34	39,1%
brak rozwiązania	30	34,5%

W sumie za wszystkie 6 zadań konkursowych rozkład wyników zawodników był następujący:

SUMA	liczba zawodników	czyli
450–600 pkt.	0	0,0%
300–449 pkt.	20	23,0%
150–299 pkt.	35	40,2%
1–149 pkt.	30	34,5%
0 pkt.	2	2,3%

W dniu 16 kwietnia 2010 roku, w siedzibie firm Asseco Poland SA i Combidata Poland w Gdyni, ogłoszono wyniki finału XVII Olimpiady Informatycznej 2009/2010 i rozdano nagrody ufundowane przez: Ministerstwo Edukacji Narodowej, Asseco Poland SA, Ogólnopolską Fundację Edukacji Komputerowej, Wydawnictwa Naukowe PWN i Wydawnictwo „Delta".

Poniżej zestawiono listę wszystkich laureatów i wyróżnionych finalistów:

(1) **Adrian Jaskółka**, 3 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza w Białymstoku, 440 pkt., laureat I miejsca

(2) **Jan Kanty Milczek**, 2 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP w Gdyni, 440 pkt., laureat I miejsca

(3) **Igor Adamski**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie, 400 pkt., laureat I miejsca

(4) **Anna Piekarska**, 3 klasa, XIV Liceum Ogólnokształcące im. Polonii Belgijskiej we Wrocławiu, 370 pkt., laureatka I miejsca

(5) **Michał Zgliczyński**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie, 350 pkt., laureat II miejsca

(6) **Dawid Dąbrowski**, 3 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP w Gdyni, 346 pkt., laureat II miejsca

(7) **Łukasz Jocz**, 2 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza w Białymstoku, 342 pkt., laureat II miejsca

(8) **Grzegorz Milka**, 3 klasa, II Liceum Ogólnokształcące im. Mikołaja Kopernika w Kędzierzynie-Koźlu, 341 pkt., laureat II miejsca

(9) **Łukasz Kalinowski**, 3 klasa, I Liceum Ogólnokształcące im. Cypriana Kamila Norwida w Bydgoszczy, 340 pkt., laureat II miejsca

(10) **Jakub Pachocki**, 3 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP w Gdyni, 340 pkt., laureat II miejsca

(11) **Janusz Wróbel**, 3 klasa, XIV Liceum Ogólnokształcące im. Polonii Belgijskiej we Wrocławiu, 340 pkt., laureat II miejsca

(12) **Krzysztof Leszczyński**, 2 klasa, I Liceum Ogólnokształcące im. Marii Konopnickiej w Suwałkach, 332 pkt., laureat II miejsca

(13) **Adam Obuchowicz**, 3 klasa, I Liceum Ogólnokształcące im. Edwarda Dembowskiego w Zielonej Górze, 326 pkt., laureat II miejsca

(14) **Alan Kutniewski**, 3 klasa, XIII Liceum Ogólnokształcące w Szczecinie, 323 pkt., laureat II miejsca

(15) **Maciej Piekarz**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie, 319 pkt., laureat II miejsca

(16) **Michał Makarewicz**, 3 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza w Białymstoku, 310 pkt., laureat II miejsca

(17) **Karol Pokorski**, 2 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP w Gdyni, 310 pkt., laureat II miejsca

(18) **Grzegorz Prusak**, 3 klasa, II Liceum Ogólnokształcące im. Generałowej Zamoyskiej i Heleny Modrzejewskiej w Poznaniu, 303 pkt., laureat III miejsca

(19) **Grzegorz Guśpiel**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie, 300 pkt., laureat III miejsca

(20) **Wojciech Lis**, 3 klasa, I Liceum Ogólnokształcące im. Juliusza Słowackiego w Chorzowie, 300 pkt., laureat III miejsca

(21) **Paweł Walczak**, 3 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP w Gdyni, 299 pkt., laureat III miejsca

(22) **Michał Krasnoborski**, 3 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP w Gdyni, 292 pkt., laureat III miejsca

(23) **Maciej Borsz**, 2 klasa, VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich w Bydgoszczy, 290 pkt., laureat III miejsca

(24) **Paweł Lipski**, 2 klasa, Liceum Ogólnokształcące „Filomata" w Gliwicach, 290 pkt., laureat III miejsca

(25) **Szymon Sidor**, 3 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP w Gdyni, 288 pkt., laureat III miejsca

(26) **Stanisław Barzowski**, 2 klasa gimnazjum, Gimnazjum nr 24 przy III LO w Gdyni, 280 pkt., laureat III miejsca

(27) **Krzysztof Pszeniczny**, 3 klasa gimnazjum, Gimnazjum im. Jana Pawła II Sióstr Prezentek w Rzeszowie, 277 pkt., laureat III miejsca

(28) **Tomasz Wiatrowski**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie, 277 pkt., laureat III miejsca

(29) **Jan Marcinkowski**, 1 klasa, XIV Liceum Ogólnokształcące im. Polonii Belgijskiej we Wrocławiu, 270 pkt., laureat III miejsca

(30) **Piotr Szefler**, 2 klasa, ZS UMK, Gimnazjum i Liceum Akademickie w Toruniu, 270 pkt., laureat III miejsca

(31) **Kamil Sałaś**, 3 klasa, II Liceum Ogólnokształcące im. Króla Jana III Sobieskiego w Krakowie, 264 pkt., laureat III miejsca

(32) **Bartłomiej Dudek**, 1 klasa, XIV Liceum Ogólnokształcące im. Polonii Belgijskiej we Wrocławiu, 260 pkt., laureat III miejsca

(33) **Michał Łowicki**, 1 klasa, III Liceum Ogólnokształcące im. Adama Mickiewicza we Wrocławiu, 260 pkt., laureat III miejsca

(34) **Adam Nieżurawski**, 2 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica w Warszawie, 260 pkt., laureat III miejsca

(35) **Krzysztof Król**, 3 klasa, XIV Liceum Ogólnokształcące im. Polonii Belgijskiej we Wrocławiu, 244 pkt., laureat III miejsca

(36) **Marcin Smulewicz**, 1 klasa, Liceum Ogólnokształcące im. Bolesława Prusa w Skierniewicach, 240 pkt., laureat III miejsca

(37) **Piotr Suwara**, 3 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica w Warszawie, 240 pkt., laureat III miejsca

(38) **Jan Wietrzykowski**, 3 klasa, III Liceum Ogólnokształcące im. św. Jana Kantego w Poznaniu, 229 pkt., laureat III miejsca

(39) **Dariusz Bukowski**, 3 klasa, XIV Liceum Ogólnokształcące im. Polonii Belgijskiej we Wrocławiu, 210 pkt., finalista z wyróżnieniem

(40) **Krzysztof Feluś**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie, 205 pkt., finalista z wyróżnieniem

(41) **Maciej Matraszek**, 2 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica w Warszawie, 202 pkt., finalista z wyróżnieniem

(42) **Franciszek Boehlke**, 2 klasa, XIII Liceum Ogólnokształcące w Szczecinie, 201 pkt., finalista z wyróżnieniem

(43) **Wojciech Łopata**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie, 200 pkt., finalista z wyróżnieniem

(44) **Wojciech Marczenko**, 2 klasa, XXVII Liceum Ogólnokształcące im. Tadeusza Czackiego w Warszawie, 200 pkt., finalista z wyróżnieniem

(45) **Michał Zając**, 1 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie, 200 pkt., finalista z wyróżnieniem

Lista pozostałych finalistów w kolejności alfabetycznej:

- **Marcin Baczyński**, 3 klasa, VIII Liceum Ogólnokształcące im. Władysława IV w Warszawie
- **Paweł Banaszewski**, 3 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP w Gdyni
- **Aleksandra Baranowska**, 3 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza w Białymstoku
- **Piotr Bejda**, 1 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie
- **Piotr Bryk**, 3 klasa, VIII Liceum Ogólnokształcące im. Stanisława Wyspiańskiego w Krakowie
- **Aleksander Bułanowski**, 3 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica w Warszawie
- **Piotr Chabierski**, 3 klasa gimnazjum, Gimnazjum nr 24 przy III LO w Gdyni

- **Andrzej Dorobisz**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie
- **Michał Flendrich**, 3 klasa, I Liceum Ogólnokształcące im. Tadeusza Kościuszki w Legnicy
- **Adam Furmanek**, 3 klasa, I Liceum Ogólnokształcące im. Króla Stanisława Leszczyńskiego w Jaśle
- **Rafał Gaweł**, 3 klasa, Liceum Ogólnokształcące w Zawoi Wilcznej
- **Karol Grodzicki**, 3 klasa, Gdańskie Liceum Autonomiczne
- **Patrick Hess**, 2 klasa, II Liceum Ogólnokształcące im. Króla Jana III Sobieskiego w Krakowie
- **Mateusz Jurczyk**, 2 klasa, VIII Liceum Ogólnokształcące im. Marii Skłodowskiej-Curie w Katowicach
- **Aleksander Kędzierski**, 3 klasa, XIII Liceum Ogólnokształcące w Szczecinie
- **Mateusz Konieczny**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie
- **Radosław Kotowski**, 2 klasa, VI Liceum Ogólnokształcące im. Króla Zygmunta Augusta w Białymstoku
- **Michał Kowalczyk**, 3 klasa gimnazjum, Gimnazjum nr 50 w Bydgoszczy
- **Aleksander Kramarz**, 1 klasa, VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich w Bydgoszczy
- **Wiktor Kuropatwa**, 1 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie
- **Jan Kwaśniak**, 3 klasa, Liceum Ogólnokształcące Sióstr Nazaretanek im. św. Jadwigi Królowej w Kielcach
- **Mateusz Lewandowski**, 3 klasa, II Liceum Ogólnokształcące im. Marii Konopnickiej w Inowrocławiu
- **Anna Lewicka**, 3 klasa, Publiczne Liceum Ogólnokształcące Stowarzyszenia Rodzin Katolickich Archidiecezji Katowickiej im. Kardynała Prymasa Augusta Hlonda w Chorzowie
- **Miłosz Makowski**, 1 klasa, ZS UMK, Gimnazjum i Liceum Akademickie w Toruniu
- **Tomasz Obuchowski**, 3 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza w Białymstoku
- **Damian Orlef**, 2 klasa, III Liceum Ogólnokształcące w Zabrzu
- **Piotr Pakosz**, 2 klasa, Zespół Szkół nr 1 im. Jana Pawła II w Przysusze
- **Łukasz Piotrowski**, 3 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica w Warszawie
- **Paweł Przytarski**, 3 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP w Gdyni
- **Marcin Regdos**, 2 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie
- **Michał Richter**, 3 klasa, Publiczne Liceum Ogólnokształcące nr II z Oddziałami Dwujęzycznymi im. Marii Konopnickiej w Opolu

- **Andrzej Rumiński**, 1 klasa, ZS UMK, Gimnazjum i Liceum Akademickie w Toruniu
- **Paweł Seta**, 3 klasa, VI Liceum Ogólnokształcące im. Jana Kochanowskiego w Radomiu
- **Łukasz Solak**, 3 klasa, VIII Liceum Ogólnokształcące im. Marii Skłodowskiej-Curie w Katowicach
- **Krzysztof Story**, 1 klasa, XIV Liceum Ogólnokształcące im. Polonii Belgijskiej we Wrocławiu
- **Kamil Sutkowski**, 3 klasa, X Liceum Ogólnokształcące we Wrocławiu
- **Maciej Szeptuch**, 2 klasa, XIV Liceum Ogólnokształcące im. Polonii Belgijskiej we Wrocławiu
- **Aleksander Szymanek**, 1 klasa, XIV Liceum Ogólnokształcące im. Polonii Belgijskiej we Wrocławiu
- **Bartosz Tarnawski**, 1 klasa, Katolickie Liceum Ogólnokształcące im. bł. ks. Emila Szramka
- **Mateusz Wasilewski**, 3 klasa, Zespół Szkół Elektronicznych i Samochodowych im. Marii Skłodowskiej-Curie w Zielonej Górze
- **Paweł Wawruch**, 3 klasa, Liceum Ogólnokształcące w Milanówku
- **Jakub Woyke**, 3 klasa, XIII Liceum Ogólnokształcące w Szczecinie

Komitet Główny Olimpiady Informatycznej przyznał następujące nagrody rzeczowe:

(1) puchary ufundowane przez Olimpiadę Informatyczną przyznano zwycięzcom XVII Olimpiady Adrianowi Jaskółce i Janowi Kantemu Milczkowi,
(2) złote, srebrne i brązowe medale ufundowane przez MEN przyznano odpowiednio laureatom I, II i III miejsca,
(3) notebooki (4 szt.) ufundowane przez Asseco Poland SA i MEN przyznano laureatom I miejsca,
(4) netbooki (7 szt.) ufundowane przez Asseco Poland SA przyznano laureatom II miejsca,
(5) dyski zewnętrzne (14 szt.) ufundowane przez Asseco Poland SA przyznano pozostałym laureatom II miejsca oraz ośmiu laureatom III miejsca,
(6) aparaty fotograficzne (20 szt.) ufundowane przez Ogólnopolską Fundację Edukacji Komputerowej przyznano następnym laureatom III miejsca oraz wyróżnionym finalistom,
(7) odtwarzacze mp3 (42 szt.) ufundowane przez Ogólnopolską Fundację Edukacji Komputerowej przyznano wszystkim finalistom,
(8) książki ufundowane przez PWN przyznano wszystkim laureatom i finalistom,
(9) roczną prenumeratę miesięcznika „Delta” przyznano wszystkim laureatom.

Komitet Główny powołał reprezentacje na:

- **Międzynarodową Olimpiadę Informatyczną IOI'2010**, która odbędzie się w Kanadzie, w miejscowości Waterloo, w terminie 14–21 sierpnia 2010 roku:
  (1) Adrian Jaskółka

(2) Jan Kanty Milczek
(3) Igor Adamski
(4) Anna Piekarska

rezerwowi:

(5) Michał Zgliczyński
(6) Dawid Dąbrowski

- **Olimpiadę Informatyczną Krajów Europy Środkowej CEOI'2010**, która odbędzie się na Słowacji, w miejscowości Koszyce, w terminie 12–19 lipca 2010 roku:

  (1) Adrian Jaskółka
  (2) Jan Kanty Milczek
  (3) Igor Adamski
  (4) Anna Piekarska

  rezerwowi:

  (5) Michał Zgliczyński
  (6) Dawid Dąbrowski

- **Bałtycką Olimpiadę Informatyczną BOI'2010**, która odbędzie się w Estonii, w miejscowości Tartu, w terminie 30 kwietnia – 4 maja 2010 roku:

  (1) Jan Kanty Milczek
  (2) Łukasz Jocz
  (3) Krzysztof Leszczyński
  (4) Karol Pokorski
  (5) Maciej Borsz
  (6) Paweł Lipski

  rezerwowi:

  (7) Stanisław Barzowski
  (8) Krzysztof Pszeniczny

  W związku z problemami z paszportem Jan Kanty Milczek zrezygnował z udziału w BOI, jego miejsce zajął Stanisław Barzowski.

  Polscy reprezentanci uzyskali następujące wyniki:

  – Paweł Lipski – złoty medal
  – Łukasz Jocz – srebrny medal
  – Krzysztof Leszczyński – srebrny medal
  – Stanisław Barzowski – brązowy medal

- **Obóz czesko-polsko-słowacki**, który odbędzie się w Pradze w terminie 22–27 czerwca 2010 roku:
  – reprezentacja na IOI i CEOI wraz z rezerwowymi.

- **Obóz im. Antoniego Kreczmara**, który odbędzie się w Krynicy Górskiej, w terminie 20 lipca – 1 sierpnia 2010 roku:

  - reprezentanci na międzynarodowe zawody informatyczne, zawodnicy rezerwowi oraz wszyscy laureaci i finaliści, którzy uczęszczają do klas niższych niż maturalna.

Sekretariat wystawił łącznie 38 zaświadczeń o uzyskaniu tytułu laureata, 7 zaświadczeń o uzyskaniu tytułu wyróżnionego finalisty oraz 42 zaświadczenia o uzyskaniu tytułu finalisty XVII Olimpiady Informatycznej.

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

- Jakub Adamek (student Uniwersytetu Jagiellońskiego, Kraków)
  - Kamil Sałaś — laureat III miejsca
- Igor Adamski (uczeń V Liceum Ogólnokształcącego im. Augusta Witkowskiego, Kraków)
  - Tomasz Wiatrowski — laureat III miejsca
- Marcin Andrychowicz (student Uniwersytetu Warszawskiego)
  - Piotr Pakosz — finalista
- Ludmiła Bałaj (X Liceum Ogólnokształcące, Wrocław)
  - Kamil Sutkowski — finalista
- Michał Bejda (student Uniwersytetu Jagiellońskiego, Kraków)
  - Kamil Sałaś — laureat III miejsca
- Mariusz Blank (Alcatel – Lucent, Bydgoszcz)
  - Łukasz Kalinowski — laureat II miejsca
- Jarosław Błasiok (student Uniwersytetu Warszawskiego)
  - Mateusz Jurczyk — finalista
  - Łukasz Solak — finalista
- Ireneusz Bujnowski (I Liceum Ogólnokształcące im. Adama Mickiewicza, Białystok)
  - Adrian Jaskółka — laureat I miejsca
  - Łukasz Jocz — laureat II miejsca
  - Michał Makarewicz — laureat II miejsca
  - Aleksandra Baranowska — finalistka
  - Radosław Kotowski — finalista
  - Tomasz Obuchowski — finalista
- Marek Cygan (doktorant Uniwersytetu Warszawskiego)
  - Maciej Borsz — laureat III miejsca
  - Michał Kowalczyk — finalista
  - Aleksander Kramarz — finalista
- Piotr Dobosiewicz (Zespół Szkół Ogólnokształcących nr 1, Bydgoszcz)
  - Łukasz Kalinowski — laureat II miejsca
- Czesław Drozdowski (XIII Liceum Ogólnokształcące, Szczecin)
  - Alan Kutniewski — laureat II miejsca

  - Franciszek Boehlke — finalista z wyróżnieniem
  - Aleksander Kędzierski — finalista
  - Jakub Woyke — finalista
- Lech Duraj (Uniwersytet Jagielloński, Kraków)
  - Igor Adamski — laureat I miejsca
  - Maciej Piekarz — laureat II miejsca
  - Michał Zgliczyński — laureat II miejsca
  - Grzegorz Guśpiel — laureat III miejsca
  - Tomasz Wiatrowski — laureat III miejsca
  - Michał Zając — finalista z wyróżnieniem
  - Piotr Bejda — finalista
  - Andrzej Dorobisz — finalista
- Andrzej Dyrek (V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków)
  - Igor Adamski — laureat I miejsca
  - Maciej Piekarz — laureat II miejsca
  - Michał Zgliczyński — laureat II miejsca
  - Grzegorz Guśpiel — laureat III miejsca
  - Tomasz Wiatrowski — laureat III miejsca
  - Krzysztof Feluś — finalista z wyróżnieniem
  - Wojciech Łopata — finalista z wyróżnieniem
  - Michał Zając — finalista z wyróżnieniem
  - Andrzej Dorobisz — finalista
  - Marcin Regdos — finalista
- Piotr Faliszewski (Akademia Górniczo-Hutnicza, Kraków)
  - Patrick Hess — finalista
- Patryk Flegel (II Liceum Ogólnokształcące im. Mikołaja Kopernika, Kędzierzyn-Koźle)
  - Grzegorz Milka — laureat II miejsca
- Bożena Janiga (I Liceum Ogólnokształcące im. Króla Stanisława Leszczyńskiego, Jasło)
  - Adam Furmanek — finalista
- Witold Jarnicki (Google, Kraków)
  - Kamil Sałaś — laureat III miejsca
- Tomasz Kociumaka (student Uniwersytetu Warszawskiego)
  - Paweł Lipski — laureat III miejsca
  - Anna Lewicka — finalistka
  - Damian Orlef — finalista
- Marcin Kościelnicki (student Uniwersytetu Warszawskiego)
  - Wojciech Lis — laureat III miejsca
- Ewa Kubik (VIII Liceum Ogólnokształcące im. Stanisława Wyspiańskiego, Kraków)

  - Piotr Bryk — finalista
- Anna Beata Kwiatkowska (Zespół Szkół Uniwersytetu Mikołaja Kopernika, Toruń)
  - Piotr Szefler — laureat III miejsca
  - Miłosz Makowski — finalista
  - Andrzej Rumiński — finalista
- Romualda Laskowska (I Liceum Ogólnokształcące im. Tadeusza Kościuszki, Legnica)
  - Michał Flendrich — finalista
- Piotr Leszczyński (student Uniwersytetu Warszawskiego)
  - Krzysztof Leszczyński — laureat II miejsca
- Paweł Lipski (uczeń Liceum Ogólnokształcącego „Filomata", Gliwice)
  - Damian Orlef — finalista
- Ryszard Lisoń (Publiczne Liceum Ogólnokształcące nr II z Oddziałami Dwujęzycznymi im. Marii Konopnickiej, Opole)
  - Michał Richter — finalista
- Jakub Łopuszański (Uniwersytet Wrocławski)
  - Michał Łowicki — laureat III miejsca
- Mirosław Mortka (VI Liceum Ogólnokształcące im. Jana Kochanowskiego, Radom)
  - Paweł Seta — finalista
- Piotr Niedźwiedź (student Uniwersytetu Warszawskiego)
  - Wojciech Marczenko — finalista z wyróżnieniem
- Rafał Nowak (Uniwersytet Wrocławski)
  - Krzysztof Story — finalista
  - Maciej Szeptuch — finalista
- Grzegorz Owsiany (Wyższa Szkoła Informatyki i Zarządzania, Rzeszów)
  - Krzysztof Pszeniczny — laureat III miejsca
- Andrzej Pezarski (doktorant Uniwersytetu Jagiellońskiego, Kraków)
  - Marcin Regdos — finalista
- Małgorzata Piekarska (VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich, Bydgoszcz)
  - Maciej Borsz — laureat III miejsca
  - Michał Kowalczyk — finalista
  - Aleksander Kramarz — finalista
- Henryk Pokorski (Starogard Gdański)
  - Karol Pokorski — laureat II miejsca
- Adam Polak (student Uniwersytetu Jagiellońskiego, Kraków)
  - Igor Adamski — laureat I miejsca
  - Maciej Piekarz — laureat II miejsca
  - Michał Zgliczyński — laureat II miejsca
  - Grzegorz Guśpiel — laureat III miejsca

  - Kamil Sałaś — laureat III miejsca
- Antoni Salamon (Katolickie Liceum Ogólnokształcące im. bł. ks. Emila Szramka, Katowice)
  - Bartosz Tarnawski — finalista
- Bartosz Szreder (student Uniwersytetu Warszawskiego)
  - Marcin Baczyński — finalista
  - Paweł Wawruch — finalista
- Ryszard Szubartowski (III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia)
  - Jan Kanty Milczek — laureat I miejsca
  - Dawid Dąbrowski — laureat II miejsca
  - Jakub Pachocki — laureat II miejsca
  - Stanisław Barzowski — laureat III miejsca
  - Michał Krasnoborski — laureat III miejsca
  - Szymon Sidor — laureat III miejsca
  - Paweł Walczak — laureat III miejsca
  - Paweł Banaszewski — finalista
  - Piotr Chabierski — finalista
  - Karol Grodzicki — finalista
  - Paweł Przytarski — finalista
- Marcin Szubert (Politechnika Poznańska)
  - Jan Wietrzykowski — laureat III miejsca
- Joanna Śmigielska (XIV Liceum Ogólnokształcące im. Stanisława Staszica, Warszawa)
  - Piotr Suwara — laureat III miejsca
  - Aleksander Bułanowski — finalista
- Przemysław Uznański (XIV Liceum Ogólnokształcące im. Polonii Belgijskiej, Wrocław)
  - Anna Piekarska — laureatka I miejsca
  - Janusz Wróbel — laureat II miejsca
  - Bartłomiej Dudek — laureat III miejsca
  - Krzysztof Król — laureat III miejsca
  - Jan Marcinkowski — laureat III miejsca
  - Dariusz Bukowski — finalista z wyróżnieniem
- Zbigniew Winiarski (II Liceum Ogólnokształcące im. Marii Konopnickiej, Inowrocław)
  - Mateusz Lewandowski — finalista
- Marcin Wrochna (student Uniwersytetu Warszawskiego)
  - Maciej Matraszek — finalista z wyróżnieniem

Zgodnie z decyzją Komitetu Głównego z dnia 16 kwietnia 2010 roku, opiekunowie naukowi laureatów i finalistów, będący nauczycielami szkół, otrzymają nagrody pieniężne.

Podobnie jak w ubiegłych latach w przygotowaniu jest publikacja zawierająca pełną informację o XVII Olimpiadzie Informatycznej, zadania konkursowe oraz wzorcowe rozwiązania. w publikacji tej znajdą się także zadania z międzynarodowych zawodów informatycznych. Ukaże się ona na początku nowego roku szkolnego.

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

Warszawa, 18 czerwca 2010 roku

Regulamin Olimpiady Informatycznej

§1 WSTĘP

Olimpiada Informatyczna, zwana dalej Olimpiadą, 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. Nr 13, poz. 125). Organizatorem Olimpiady Informatycznej jest Fundacja Rozwoju Informatyki, zwana dalej Organizatorem. W organizacji Olimpiady Fundacja Rozwoju Informatyki współdziała z Wydziałem Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego, Instytutem Informatyki Uniwersytetu Wrocławskiego, Katedrą Algorytmiki Uniwersytetu Jagiellońskiego, Wydziałem Matematyki i Informatyki Uniwersytetu im. Mikołaja Kopernika w Toruniu, Instytutem Informatyki Wydziału Automatyki, Elektroniki i Informatyki Politechniki Śląskiej w Gliwicach, Wydziałem Informatyki i Zarządzania Politechniki Poznańskiej, a także z innymi 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;
- 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 Głównym.

(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.

(8) 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.

(9) 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, czas działania tego programu oraz ilość wymaganej przez program pamięci. Jeśli rozwiązaniem zadania jest plik z danymi, wówczas ocenia się poprawność danych i na tej podstawie przyznaje punkty. Za każde zadanie uczeń może zdobyć maksymalnie 100 punktów, gdzie 100 jest sumą maksymalnych liczb punktów za poszczególne testy (lub dane z wynikami) dla tego zadania. Oceną rozwiązań ucznia jest suma punktów za poszczególne zadania. Oceny rozwiązań uczniów są podstawą utworzenia listy rankingowej zawodników po zawodach każdego stopnia.

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

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

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

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

(14) 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.

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

(16) 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.

(17) Na podstawie analizy rozwiązań zadań w zawodach III stopnia i listy rankingowej Komitet Główny przyznaje tytuły laureatów Olimpiady Informatycznej: I stopnia (prace na poziomie złotych medalistów Międzynarodowej Olimpiady Informatycznej), II stopnia (prace na poziomie srebrnych medalistów Międzynarodowej Olimpiady Informatycznej), III stopnia (prace na poziomie brązowych medalistów Międzynarodowej Olimpiady Informatycznej) i nagradza ich medalami, odpowiednio, złotymi, srebrnymi i brązowymi. Liczba laureatów nie przekracza połowy uczestników zawodów finałowych.

(18) W przypadku bardzo wysokiego poziomu finałów Komitet Główny może dodatkowo wyróżnić uczniów niebędących laureatami.

(19) Zwycięzcą Olimpiady Informatycznej zostaje osoba, która osiągnęła najlepszy wynik w zawodach finałowych.

§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) Komitet pracuje w dotychczasowym składzie, powołanym w roku 2008/2009.

(3) Komitet wybiera ze swego grona Prezydium. 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) zatwierdza listy rankingowe oraz listy laureatów i wyróżnionych uczestników;

   (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 zgodą Organizatora i za pośrednictwem kierownika organizacyjnego Olimpiady.

(12) Komitet przyjmuje plan finansowy na pierwszym posiedzeniu Komitetu w nowym roku szkolnym.

(13) Komitet przyjmuje sprawozdanie finansowe z przebiegu Olimpiady i przedkłada je Organizatorowi.

(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 Główny.

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

§6 PRZEBIEG OLIMPIADY

(1) Komitet Główny 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 Główny 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 MEN z 30 kwietnia 2007 r. w sprawie warunków i sposobu oceniania, klasyfikowania i promowania uczniów i słuchaczy oraz przeprowadzania sprawdzianów i egzaminów w szkołach publicznych (Dz. U. z 2007 r. Nr 83, poz. 562, §§ 20 i 60).

(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 27 lipca 2005 r. „Prawo o szkolnictwie wyższym", na zasadach zawartych w tych uchwałach (Dz. U. Nr 164, poz. 1365).

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

(6) Komitet Główny może nagrodzić opiekunów, których praca przy przygotowaniu uczestnika Olimpiady zostanie oceniona przez Komitet jako wyróżniająca.

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

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

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

§8 FINANSOWANIE OLIMPIADY

(1) Komitet Główny finansuje działania Olimpiady zgodnie z umową podpisaną przez Ministerstwo Edukacji Narodowej i Fundację Rozwoju Informatyki. Komitet Główny będzie także zabiegał o pozyskanie dotacji z innych organizacji wspierających.

§9 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 Główny zatwierdza sprawozdanie merytoryczne i przedstawia je Organizatorowi celem przedłożenia Ministerstwu Edukacji Narodowej.

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

Zasady organizacji zawodów XVII Olimpiady Informatycznej w roku szkolnym 2009/2010

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 2009/2010.

§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 Fundacja Rozwoju Informatyki. W organizacji Olimpiady Fundacja współdziała ze środowiskami akademickimi, zawodowymi i oświatowymi działającymi w sprawach edukacji informatycznej oraz firmą Asseco Poland SA.

§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 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 350 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 30%.

(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 — 19.10–16.11.2009 r.
  ogłoszenie wyników:
  - w witrynie Olimpiady — 11.12.2009 r.,
  - rozesłanie pocztą wyników oraz materiałów Olimpiady i Asseco — 15.12.2009 r.
- zawody II stopnia — 09–11.02.2010 r.
  ogłoszenie wyników w witrynie Olimpiady — 26.02.2010 r.
- zawody III stopnia — 13–17.04.2010 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`

(6) Programy w *C*/*C++* będą kompilowane w systemie Linux za pomocą kompilatora GCC/G++. Programy w *Pascalu* będą kompilowane w systemie Linux za pomocą kompilatora FreePascal. 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
```

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

(8) 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, zwany dalej SIO, o adresie: `http://sio.mimuw.edu.pl`. Komitet Główny nie ponosi odpowiedzialności za brak możliwości przekazania rozwiązań przez Internet w sytuacji nadmiernego obciążenia lub awarii systemu. Odbiór przesyłki zostanie potwierdzony przez SIO 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 rozwiązań i związanej z tym rejestracji będą podane w SIO.
- pocztą, jedną 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 16 listopada 2009 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 SIO 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 SIO 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.
- wypełniony dokument zgłoszeniowy (dostępny w witrynie internetowej Olimpiady).

(3) W trakcie rozwiązywania zadań można korzystać z dowolnej literatury oraz ogólnodostępnych kodów źródłowych. Należy wówczas podać w rozwiązaniu, w komentarzu, odnośnik do wykorzystanej literatury lub kodu.

(4) Podczas korzystania z SIO zawodnik postępuje zgodnie z instrukcjami umieszczonymi w witrynie systemu. W szczególności, warunkiem koniecznym do kwalifikacji zawodnika do dalszych etapów jest podanie lub aktualizacja w SIO wszystkich wymaganych danych osobowych.

(5) Każdy uczestnik powinien założyć w witrynie SIO dokładnie jedno konto. Zawodnicy korzystający z więcej niż jednego konta mogą zostać zdyskwalifikowani.

(6) Każde zadanie można zgłosić w SIO co najwyżej 10 razy. Spośród tych zgłoszeń ocenianie jest jedynie najpóźniejsze. Po wyczerpaniu tego limitu kolejne rozwiązanie może zostać zgłoszone już tylko zwykłą pocztą.

(7) W SIO 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 SIO. Komitet Główny może nie udzielić odpowiedzi na pytanie z ważnych przyczyn, m.in. gdy jest ono niejednoznaczne lub dotyczy sposobu rozwiązania zadania.

(8) Przez witrynę SIO udostępniane są narzędzia do sprawdzania rozwiązań pod względem formalnym. Szczegóły dotyczące sposobu postępowania są dokładnie podane w witrynie.

(9) Od 30.11.2009 r. poprzez System Internetowy Olimpiady każdy zawodnik będzie mógł zapoznać się ze wstępną oceną swojej pracy.

(10) Do 04.12.2009 r. (włącznie) poprzez SIO każdy zawodnik będzie mógł zgłaszać uwagi do wstępnej oceny swoich rozwiązań. Reklamacji nie podlega jednak dobór testów, limitów czasowych, kompilatorów i sposobu oceny.

(11) Reklamacje złożone po 04.12.2009 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) Komisje Regulaminowe powołane przez komitety okręgowe lub Komitet Główny czuwają nad prawidłowością przebiegu zawodów i pilnują przestrzegania Regulaminu Olimpiady i Zasad Organizacji Zawodów.

(5) Zawody II i III stopnia są przeprowadzane za pomocą 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 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 zadawać pytania, w ustalony przez Jury sposób, 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 dwóch godzin 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 punktach 1. i 2. przysługują na zasadach określonych w rozporządzeniu MEN z 30 kwietnia 2007 r. w sprawie warunków i sposobu oceniania, klasyfikowania i promowania uczniów i słuchaczy oraz przeprowadzania sprawdzianów i egzaminów w szkołach publicznych (Dz. U. z 2007 r. Nr 83, poz. 562, §§ 20 i 60).

(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 XXII Międzynarodową Olimpiadę Informatyczną w 2010 roku na podstawie wyników olimpiady oraz regulaminu tej Olimpiady Międzynarodowej.

(7) Komitet Główny może nagrodzić opiekunów, których praca przy przygotowaniu uczestnika Olimpiady zostanie oceniona przez ten Komitet jako wyróżniająca.

(8) Wyznaczeni przez Komitet Główny reprezentanci Polski na olimpiady międzynarodowe zostaną zaproszeni do nieodpłatnego udziału w XI Obozie Naukowo-Treningowym im. Antoniego Kreczmara, który odbędzie się w czasie wakacji 2010 r. Do nieodpłatnego udziału w Obozie Komitet Główny może zaprosić także innych finalistów, którzy nie są w ostatniej programowo klasie swojej szkoły, w zależności od uzyskanych wyników.

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

§7 PRZEPISY KOŃCOWE

(1) Komitet Główny zawiadamia wszystkich uczestników zawodów I i II stopnia o ich wynikach. Wszyscy uczestnicy zawodów I stopnia zostaną zawiadomieni o swoich wynikach zwykłą pocztą, a poprzez SIO będą mogli zapoznać się ze szczegółowym raportem ze sprawdzania ich rozwiązań.

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

(3) 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

Marcin Pilipczuk
Treść zadania, Opracowanie

Bartłomiej Wołowiec
Program

Dostępna pamięć: 64 MB.

OI, Etap I, 19.10–16.11.2009

Gildie

Król Bajtazar ma nie lada problem. Gildia Szwaczek oraz Gildia Krawców jednocześnie poprosiły o pozwolenie na otwarcie swoich filii w każdym z miast królestwa.

W Bajtocji jest n miast. Niektóre z nich są połączone dwukierunkowymi drogami. Każda z gildii wysunęła postulat, aby dla każdego miasta:

- *znajdowała się w nim filia danej gildii, lub*
- *miasto było bezpośrednio połączone drogą z miastem, w którym znajduje się filia tej gildii.*

Z drugiej strony, Król Bajtazar wie, że jeśli pozwoli w jednym mieście otworzyć filie obu gildii, to zapewne doprowadzi to do zmowy gildii i zmonopolizowania rynku odzieżowego. Dlatego też poprosił Cię o pomoc.

Wejście

W pierwszym wierszu standardowego wejścia podane są dwie liczby całkowite n oraz m ($1 \leqslant n \leqslant 200\,000$, $0 \leqslant m \leqslant 500\,000$), oznaczające odpowiednio liczbę miast i liczbę dróg w Bajtocji. Miasta są ponumerowane od 1 do n. W $(i+1)$-szym wierszu wejścia znajduje się opis i-tej drogi; zawiera on liczby a_i oraz b_i ($1 \leqslant a_i, b_i \leqslant n$, $a_i \neq b_i$) oznaczające, że i-ta droga łączy miasta a_i oraz b_i. Każda para miast jest połączona co najwyżej jedną drogą. Drogi nie krzyżują się — jedynie mogą spotykać się w miastach — choć mogą prowadzić tunelami i estakadami.

Wyjście

W pierwszym wierszu standardowego wyjścia Twój program powinien wypisać jedno słowo: `TAK` *— jeśli da się rozmieścić filie gildii w miastach zgodnie z warunkami zadania, lub* `NIE` *— w przeciwnym przypadku. W przypadku odpowiedzi* `TAK`*, w kolejnych n wierszach powinien znaleźć się opis przykładowego rozmieszczenia filii. $(i+1)$-szy wiersz powinien zawierać:*

- *literę* `K`*, jeśli w mieście i ma się znaleźć filia gildii krawców, lub*
- *literę* `S`*, jeśli w mieście i ma się znaleźć filia gildii szwaczek, lub*
- *literę* `N`*, jeśli w mieście i nie ma się znaleźć filia żadnej z dwóch gildii.*

Przykład

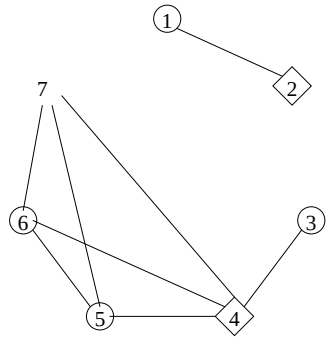
Dla danych wejściowych:

```
7 8
1 2
```

```
3 4
5 4
6 4
7 4
5 6
5 7
6 7
```

poprawnym wynikiem jest:

```
TAK
K
S
K
S
K
K
N
```



Miasta, w których ma zostać otwarta gildia krawców, są zaznaczone kółkami, a te, w których ma zostać otwarta gildia szwaczek, są zaznaczone rombami.

Rozwiązanie

Wprowadzenie

Zadanie *Gildie* było zadaniem prostym, choć dość nietypowym. W przypadku odpowiedzi `TAK`, przykładowe rozmieszczenie gildii można skonstruować na wiele różnych sposobów — intuicyjnie, jest duża swoboda w trakcie konstrukcji. Główną częścią rozwiązania było właśnie wyznaczenie poprawnego rozmieszczenia.

Zacznijmy od prostej obserwacji. Załóżmy, że mamy dane przykładowe poprawne rozmieszczenie filii (takie, w którym postulaty obu gildii są spełnione). Jeśli w każdym mieście, w którym nie postawiliśmy żadnej gildii (literka `N` na wyjściu), postawimy dowolną z gildii, rozmieszczenie wciąż pozostanie poprawne. Dlatego też wystarczy rozważać tylko te rozmieszczenia, w których w każdym mieście jest filia jednej z gildii.

Przetłumaczmy teraz treść zadania na język teorii grafów. Mamy dany nieskierowany graf G, w którym miasta Bajtocji tworzą zbiór wierzchołków V, a drogi łączące miasta — zbiór krawędzi E. Naszym zadaniem jest podzielić zbiór wierzchołków V na dwie rozłączne części V_K i V_S — lokalizacje dla filii odpowiednio gildii krawców i szwaczek — tak, by każdy wierzchołek $v \in V$ spełniał dwa warunki:

- $v \in V_K$ lub jakiś sąsiad v należy do V_K,
- $v \in V_S$ lub jakiś sąsiad v należy do V_S.

Wprowadźmy teraz kilka pojęć z teorii grafów. Powiemy, że wierzchołek v *dominuje* wierzchołek w, jeśli $v = w$ lub v jest sąsiadem w. Dla danego zbioru wierzchołków S powiemy, że wierzchołek w jest *zdominowany* przez S, jeśli istnieje w S wierzchołek dominujący w. Zbiór wierzchołków $D \subseteq V$ nazwiemy *zbiorem dominującym*, jeśli

dominuje on wszystkie wierzchołki w grafie. Wyposażeni w nową definicję, możemy przeformułować nasze zadanie tak: mamy podzielić zbiór wierzchołków V na dwa rozłączne zbiory dominujące V_K i V_S.

Liczba domatyczna (ang. *domatic number*) grafu G to największa liczba całkowita k taka, że zbiór wierzchołków V da się podzielić na k parami rozłącznych zbiorów dominujących. W zadaniu mamy więc sprawdzić, czy liczba domatyczna podanego grafu (miast i dróg w Bajtocji) wynosi co najmniej 2. W ogólności, wyznaczenie liczby domatycznej grafu jest problemem NP-trudnym, to znaczy, że prawdopodobnie nie da się jej obliczyć w czasie wielomianowym ze względu na wielkość grafu. Jednak, jak pokażemy, rozstrzygnięcie, czy liczba domatyczna wynosi 1, czy też jest nie mniejsza niż 2, jest dość proste.

Zauważmy, iż jeśli w grafie istnieje wierzchołek, z którego nie wychodzą żadne krawędzie (tzw. wierzchołek izolowany), to odpowiedź brzmi `NIE`. Istotnie, jeśli w tym wierzchołku postawimy filię jednej z gildii, to nie mamy gdzie postawić filii drugiej gildii tak, by zdominowała ten wierzchołek. Nietrudno dostrzec, że sprawdzenie, czy w grafie jest wierzchołek izolowany, możemy wykonać w czasie $O(|V|+|E|)$. Wystarczy zliczyć stopnie wierzchołków.

Pokażemy, że jeśli w grafie nie ma wierzchołków izolowanych, to odpowiedź w zadaniu brzmi `TAK`. Dowodząc tego, opiszemy trzy różne sposoby konstrukcji podziału zbioru wierzchołków na dwa zbiory dominujące V_K i V_S. Wszystkie konstrukcje będą działały w czasie $O(|V| + |E|)$. Otrzymamy więc trzy rozwiązania o liniowej złożoności czasowej. Istnieje też mnóstwo innych konstrukcji działających w czasie liniowym. Uznajmy jednak, że na potrzeby tego opracowania wystarczy nam opis trzech.

Rozwiązania wzorcowe

Jak już zauważyliśmy we wprowadzeniu, jeśli w Bajtocji istnieje miasto, z którego nie wychodzi żadna droga, odpowiedź brzmi `NIE`. Pokażemy trzy różne konstrukcje rozmieszczenia filii gildii, przy założeniu, że z każdego miasta wychodzi choć jedna droga. Każda konstrukcja będzie wypełniała tablicę *gildia*, przypisując gildie kolejnym miastom. Początkowo tablica *gildia* jest zainicjowana na wartości N (brak przypisania), w miarę działania algorytmu będziemy przypisywać miastom wartości K (gildia krawców) lub S (gildia szwaczek).

Konstrukcja pierwsza

W tym algorytmie przeglądamy listę krawędzi w dowolnej kolejności. Dla każdej krawędzi:

1. jeśli oba końce krawędzi mają już przypisane gildie, nic nie robimy,
2. jeśli jeden koniec krawędzi ma przypisaną gildię, przypisujemy inną gildię drugiemu końcowi tej krawędzi,
3. jeśli żaden z końców krawędzi nie ma przypisanej gildii, przypisujemy w dowolny sposób obu końcom różne gildie.

Oto pseudokod powyższego rozwiązania:

```
1:  procedure UstawPrzeciwna(v, w)
2:  begin
3:    if gildia[v] = N then begin
4:      if gildia[w] = S then gildia[v] := K
5:      else gildia[v] := S;
6:    end
7:  end
8:
9:  begin
10:   for v := 1 to n do gildia[v] := N;
11:   foreach (v, w) ∈ E do begin
12:     UstawPrzeciwna(v, w);
13:     UstawPrzeciwna(w, v);
14:   end
15: end
```

Spróbujmy teraz wykazać, że powyższe rozwiązanie konstruuje poprawne przypisanie gildii miastom. Wpierw zauważmy, że każde miasto będzie miało przypisaną jakąś gildię: skoro nie ma wierzchołków izolowanych, każdy wierzchołek jest incydentny z jakąś krawędzią, a zatem w momencie przeglądania tej krawędzi przypiszemy wierzchołkowi gildię, jeśli nie była przypisana wcześniej.

By dokończyć dowód poprawności tej konstrukcji, wystarczy zauważyć jedno: w momencie, gdy przypisujemy wierzchołkowi v jakąś gildię (dla ustalenia uwagi: gildię krawców), wierzchołek v ma już pewnego sąsiada w, któremu przypisana jest gildia szwaczek (być może właśnie ją przypisujemy). Istotnie, w chwili, gdy przeglądamy krawędź (v, w) i przypisujemy w wyniku tego przeglądania wierzchołkowi v gildię krawców, to albo do w już była przypisana gildia szwaczek (przypadek drugi algorytmu), albo przypiszemy ją w bieżącym kroku (przypadek trzeci algorytmu). Tą sprytną obserwacją zakończyliśmy dowód poprawności pierwszej konstrukcji.

Implementację takiego rozwiązania można znaleźć w pliku `gil1.cpp`.

Konstrukcja druga

W tym algorytmie przeglądamy listę wierzchołków w dowolnej kolejności. Dla każdego wierzchołka, jeśli nie ma on jeszcze przypisanej gildii, przypisujemy mu gildię krawców, zaś wszystkim jego sąsiadom, którym jeszcze nie przypisaliśmy gildii, przypisujemy gildię szwaczek. Oto pseudokod:

```
1:  begin
2:    for v := 1 to n do gildia[v] := N;
3:    for v := 1 to n do
4:      if gildia[v] = N then begin
5:        gildia[v] := K;
6:        foreach w : (v, w) ∈ E do
7:          if gildia[w] = N then
```

```
8:              gildia[w] := S;
9:      end
10: end
```

Przejdźmy do dowodu poprawności algorytmu. Oczywiście, każde miasto będzie miało przypisaną jakąś gildię. Spójrzmy teraz na dowolny wierzchołek w, któremu przypisaliśmy gildię szwaczek. Skoro to uczyniliśmy, musiał on być sąsiadem jakiegoś wierzchołka v, któremu przypisaliśmy gildię krawców. Zatem zbiór wierzchołków, którym przypisaliśmy gildię krawców, jest zbiorem dominującym.

Z drugiej strony zauważmy, że żadne dwa wierzchołki, którym przypisaliśmy gildię krawców, nie mogą być sąsiadujące. Jeśli bowiem jakiemuś wierzchołkowi przypisaliśmy gildię krawców, automatycznie wszyscy jego nieprzypisani sąsiedzi otrzymali filię gildii szwaczek. Innymi słowy, wszyscy sąsiedzi wierzchołków z filią gildii krawców mają przypisaną gildię szwaczek. Skoro każdy wierzchołek ma co najmniej jednego sąsiada, miasta z gildią szwaczek tworzą zbiór dominujący.

Można powyższą konstrukcję wyrazić trochę inaczej. *Zbiorem niezależnym* w grafie G nazwiemy taki zbiór wierzchołków I, że żadne dwa wierzchołki z tego zbioru nie są połączone krawędzią. Powyższy algorytm można zapisać następująco:

1. Znajdź dowolny maksymalny w sensie zawierania zbiór niezależny I.
2. Przypisz wszystkim wierzchołkom z I gildię krawców, a pozostałym wierzchołkom gildię szwaczek.

Implementację tej wersji rozwiązania wzorcowego można znaleźć w pliku `gil3.cpp`.

Konstrukcja trzecia

W tym algorytmie przeglądamy listę wierzchołków w dowolnej kolejności. Załóżmy, że rozpatrujemy wierzchołek v, który jeszcze nie ma przypisanej gildii. Wówczas będziemy przypisywać gildie wszystkim wierzchołkom w całej spójnej składowej, do której należy v. Innymi słowy, przeglądamy wszystkie spójne składowe grafu G i dla każdej z nich niezależnie wybieramy przypisanie miastom gildii.

Załóżmy więc, że analizujemy wierzchołek v należący do pewnej spójnej składowej H. Konstruujemy dowolne drzewo rozpinające H (np. drzewo przeszukiwania w głąb) i ukorzeniamy je w v. Wierzchołkowi w przyporządkowujemy gildię krawców, jeśli jego odległość od v w drzewie rozpinającym jest parzysta, zaś gildię szwaczek, jeśli nieparzysta. Oto pseudokod tego rozwiązania, wykorzystujący drzewo przeszukiwania w głąb:

```
1: procedure DFS(v, g)
2: begin
3:   if gildie[v] = N then begin
4:     gildie[v] := g;
5:     if g = S then g := K else g := S;
6:     foreach w : (v, w) ∈ E do
7:       DFS(w, g);
8:   end
```

```
 9: end
10:
11: begin
12:    for v := 1 to n do gildia[v] := N;
13:    for v := 1 to n do DFS(v, K);
14: end
```

Uzasadnijmy poprawność przedstawionej konstrukcji. Oczywiście, każdemu wierzchołkowi przypiszemy jakąś gildię. Jeśli wierzchołek v nie był korzeniem drzewa rozpinającego, to jego ojcu w tym drzewie przypiszemy inną gildię niż v, czyli wierzchołek v będzie zdominowany przez obie gildie. Z drugiej strony, jeśli v jest korzeniem drzewa rozpinającego, to ma co najmniej jednego syna w tym drzewie (v nie jest izolowany), czyli ma sąsiada o innej gildii niż on sam. Tym samym, opisany algorytm konstruuje poprawne przyporządkowanie miastom gildii.

Implementacje takiego rozwiązania można znaleźć w plikach `gil.cpp` i `gil2.pas`.

Rozwiązania alternatywne

Rozwiązanie polegające na rozpatrzeniu wszystkich możliwych przyporządkowań miastom gildii działa w czasie $O\left((|V|+|E|)\cdot 3^{|V|}\right)$ lub $O\left((|V|+|E|)\cdot 2^{|V|}\right)$, jeśli uwzględnimy obserwację, że możemy w każdym mieście umieścić filię którejś gildii. Takie rozwiązania otrzymywały w tym zadaniu pojedyncze punkty.

Nie znamy żadnych ciekawych rozwiązań *pośrednich*. Wszystkie rozwiązania działające istotnie szybciej niż rozwiązania przeszukujące wszystkie możliwe odpowiedzi, lecz istotnie wolniej niż wzorcowe, okazywały się niepotrzebnie przekomplikowanymi rozwiązaniami wzorcowymi.

W tym zadaniu można też było wymyślić różnorakie rozwiązania niepoprawne. Dla przykładu, rozważmy następującą heurystykę. Przeglądamy wszystkie wierzchołki w dowolnej kolejności. Jeśli dany wierzchołek v ma jakiegoś sąsiada z przypisaną gildią, przyporządkowujemy v inną gildię niż ma ów sąsiad. W przeciwnym wypadku przyporządkowujemy v gildię inną niż ostatnio przypisana. To rozwiązanie może działać niepoprawnie już dla bardzo małego grafu: ścieżki złożonej z trzech wierzchołków. Jeśli wpierw rozważymy oba końce tej ścieżki, przypiszemy im różne gildie. Wówczas cokolwiek przypiszemy środkowemu wierzchołkowi, nie otrzymamy poprawnego przyporządkowania.

Testy

Zadanie było sprawdzane na 22 zestawach danych testowych, pogrupowanych w 12 grup testów.

Nazwa	n	m	Opis
gil1a.in	100	95	drzewa i ścieżki po 10 miast, dodatkowe losowe krawędzie

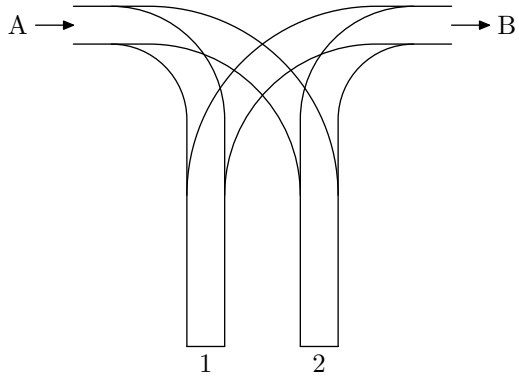
Nazwa	n	m	Opis
gil1b.in	1	0	najmniejszy test
gil2a.in	972	946	drzewa i ścieżki po 2-50 miast, dodatkowe losowe krawędzie
gil2b.in	982	946	test *2a* plus 10 wierzchołków izolowanych
gil3a.in	1 073	1 827	drzewa po 2-30 miast, ponad 1800 losowych krawędzi
gil3b.in	1 074	1 827	test *3a* plus jeden wierzchołek izolowany
gil4a.in	183 736	173 226	drzewa 5-30 miast
gil4b.in	200 000	0	duży test bez krawędzi
gil5a.in	191 726	497 272	ścieżka łącząca wszystkie miasta, dodatkowo dużo krawędzi losowych
gil5b.in	191 727	497 272	test *5a* plus jeden wierzchołek izolowany
gil6.in	192 837	499 231	duży losowy test
gil7a.in	58 263	291 314	losowy test, odpowiedź `NIE`
gil7b.in	59 273	49 422	dużo małych drzew po 2-10 miast
gil8a.in	196 372	196 371	ścieżka łącząca wszystkie miasta
gil8b.in	197 253	197 335	ścieżka łącząca wszystkie miasta, dodatkowo losowe krawędzie
gil9.in	198 372	198 356	ścieżki po 1 000-10 000 miast
gil10a.in	196 382	196 446	ścieżki po 1 000-30 000 miast, dodatkowe losowe krawędzie
gil10b.in	200 000	500 000	cykl z wszystkimi miastami, dodatkowo drogi między miastami odległymi o 2 i między co drugą parą miast odległych o 3
gil11a.in	197 332	197 318	ścieżki i drzewa po 2-50 000 miast
gil11b.in	197 382	197 318	test *11a* plus 50 wierzchołków izolowanych
gil12a.in	199 233	365 871	zawiera pełną klikę 600 wierzchołków, a poza tym losowe ścieżki i drzewa po 2-30 wierzchołków
gil12b.in	199 234	365 871	test *12a* plus jeden wierzchołek izolowany

Bartosz Walczak — Treść zadania, Opracowanie
Michał Pilipczuk — Program

Dostępna pamięć: 128 MB. *OI, Etap I, 19.10–16.11.2009*

Kolej

Bocznica kolejowa składa się z dwóch ślepo zakończonych torów 1 i 2. Wjazd na bocznicę odbywa się torem A, a wyjazd torem B (patrz rysunek poniżej).



Na torze A stoi n wagonów ponumerowanych od 1 do n. Są one ustawione w kolejności $a_1, a_2, \ldots, a_n$ *(tzn. w takiej kolejności wjeżdżają na bocznicę). Trzeba je tak przetoczyć przez bocznicę, aby opuściły ją torem B w kolejności* $1, 2, \ldots, n$*. Każdy wagon można dokładnie raz przetoczyć z toru A na jeden z torów 1 lub 2, oraz dokładnie raz z toru 1 lub 2 na tor B. W każdej chwili na każdym z torów 1, 2 może czekać dowolnie wiele wagonów.*

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę naturalną n *(*$1 \leqslant n \leqslant 100\,000$*), oznaczającą liczbę wagonów do przetoczenia. W drugim wierszu znajdują się liczby* $a_1, a_2, \ldots, a_n$ *będące permutacją liczb* $1, 2, \ldots, n$ *(czyli każda z liczb* a_i *należy do zbioru* $\{1, 2, \ldots, n\}$ *oraz wszystkie te liczby są różne), pooddzielane pojedynczymi odstępami.*

Wyjście

Pierwszy wiersz standardowego wyjścia powinien zawierać słowo `TAK`*, jeśli istnieje sposób uporządkowania wagonów w kolejności* $1, 2, \ldots, n$ *poprzez przetaczanie ich przez bocznicę, albo słowo* `NIE`*, jeśli taki sposób nie istnieje. W przypadku, gdy odpowiedzią jest* `TAK`*, drugi wiersz powinien zawierać pooddzielane pojedynczymi odstępami numery torów bocznicy (1 lub 2), na które są wtaczane kolejne wagony* $a_1, a_2, \ldots, a_n$ *w pewnym poprawnym sposobie ich uporządkowania. Jeżeli istnieje wiele możliwych sposobów uporządkowania wagonów, należy wypisać dowolny z nich.*

Przykład

Dla danych wejściowych:	*jednym z poprawnych wyników jest:*
`4`	`TAK`
`1 3 4 2`	`1 1 2 1`
Natomiast dla danych wejściowych:	*poprawnym wynikiem jest:*
`4`	`NIE`
`2 3 4 1`	

Wyjaśnienie do przykładu: *W pierwszym przykładzie zaczynamy od odstawienia wagonu numer 1 na pierwszą bocznicę, po czym zaraz ten sam wagon zjeżdża na tor B. Następnie wagon numer 3 odstawiamy na pierwszą bocznicę, wagon numer 4 — na drugą, wreszcie wagon numer 2 trafia na pierwszą bocznicę. Na koniec z pierwszej bocznicy zjeżdżają na tor B kolejno wagony o numerach 2 i 3, po czym z drugiej bocznicy zjeżdża wagon numer 4.*

Rozwiązanie

Wprowadzenie

Problem z zadania można łatwo przetłumaczyć na język matematyki. Dana jest permutacja $a_1, \ldots, a_n$ liczb od 1 do n. Kolejne jej elementy możemy odkładać na jeden z dwóch stosów. Możemy także zdejmować elementy ze stosów, tworząc drugą permutację. Przy tym z każdego stosu możemy zdjąć tylko element znajdujący się na jego szczycie, czyli element, który został odłożony na ten stos najpóźniej. Pytanie brzmi: czy za pomocą takich dwóch stosów możemy posortować permutację $a_1, \ldots, a_n$? A jeżeli możemy, to jak wygląda jej *algorytm sortowania*, czyli sekwencja ruchów polegających na odkładaniu elementów na stosy i zdejmowaniu elementów ze stosów, prowadzących do posortowania permutacji?

Obserwacja 1. *Każdy algorytm sortowania permutacji utrzymuje na obu stosach malejący porządek elementów (przeglądając stos od spodu do szczytu).*

Dowód: Jeżeli $a_i < a_j$, to element a_i jest zdejmowany ze stosu wcześniej niż a_j. Jeżeli więc a_i i a_j są na tym samym stosie, to a_i musi być bliżej szczytu stosu niż a_j. ■

Działanie każdego algorytmu sortowania permutacji można podzielić na n kroków. W i-tym kroku algorytm odkłada element a_i na jeden ze stosów, po czym zdejmuje ze stosów pewne elementy (być może nie zdejmuje żadnego).

Dla każdego i zdefiniujmy $\ell(i) = \max\{j : a_j \leqslant a_i\}$, jest to indeks w permutacji $a_1, \ldots, a_n$ ostatniego elementu nie większego niż a_i. Żaden element a_i nie może być zdjęty ze stosu wcześniej niż w kroku $\ell(i)$, gdyż dopiero w tym kroku przychodzi element $a_{\ell(i)} \leqslant a_i$. Poniższa obserwacja pokazuje, że można go zdjąć dokładnie w kroku $\ell(i)$.

Obserwacja 2. *Każdy algorytm sortowania permutacji $a_1, \ldots, a_n$ można tak zmodyfikować, aby zdejmował ze stosu każdy element a_i w kroku $\ell(i)$.*

Dowód: Niech A będzie dowolnym algorytmem sortowania permutacji $a_1, \ldots, a_n$. Algorytm A' działa następująco: każdy przychodzący element odkłada na ten sam stos, co algorytm A, oraz każdy element a_i usuwa ze stosu w kroku $\ell(i)$ (bez względu na to, czy element ten jest na szczycie stosu). Przy tym elementy przeznaczone do usunięcia w jednym kroku są usuwane w kolejności rosnącej. Pokażemy, że jest to poprawny algorytm sortowania permutacji $a_1, \ldots, a_n$ za pomocą dwóch stosów.

Ponieważ dla $a_i < a_j$ mamy $\ell(i) \leqslant \ell(j)$, więc elementy są usuwane ze stosów we właściwej kolejności. Musimy jeszcze uzasadnić, że gdy algorytm próbuje usunąć element ze stosu, operacja ta jest dozwolona, czyli usuwany element jest na szczycie stosu. Ponieważ (z definicji) $i \leqslant \ell(i)$, więc usuwany element rzeczywiście znajduje się na stosie. Ponadto, algorytm A' utrzymuje malejący porządek na stosach, ponieważ odkłada elementy na stosy tak samo jak algorytm A i zdejmuje je nie później niż algorytm A, a algorytm A utrzymuje malejący porządek na stosach na podstawie Obserwacji 1. Zatem w chwili, gdy algorytm A' próbuje usunąć element ze stosu, jest to najmniejszy element ze wszystkich pozostałych na stosach, więc znajduje się na szczycie stosu. ■

Na koniec wstępnych rozważań zauważmy jeszcze, że wszystkie indeksy $\ell(i)$ można obliczyć w czasie liniowym. W poniższym pseudokodzie w polu $arrived[j]$ zapisujemy, czy w dotychczas przejrzanym fragmencie ciągu a wystąpiła liczba j, i jeśli tak, na której była pozycji. W zmiennej $next$ jest natomiast najmniejsza wartość, która jeszcze nie wystąpiła przy przeglądaniu permutacji.

```
1:  Algorytm obliczania wartości ℓ(i):
2:    arrived[1..n] := (0, 0, ..., 0);
3:    next := 1;
4:    for i := 1 to n do begin
5:      arrived[a[i]] := i;
6:      while (next ⩽ n) and (arrived[next] ≠ 0) do begin
7:        ℓ[arrived[next]] := i;
8:        next := next + 1;
9:      end
10:   end
```

Redukcja do kolorowania grafu

Wiemy już, kiedy zdejmować elementy ze stosów. Spróbujmy teraz wywnioskować coś o tym, na który stos należy je kłaść. Zacznijmy od prostej obserwacji. Element a_i wkładamy w kroku i-tym, a zdejmujemy w $\ell(i)$-tym, więc, aby zachować malejący porządek stosów, wszystkie a_j, takie że $i < j \leqslant \ell(i)$ oraz $a_i < a_j$, muszą znaleźć się na innym stosie niż a_i.

Otrzymaliśmy zatem zbiór warunków postaci „a_i oraz a_j muszą być odłożone na różne stosy". Przedstawmy je w postaci grafu nieskierowanego $G = (V, E)$, którego wierzchołkami są indeksy $1, \ldots, n$. Krawędzie ilustrują nasze warunki, tzn. dla dowolnych dwóch indeksów $i < j$ istnieje krawędź (i, j) wtedy i tylko wtedy, gdy $a_i < a_j$ oraz $\ell(i) \geqslant j$. Przypomnijmy, że naszym celem jest teraz przyporządkowanie

elementom permutacji stosów. Zamiast tego będziemy mówić o przypisaniu każdemu wierzchołkowi grafu jednego z dwóch kolorów. Graf skonstruowaliśmy w ten sposób, że wierzchołki połączone krawędzią trzeba pokolorować na różne kolory.

W teorii grafów mówimy, że funkcja $k : V \to \{1, 2\}$ jest *poprawnym dwukolorowaniem* grafu $G = (V, E)$, jeżeli dla każdej krawędzi $(i, j) \in E$ zachodzi $k(i) \neq k(j)$. Graf G jest *dwukolorowalny*, gdy istnieje jego poprawne dwukolorowanie.

Stąd, aby permutację a dało się posortować przy pomocy dwóch stosów, odpowiadający jej graf G musi być dwukolorowalny. Naturalne wydaje się teraz pytanie, czy ów warunek jest wystarczający. Okazuje się, że odpowiedź na to pytanie jest twierdząca.

Twierdzenie 1. *Istnieje algorytm sortowania permutacji $a_1, \ldots, a_n$ za pomocą dwóch stosów wtedy i tylko wtedy, gdy odpowiadający jej graf G jest dwukolorowalny. Jeżeli ponadto G jest dwukolorowalny, to istnieje algorytm sortowania permutacji $a_1, \ldots, a_n$, który każdy element a_i odkłada na stos określony przez kolor wierzchołka i.*

Dowód: Konieczność warunku dwukolorowalności grafu G uzasadniliśmy powyżej. Przejdźmy więc od razu do dowodu jego dostateczności. Załóżmy, że graf G można pokolorować dwoma kolorami 1 i 2. Niech $k(i)$ będzie kolorem wierzchołka i. Rozważmy algorytm, który każdy element a_i odkłada na stos o numerze $k(i)$ oraz usuwa ze stosu w kroku $\ell(i)$, przy czym elementy przeznaczone do usunięcia w jednym kroku są usuwane w kolejności rosnącej. Pokażemy (analogicznie jak w dowodzie Obserwacji 2), że jest to poprawny algorytm sortowania permutacji $a_1, \ldots, a_n$ za pomocą dwóch stosów. Ponieważ dla $a_i < a_j$ mamy $\ell(i) \leqslant \ell(j)$, więc elementy są usuwane ze stosów we właściwej kolejności. Musimy jeszcze uzasadnić, że gdy algorytm chce usunąć element ze stosu, operacja ta jest dozwolona. W tym celu wystarczy udowodnić, że algorytm utrzymuje malejący porządek elementów na stosach.

Popatrzmy na dowolne elementy a_i, a_j, takie że $i < j$, $a_i < a_j$ i $k(i) = k(j)$. Wówczas $\ell(i) < j$. W przeciwnym przypadku w grafie G mielibyśmy krawędź (i, j), więc wierzchołki i, j nie mogłyby otrzymać tego samego koloru. Zatem w chwili odłożenia na stos elementu a_j, elementu a_i już na tym stosie nie ma (został zdjęty wcześniej). ■

Twierdzenie to naturalnie uogólnia się na dowolną liczbę stosów: problem sortowania permutacji za pomocą k stosów jest równoważny problemowi k-kolorowania grafu G.

Można by pomyśleć, że zadanie jest już rozwiązane: budujemy graf G i kolorujemy go dwoma kolorami na przykład za pomocą przeszukiwania w głąb[1]. Ale czy na pewno? Niestety, istnieją permutacje (również takie z odpowiedzią pozytywną), dla których graf G ma kwadratową liczbę krawędzi. Oto przykład:

$$m, m-1, \ldots, 2, n, n-1, \ldots, m+1, 1, \quad \text{gdzie} \quad m \approx \tfrac{n}{2}.$$

Zatem złożoność obliczeniowa takiego algorytmu w pesymistycznym przypadku wynosi $\Theta(n^2)$ i jest niewystarczająca do potrzeb naszego zadania.

[1] Mowa o algorytmie DFS, opisanym np. w książce [21].

Redukcja rozmiaru grafu

Okazuje się, że zamiast pełnego grafu G wystarczy skonstruować jego *las rozpinający* — podgraf acykliczny grafu G, który ma dokładnie te same spójne składowe co G. Taki podgraf ma rozmiar liniowy względem liczby wierzchołków. Oczywiście, każde poprawne dwukolorowanie grafu G indukuje poprawne dwukolorowanie lasu rozpinającego. Prawdziwa jest również odwrotna własność.

Obserwacja 3. *Niech L będzie lasem rozpinającym graf $G = (V, E)$ i niech $k : V \to \{1, 2\}$ będzie poprawnym dwukolorowaniem lasu L. Wówczas albo k jest poprawnym dwukolorowaniem grafu G, albo graf G nie jest dwukolorowalny.*

Dowód: Załóżmy, że dla pewnej krawędzi $(i, j) \in E$ zachodzi $k(i) = k(j)$. Wierzchołki i, j są w tej samej spójnej składowej lasu L, więc są w L połączone ścieżką. Wierzchołki na tej ścieżce mają kolory na przemian równe $k(i)$ i różne od $k(i)$, więc skoro $k(i) = k(j)$, ścieżka ta ma parzystą liczbę krawędzi. Zatem wraz z krawędzią (i, j) tworzy w grafie G cykl nieparzystej długości, który nie jest dwukolorowalny. ■

Tak dochodzimy do następującego schematu rozwiązania:

1. Dla podanej na wejściu permutacji $a_1, \ldots, a_n$ skonstruuj las L rozpinający graf G.
2. Znajdź poprawne dwukolorowanie lasu L.
3. Sprawdź, czy znalezione dwukolorowanie jest poprawnym dwukolorowaniem grafu G.

Realizacja punktu 2 jest bardzo łatwa — wystarczy przeszukać las L algorytmem przeszukiwania w głąb, kolorując napotykane wierzchołki na przemian liczbami 1 i 2. W punkcie 3 oczywiście nie możemy sprawdzać osobno każdej krawędzi grafu G, gdyż prowadziłoby to ponownie do rozwiązania kwadratowego. Jednak z Twierdzenia 1 wiemy, że w poprawnym dwukolorowaniu kolor każdego wierzchołka i jest numerem stosu, na który należy odłożyć element a_i w algorytmie sortowania permutacji. Sprawdzamy zatem, czy otrzymany z dwukolorowania lasu L algorytm sortowania permutacji jest poprawny.

```
1:  S[1].push(∞); S[2].push(∞);
2:  next := 1; { następny element do zdjęcia }
3:  for i := 1 to n do begin
4:    if a[i] > S[k[i]].top then return NIE;
5:    S[k[i]].push(a[i]);
6:    { zdejmowanie ze stosów kolejnych elementów permutacji posortowanej: }
7:    while (S[1].top = next) or (S[2].top = next) do begin
8:      if S[1].top = next then S[1].pop else S[2].pop;
9:      next := next + 1;
10:   end
11: end
12: return TAK, k[1..n];
```

Pozostaje jeszcze do zrealizowania punkt 1 przedstawionego powyżej schematu rozwiązania.

Warto przy okazji zaznaczyć, że ta sama idea (dwukolorowanie lasu rozpinającego graf pomocniczy) wystąpiła w rozwiązaniu zadania *Autostrady* z II etapu X Olimpiady Informatycznej [10].

Konstrukcja lasu rozpinającego – rozwiązanie wzorcowe

Zauważmy, że zwykłe przeszukiwanie w głąb grafu G konstruuje las rozpinający graf G — jest to tak zwany las przeszukiwania w głąb. Krawędź (i, j) w tym lesie oznacza, że procedura przeszukiwania w głąb, przeglądając sąsiedztwo wierzchołka i, znalazła nieodwiedzony wierzchołek j (albo na odwrót) i wywołała się dla niego rekurencyjnie. Spróbujmy więc zasymulować przeszukiwanie w głąb grafu G. Zauważmy przy tym, że poza znajdowaniem nieodwiedzonego sąsiada algorytm DFS działa w czasie $O(n)$. Potrzebujemy zatem odpowiednich struktur przyspieszających to wąskie gardło rozwiązania. Poniższy pseudokod pokazuje, jak za ich pomocą znaleźć las rozpinający i od razu pokolorować go na dwa kolory.

```
1:  procedure modified_dfs(i, color)
2:  begin
3:    k[i] := color;
4:    delete(i);
5:    while true do begin
6:      j := get_edge(i);
7:      if j = nil then break;
8:      modified_dfs(j, 3 − color);
9:    end
10: end
```

Funkcja *get_edge*(i) zwraca dowolnego nieodwiedzonego sąsiada wierzchołka i (albo **nil**, jeżeli wszystkie sąsiednie wierzchołki są już odwiedzone), natomiast procedura *delete*(i) usuwa wierzchołek i ze struktury przechowującej nieodwiedzone wierzchołki.

Implementacja operacji *get_edge* i *delete* musi oczywiście wykorzystywać szczególny sposób powstawania grafu G z wejściowej permutacji. Dla wygody wprowadźmy relację porządku $\prec$ na wierzchołkach grafu G: niech $i \prec j$ oznacza, że $a_i < a_j$. Dla każdego wierzchołka i grafu G incydentne do niego krawędzie dzielimy na dwie kategorie:

- *krawędzie w przód*, czyli krawędzie $(i, j) \in E$, takie że $i < j \leqslant \ell(i)$ oraz $i \prec j$,
- *krawędzie w tył*, czyli krawędzie $(i, j) \in E$, takie że $j < i \leqslant \ell(j)$ oraz $j \prec i$.

Do przechowywania obu typów krawędzi prowadzących do nieodwiedzonych sąsiadów użyjemy dwóch osobnych struktur danych.

Krawędzie w przód Funkcja *get_edge*(i) ograniczona do krawędzi w przód powinna zwrócić nieodwiedzony wierzchołek j, taki że $i < j \leqslant \ell(i)$ oraz $j \succ i$. W tym celu wystarczy znaleźć wśród nieodwiedzonych wierzchołków w przedziale $[i+1, \ell(i)]$ maksymalny w porządku $\prec$ wierzchołek j, po czym sprawdzić, czy $a_j > a_i$.

Strukturę, która pozwala efektywnie znajdować maksimum na przedziale, można zbudować na schemacie *drzewa przedziałowego*[2]. To powszechne jego zastosowanie bywa nazywane *drzewem turniejowym*. Każdy jego liść i posiada flagę *visited*$[i]$, która mówi, czy wierzchołek i grafu G jest odwiedzony. Ponadto w każdym węźle w jest zapisany indeks $max[w]$ maksymalnego (w porządku $\prec$) nieodwiedzonego liścia w poddrzewie o korzeniu w. Istota drzewa turniejowego polega na łatwości obliczenia $max[w]$ — dla liścia jest to w, gdy *visited*$[w]$ = **false**, oraz **nil** w przeciwnym razie, zaś dla węzła wewnętrznego jest to po prostu większy (w porządku $\prec$) z dwóch indeksów *max* zapisanych w jego dzieciach (zakładamy przy tym, że **nil** jest mniejszy od jakiegokolwiek innego indeksu).

Korzystając z opisanej struktury, operację wyszukiwania maksymalnego nieodwiedzonego wierzchołka $j \in [i+1, \ell(i)]$ można zrealizować w czasie $O(\log n)$. Wystarczy bowiem sprawdzić indeksy $max[w]$ zapisane w korzeniach maksymalnych (w sensie zawierania) poddrzew, których przedział $[x_w, y_w]$ zawiera się w $[i+1, \ell(i)]$, a takich poddrzew jest $O(\log n)$. Zmiana wartości flagi *visited*$[i]$ (wykonywana w procedurze *delete*(i)) również zajmuje czas $O(\log n)$, gdyż wymaga ona aktualizacji indeksów $max[w]$ tylko w węzłach w leżących na ścieżce od korzenia do liścia i (a każda taka ścieżka ma długość $O(\log n)$).

Krawędzie w tył Wyszukiwanie krawędzi w tył prowadzących do nieodwiedzonych sąsiadów jest nieco bardziej skomplikowane. Funkcja *get_edge*(i) ograniczona do krawędzi w tył musi dla danego i znaleźć taki nieodwiedzony wierzchołek j, że $j < i \leqslant \ell(j)$ oraz $j \prec i$. Będziemy więc szukać minimalnego w porządku $\prec$ nieodwiedzonego wierzchołka j, takiego że $j < i \leqslant \ell(j)$.

Tu wykorzystamy strukturę zwaną *drzewem z nasłuchiwaczami*, również opartą na drzewie przedziałowym. W każdym jego węźle znajduje się lista *nasłuchiwaczy*. Każdy nasłuchiwacz na liście węzła w reprezentuje nieodwiedzony wierzchołek j, taki że $[x_w, y_w] \subseteq [j+1, \ell(j)]$. Każdy nieodwiedzony wierzchołek j ma swoje nasłuchiwacze tylko w tych węzłach w, które są korzeniami maksymalnych (w sensie zawierania) poddrzew zawierających się w $[j+1, \ell(j)]$. Zatem każdy nieodwiedzony wierzchołek j generuje $O(\log n)$ nasłuchiwaczy oraz na każdej ścieżce od korzenia do liścia w przedziale $[j+1, \ell(j)]$ znajduje się nasłuchiwacz reprezentujący j.

Aby znaleźć krawędź w tył z wierzchołka i do nieodwiedzonego sąsiada, szukamy na ścieżce od korzenia do liścia i nasłuchiwacza minimalnego w porządku $\prec$ wierzchołka j. Jako że listy nasłuchiwaczy mogą być długie, utrzymujemy dodatkowy niezmiennik: nasłuchiwacze na każdej liście są posortowane zgodnie z porządkiem $\prec$. Wówczas znalezienie minimalnego nasłuchiwacza wymaga przejrzenia tylko pierwszych elementów list w węzłach na ścieżce od korzenia do i, co zajmuje łącznie czas

[2] Jest to pełne, całkowicie zrównoważone drzewo binarne, którego liśćmi są kolejne elementy przedziału $[1, n]$, zaś każdy węzeł wewnętrzny w reprezentuje przedział $[x_w, y_w]$ wszystkich elementów, które są liśćmi poddrzewa o korzeniu w. Bardziej szczegółowy opis drzew przedziałowych można znaleźć np. w opracowaniu zadania *Latarnia* w tej książeczce oraz w podanych tam odnośnikach.

$O(\log n)$. Aby w procedurze budowania drzewa skonstruować posortowane listy nasłuchiwaczy, przeglądamy wierzchołki $j \in [1, n]$ w kolejności rosnącej względem $\prec$ i dla każdego z nich dodajemy wszystkie jego nasłuchiwacze na koniec odpowiednich list. Tym samym konstrukcja drzewa z nasłuchiwaczami zajmuje czas $\Theta(n \log n)$.

Procedura *delete*(i) wymaga usunięcia wszystkich nasłuchiwaczy wierzchołka i. Aby umożliwić efektywne wykonanie tej operacji, podczas konstrukcji drzewa łączymy wszystkie nasłuchiwacze wierzchołka i w dodatkową listę. Wówczas usunięcie każdego nasłuchiwacza zajmuje czas stały, więc łączny czas uaktualnienia drzewa przez procedurę *delete*(i) wynosi $O(\log n)$ [3].

Pokazaliśmy, że obie operacje *get_edge* i *delete* możemy wykonać w czasie $O(\log n)$, używając struktur danych, które mają całkowity rozmiar $\Theta(n \log n)$ i których inicjalizacja zajmuje czas $\Theta(n \log n)$. Ponieważ przeszukiwanie całego grafu procedurą *modified_dfs* wykonuje liniową liczbę operacji *get_edge* i *delete*, więc złożoność czasowa i pamięciowa rozwiązania wzorcowego wynosi $\Theta(n \log n)$. Rozwiązanie to zostało zaimplementowane w pliku `kol.cpp`.

Konstrukcja lasu rozpinającego – algorytm on-line

Oznaczmy przez G_i podgraf grafu G, który powstaje przez obcięcie G do podzbioru wierzchołków $\{1, \ldots, i\}$. Rozwiązanie, które teraz pokażemy, działa *on-line* — konstruuje kolejno lasy rozpinające grafy $G_1, \ldots, G_n$, przy czym do obliczenia lasu rozpinającego graf G_i wykorzystuje tylko informację o elementach $a_1, \ldots, a_i$ wejściowej permutacji.

W i-tym kroku algorytmu dokładamy wierzchołek i oraz uaktualniamy las rozpinający G_{i-1} do lasu rozpinającego G_i. W tym celu musimy wierzchołek i połączyć krawędzią z pewnym wierzchołkiem każdej spójnej składowej grafu G_{i-1}, która w grafie G_i ma krawędź do i. Oznaczmy zbiór $U_i = \{j : \ell(j) > i\}$. Łatwo sprawdzić, że wszystkie wierzchołki j, które są w grafie G połączone z którymkolwiek z wierzchołków $i, \ldots, n$, mają $\ell(j) \geqslant i$, więc należą do U_{i-1}. Zatem bezpośrednio przed i-tym krokiem algorytmu możemy na zawsze zapomnieć o wierzchołkach nienależących do U_{i-1} — już nie mogą być wykorzystane przy łączeniu składowych.

Mimo że do obliczenia indeksów $\ell(i)$ musimy znać całą permutację $a_1, \ldots, a_n$, zbiory U_i możemy wyliczać on-line. Zdefiniujmy $next(i) = \min\{a_{i+1}, \ldots, a_n\}$: jest to najmniejszy element, który nie występuje wśród $a_1, \ldots, a_i$. Z definicji $\ell(j) > i$ wtedy i tylko wtedy, gdy $a_j \geqslant next(i)$. Zatem $U_i = \{j : a_j \geqslant next(i)\}$.

Przypomnijmy nasz pomocniczy porządek na indeksach (wierzchołkach): $i \prec j$ oznacza, że $a_i < a_j$. Wierzchołek i jest w grafie G_i połączony z tymi wierzchołkami $j < i$, dla których $\ell(j) \geqslant i$ oraz $j \prec i$. Wybierzmy z każdej spójnej składowej grafu G_{i-1} ograniczonej do zbioru U_{i-1} minimalny w porządku $\prec$ wierzchołek j. Jeżeli $j \prec i$, to do lasu rozpinającego dodajemy krawędź (i, j). Jeżeli natomiast $j \succ i$, to rozpatrywana składowa grafu G_{i-1} nie jest połączona z i w grafie G_i. Wszystkie

[3] Można także zrealizować tę operację w zamortyzowanym czasie $O(\log n)$, ale za to bez konieczności użycia dodatkowych list. W tym celu, w operacji *delete* zaznaczamy jedynie, że wierzchołek j został usunięty, natomiast podczas wykonywania operacji *get_edge* usuwamy z początków przeglądanych list wszystkie nieaktualne nasłuchiwacze.

składowe grafu G_{i-1}, które połączyliśmy krawędzią z wierzchołkiem i, tworzą wraz z nim jedną spójną składową w grafie G_i.

Okazuje się, że spójne składowe grafu G_i ograniczone do zbioru U_i nie „przecinają się" w porządku $\prec$ — jeżeli C_1, C_2 są dwiema różnymi składowymi grafu G_i ograniczonymi do zbioru U_i, to albo wszystkie wierzchołki z C_1 są mniejsze od wszystkich z C_2 ($C_1 \prec C_2$), albo wszystkie wierzchołki z C_2 są mniejsze od wszystkich z C_1 ($C_2 \prec C_1$). Nie będziemy osobno dowodzić tej własności — algorytm, który pokażemy, po prostu utrzymuje taki niezmiennik.

Algorytm konstrukcji lasu rozpinającego on-line wygląda następująco. Na początku i-tego kroku na stosie S znajdują się spójne składowe grafu G_{i-1} ograniczone do zbioru U_{i-1}, uporządkowane malejąco względem $\prec$ (tzn. najmniejsza składowa jest na szczycie stosu). Algorytm zdejmuje ze stosu wszystkie ograniczone składowe, których najmniejszy w porządku $\prec$ wierzchołek jest mniejszy od i, łącząc te wierzchołki z i krawędziami lasu rozpinającego. Zbiory zdjęte ze stosu łączy wraz z wierzchołkiem i w jedną składową grafu G_i ograniczoną do zbioru U_{i-1}, którą następnie odkłada na stos. Wszystkie inne zbiory, które pozostały na stosie, są w całości większe od i, więc niezmiennik jest zachowany. Na koniec ze zbiorów na stosie usuwane są wszystkie wierzchołki nienależące do U_i. Zbiór U_i ma postać $\{j : j \succ M\}$, przy czym M to najmniejszy w porządku $\prec$ wierzchołek spośród $\{i+1, \ldots, n\}$, więc wystarczy usuwać najmniejszy (według $\prec$) element ze zbioru na szczycie stosu.

```
1:  next := 1;
2:  for i := 1 to n do begin
3:    current := {i};
4:    while (not S.empty) and (a[S.top.min] < a[i]) do begin
5:      add_edge(i, S.top.min);
6:      current := current ∪ S.top; S.pop;
7:    end
8:    S.push(current);
9:    while (not S.empty) and (a[S.top.min] = next) do begin
10:     S.top.extract_min;
11:     if S.top = ∅ then S.pop;
12:     next := next + 1;
13:   end
14: end
```

W algorytmie tym kluczowa jest efektywna implementacja następujących operacji na zbiorach przechowywanych na stosie S i w zmiennej *current*:

- Utwórz zbiór złożony z pojedynczego wierzchołka (operacja w wierszu 3).
- Połącz dwa zbiory w jeden (operacja w wierszu 6).
- Pobierz najmniejszy (według $\prec$) wierzchołek zbioru (operacja *min*).
- Usuń najmniejszy wierzchołek ze zbioru (operacja *extract_min*).

Są to operacje tak zwanej *kolejki złączalnej*. Przykładem struktury, która pozwala uzyskać ograniczenie $O(\log n)$ na czas wykonania każdej z nich, jest *drzewo lewicowe*.

Pojedyncze drzewo lewicowe reprezentuje jeden zbiór. Jest to pełne drzewo binarne, którego każdy węzeł wewnętrzny przechowuje jeden element zbioru oraz którego liście nie przechowują żadnej informacji (są sztucznie dodane dla czytelności opisu). Każde jego poddrzewo spełnia następujące własności:

(1) *Własność kopca*: element w korzeniu poddrzewa jest najmniejszy (w porządku $\prec$) ze wszystkich elementów w poddrzewie.

(2) Skrajnie prawa ścieżka poddrzewa (czyli ścieżka wychodząca z korzenia poddrzewa i przechodząca w każdym węźle do prawego dziecka) jest najkrótsza ze wszystkich ścieżek od korzenia do liścia (w tym poddrzewie).

Utworzenie jednoelementowego drzewa nie przedstawia żadnej trudności. Operacja *min* zwraca po prostu korzeń drzewa. Operację *extract_min* możemy sprowadzić do łączenia dwóch drzew: usuwamy korzeń i łączymy poddrzewa w jego dzieciach w jedno drzewo. Pozostaje do zrealizowania operacja połączenia dwóch drzew.

W każdym węźle w zapisujemy $length[w]$ — długość skrajnie prawej ścieżki poprowadzonej z w. Z własności (2) natychmiast wynika, że jest ona logarytmiczna względem rozmiaru poddrzewa o korzeniu w. Jeżeli w jest liściem, to $length[w] = 0$. Połączenie dwóch drzew lewicowych w jedno realizuje następująca funkcja rekurencyjna:

```
1: function union(u, w)
2: begin
3:    if w ≺ u then zamień u ↔ w;
4:    if right[u] jest liściem then right[u] := w else right[u] := union(right[u], w);
5:    if length[left[u]] < length[right[u]] then zamień left[u] ↔ right[u];
6:    length[u] := length[right[u]] + 1;
7:    return u;
8: end
```

Funkcja ta przyjmuje jako argumenty korzenie dwóch drzew lewicowych i zwraca korzeń nowego drzewa powstałego z ich połączenia. Liczba wywołań rekurencyjnych funkcji *union* dla pojedynczej operacji łączenia drzew jest ograniczona przez $length[u] + length[w] = O(\log n)$.

Uzyskany algorytm konstrukcji lasu rozpinającego działa w czasie $O(n \log n)$ i pamięci liniowej. Implementacja kolejki złączalnej za pomocą kopców dwumianowych (patrz [21]) dałaby taki sam czas działania. Opisane rozwiązanie znajduje się w pliku `kol2.cpp`.

Ulepszony algorytm on-line

Poprzednie rozwiązanie konstruuje las rozpinający graf G bez względu na to, czy graf ten jest dwukolorowalny. Jednak w przypadku, gdy graf G nie jest dwukolorowalny, jego las rozpinający nie jest nam w sumie potrzebny — możemy od razu odpowiedzieć `NIE`.

Załóżmy zatem, że graf G_{i-1} jest dwukolorowalny. Oczywiście, dwukolorowanie każdej spójnej składowej grafu G_{i-1} jest wyznaczone jednoznacznie (z dokładnością

do zamiany kolorów). Każdy zbiór występujący w algorytmie (na stosie S lub w zmiennej *current*) reprezentujemy za pomocą dwóch list: każda z nich zawiera wierzchołki jednego koloru uporządkowane zgodnie z relacją $\prec$ (z minimalnym na początku listy). Operacja utworzenia nowego zbioru $\{i\}$ wstawia i do jednej z list, a drugą listę pozostawia pustą. Operacja *min* pobiera mniejszy wierzchołek z początków obu list, zaś operacja *extract_min* usuwa ten wierzchołek. Natomiast operacja łączenia dwóch zbiorów w wierszu 6 dokleja listy reprezentujące zbiór $S.top$ na koniec list reprezentujących zbiór *current* w taki sposób, aby zachowany był rosnący porządek elementów w obu wynikowych listach. Pokażemy, że takie połączenie list jest możliwe, o ile graf G_i jest dwukolorowalny.

Niech j_1 będzie minimalnym wierzchołkiem w zbiorze $S.top$. Połączenie zbiorów *current* i $S.top$ następuje wtedy, gdy $j_1 \prec i$. Wszystkie wierzchołki zbioru *current* różne od i pochodzą z połączenia składowych zdjętych ze stosu S — są mniejsze od wszystkich wierzchołków w zbiorze $S.top$ (w szczególności od j_1), a więc także od i. Zatem i jest ostatnim elementem swojej listy[4]. Jeżeli w zbiorze $S.top$ druga lista (tzn. ta, która nie zaczyna się od j_1) jest pusta lub jej początkowy (minimalny) element j_2 jest większy od i, nie ma problemu: doklejamy tę listę do listy zawierającej i, a listę zaczynającą się elementem j_1 doklejamy do drugiej listy zbioru *current*. Jeżeli natomiast $j_1, j_2 \prec i$, to graf G_i nie jest dwukolorowalny — mamy w nim krawędzie (i, j_1) i (i, j_2), a więc w poprawnym dwukolorowaniu wierzchołki j_1, j_2 powinny otrzymać ten sam kolor, co nie jest możliwe, gdyż w dwukolorowaniu składowej $S.top$ dostały różne kolory. W takim przypadku przerywamy algorytm i odpowiadamy `NIE`.

W ten sposób wszystkie operacje na zbiorach wykonywane przez nasz algorytm udało nam się zaimplementować w czasie stałym. Otrzymaliśmy rozwiązanie zadania działające w czasie *liniowym*! Co więcej, jeżeli opisany algorytm kończy swoje działanie bez odpowiedzi `NIE`, to graf $G_n = G$ na pewno jest dwukolorowalny — nie musimy w ogóle sprawdzać, czy dwukolorowanie otrzymanego lasu rozpinającego daje poprawny algorytm sortowania permutacji.

Jeszcze jeden algorytm dwukolorowania

Spójrzmy na przedstawiony powyżej ulepszony algorytm on-line z innej strony. Obsługuje on kolejno przychodzące elementy permutacji $a_1, \ldots, a_n$, starając się przydzielać je do jednego z dwóch stosów. Element a_1 oczywiście można odłożyć na dowolny stos. Jeżeli w momencie przyjścia elementu a_i elementy na szczytach obu stosów są mniejsze, odpowiedzią jest `NIE`. Jeżeli element na szczycie jednego stosu jest mniejszy od a_i, a element na szczycie drugiego stosu jest większy od a_i, to trzeba odłożyć a_i na drugi stos. Natomiast jeżeli elementy na szczytach obu stosów są większe od a_i, wówczas nie wiadomo (bez dodatkowej informacji), na który stos należy odłożyć a_i. Algorytm on-line odkłada tę decyzję na później, gdyż potrzebna informacja może nadejść dopiero wraz z dalszymi elementami permutacji. Przedstawimy teraz algorytm, który na podstawie bardzo prostej informacji dodatkowej potrafi od razu stwierdzić, na który stos powinien odłożyć element a_i.

[4] Jednocześnie jest pierwszym (a więc jedynym) elementem swojej listy, gdyż znajduje się w niej od początku istnienia listy, a kolejne elementy mogą być dokładane tylko na koniec.

Dla każdego i niech $m(i)$ będzie indeksem maksymalnego elementu permutacji między i a $\ell(i)$ włącznie: $a_{m(i)} = \max\{a_i, a_{i+1}, \ldots, a_{\ell(i)}\}$. Algorytm wygląda następująco:

```
 1: S[1].push(∞); S[2].push(∞);
 2: next := 1;
 3: for i := 1 to n do begin
 4:   if (a[i] > S[1].top) and (a[i] > S[2].top) then return NIE;
 5:   least := min(S[1].top, S[2].top);
 6:   if S[1].top < S[2].top then greater_stack := 2 else greater_stack := 1;
 7:   j := i;
 8:   same := true; { czy i, j muszą znaleźć się na tym samym stosie? }
 9:   while (a[j] < least) and (m[j] ≠ j) do begin
10:     j := m[j];
11:     same := not same;
12:   end
13:   { j znajdzie się na stosie greater_stack }
14:   if (same) then k[i] := greater_stack else k[i] := 3 − greater_stack;
15:   S[k[i]].push(a[i]);
16:   { zdejmowanie ze stosów kolejnych elementów permutacji posortowanej: }
17:   while (S[1].top = next) or (S[2].top = next) do begin
18:     if S[1].top = next then S[1].pop else S[2].pop;
19:     next := next + 1;
20:   end
21: end
22: return TAK, k[1..n];
```

Przyjrzyjmy się pojedynczej iteracji pętli **for**. Zmienna *same* mówi nam, czy elementy o indeksach i, j muszą znaleźć się na tym samym stosie (wartość **true**), czy na różnych stosach (wartość **false**). Na początku $i = j$, więc *same* = **true**. Zauważmy, że jeżeli $m(j) \neq j$, to $j < m(j) \leqslant \ell(j)$ oraz $a_j < a_{m(j)}$, więc w grafie G istnieje krawędź $(j, m(j))$, a zatem elementy o indeksach $j, m[j]$ muszą znaleźć się na różnych stosach. Dlatego w pojedynczej iteracji pętli **while** zmienna *same* zmienia wartość na przeciwną. Są dwie możliwości zakończenia pętli **while** w wierszach 9-12:

- $a_j >$ *least*. Niech *least* $= a_r$. Pokażemy, że $r < j \leqslant \ell(r)$. Mamy $r < i \leqslant j$, gdyż zawsze $x \leqslant m(x)$. Jeśli $i = j$, oczywista jest również prawa nierówność — element a_r nie został jeszcze zdjęty z S. Gdy natomiast $j \neq i$, to $j = m(j')$ dla pewnego j' spełniającego warunki w wierszu 8. Wobec tego $a_{\ell(j')} \leqslant a_{j'} < a_r$, a więc $j = m(j') \leqslant \ell(j') \leqslant \ell(r)$. Ostatecznie otrzymujemy, że (j, r) jest krawędzią grafu G, czyli elementy o indeksach j, r muszą znaleźć się na różnych stosach. To oraz wartość zmiennej *same* jednoznacznie wyznacza numer stosu, na który należy odłożyć i-ty element (wiersz 14).

- $a_j <$ *least* i $m(j) = j$. Wówczas $a_j = \max\{a_i, \ldots, a_{\ell(j)}\}$, ponieważ dla dowolnego t zachodzi $a_{m^t(i)} = \max\{a_i, \ldots, a_{\ell(m^{t-1}(i))}\}$ (łatwa indukcja). Stąd dla każdego $h \in \{i, \ldots, \ell(j)\}$ zachodzi $\ell(h) \leqslant \ell(j)$. Zatem w grafie G nie ma krawędzi łączącej zbiór $\{i, \ldots, \ell(j)\}$ z $\{\ell(j) + 1, \ldots, n\}$. Co więcej, wszystkie $a_i, \ldots, a_{\ell(j)}$

są mniejsze od wszystkich elementów na stosach oraz, oczywiście, większe od wszystkich elementów już zdjętych ze stosów, wobec czego żadna krawędź nie opuszcza zbioru $\{i, \ldots, \ell(j)\}$. Możemy zatem odłożyć i-ty element na dowolny stos.

Powyższy algorytm, mając obliczone indeksy $m(i)$, działa w czasie liniowym. Jest tak, ponieważ dla każdego j przypisanie $j := m[j]$ w wierszu 10 jest wykonywane co najwyżej raz. Istotnie, jeżeli w pewnym kroku i pętli **for** wykonuje się to przypisanie (dla wybranego j), to w kolejnych krokach $i+1, \ldots, j$ co najmniej jeden z elementów $a_i, a_{m(i)}, \ldots, a_{m^{s-1}(i)}$ (gdzie $m^s(i) = j$), wszystkich mniejszych od a_j, znajduje się na stosie, więc pętla **while** zatrzyma się na warunku w wierszu 9, zanim dojdzie do podstawienia $j := m[j]$.

Wszystkie indeksy $m(i)$ można wyliczyć z definicji w czasie $O(n \log n)$ za pomocą takich samych drzew turniejowych, jakich używaliśmy w rozwiązywaniu wzorcowym do wyszukiwania krawędzi w przód. Znajdowanie maksimów na przedziałach off-line można też łatwo sprowadzić do problemu szukania najniższych wspólnych przodków w drzewie, który można rozwiązać w czasie $O(n \log^\star n)$ algorytmem Tarjana [21], a nawet w czasie liniowym. Opis tego sprowadzenia oraz rozwiązania liniowego można znaleźć w dwuczęściowym artykule poświęconym problemom RMQ i LCA, w numerach 9/2007 i 11/2007 czasopisma *Delta*[5].

Rozwiązania niepoprawne

Próba zachłannego rozwiązywania zadania — przydzielania elementów a_i do stosów na podstawie samego tylko początkowego fragmentu permutacji $a_1, \ldots, a_i$ — jest z góry skazana na niepowodzenie. Wystarczy rozważyć dwie następujące permutacje:

$$5, 2, 6, 1, 4, 3 \quad \text{oraz} \quad 5, 2, 4, 1, 6, 3.$$

Każdy poprawny algorytm sortowania odkłada elementy 5 i 2 na ten sam stos w przypadku pierwszej permutacji oraz na różne stosy w przypadku drugiej.

Błędne są również rozwiązania, które decyzję w i-tym kroku podejmują na podstawie samych tylko elementów $a_1, \ldots, a_{\ell(i)}$. Świadczą o tym na przykład takie dwie permutacje:

$$7, 2, 4, 1, 6, 3, 8, 5 \quad \text{oraz} \quad 7, 2, 4, 1, 8, 3, 6, 5.$$

W obu $\ell(2) = 4$ oraz początkowe fragmenty a_1, a_2, a_3, a_4 są identyczne. Mimo to każdy poprawny algorytm sortowania odkłada elementy 7 i 2 na ten sam stos w pierwszym przypadku oraz na różne stosy w drugim przypadku.

Testy

Programy nadesłane przez zawodników były sprawdzane na 10 grupach testów. Każda z tych grup składa się z trzech testów: losowego testu typu a z odpowiedzią pozytywną, losowego testu typu b z odpowiedzią negatywną oraz testu typu c z odpowiedzią

[5] Artykuły dostępne także na stronie internetowej czasopisma: `http://www.mimuw.edu.pl/delta/`

pozytywną, który odrzuca niepoprawne rozwiązania drugiego typu opisanego powyżej (wyłączając test *1c*, reprezentujący przypadek brzegowy). Oto zestawienie wielkości danych wejściowych w poszczególnych testach:

Nazwa	n
kol1a.in	19
kol1b.in	15
kol1c.in	1
kol2a.in	60
kol2b.in	60
kol2c.in	60
kol3a.in	286
kol3b.in	268
kol3c.in	290

Nazwa	n
kol4a.in	813
kol4b.in	770
kol4c.in	813
kol5a.in	12 196
kol5b.in	12 155
kol5c.in	12 000
kol6a.in	27 193
kol6b.in	27 122
kol6c.in	25 000
kol7a.in	49 178
kol7b.in	49 126

Nazwa	n
kol7c.in	50 000
kol8a.in	69 281
kol8b.in	69 187
kol8c.in	70 000
kol9a.in	95 398
kol9b.in	95 402
kol9c.in	90 000
kol10a.in	99 838
kol10b.in	96 051
kol10c.in	99 000

Brutalne (wykładnicze) rozwiązania mogły zaliczyć tylko testy z grup 1 i 2, przy czym rozwiązanie sprawdzające wszystkie 2^n możliwości zaliczało tylko testy z grupy 1. Testy z grup 3 i 4 były dostępne dla rozwiązań działających w czasie $O(n^2)$. Testy z grupy 5 przechodziło rozwiązanie konstruujące cały graf G w czasie $O((n+m)\log n)$, gdzie m jest liczbą krawędzi w grafie G. Natomiast testy w grupach 5-10 zostały tak dobrane, aby graf G miał rozmiar kwadratowy, i wobec tego nie zaliczały ich rozwiązania oparte na generowaniu całego grafu G. Przewidziane przez organizatorów rozwiązania niepoprawne nie przechodziły żadnej grupy testów.

Tomasz Waleń
Treść zadania, Opracowanie

Wojciech Tyczyński
Program

Dostępna pamięć: 64 MB.

OI, Etap I, 19.10–16.11.2009

Korale

Bajtazar postanowił zająć się produkcją naszyjników. Udało mu się okazyjnie kupić bardzo długi sznur różnokolorowych korali. Dysponuje też maszyną, która dla zadanego k ($k > 0$) potrafi pociąć sznur na odcinki składające się z k korali (czyli pierwszy odcinek składa się z korali o numerach $1, \ldots, k$, drugi $k+1, \ldots, 2k$, itd.). Jeśli sznur korali ma długość, która nie jest wielokrotnością k, to ostatni odcinek, który ma długość mniejszą niż k, jako niepełnowartościowy, nie jest wykorzystywany. Kolory korali oznaczamy dalej dodatnimi liczbami całkowitymi.

Bajtazar lubi różnorodność i zastanawia się, jak dobrać liczbę k, tak by otrzymać jak najwięcej różnych sznurów korali. Sznur korali, który ma zostać pocięty, ma wyraźnie określony koniec, od którego zaczynamy odcinać krótsze sznury. Jednak każdy odcięty sznur może zostać obrócony — inaczej mówiąc, sznury $(1, 2, 3)$ i $(3, 2, 1)$ są dla Bajtazara takie same. Napisz program, który pomoże mu wyznaczyć optymalną wartość parametru k.

Przykładowo, dla sznura korali postaci:

$$(1, 1, 1, 2, 2, 2, 3, 3, 3, 1, 2, 3, 3, 1, 2, 2, 1, 3, 3, 2, 1),$$

- *stosując $k = 1$, możemy otrzymać 3 różne sznury korali: (1), (2), (3),*
- *stosując $k = 2$, możemy otrzymać 6 różnych sznurów korali: $(1, 1)$, $(1, 2)$, $(2, 2)$, $(3, 3)$, $(3, 1)$, $(2, 3)$,*
- *stosując $k = 3$, możemy otrzymać 5 różnych sznurów korali: $(1, 1, 1)$, $(2, 2, 2)$, $(3, 3, 3)$, $(1, 2, 3)$, $(3, 1, 2)$,*
- *stosując $k = 4$, możemy otrzymać 5 różnych sznurów korali: $(1, 1, 1, 2)$, $(2, 2, 3, 3)$, $(3, 1, 2, 3)$, $(3, 1, 2, 2)$, $(1, 3, 3, 2)$,*
- *dla większych wartości k możemy uzyskać co najwyżej 3 różne sznury korali.*

Wejście

W pierwszym wierszu standardowego wejścia znajduje się liczba całkowita n ($1 \leqslant n \leqslant 200\,000$), oznaczająca długość sznura korali. W drugim wierszu znajduje się n dodatnich liczb całkowitych a_i ($1 \leqslant a_i \leqslant n$), pooddzielanych pojedynczymi odstępami i oznaczających kolory kolejnych korali w sznurze Bajtazara.

Wyjście

W pierwszym wierszu standardowego wyjścia należy wypisać dwie liczby całkowite oddzielone pojedynczym odstępem: liczbę różnych sznurów korali, które można uzyskać przy optymalnym wyborze parametru k, oraz liczbę l różnych wyborów parametru k prowadzących do uzyskania takiej liczby sznurów. Drugi wiersz powinien zawierać l liczb pooddzielanych pojedynczymi odstępami: wartości parametru k, dla których uzyskujemy optymalne rozwiązanie — mogą one zostać wypisane w dowolnej kolejności.

Przykład

Dla danych wejściowych:

```
21
1 1 1 2 2 2 3 3 3 1 2 3 3 1 2 2 1 3 3 2 1
```

poprawnym wynikiem jest:

```
6 1
2
```

Rozwiązanie

Pierwsze rozwiązanie

Pierwszym pomysłem, jaki nasuwa się w związku z zadaniem, jest bezpośrednie wykonanie poleceń podanych w treści zadania.

Musimy sprawdzić wszystkie możliwe długości podsłów k ($k = 1, 2, \ldots, n$). Przez *podsłowa* rozumiemy tu spójne fragmenty wyjściowego ciągu korali, który to ciąg będziemy w związku z tym nazywać *słowem*. Dla danej długości k przeglądamy kolejne podsłowa tej długości, wybrane zgodnie z warunkami zadania. W trakcie przeglądania musimy sprawdzać, czy aktualnie analizowane podsłowo występowało (w identycznej formie, lub też odwrócone) wcześniej w podziale.

Rozwiązanie to, mimo iż wygląda na niezbyt optymalne, w rzeczywistości nie jest najgorsze. Sprawdzenie, ile jest różnych podsłów zadanej długości k wybranych zgodnie z treścią zadania, ma bowiem łączny koszt czasowy rzędu:

$$T_k = \sum_{i=1}^{\lfloor n/k \rfloor} (i-1) \cdot k = k \cdot \sum_{i=1}^{\lfloor n/k \rfloor} (i-1) = k \cdot O\left(\frac{n^2}{k^2}\right) = O\left(\frac{n^2}{k}\right).$$

Powyższe oszacowanie bierze się stąd, że dla i-tego z kolei podsłowa musimy sprawdzić, czy wystąpiło wśród dotychczas rozpatrzonych $i-1$ podsłów, a dla każdej pary podsłów sprawdzenie, czy są one równe zgodnie z naszymi kryteriami, zajmuje czas $O(k)$. Zatem złożoność czasowa całego algorytmu wynosi:

$$T = \sum_{k=1}^{n} T_k = O\left(\sum_{k=1}^{n} \frac{n^2}{k}\right) = O(n^2) \cdot O\left(\sum_{k=1}^{n} \frac{1}{k}\right) = O(n^2) \cdot O(\log n) = O(n^2 \log n).$$

W tym oszacowaniu wykorzystujemy znaną własność tzw. liczb harmonicznych, patrz np. książka [23]:

$$\sum_{k=1}^{n} \frac{1}{k} = O(\log n). \tag{1}$$

Implementację przedstawionego rozwiązania można znaleźć w pliku `kors1.cpp`. Na zawodach tego typu programy zdobywały około 40 punktów.

Rozwiązanie wzorcowe

Zastanówmy się teraz, w jaki sposób można ulepszyć poprzednie rozwiązanie. Najbardziej pracochłonną częścią tego algorytmu było porównywanie podsłów długości k. Aby usprawnić ten fragment rozwiązania, możemy zastosować algorytm Karpa-Millera-Rosenberga (KMR), który buduje tzw. *słownik podsłów bazowych*[1]. W ten sposób dla zadanego słowa s, w czasie $O(n \log n)$ (lub $O(n \log^2 n)$, w zależności od tego, czy w algorytmie używamy sortowania pozycyjnego, czy jakiegoś o złożoności czasowej liniowo-logarytmicznej) przygotowujemy strukturę danych, która pozwala na porównywanie dowolnych dwóch podsłów słowa s w czasie $O(1)$.

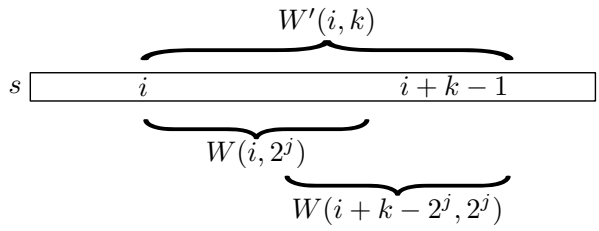
Struktura ta, dla wejściowego słowa s ($s = s[1..n]$) oraz dowolnych liczb całkowitych $0 \leqslant j \leqslant \log n$ oraz $1 \leqslant i \leqslant n + 1 - 2^j$, generuje liczbowy identyfikator $W(i, 2^j)$ dla podsłowa $s[i..i + 2^j - 1]$. Identyfikatory te są niewielkimi liczbami całkowitymi (z zakresu $[0..n - 1]$) oraz mają tę własność, że

$$W(i, 2^j) = W(p, 2^j) \quad \Leftrightarrow \quad s[i..i + 2^j - 1] = s[p..p + 2^j - 1].$$

Dzięki takiej informacji możemy również jednoznacznie identyfikować podsłowa *dowolnej* długości k. Faktycznie, niech j oznacza największą liczbę całkowitą, taką że $2^j \leqslant k$. Wtedy podsłowo $s[i..i + k - 1]$ możemy jednoznacznie zidentyfikować z parą

$$W'(i, k) \stackrel{\text{def}}{=} (W(i, 2^j), W(i + k - 2^j, 2^j)),$$

zobacz rys. 1. Oczywiście, nadal ma miejsce własność, że dwa podsłowa długości k są identyczne wtedy i tylko wtedy, gdy odpowiadające im identyfikatory są równe.



Rys. 1: Funkcja $W'(i, k)$.

Zadanie wymaga od nas utożsamiania słów postaci w oraz w^R (odwrócone słowo w). Można pokonać to utrudnienie, stosując algorytm KMR dla słowa $s\#s^R$ (# to dowolny symbol różny od wszystkich symboli wejściowych). Musimy również zmienić definicję funkcji $W'(i, k)$. Każde podsłowo $w = s[i..i + k - 1]$ występuje w słowie $s\#s^R$ w dwóch miejscach:

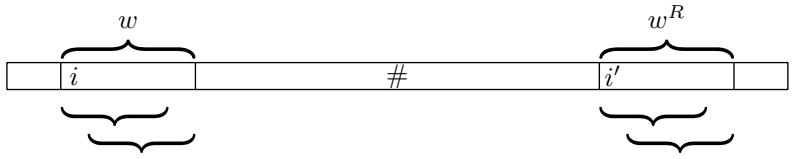
- na pozycji i — jako w,
- oraz na pozycji $i' = 2n + 2 - (i + k - 1)$ — jako w^R.

Jako identyfikator $W'(i, k)$, dla $1 \leqslant i$ oraz $i + k - 1 \leqslant n$, powinniśmy wybrać np. mniejszą (leksykograficznie) z par:

$$(W(i, 2^j), W(i + k - 2^j, 2^j)) \quad \text{oraz} \quad (W(i', 2^j), W(i' + k - 2^j, 2^j)),$$

[1] Patrz np. książka [19] lub opis rozwiązania zadania *Powtórzenia* z VII OI [7].

gdzie $2^j \leqslant k < 2^{j+1}$, patrz także rys. 2.



Rys. 2: Konstrukcja funkcji $W'(i,k)$ dla słowa $s\#s^R$.

Kolejnym krokiem, który musimy zastosować, jest efektywniejsze badanie liczby różnych podsłów w zadanym podziale. Dla ustalonego k, za pomocą poprzedniego kroku utożsamiamy podsłowa z parami liczb całkowitych, czyli otrzymujemy ciąg złożony z $\lfloor \frac{n}{k} \rfloor$ par. Następnie taki ciąg możemy uporządkować leksykograficznie (tzn. posortować pary w pierwszej kolejności względem pierwszej współrzędnej, a w razie remisów względem drugiej). W uporządkowanym ciągu już łatwo wyznaczamy liczbę różnych elementów. Takie postępowanie możemy zaimplementować łatwo w czasie liniowo-logarytmicznym względem liczby elementów ciągu, równej $\lfloor \frac{n}{k} \rfloor$ dla danej długości podsłowa k. Można się także bardziej postarać i zamiast sortowania liniowo-logarytmicznego użyć algorytmu sortowania pozycyjnego. Wówczas należy *wszystkie* ciągi par, skonstruowane dla poszczególnych wartości parametru k, posortować za jednym razem, dodając do każdej z par trzecią współrzędną oznaczającą wartość parametru k, której ona odpowiada. Dzięki temu koszt czasowy sortowania spada do:

$$O\left(\sum_{k=1}^{n} \frac{n}{k}\right),$$

czyli możemy powiedzieć, że na daną wartość parametru k przypada $O(\frac{n}{k})$ operacji.

Podsumowując, zadanie możemy rozwiązać, używając następującego postępowania:

1: uruchom algorytm KMR dla słowa $s\#s^R$
2: **for** $k := 1$ **to** n **do begin**
3: wygeneruj ciąg par identyfikujących kolejne podsłowa o długości k:
4: $C_k = (W'(i,k) \ : \ i = 1,\ k+1,\ 2k+1, \ldots)$
5: **end**
6: uporządkuj ciągi C_k przy pomocy sortowania pozycyjnego
7: **for** $k := 1$ **to** n **do**
8: wyznacz liczbę różnych elementów w ciągu C_k

Zastanówmy się, jaka jest złożoność takiego rozwiązania. Czas potrzebny na wstępne obliczenia w algorytmie KMR wynosi $O(n \log n)$. Następnie, wyznaczenie liczby różnych podsłów długości k wybranych zgodnie z wymaganiami zadania zajmuje czas $O(\frac{n}{k})$. Korzystając ponownie z oszacowania (1), otrzymujemy koszt czasowy całego algorytmu:

$$T = O(n \log n) + \sum_{k=1}^{n} O\left(\frac{n}{k}\right) = O(n \log n) + O(n) \cdot O\left(\sum_{k=1}^{n} \frac{1}{k}\right) = O(n \log n).$$

Implementacje rozwiązania wzorcowego używające sortowania liniowo-logarytmicznego, czyli działające w złożoności czasowej $O(n \log^2 n)$, można znaleźć w plikach `kor.cpp` i `kor1.pas`.

W tym miejscu warto dodać, że równie efektywne rozwiązanie można otrzymać, korzystając z metody haszowania, która — podobnie jak algorytm KMR — pozwala na wyznaczanie identyfikatorów zadanej długości podsłów słowa s, jednakże jest obarczona mało prawdopodobną możliwością błędnego stwierdzenia, że dwa różne podsłowa są równe. Więcej o tej metodzie można przeczytać w opisie rozwiązania zadania *Antysymetria* w niniejszej książeczce oraz w podanych tam odnośnikach.

Testy

Rozwiązania były sprawdzane na 12 grupach danych testowych. Poniżej przedstawiona została tabela z podstawowymi parametrami testów, w której n oznacza długość sznura korali, a m — liczbę różnych kolorów korali.

Nazwa	n	m
kor1a.in	20	13
kor1b.in	20	2
kor1c.in	20	1
kor2a.in	100	17
kor2b.in	100	8
kor2c.in	100	1
kor3a.in	200	60
kor3b.in	200	4
kor4a.in	500	252
kor4b.in	500	9
kor5a.in	16 000	726
kor5b.in	16 000	16
kor6a.in	20 000	775
kor6b.in	20 000	48

Nazwa	n	m
kor7a.in	40 000	2 035
kor7b.in	40 000	96
kor8a.in	50 000	12 303
kor8b.in	50 000	120
kor9a.in	80 000	34 267
kor9b.in	80 000	18
kor10a.in	140 000	88 456
kor10b.in	140 000	3
kor11a.in	170 000	101 623
kor11b.in	170 000	13
kor12a.in	200 000	126 290
kor12b.in	200 000	2
kor12c.in	200 000	1

Jakub Radoszewski — Treść zadania, Opracowanie
Jakub Radoszewski, Bartłomiej Wołowiec — Program

Dostępna pamięć: 64 MB. *OI, Etap I, 19.10–16.11.2009*

Najdzielniejszy dzielnik

Dana jest liczba całkowita $N > 1$. Powiemy, że liczba całkowita $d > 1$ jest dzielnikiem N z krotnością $k > 0$ (k całkowite), jeżeli $d^k \mid N$ oraz $d^{k+1} \nmid N$. Dla przykładu, liczba $N = 48 = 16 \cdot 3$ ma następujące dzielniki: 2 z krotnością 4, 3 z krotnością 1, 4 z krotnością 2, 6 z krotnością 1 itd.

Powiemy, że liczba d jest **najdzielniejszym dzielnikiem** *liczby N, jeżeli d jest dzielnikiem N z krotnością k i N nie ma dzielników z krotnościami większymi niż k. Przykładowo, najdzielniejszym dzielnikiem liczby 48 jest 2 (z krotnością 4), a najdzielniejszymi dzielnikami liczby 6 są: 2, 3 i 6 (każdy z krotnością 1).*

Twoim zadaniem jest wyznaczenie krotności najdzielniejszego dzielnika liczby N oraz wyznaczenie liczby wszystkich najdzielniejszych dzielników N.

Wejście

Na standardowym wejściu znajduje się trochę nietypowy opis liczby N. Pierwszy wiersz zawiera jedną liczbę całkowitą n ($1 \leqslant n \leqslant 600$). Drugi wiersz zawiera n liczb całkowitych a_i ($2 \leqslant a_i \leqslant 10^{18}$) pooddzielanych pojedynczymi odstępami. Opis ten oznacza, że $N = a_1 \cdot a_2 \cdot \ldots \cdot a_n$.

Wyjście

Pierwszy wiersz standardowego wyjścia powinien zawierać największą liczbę całkowitą dodatnią k, dla której istnieje dzielnik d liczby N, taki że $d^k \mid N$. Drugi wiersz powinien zawierać jedną liczbę całkowitą dodatnią D będącą liczbą (najdzielniejszych) dzielników N o krotności k.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`3`	`4`
`4 3 4`	`1`
natomiast dla danych:	*poprawnym wynikiem jest:*
`1`	`1`
`6`	`3`

Ocenianie

Jeżeli Twój program wypisze poprawną krotność k najdzielniejszego dzielnika liczby N, natomiast nie wypisze w drugim wierszu liczby najdzielniejszych dzielników D lub wypisana przez niego liczba tych dzielników będzie niepoprawna, to uzyska 50% punktów za dany test (oczywiście, odpowiednio przeskalowane w przypadku przekroczenia połowy limitu czasowego).

Rozwiązanie

Wprowadzenie

Oznaczmy przez $kr(d, N)$ zdefiniowaną w treści zadania krotność dzielnika d w ramach liczby N. Naszym zadaniem jest, wśród wszystkich dzielników danej (w pośredni sposób) liczby N, wskazanie takich dzielników, których krotność w N jest największa, i obliczenie pewnych statystyk dotyczących tych dzielników. Ponieważ liczba N może być bardzo duża (po wymnożeniu wszystkich liczb a_i możemy otrzymać nawet liczbę rzędu $10^{10\,800}$), a także może mieć bardzo wiele dzielników, więc widać wyraźnie, że musimy wymyślić rozwiązanie istotnie sprytniejsze niż bezpośrednie przeglądanie wszystkich dzielników N.

Intuicja może nam podpowiadać, że coś wspólnego z rozwiązaniem mają *liczby pierwsze*, gdyż są one „najmniejszymi" możliwymi dzielnikami, a zatem ich krotności w ramach N powinny być duże. Jednakże drugi przykład z treści zadania wyraźnie pokazuje, że wśród najdzielniejszych dzielników liczby mogą pojawiać się także liczby złożone. Warto więc nieco dokładniej przyjrzeć się temu zagadnieniu.

Fakt 1. *Jeżeli liczba d jest najdzielniejszym dzielnikiem N, to również każdy jej dzielnik pierwszy p jest najdzielniejszym dzielnikiem N.*

Dowód: Faktycznie, jeśli $d^a \mid N$ oraz $d = p \cdot r$, to także $p^a \mid N$, a więc $kr(p, N) \geqslant kr(d, N)$. ■

Fakt 2. *Jeżeli d jest najdzielniejszym dzielnikiem N, to d nie jest podzielne przez kwadrat żadnej liczby całkowitej większej niż 1, równoważnie: w rozkładzie d na czynniki pierwsze każda liczba pierwsza występuje co najwyżej raz.*

Dowód: Zauważmy na wstępie, iż $kr(d, N) > 0$. Gdyby zatem jakaś liczba pierwsza p wystąpiła w rozkładzie d na czynniki pierwsze co najmniej dwukrotnie, to mielibyśmy:

$$kr(p, N) \;\geqslant\; 2 \cdot kr(d, N) \;>\; kr(d, N),$$

czyli d nie byłby najdzielniejszym dzielnikiem N. ■

Fakt 3. *Jeśli parami różne liczby pierwsze $p_1, p_2, \ldots, p_a$ są najdzielniejszymi dzielnikami N, to także ich iloczyn $p_1 p_2 \ldots p_a$ jest najdzielniejszym dzielnikiem N.*

Dowód: Oczywisty. ■

Na podstawie spostrzeżeń zawartych w Faktach 1-3 możemy już dużo lepiej wyobrazić sobie, jak wygląda zbiór $ndz(N)$ wszystkich najdzielniejszych dzielników N. Jeżeli $p_1, p_2, \ldots, p_c$ są wszystkimi liczbami pierwszymi zawartymi w $ndz(N)$, to

$$ndz(N) = \left\{ \prod_{\substack{j>0 \\ 1 \leqslant i_1 < i_2 < \ldots < i_j \leqslant c}} p_{i_1} p_{i_2} \ldots p_{i_j} \right\},$$

czyli jest to zbiór o liczności $2^c - 1$ złożony ze wszystkich (różnych od 1) iloczynów pewnych spośród liczb p_i, które to iloczyny zawierają każdą z liczb p_i co najwyżej raz.

Aby rozwiązać nasze zadanie, wystarczy zatem rozważyć wszystkie liczby pierwsze dzielące N, dla każdej z nich obliczyć jej krotność w ramach N, wziąć maksimum z tych krotności i wyznaczyć liczbę liczb pierwszych odpowiadających temu maksimum. I tak jak wszystkich dzielników liczba N może mieć wiele, tak dzielników *pierwszych* ma już zdecydowanie mniej — łatwo widać, że każda z liczb a_i ma co najwyżej $\log_2 a_i$ dzielników pierwszych (pytanie kontrolne: dlaczego?), a różnych dzielników pierwszych a_i jest w rzeczywistości jeszcze mniej.

Pozostaje pytanie, jak wyznaczyć wszystkie te dzielniki pierwsze. Jest to tzw. problem *faktoryzacji* liczby, który w ogólności jest, niestety, trudny — nie jest znany żaden algorytm rozwiązujący ten problem w złożoności czasowej wielomianowej względem *długości zapisu* rozkładanej liczby (czyli, innymi słowy, względem logarytmu z tej liczby). Stąd kiepskim pomysłem byłoby wymnożenie wszystkich liczb a_i i operowanie na otrzymanej, bardzo dużej liczbie N. Zamiast tego będziemy operować na reprezentacji liczby N z treści zadania, czyli właśnie na ciągu a.

Rozwiązanie wzorcowe

Faza I: pierwiastek sześcienny

Oznaczmy:

$$M \stackrel{\text{def}}{=} \max\{a_1, a_2, \ldots, a_n\}.$$

Jest to liczba nieprzekraczająca 10^{18}. Najpopularniejszy algorytm rozkładu dowolnej liczby z na czynniki pierwsze polega na rozważaniu kolejnych liczb pierwszych do pierwiastka z z i dzieleniu przez nie liczby z do skutku (trochę mniej efektywne rozwiązanie otrzymujemy, rozważając wszystkie liczby naturalne do $\sqrt{z}$, a nie tylko te pierwsze). Na końcu tego procesu pozostanie albo jedynka, albo jakaś liczba pierwsza większa od $\sqrt{z}$ (pytanie kontrolne do Czytelnika: dlaczego nie może pozostać liczba złożona?). Gdyby zastosować taką właśnie metodę do wszystkich liczb a_i, otrzymalibyśmy algorytm o złożoności czasowej $O(n\sqrt{M})$. To trochę za wolno jak na ograniczenia z zadania.

Skoro nie możemy pozwolić sobie na sprawdzenie tak dużej liczby liczb pierwszych, spróbujemy ją ograniczyć. Okazuje się, że niezłym pomysłem jest rozważenie jedynie liczb pierwszych nieprzekraczających

$$m \stackrel{\text{def}}{=} \sqrt[3]{M}.$$

Dla każdej z tych liczb pierwszych wykonamy operację *skrócenia*, polegającą na wyznaczeniu jej krotności w ramach każdej z liczb a_i, zsumowaniu tych krotności i porównaniu wyniku sumowania (będącego zarazem krotnością tej liczby pierwszej w ramach N) z najlepszym dotychczas otrzymanym — patrz poniższy pseudokod.

```
1: { Zmienne globalne (obie początkowo równe 0): }
2: {   k — maksymalna znaleziona krotność }
3: {   c — liczba wykrytych liczb pierwszych odpowiadających aktualnemu k }
```

```
 4:
 5: procedure skrócenie(p : pierwsza)
 6: begin
 7:   kr := 0;
 8:   for i := 1 to n do
 9:     while a_i mod p = 0 do begin
10:       a_i := a_i div p;
11:       kr := kr + 1;
12:     end
13:   if kr > k then begin
14:     k := kr; c := 1;
15:   end else if kr = k then c := c + 1;
16: end
```

Można by zapytać, dlaczego tak a nie inaczej dobraliśmy górną granicę zakresu rozważanych liczb pierwszych, tj. m. Odpowiedź na to pytanie uzyskamy, jeśli zastanowimy się, jak będą wyglądać liczby a_i po wszystkich wykonanych skróceniach. Otóż każda będzie iloczynem co najwyżej *dwóch* liczb pierwszych — faktycznie, gdyby po skróceniach było $a_i = p{\cdot}q{\cdot}r$ dla liczb naturalnych $p, q, r > 1$, to mielibyśmy $p, q, r > m$, czyli $a_i > m^3 = M$, co nie jest możliwe. Stąd w dalszej części rozwiązania musimy już tylko rozważyć te pozostałe liczby pierwsze. Zanim to uczynimy, oszacujmy jeszcze złożoność czasową Fazy I.

Zacznijmy od tego, że wszystkie liczby pierwsze nie większe niż m możemy wyznaczyć za pomocą sita Eratostenesa[1], którego złożoność czasowa to

$$O(m \log\log m). \tag{1}$$

Liczba rozważanych liczb pierwszych jest rzędu $O\left(\frac{m}{\log m}\right)$ [2]. Dla każdej z tych liczb wykonujemy skrócenie:

```
1: Faza I:
2:   P := sito(1, 2, ..., ⌊m⌋);
3:   foreach p ∈ P do skrócenie(p);
```

Koszt czasowy tych wszystkich skróceń zależy od liczby obrotów pętli **for** oraz **while**, z których ta druga odpowiada wykonywaniu dzieleń przez p: sekwencji udanych dzieleń zakończonej jednym nieudanym. Udanych dzieleń (wiersze 10-11 w pseudokodzie funkcji *skrócenie*) jest łącznie co najwyżej

$$n \log M = n \log (m^3) = 3n \log m = O(n \log m), \tag{2}$$

[1] Więcej o tym algorytmie można przeczytać np. w opisie rozwiązania zadania *Zapytania* z XIV Olimpiady Informatycznej [14].

[2] To oszacowanie można znaleźć w książce [23], a także w różnych książkach poświęconych teorii liczb.

gdyż tyle maksymalnie (niekoniecznie różnych) dzielników pierwszych ma liczba N. Z kolei dzieleń nieudanych jest co najwyżej

$$O(|P| \cdot n) = O\left(\frac{m}{\log m} \cdot n\right). \tag{3}$$

Łączny koszt czasowy Fazy I otrzymujemy jako sumę kosztów częściowych (1), (2) oraz (3):

$$O\left(m \log\log m + n \log m + \frac{m}{\log m} \cdot n\right) = O(m \log\log m + mn/\log m). \tag{4}$$

Faza II: wspólne dzielniki

Po wykonaniu pierwszej fazy każda z liczb a_i jest jedynką, liczbą pierwszą lub iloczynem dwóch liczb pierwszych. Niestety, wszystkie występujące liczby pierwsze są duże, więc musimy poszukiwać ich jakoś sprytniej niż dotychczas. W tej fazie pozbędziemy się *wspólnych* dzielników pierwszych występujących w różnych liczbach a_i. Skorzystamy z tego, że największy wspólny dzielnik dwóch liczb umiemy obliczać bardzo efektywnie, za pomocą algorytmu Euklidesa.

Załóżmy, że pewne dwa wyrazy ciągu a, tj. a_i oraz a_j, posiadają wspólny dzielnik pierwszy i nie są równe, czyli, bez straty ogólności, zachodzi $1 < a_i < a_j$. Widać, że mogą one mieć co najwyżej jeden taki wspólny dzielnik pierwszy p. Wówczas $a_i = p \cdot q$, $a_j = p \cdot r$ i każda z liczb q, r, $q \neq r$, jest albo pierwsza, albo równa 1. To oznacza, że $\mathrm{nwd}(a_i, a_j)$ jest równe p, czyli że jest szukanym wspólnym dzielnikiem pierwszym.

Ilustrację tego rozumowania stanowi poniższy pseudokod.

```
1: Faza II:
2:   for i := 1 to n − 1 do
3:     for j := i + 1 to n do begin
4:       d := nwd(a_i, a_j);
5:       if (d > 1) and (d < max(a_i, a_j)) then skrócenie(d);
6:     end
```

Na złożoność czasową powyższego algorytmu składa się obliczanie największych wspólnych dzielników oraz wykonywanie skróceń. Algorytm Euklidesa wywołujemy $O(n^2)$ razy, każde wywołanie ma złożoność czasową $O(\log M) = O(\log m)$, więc łączny koszt obliczania nwd to:

$$O(n^2 \log m). \tag{5}$$

Aby oszacować łączny koszt czasowy wszystkich skróceń, zauważmy, że łączna liczba liczb pierwszych występujących w ramach a_i po wykonaniu Fazy I nie przekracza $2n$. Stąd będzie co najwyżej $2n$ skróceń, z których każde wykonamy w czasie $O(n)$, co daje łączny koszt czasowy wszystkich skróceń rzędu

$$O(n^2). \tag{6}$$

To pokazuje, że złożoność czasowa Fazy II to $O(n^2 \log m)$.

Faza III: kwadraty

Liczby pierwsze mogą powtarzać się także w ramach jednej liczby a_i, tzn. wtedy, gdy jest ona kwadratem liczby pierwszej. W tym przypadku bardzo łatwo zidentyfikować taką liczbę pierwszą i wykonać odpowiednie skrócenie:

```
1: Faza III:
2:   for i := 1 to n do
3:     if a_i > 1 then begin
4:       d := ⌊√a_i⌋;
5:       if d^2 = a_i then skrócenie(d);
6:     end
```

Łączny koszt czasowy obliczania wartości d to $O(n)$ lub $O(n \log m)$, w zależności od tego, czy mamy do dyspozycji operację pierwiastkowania w czasie stałym, czy też musimy ją sami zaimplementować za pomocą wyszukiwania binarnego. W językach programowania dostępnych na zawodach dostępna jest stosowna funkcja `sqrt`. To pokazuje, że dominujący w tej fazie jest tak naprawdę koszt czasowy operacji skrócenia, który w poprzedniej fazie oszacowaliśmy na $O(n^2)$.

Faza IV, ostatnia

W poprzednich fazach pozbyliśmy się różnych rodzajów dzielników pierwszych liczby N. Zastanówmy się, jak mogą wyglądać otrzymane w rezultacie liczby a_i. Widzimy, że każda z nich jest albo jedynką, albo liczbą pierwszą, albo iloczynem dwóch *różnych* liczb pierwszych. Co więcej, jeśli ustalimy pewne $a_i > 1$, to wszystkie pozostałe elementy ciągu są albo równe a_i, albo względnie pierwsze z a_i (czyli nie posiadają wspólnych dzielników pierwszych z a_i). To oznacza, że każdą grupę równych wartości w ciągu a_i możemy rozważyć oddzielnie. I teraz, jeśli wartość a_i występuje w ciągu b razy, to dostarcza nam ona z krotnością b albo jedną, albo dwie liczby pierwsze, różne od dotychczas rozważonych. Żeby jednak sprawdzić, czy jedną, czy dwie, musimy rozłożyć liczbę a_i na czynniki pierwsze... Niestety, jest to smutna wiadomość, gdyż, jak wcześniej stwierdziliśmy, w ogólności nie jest to problem, który potrafimy łatwo rozwiązać.

Na szczęście możemy trochę „oszukać". Zauważmy, że do obliczenia wyniku (parametry k oraz c) wystarczy nam tylko informacja, *czy* dane a_i jest liczbą pierwszą, czy iloczynem dwóch liczb pierwszych, a w ogóle nie interesuje nas to, *jakie* są to liczby! To z kolei można sprowadzić do pytania, czy liczba a_i jest pierwsza czy złożona. Zamiast problemu faktoryzacji otrzymujemy zatem problem badania *pierwszości* liczby, a to już jest dużo lepsza sytuacja. Otóż istnieją wielomianowe (znów, względem długości zapisu liczby) algorytmy testujące pierwszość liczb. Najpopularniejsze z nich to algorytmy *randomizowane*[3]: algorytm Millera-Rabina i Solovaya-Strassena. Więcej o tych

[3]Od niedawna (2002 r.) znany jest także deterministyczny (czyli nierandomizowany) algorytm testujący pierwszość liczb, a mianowicie algorytm AKS (nazwa pochodzi od pierwszych liter nazwisk twórców: Agrawal, Kayal, Saxena). Algorytm ten korzysta jednak z zaawansowanych narzędzi i ma bardzo dużą, choć wielomianową, złożoność czasową, więc w praktyce nie jest obecnie wykorzystywany.

algorytmach można przeczytać w Internecie (patrz np. Wikipedia), a dokładny opis algorytmu Millera-Rabina znajduje się m.in. w książce [21]. W tym miejscu ograniczymy się do przedstawienia skróconego opisu wspólnej struktury tych algorytmów.

W algorytmie dysponujemy pewną losową procedurą, której przekazujemy rozważaną liczbę i która zwraca jeden z komunikatów: „na pewno złożona" albo „nie wiem, czy pierwsza czy złożona". Wiemy, że procedura ta nigdy nie zwraca nieprawdy, a jeśli liczba faktycznie jest złożona, to procedura zwraca wynik „na pewno złożona" z pewnym dodatnim prawdopodobieństwem p (w teście Millera-Rabina mamy $p \geqslant 3/4$). I teraz uruchamiamy wspomnianą procedurę wielokrotnie (s razy); jeśli kiedykolwiek orzeknie, że testowana liczba jest złożona, to tak też jest w rzeczywistości, a w przeciwnym przypadku zakładamy, że liczba jest pierwsza. Zauważmy, że prawdopodobieństwo tego, że po wykonaniu s prób zakończonych odpowiedzią „nie wiem" mamy wciąż do czynienia z liczbą złożoną, to $(1-p)^s$, zakładając, że wyniki zwracane przez rozważaną procedurę są niezależne (jako zmienne losowe). Dla algorytmu Millera-Rabina i $s = 50$ powtórzeń mamy konkretnie:

$$(1-p)^s \leqslant \frac{1}{4^{50}} \approx 0.00000000000000000000000000000079,$$

czyli to prawdopodobieństwo jest naprawdę znikome.

Wystarczy teraz użyć dowolnego z wspomnianych testów pierwszości (funkcja *pierwsza*) i mamy gotowe rozwiązanie Fazy IV:

```
1:  Faza IV:
2:    for i := 1 to n do
3:      if a_i > 1 then begin
4:        kr := 1;
5:        for j := i + 1 to n do
6:          if a_j = a_i then begin
7:            a_j := 1; kr := kr + 1;
8:          end
9:        if pierwsza(a_i) then ile := 1 else ile := 2;
10:       a_i := 1;   { dla porządku }
11:       if kr > k then begin
12:         k := kr; c := ile;
13:       end else if kr = k then c := c + ile;
14:     end
```

Koszt czasowy tej fazy to $O(n^2)$ (wiersze 5-8) plus n wywołań testu pierwszości. Jeżeli, dla przykładu, użyjemy testu Millera-Rabina, którego złożoność czasowa to $O(s \log^3 M)$, przy czym s to liczba powtórzeń losowej procedury testującej, to koszt czasowy Fazy IV wyniesie:

$$O(n^2 + ns \log^3 m). \tag{7}$$

Podsumowanie

Po wykonaniu wszystkich czterech faz znamy wartości dwóch parametrów: k — krotności każdej z liczb ze zbioru $ndz(N)$, oraz c — liczby liczb pierwszych w zbiorze

$ndz(N)$. Na mocy wcześniejszych rozważań, oznacza to, że powinniśmy wypisać kolejno liczby k oraz $2^c - 1$. Pojawia się tu dodatkowo jeden drobny kłopot: liczba $2^c - 1$ może być bardzo duża. Aby ją obliczyć, musimy zaimplementować własne operacje arytmetyczne na dużych liczbach. Wystarczą nam zaledwie mnożenie przez 2 i odejmowanie jedynki, a każdą z tych operacji implementujemy jak odpowiednie działanie pisemne. Ponieważ koszt czasowy każdej z tych operacji to $O(\log(2^c - 1)) = O(c)$, a musimy ich łącznie wykonać c, więc złożoność czasowa obliczania $2^c - 1$ wynosi:

$$O(c^2) = O((n \log M)^2) = O(n^2 \log^2 m). \tag{8}$$

Możemy już teraz wyznaczyć łączną złożoność czasową całego algorytmu. Otrzymujemy ją, sumując składniki (4)–(8):

$$O\left(m \log\log m + mn/\log m + ns \log^3 m + n^2 \log^2 m\right).$$

Wygląda to dosyć skomplikowanie, ale wystarczy zauważyć, że każdy ze składników, przy ograniczeniach z zadania, daje rozsądnego rzędu wielkości liczbę wykonywanych operacji.

Implementacje tego rozwiązania można znaleźć w plikach: `naj.cpp`, `naj2.cpp`, `naj3.pas`, `naj4.pas` i `naj5.cpp` (różne implementacje testu Millera-Rabina) oraz `naj1.cpp` (test Solovaya-Strassena).

Warto też na koniec przypomnieć sobie o dodatkowym warunku z treści zadania, że rozwiązanie wypisujące poprawnie wartość k uzyskuje za zadanie 50% punktów. Wbrew pozorom, pominięcie parametru D nie tylko pozwala uniknąć implementacji arytmetyki dużych liczb, lecz także i złożonych algorytmów testowania pierwszości! Faktycznie, wynik testu pierwszości (wiersz 9 w implementacji Fazy IV) jest używany wyłącznie do obliczania wartości parametru c. Implementację tak uproszczonego rozwiązania można znaleźć w pliku `najb4.cpp`.

Rozwiązanie alternatywne

Znajomość jeszcze jednego klasycznego algorytmu teorioliczbowego pozwala skonstruować inne, również całkiem efektywne rozwiązanie. Chodzi tutaj o heurystykę „ro" Pollarda[4], służącą do faktoryzacji liczb. Załóżmy, że dana jest liczba złożona z, której najmniejszy dzielnik pierwszy jest równy p. Wówczas heurystyka Pollarda znajdzie ten dzielnik (lub pewną jego wielokrotność, jednakże niebędącą wielokrotnością z) w oczekiwanej liczbie kroków rzędu $O(\sqrt{p})$, przy czym każdy krok jest wykonywany w czasie $O(\log^2 z)$.

Okazuje się, że można użyć tej heurystyki zamiast Faz II-IV rozwiązania wzorcowego. W tym celu wykonujemy następujące kroki:

- Na początku wykonujemy Fazę I, dokładnie tak samo jak w rozwiązaniu wzorcowym.
- Następnie dla każdej liczby $a_i > 1$ sprawdzamy, czy jest liczbą pierwszą, czy złożoną, używając do tego celu np. testu Millera-Rabina.

[4]Opis tego algorytmu można znaleźć w książce [21].

- Po wykonaniu poprzednich kroków pozostają nam do rozłożenia liczby a_i postaci $p \cdot q$, przy czym $p \leqslant q$ są liczbami pierwszymi. Wówczas zachodzi $p \leqslant \sqrt{M}$, a zatem heurystyka Pollarda pozwala znaleźć nietrywialną wielokrotność pewnego dzielnika pierwszego liczby a_i w oczekiwanym czasie $O(\sqrt{p}) = O(\sqrt[4]{M})$. Na tej podstawie za pomocą operacji nwd rozkładamy a_i na czynniki pierwsze, po czym dokonujemy odpowiednich skróceń w ciągu a.

Łączny oczekiwany koszt czasowy tego rozwiązania jest taki sam jak koszt rozwiązania wzorcowego:

$$O\left(m \log\log m + mn/\log m + ns\log^3 m + nM^{1/4}\log^2 m + n^2\log^2 m\right) = \\ = O\left(m \log\log m + mn/\log m + ns\log^3 m + n^2\log^2 m\right). \quad (9)$$

W praktyce jest ono co najwyżej kilkukrotnie wolniejsze od rozwiązania wzorcowego, ale jest to różnica na tyle nieznaczna, że na zawodach była mu przyznawana maksymalna punktacja. Implementacje tego rozwiązania można znaleźć w plikach `naj6.cpp` i `naj7.pas`.

Inne rozwiązania

W gruncie rzeczy, w tym zadaniu wszystkie istotne rozwiązania powolne tudzież błędne są gorszymi implementacjami rozwiązania wzorcowego: praktycznie każdą z jego faz można wykonać w gorszej złożoności czasowej, niepoprawnie albo w ogóle o niej zapomnieć. Zainteresowany Czytelnik znajdzie rozmaite przykłady takich rozwiązań w plikach `najs[1-5].[cpp|pas]` oraz `najb[1-11].cpp`.

Testy

Zadanie było sprawdzane na 14 zestawach danych testowych, których podstawowe parametry są wymienione w poniższej tabeli. Główną grupę stanowią testy zwane *ogólnymi*, generowane poprzez wyszczególnienie liczby liczb pierwszych w $ndz(N)$ znajdowanych w poszczególnych fazach rozwiązania wzorcowego oraz maski informującej o obecności (lub jej braku) liczb pierwszych z poszczególnych faz w danym ciągu a. Poza tym w zestawie występują: testy maksymalizujące wynikowe k (typ $\max \mathbf{k}$) i wynikowe D (typ $\max \mathbf{c}$), testy wydajnościowe dla algorytmów badania pierwszości skonstruowane tylko za pomocą liczb pierwszych z Fazy IV, wreszcie test zawierający pseudolosowe liczby a_i oraz małe testy generowane ręcznie.

Nazwa	n	k	c	Opis
naj1a.in	7	4	2	$N = 941\,412\,010\,017\,164\,400$
naj1b.in	2	3	2	$N = 719\,986\,312\,752\,603\,624$
naj2.in	200	17	5	test ogólny, tylko Faza I
naj3a.in	200	2	89	test ogólny, tylko Faza I

Nazwa	n	k	c	Opis
naj3b.in	300	17 700	1	test max **k**
naj4.in	170	16	9	test ogólny, wszystkie fazy
naj5.in	300	3	63	test ogólny, bez Fazy III
naj6.in	299	24	9	test ogólny, tylko Fazy I i III
naj7a.in	300	14	21	test ogólny, bez Fazy II
naj7b.in	289	284	1	test „zupełnie losowy”
naj8a.in	300	56	4	test ogólny, wszystkie fazy
naj8b.in	300	1	1 346	test max **c**
naj8c.in	100	4	20	test ogólny, wszystkie fazy
naj9a.in	300	4	72	test ogólny, wszystkie fazy
naj9b.in	300	1	1 450	test max **c**
naj9c.in	268	15	16	test ogólny, bez Fazy III
naj9d.in	298	1	305	test wydajnościowy dla badania pierwszości
naj10a.in	300	2	191	test ogólny, wszystkie fazy
naj10b.in	300	12	20	test ogólny, wszystkie fazy
naj10c.in	300	1	304	test wydajnościowy dla badania pierwszości
naj11.in	300	12	18	test ogólny, bez Fazy IV
naj12a.in	300	10	28	test ogólny, wszystkie fazy
naj12b.in	300	22	11	test ogólny, bez Fazy I
naj13.in	300	9	21	test ogólny, wszystkie fazy
naj14a.in	300	3	90	test ogólny, bez Fazy IV
naj14b.in	300	5	57	test ogólny, bez Fazy IV

Jakub Radoszewski
Treść zadania, Opracowanie

Bartosz Górski
Program

Dostępna pamięć: 64 MB.

OI, Etap I, 19.10–16.11.2009

Test na inteligencję

Jedno z zadań w bajtockim teście na inteligencję polega na wykreślaniu liczb z zadanego początkowego ciągu tak, aby otrzymywać w ten sposób różne inne zadane sekwencje. Bajtazar chciałby zostać bajtockim mistrzem IQ, ale wyjątkowo kiepsko radzi sobie z zadaniami tego typu. Zamierza dużo ćwiczyć i poprosił Cię o napisanie programu, który pomoże mu szybko sprawdzać odpowiedzi.

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą m ($1 \leqslant m \leqslant 1\,000\,000$). Drugi wiersz zawiera ciąg m liczb całkowitych $a_1, a_2, \ldots, a_m$ ($1 \leqslant a_i \leqslant 1\,000\,000$ dla $1 \leqslant i \leqslant m$) pooddzielanych pojedynczymi odstępami, tworzących początkowy ciąg w zadaniu z testu. Trzeci wiersz wejścia zawiera jedną dodatnią liczbę całkowitą n. Kolejne $2n$ wierszy zawiera opisy ciągów, które mają powstać w wyniku wykreślania różnych liczb z początkowego ciągu. Opis każdego z tych ciągów zajmuje po dwa kolejne wiersze. W pierwszym wierszu każdego opisu znajduje się liczba całkowita m_i ($1 \leqslant m_i \leqslant 1\,000\,000$). Drugi wiersz zawiera m_i-elementowy ciąg liczb całkowitych $b_{i,1}, b_{i,2}, \ldots, b_{i,m_i}$ ($1 \leqslant b_{i,j} \leqslant 1\,000\,000$ dla $1 \leqslant j \leqslant m_i$) pooddzielanych pojedynczymi odstępami. Możesz założyć, że suma długości podanych n sekwencji nie przekroczy 1 000 000.

Wyjście

Twój program powinien wypisać na standardowe wyjście n wierszy. Wiersz o numerze i (dla $1 \leqslant i \leqslant n$) powinien zawierać jedno słowo „`TAK`" lub „`NIE`" (bez cudzysłowów), w zależności od tego, czy i-ta sekwencja z wejścia może powstać w wyniku wykreślenia (tj. usunięcia) pewnych (niekoniecznie kolejnych) liczb z początkowego ciągu. Oczywiście, kolejność liczb w powstałym po wykreśleniach ciągu ma znaczenie (patrz przykład).

Przykład

Dla danych wejściowych:

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

```
1 5 7 4
```
poprawnym wynikiem jest:
```
TAK
NIE
TAK
NIE
```

Rozwiązanie

Analiza problemu

Zacznijmy od sformułowania właściwego problemu, który mamy do rozwiązania. Dany jest ciąg a o długości m oraz n ciągów b_i o długościach m_i, wszystkie indeksowane od jedynki. Ograniczenia z zadania sugerują wprowadzenie dodatkowych parametrów:

- $len = m_1 + m_2 + \ldots + m_n$, czyli suma długości wszystkich sekwencji b_i,
- s — maksimum wartości elementów wszystkich ciągów,
- $S = \{1, 2, \ldots, s\}$.

Treść zadania gwarantuje, że $m, n, len, s \leqslant 10^6$ oraz że elementy wszystkich ciągów należą do zbioru S.

Zdefiniowana w zadaniu operacja wykreślania elementów z ciągu a sprowadza się do tego, że chcemy dla każdego ciągu b_i sprawdzić, czy jest on *podciągiem* ciągu a. Przypomnijmy, że ciąg $(c_i)_{i=1}^{p}$ nazywamy podciągiem ciągu $(d_j)_{j=1}^{q}$, jeśli można wybrać takie indeksy $j_1, j_2, \ldots, j_p$, że:

$$1 \leqslant j_1 < j_2 < \ldots < j_p \leqslant q$$

oraz:

$$c_1 = d_{j_1}, \quad c_2 = d_{j_2}, \quad \ldots, \quad c_p = d_{j_p}.$$

Zanim zajmiemy się konstruowaniem wymyślnych algorytmów, warto zacząć od najprostszego możliwego rozwiązania, w którym dla każdego kolejnego ciągu b_i w bezpośredni sposób sprawdzamy, czy jest on podciągiem ciągu a. Każde takie sprawdzenie zaczynamy od znalezienia w ciągu a pierwszego wystąpienia elementu $b_{i,1}$, następnie w dalszej części ciągu a poszukujemy pierwszego wystąpienia elementu $b_{i,2}$ itd. Poniższy pseudokod zawiera implementację takiego podejścia.

```
1: function podciąg(a, m, b_i, m_i)
2: begin
3:   k := 1; { pozycja w ciągu b_i }
4:   for j := 1 to m do
5:     if (k ⩽ m_i) and (a_j = b_{i,k}) then
6:       k := k + 1;
7:   if k > m_i then return TAK else return NIE;
8: end
```

Łączna liczba operacji wykonywanych w funkcji *podciąg* jest rzędu $O(m)$. Wywołując tę funkcję dla każdego kolejnego ciągu b_i, otrzymujemy zatem rozwiązanie o złożoności czasowej $O(n \cdot m)$, co wobec ograniczeń z zadania, jeszcze nas nie satysfakcjonuje.

Powyższa funkcja jest zapisana niezbyt efektywnie; można ją próbować usprawniać, korzystając z rozmaitych spostrzeżeń:

- jeśli $m_i > m$, to można od razu zwrócić `NIE`;
- jeśli w pewnym momencie wykonywania pętli **for** zachodzi $k > m_i$, to możemy od razu przerwać wykonywanie pętli i zwrócić `TAK`;
- (bardziej pomysłowe:) możemy na wstępie sprawdzać, czy w ogóle multizbiór (czyli zbiór z powtórzeniami) elementów ciągu b_i jest podzbiorem multizbioru elementów ciągu a, a jeśli nie, to od razu zwracać odpowiedź `NIE` (implementację tego usprawnienia w dodatkowym, łącznym koszcie czasowym $O(m + s + len)$ pozostawiamy Czytelnikowi).

Niestety, żadne z powyższych ulepszeń nie poprawia złożoności czasowej naszego rozwiązania, o czym można przekonać się, biorąc pod uwagę np. ciągi $a = (1, 1, \ldots, 1, 2)$, $b_i = (2)$. Tego typu rozwiązania zdobywały na zawodach 20-30% punktów. Stosowne implementacje można znaleźć w plikach `tess4.cpp` oraz `tess5.pas`.

Pierwsze rozwiązanie wzorcowe

Aby efektywniej sprawdzać, czy dany ciąg b_i jest podciągiem ciągu a, posłużymy się pomocniczą strukturą danych, którą zbudujemy raz na samym początku rozwiązania. Naszym celem jest zredukowanie kosztu czasowego pojedynczego sprawdzenia z $O(m)$ do kosztu zależnego od m_i — wówczas suma kosztów wszystkich sprawdzeń będzie zależała już nie od $n \cdot m$, ale od parametru $len = m_1 + m_2 + \ldots + m_n$.

Zauważmy, że na funkcję *podciąg* możemy spojrzeć jak na m_i-krotne wykonanie operacji: „znajdź pierwszy element w ciągu a położony za a_j i równy $b_{i,k}$" — jeśli któraś z tych m_i operacji nie powiedzie się, możemy od razu zwrócić odpowiedź `NIE`. Skorzystajmy z tego, że rozmiar zbioru S jest nieduży, i dla każdego elementu $c \in S$ zapiszmy indeksy jego kolejnych wystąpień w ciągu a. Oznaczmy taki (posortowany) ciąg przez ℓ_c. Wówczas żądaną operację możemy wykonać, znajdując w $\ell_{b_{i,k}}$ pierwszy element większy niż j. Jeśli dodatkowo struktury ℓ_c będą zorganizowane jak tablice, tzn. będą dopuszczały swobodny dostęp do poszczególnych elementów (czyli dostęp w czasie stałym), to ów indeks można będzie wyznaczyć efektywnie za pomocą wyszukiwania binarnego.

Przykład 1. W przypadku ciągu $a = (2, 1, 4, 2, 1, 5, 4, 1, 2)$ i zbioru $S = \{1, 2, 3, 4, 5\}$ ciągi ℓ_c mają postać: $\ell_1 = (2, 5, 8)$, $\ell_2 = (1, 4, 9)$, $\ell_3 = ()$, $\ell_4 = (3, 7)$, $\ell_5 = (6)$.

Spróbujmy zapisać pseudokod powyższego rozwiązania, pomijając na razie to, jak konkretnie reprezentujemy strukturę danych ℓ_c oraz jak ją tworzymy — załóżmy tylko, że kolejne elementy ciągu ℓ_c to $\ell_c[1], \ell_c[2], \ldots, \ell_c[r]$, przy czym $r = size(\ell_c)$ to długość tego ciągu.

```
1:  { Wyszukiwanie binarne — funkcja zwraca pierwszy element ℓc[i] ciągu }
2:  { rosnącego ℓc większy niż j lub BRAK, jeśli takiego elementu nie ma. }
3:  function pierwszy_większy(ℓc, j)
4:  begin
5:    lewy := 1; prawy := size(ℓc);
6:    while lewy < prawy do begin
7:      śr := (lewy + prawy) div 2;
8:      if ℓc[śr] ⩽ j then lewy := śr + 1
9:      else prawy := śr;
10:   end
11:   if ℓc[lewy] > j then return ℓc[lewy]
12:   else return BRAK;
13: end
14:
15: function podciąg2(bi, mi)
16: begin
17:   j := 0;
18:   for k := 1 to mi do begin
19:     j := pierwszy_większy(ℓb_{i,k}, j);
20:     if j = BRAK then return NIE;
21:   end
22:   return TAK;
23: end
```

Koszt czasowy funkcji *pierwszy_większy* to $O\big(\log(size(\ell_c))\big) = O(\log m)$, oczywiście o ile parametr ℓ_c nie jest przekazywany bezpośrednio, lecz przez wskaźnik bądź referencję. Stąd złożoność czasowa funkcji *podciąg2* to $O(m_i \log m)$. Pozostaje pytanie, w jaki sposób reprezentować ciągi ℓ_c.

Programujący w języku C++ mogą do przechowywania struktury ℓ użyć tablicy vectorów; `vector` to kontener z biblioteki STL, który zachowuje się dokładnie jak tablica, tyle że dodatkowo umożliwia dodawanie nowych elementów na koniec[1]. Zakładając jednak, że nie mamy takich udogodnień do dyspozycji, możemy upakować całą strukturę ℓ do jednej, dużej tablicy $t[1..m]$: na początku umieścimy w niej ℓ_1, potem ℓ_2, itd. aż do ℓ_s. Konstrukcję tablicy t wykonujemy za pomocą poniższego pseudokodu.

```
1: Algorytm wypełniania tablicy t[1..m]:
2:   for i := 1 to s do count[i] := 0;
3:   for j := 1 to m do count[aj] := count[aj] + 1;
4:   pocz[1] := 1;
5:   for i := 2 to s do pocz[i] := pocz[i − 1] + count[i − 1];
6:   for i := 1 to s do kon[i] := pocz[i] − 1;
7:   for j := 1 to m do begin
8:     kon[aj] := kon[aj] + 1;
```

[1] W STL-u można także znaleźć funkcję działającą podobnie jak nasza *pierwszy_większy*, a mianowicie `upper_bound`.

```
 9:      t[kon[a_j]] := j;
10:  end
```

W powyższym algorytmie, w wierszach 2-3 zliczamy wystąpienia poszczególnych elementów w ciągu a i zapamiętujemy wyniki w tablicy $count[1..s]$. W dwóch kolejnych wierszach na podstawie tych wartości wypełniamy tablicę *pocz* indeksów początków tablic $\ell_1, \ell_2, \ldots, \ell_s$ w ramach t. Wreszcie w wierszach 6-10 wypełniamy tablicę t, przy okazji obliczając końce wystąpień poszczególnych tablic ℓ_i w ramach t (tablica *kon*) — po zakończeniu tej procedury mamy $\ell_i = t[pocz[i]..kon[i]]$. (Przy czym przedział postaci $[x..x-1]$ reprezentuje ciąg pusty). Warto jeszcze wspomnieć, że powyższy algorytm działa bardzo podobnie do sortowania kubełkowego (patrz np. książka [21]).

Nie da się ukryć, że koszt czasowy wypełnienia tablicy t to $O(m+s)$. Po tych wstępnych obliczeniach wyszukiwanie binarne w ciągu ℓ_c można wykonywać na fragmencie tablicy t od $pocz[c]$ do $kon[c]$. W ten sposób złożoność czasowa funkcji *podciąg2* pozostaje zgodna z wcześniejszymi przewidywaniami, tzn. $O(m_i \log m)$ w jednym wywołaniu, a łącznie $O(len \cdot \log m)$. Ostateczna złożoność czasowa tego rozwiązania to zatem $O(len \cdot \log m + m + s)$. Łatwo sprawdzić, że jego złożoność pamięciowa szacuje się przez $O(len + m + s)$.

Implementacje rozwiązań opartych na tym podejściu można znaleźć w plikach `tess1.c`, `tess2.cpp` oraz `tess3.pas`.

Drugie rozwiązanie wzorcowe

W tym rozwiązaniu również będziemy przyspieszać podany na początku algorytm bezpośredni, tym razem jednak w zupełnie inny sposób, a mianowicie rozpatrując wszystkie ciągi b_i naraz. Będzie to wyglądało mniej więcej tak, jakbyśmy równolegle uruchomili funkcję *podciąg* dla wszystkich ciągów b_i i śledzili tylko, dla każdego z tych ciągów, w którym jego miejscu znajduje się jego aktualnie „oczekiwany" (tj. poszukiwany w ciągu a) element, wskazywany w oryginalnej funkcji przez zmienną k. Skorzystamy z faktu, że każdy z tych oczekiwanych elementów znajduje się w zbiorze S, dzięki czemu ciągi b_i będziemy mogli w każdym momencie podzielić na grupy oczekujących na poszczególne elementy $c \in S$.

W rozwiązaniu wykorzystamy trochę nietypową strukturę danych, a mianowicie *worek*. Jak sama nazwa wskazuje, jej zadaniem jest przechowywanie elementów z pewnego zbioru. Worek udostępnia dwie operacje: można włożyć do niego podany element oraz wyciągnąć z niego jakiś (bliżej nieokreślony) element. Będziemy tu dodatkowo zakładać, iż nigdy nie będziemy chcieli wstawić do worka czegoś, co już w nim się znajduje. Dla każdego elementu $c \in S$ będziemy utrzymywać właśnie taki worek numerów ciągów b_i, które w danej chwili oczekują na pojawienie się w ciągu a elementu c. I teraz w miarę przeglądania kolejnych elementów ciągu a będziemy przerzucać elementy pomiędzy workami, symulując przesunięcie wszystkim ciągom z worka odpowiadającego danemu a_i wskaźnika aktualnie oczekiwanego elementu na następny.

Poniższy pseudokod realizuje opisane podejście. Tablica *worki* przechowuje aktualne zawartości worków związanych z poszczególnymi elementami $c \in S$, tablica

zadowolony — informacje o tym, które spośród ciągów b_i okazały się już podciągami ciągu a, wreszcie w tablicy k pamiętamy, dla każdego ciągu b_i, indeks aktualnie oczekiwanego elementu tego ciągu (zbieżność nazwy tej tablicy ze zmienną k w funkcji *podciąg* jest nieprzypadkowa).

```
1:  Rozwiązanie wzorcowe z użyciem worków:
2:    worki[1..s] := (∅, ∅, ..., ∅);
3:    zadowolony[1..n] := (false, false, ..., false);
4:    k[1..n] := (1, 1, ..., 1);
5:    for i := 1 to n do worki[b_{i,1}].insert(i);
6:    for j := 1 to m do begin
7:      W := worki[a_j]; { kopiujemy cały worek, a nie referencję do niego }
8:      worki[a_j] := ∅;
9:      foreach i ∈ W do begin
10:       k[i] := k[i] + 1;
11:       if k[i] > m_i then zadowolony[i] := true
12:       else worki[b_{i,k[i]}].insert(i);
13:     end
14:   end
15:   for i := 1 to n do
16:     if zadowolony[i] then write(TAK) else write(NIE);
```

Do zakończenia opisu powyższego algorytmu pozostał jeszcze jeden drobiazg — kwestia reprezentacji naszych worków. Odpowiednia struktura danych powinna umożliwiać wstawianie (metoda *insert* powyżej) oraz usuwanie elementów, w jakiejkolwiek kolejności. Zauważmy, że za pomocą tych operacji można bez problemów symulować przeniesienie zawartości jednego worka do innego (wiersze 7-8), a także jednokrotne przejrzenie wszystkich elementów worka (pętla **foreach** w wierszu 9). Do tego celu świetnie nadaje się większość klasycznych dynamicznych struktur danych: stos (kolejka LIFO), kolejka (FIFO), lista itp. — o tych i innych strukturach dynamicznych można poczytać w większości książek poświęconych algorytmom, np. [19] czy [21]. Każda z nich umożliwia wstawianie i usuwanie elementów w czasie stałym. Dodajmy tylko, że programujący w C++ mogą korzystać z gotowych implementacji dynamicznych struktur danych w bibliotece STL, takich jak `stack`, `queue`, `deque`, `list`, czy nawet wspomniany już wcześniej `vector`.

Spróbujmy wreszcie przeanalizować złożoność czasową opisanego rozwiązania. Widać wyraźnie, że główny koszt czasowy stanowią pętle w wierszach 6-14. Można by szacować ten koszt zgrubnie — pętla **for** wykonuje m obrotów, a w każdym z nich przeglądamy jakiś worek, którego rozmiar nie przekracza n — i w ten sposób otrzymać oszacowanie $O(n \cdot m)$. Pokażemy jednak, że złożoność czasowa opisanego rozwiązania jest istotnie mniejsza.

W tym celu zastosujemy analizę kosztu zamortyzowanego (patrz np. książka [21]), tylko w bardzo, bardzo prostej postaci. Pokażemy, że jakkolwiek pojedyncza sekwencja obrotów pętli **foreach** (wiersz 9) może być długa, to *łączna* liczba jej obrotów jest ograniczona. Zauważmy, że w wyniku każdego takiego obrotu wskaźnik oczekiwanego elementu w jakimś ciągu (tj. $k[i]$) przesuwa się o 1. Łączna liczba zwiększeń

danego $k[i]$ nie może przekroczyć m_i, co pokazuje, że sumaryczna liczba obrotów pętli **foreach** jest ograniczona przez $m_1 + m_2 + \ldots + m_n = len$. Dodając do tego koszt czasowy inicjacji struktur danych (wiersze 2-5), wypisywania wyniku (wiersze 15-16) i „jałowych" obrotów pętli **for** (pusty worek), otrzymujemy łączną złożoność czasową tego rozwiązania: $O(len + m + s)$. Jest ona lepsza niż w przypadku poprzedniego algorytmu, ale w praktyce różnica jest na tyle nieznaczna, że oba zostały uznane za wzorcowe. Dodajmy, że złożoność pamięciowa jest tu taka sama jak czasowa.

Implementacje tego rozwiązania można znaleźć w plikach `tes.cpp`, `tes1.c`, `tes2.pas` i `tes3.cpp`.

Testy

Rozwiązania niniejszego zadania były sprawdzane na 10 zestawach danych testowych. Testy *a* to testy losowe, które w tym zadaniu pełnią też rolę testów poprawnościowych. Testy *b* zawierają dużo zapytań o w miarę krótkie podciągi i pełnią rolę testów wydajnościowych. Test *10c* to maksymalny i zarazem skrajny przypadek testowy.

Nazwa	m	n	len	odpowiedzi tak/nie
tes1.in	20	10	100	3 / 7
tes2.in	1 000	100	100 000	10 / 90
tes3a.in	10 000	10 000	100 000	1 460 / 8 540
tes3b.in	100 000	2 500	100 000	1 900 / 600
tes4a.in	100 000	5 000	800 000	422 / 4 578
tes4b.in	200 000	5 000	200 000	3 785 / 1 215
tes5a.in	300 000	9 000	1 000 000	625 / 8 375
tes5b.in	300 000	7 500	300 000	5 560 / 1 940
tes6a.in	500 000	1 000	500 000	999 / 1
tes6b.in	500 000	12 500	500 000	9 358 / 3 142
tes7a.in	700 000	2 000	800 000	887 / 1 113
tes7b.in	700 000	17 500	700 000	13 141 / 4 359
tes8a.in	800 000	4 000	900 000	1 868 / 2 132
tes8b.in	800 000	20 000	800 000	14 944 / 5 056
tes9a.in	900 000	10 000	1 000 000	9 802 / 198
tes9b.in	900 000	22 500	900 000	16 924 / 5 576
tes10a.in	1 000 000	100 000	1 000 000	95 740 / 4 260
tes10b.in	1 000 000	25 000	1 000 000	18 700 / 6 300
tes10c.in	1 000 000	1	1 000 000	1 / 0

Zawody II stopnia

opracowania zadań

Jakub Radoszewski, Wojciech Rytter — **Treść zadania, Opracowanie**
Mirosław Michalski — **Program**

Dostępna pamięć: 32 MB. *OI, Etap II, dzień próbny, 9.02.2010*

Antysymetria

Bajtazar studiuje różne napisy złożone z zer i jedynek. Niech x będzie takim napisem, przez x^R będziemy oznaczać odwrócony (czyli „czytany wspak") napis x, a przez $\overline{x}$ będziemy oznaczać napis powstały z x przez zamianę wszystkich zer na jedynki, a jedynek na zera.

Bajtazara interesuje **antysymetria**, *natomiast niezbyt lubi wszystko co symetryczne. Antysymetria nie jest tylko prostym zaprzeczeniem symetrii. Powiemy, że (niepusty) napis x jest* **antysymetryczny**, *jeżeli dla każdej pozycji i w x, i-ty znak od końca jest różny od i-tego znaku, licząc od początku. W szczególności, niepusty napis x złożony z zer i jedynek jest antysymetryczny wtedy i tylko wtedy, gdy $x = \overline{x}^R$. Na przykład, napisy* `00001111` *i* `010101` *są antysymetryczne, natomiast* `1001` *nie jest.*

W zadanym napisie złożonym z zer i jedynek chcielibyśmy wyznaczyć liczbę jego spójnych (tj. jednokawałkowych) niepustych fragmentów, które są antysymetryczne. Jeżeli różne fragmenty odpowiadają takim samym słowom, to i tak należy je policzyć wielokrotnie.

Wejście

Pierwszy wiersz standardowego wejścia zawiera liczbę n ($1 \leqslant n \leqslant 500\,000$), oznaczającą długość napisu. Drugi wiersz zawiera napis złożony z liter 0 i/lub 1 o długości n. Napis ten nie zawiera żadnych odstępów.

Wyjście

Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą, oznaczającą liczbę spójnych fragmentów wczytanego napisu, które są antysymetryczne.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`8` `11001011`	`7`

Antysymetryczne fragmenty to: `01` *(pojawia się dwukrotnie),* `10` *(także dwukrotnie),* `0101`, `1100` *oraz* `001011`.

Rozwiązanie

Analiza problemu

Zastanówmy się, czym tak naprawdę jest napis antysymetryczny. No dobrze, a czym jest napis *symetryczny*? Napis symetryczny spełnia warunek $x = x^R$, czyli jest niczym

innym jak *palindromem* — napisem czytanym tak samo normalnie i wspak. Na palindrom parzystej długości (tzw. palindrom parzysty) możemy także spojrzeć jak na napis postaci $u \cdot u^R = uu^R$, czyli sklejenie pierwszej połówki z drugą połówką będącą jej odwróceniem. Palindromy nieparzyste różnią się od parzystych tylko tym, że mają w środku jeszcze jedną, dodatkową cyfrę c, czyli są postaci ucu^R.

Przez analogię, napisy antysymetryczne będziemy nazywali *antypalindromami*. Binarny antypalindrom parzysty również możemy podzielić na połówki, z których druga jest tym razem odwróceniem i negacją (zamiana zer na jedynki i odwrotnie) pierwszej, czyli jest to napis postaci $u\overline{u}^R$. Faktycznie, wynika to wprost z warunku, że i-ta cyfra od początku musi być inna od i-tej cyfry od końca, co dla napisów złożonych wyłącznie z cyfr $\{0,1\}$ oznacza, że cyfry te są swoimi negacjami. Przykładowo, $\mathtt{11010100} = \mathtt{1101}\,\overline{\mathtt{1011}}$. Z kolei antypalindrom nieparzystej długości $2k+1$ powinien być postaci $uc\overline{u}^R$ dla pewnej cyfry c. Powinien, ale tak być nie może: z definicji wynika, że $(k+1)$-sza cyfra od początku tego napisu, czyli c, powinna być różna od $(k+1)$-szej cyfry od końca, czyli także c — a wszakże c od c różne być nie może. To pokazuje, że antypalindromy nieparzystej długości nie istnieją!

Podsłowem napisu (słowa) zerojedynkowego s nazwiemy dowolny jego spójny fragment. Jeżeli $s = s_1 s_2 \ldots s_n$, to przez $s[i..j]$ oznaczymy podsłowo $s_i s_{i+1} \ldots s_j$. Widzimy, że aby rozwiązać zadanie, musimy wyznaczyć liczbę podsłów wyjściowego słowa s będących antypalindromami parzystymi. A jak rozwiązywalibyśmy podobne zadanie, w którym mielibyśmy zliczyć podsłowa będące *palindromami* parzystymi? Bardzo krótka odpowiedź na to pytanie brzmi: obliczamy promienie palindromów parzystych za pomocą algorytmu Manachera i wypisujemy ich sumę, co można wykonać w czasie liniowym względem długości słowa s. W przypadku antypalindromów zrobimy generalnie to samo, tylko użyjemy odpowiednio zmodyfikowanego algorytmu Manachera... Ale po kolei.

Rozwiązanie wzorcowe

O algorytmie Manachera można poczytać przede wszystkim w książce [19], a także na różnych, łatwych do wyszukania stronach internetowych. Pojawił się on także w rozwiązaniu zadania *Osie symetrii* z XIV Olimpiady Informatycznej [14]. W tej sekcji opiszemy ten algorytm zmodyfikowany do wyznaczania antypalindromów parzystych (działający całkiem analogicznie jak oryginalna wersja z palindromami).

Promienie antypalindromów

Dane jest n-literowe słowo zerojedynkowe $s = s_1 s_2 \ldots s_n$. Powiemy, że parzystej długości słowo v jest podsłowem słowa s o *środku* na pozycji i, jeżeli

$$v \;=\; s[i-j+1..i+j] \;=\; s[i-j+1..i] \cdot s[i+1..i+j].$$

Jak wygląda zbiór wszystkich antypalindromów parzystych o środku na ustalonej pozycji i? Zauważmy, że jeżeli $s[i-j+1..i+j]$ jest takim antypalindromem, to wszystkie inne podsłowa o środku na i-tej pozycji i długości mniejszej niż $2j$ są antypalindromami parzystymi. Wynika to z faktu, że po usunięciu pierwszej i ostatniej

cyfry antypalindrom zmienia się także w antypalindrom, tyle że krótszy (proste uzasadnienie tej obserwacji pozostawiamy Czytelnikowi).

Możemy teraz zdefiniować *promień* antypalindromu na i-tej pozycji słowa (oznaczenie: $R[i]$) jako największe takie $j \geqslant 0$, że podsłowo $s[i-j+1..i+j]$ jest antypalindromem, patrz także tabelka na rys. 1. Dzięki wcześniej poczynionej obserwacji opisującej zbiór antypalindromów o środku na danej pozycji wiemy, że tak zdefiniowany promień ma następującą, intuicyjną własność: wszystkie podsłowa o środku na pozycji i i długości nieprzekraczającej $2R[i]$ są antypalindromami, a wszystkie dłuższe — już nie. To oznacza, że mamy łącznie $R[i]$ antypalindromów o środku na danej pozycji i, a zatem poszukiwana liczba wszystkich podsłów słowa s będących antypalindromami jest równa sumie wszystkich promieni antypalindromów, czyli

$$wynik = R[1] + R[2] + \ldots + R[n-1]. \tag{1}$$

Wniosek z tego prosty: aby rozwiązać zadanie, wystarczy obliczyć promienie antypalindromów na wszystkich pozycjach słowa.

i	1	2	3	4	5	6	7	8	9	10
s_i	**0**	**1**	**1**	**0**	**1**	**0**	**1**	**0**	**0**	**0**
$R[i]$	1	0	1	2	4	2	1	0	0	

Rys. 1: Promienie antypalindromów na poszczególnych pozycjach słowa 0110101000. Przykładowo, promień na pozycji 5 jest równy 4, ponieważ najdłuższym antypalindromem o środku na pozycji 5 jest 11010100.

Obliczanie promieni antypalindromów

Promienie antypalindromów będziemy obliczać kolejno od lewej do prawej. Załóżmy zatem, że obliczyliśmy już wartości $R[1], R[2], \ldots, R[i]$. W tej sekcji zastanowimy się, czy korzystając z nich, możemy efektywnie obliczać promienie na kolejnych pozycjach słowa. W tym celu pracowicie wyprowadzimy pewien techniczny lemat, który stanowi podstawę działania algorytmu Manachera (Czytelników pragnących od razu poznać sam algorytm odsyłamy do kolejnej sekcji).

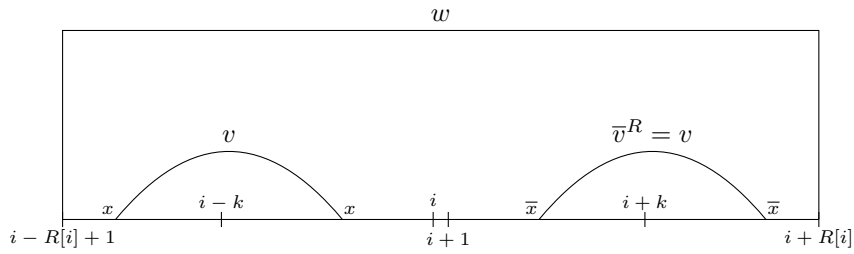
Rozważmy pozycję $i+k$ dla $k < R[i]$, czyli mieszczącą się wewnątrz największego antypalindromu o środku na pozycji i. Okazuje się, że przy obliczaniu $R[i+k]$ możemy skorzystać z wartości $R[i-k]$. Intuicyjnie, dowolny antypalindrom v o środku na pozycji $i-k$, mieszczący się wewnątrz antypalindromu

$$w = s[i - R[i] + 1..i + R[i]],$$

możemy odbić (anty)symetrycznie względem środka antypalindromu w, otrzymując $\overline{v}^R$, czyli dokładnie antypalindrom v, o środku na pozycji $i+k$.

Spróbujmy sformalizować podaną intuicję — ilustracja poniższego rozumowania znajduje się na rys. 2. Załóżmy na początek, że $R[i-k] < R[i]-k$, czyli że największy antypalindrom

$$v = s[(i-k) - R[i-k] + 1..(i-k) + R[i-k]]$$



Rys. 2: Obliczanie promienia $R[i+k]$ w przypadku $R[i-k] < R[i]-k$. Wówczas $R[i+k] = R[i-k]$. Na tym rysunku wystąpienia antypalindromu v akurat nie nachodzą na siebie, ale równie dobrze mogłyby.

o środku na pozycji $i-k$ mieści się ściśle wewnątrz antypalindromu w, a zatem, w szczególności, v nie dotyka brzegu w. Ze względu na to, iż w jest antypalindromem, mamy, że dla dowolnego $a = 1, 2, \ldots, R[i]$:

$$s_{i+a} = \overline{s_{i-a+1}}.$$

Stąd w szczególności:

$$\begin{aligned} s_{i+k+R[i-k]} &= \overline{s_{i-k-R[i-k]+1}}, \\ s_{i+k+R[i-k]-1} &= \overline{s_{i-k-R[i-k]+2}}, \\ &\vdots \\ s_{i+k-R[i-k]+1} &= \overline{s_{i-k+R[i-k]}}, \end{aligned}$$

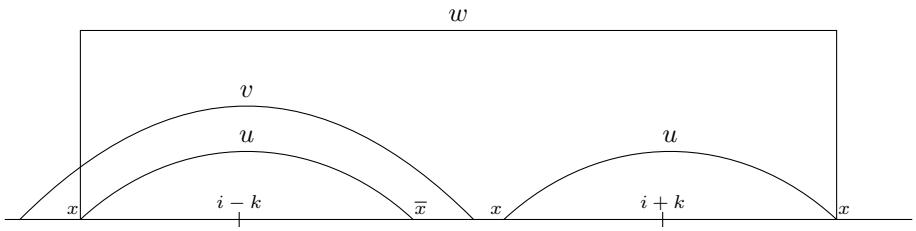
co pokazuje, że

$$s[(i+k) - R[i-k] + 1\,..\,(i+k) + R[i-k]] \;=\; \overline{v}^R \;=\; v.$$

Stąd $R[i+k] \geqslant R[i-k]$.

Pokażemy, że nie może zachodzić $R[i+k] > R[i-k]$, czyli że w rzeczy samej $R[i+k] = R[i-k]$. W tym celu rozważmy cyfry sąsiadujące z lewej i z prawej z wystąpieniem antypalindromu v o środku na pozycji $i-k$, to znaczy $s_{i-k-R[i-k]}$ oraz $s_{i-k+R[i-k]+1}$. Na mocy definicji $R[i-k]$ wiemy, że te cyfry muszą być takie same (na rys. 2 obie oznaczone są przez x). Zauważmy, że obie te cyfry mieszczą się wewnątrz antypalindromu w. Nie powinno to już stanowić dla Czytelnika niespodzianki, że ich odbicia względem środka antypalindromu w, to znaczy odpowiednio $s_{i+k+R[i-k]+1}$ oraz $s_{i+k-R[i-k]}$, są dokładnie cyframi sąsiadującymi z wystąpieniem antypalindromu v na pozycji $i+k$. Te cyfry są negacjami dwóch wcześniej rozważanych, a zatem także są równe ($\overline{x}$ na rysunku), co pokazuje, że tego wystąpienia antypalindromu v także nie można rozszerzyć. Ostatecznie mamy, że jeśli $R[i-k] < R[i]-k$, to po prostu $R[i+k] = R[i-k]$, co przy wyznaczaniu promieni antypalindromów może być nadzwyczaj pomocne.

Dotychczas poszło nam całkiem nieźle, możemy spróbować przeanalizować kolejne przypadki. Przypadek, w którym $R[i-k] = R[i]-k$ (v stanowi *prefiks*, czyli początkowy fragment, słowa w), zostawimy sobie na deser, wcześniej rozpatrzymy przypadek,



Rys. 3: Obliczanie promienia $R[i+k]$ w przypadku $R[i-k] > R[i]-k$. Wówczas $R[i+k] = R[i]-k$.

w którym $R[i-k] > R[i]-k$, czyli antypalindrom v wystaje poza antypalindrom w — patrz rys. 3. Nie musimy już chyba formalnie dowodzić, że wówczas środkowy fragment u antypalindromu v mieszczący się wewnątrz w, będący antypalindromem o długości $R[i]-k$, możemy odbić antysymetrycznie wewnątrz w, otrzymując antypalindrom o środku na pozycji $i+k$, co oznacza, że $R[i+k] \geqslant R[i]-k$. Pokażemy, że tę nierówność możemy zamienić w istocie w równość.

W tym celu musimy ponownie przyjrzeć się cyfrom sąsiadującym bezpośrednio z wystąpieniem u na pozycji $i-k$: są to $s_{i-R[i]}$ i jeszcze jedna cyfra s_j mieszcząca się w ramach w (dokładny wzór na indeks j celowo pomijamy). Przypomnijmy, że antypalindrom u możemy rozszerzyć do dłuższego antypalindromu v, co pokazuje, że rozważane cyfry muszą być różne: $s_{i-R[i]} = \overline{s_j}$ (odpowiednio x i $\overline{x}$ na rysunku). A jakie są cyfry sąsiadujące bezpośrednio z wystąpieniem u o środku na pozycji $i+k$? Jedna z nich to na pewno negacja cyfry s_j. Druga to cyfra $s_{i+R[i]+1}$, następująca bezpośrednio za wystąpieniem w w ramach s. Skoro tak, to czy możemy cokolwiek o niej powiedzieć? Okazuje się, że tak: ponieważ antypalindromu w nie można rozszerzyć, więc $s_{i+R[i]+1} = s_{i-R[i]}$. Ostatecznie, cyfry sąsiadujące z wystąpieniem u o środku na pozycji $i+k$ muszą być równe:

$$s_{i+R[i]+1} \;=\; s_{i-R[i]} \;=\; \overline{s_j},$$

co pokazuje, że antypalindromu u o środku na pozycji $i+k$ nie można już rozszerzyć. Uzasadniliśmy zatem, że jeżeli $R[i-k] > R[i]-k$, to $R[i+k] = R[i]-k$.

Uważny Czytelnik mógł dostrzec w ostatnim rozumowaniu pewną lukę: zastanawiamy się nad cyframi $s_{i-R[i]}$ oraz $s_{i+R[i]+1}$, choć właściwie nie jest nigdzie powiedziane, że te cyfry w ogóle istnieją! Nie jest to jednak większy problem — $s_{i-R[i]}$ na pewno istnieje, gdyż mieści się w v. Jeśli natomiast $s_{i+R[i]+1}$ jest już poza naszym słowem, to tym bardziej nierówność $R[i+k] \geqslant R[i]-k$ nie może być ostra — wszak antypalindrom musi mieścić się w słowie.

Po tym uzupełnieniu możemy przejść do ostatniego nierozważonego przypadku, w którym $R[i-k] = R[i]-k$ (v jest prefiksem w). Wiemy już, że wówczas antypalindrom v występuje także na pozycji $i+k$, co pokazuje, że $R[i+k] \geqslant R[i-k]$. Łatwo przekonać się, że tym razem argument dotyczący sąsiadujących cyfr nie zadziała, czyli nie jesteśmy w stanie powiedzieć, czy ta nierówność jest równością, czy też nie. Trudno, pozostańmy z tym, co mamy.

Algorytm Manachera

W wyniku powyższego, całkiem obszernego rozumowania uzyskaliśmy pewne przydatne zależności, które możemy podsumować w skondensowany sposób za pomocą następującego lematu (w naszym rozumowaniu nie rozważaliśmy przypadku $k = R[i]$, lecz łatwo sprawdzić, że wówczas także teza lematu zachodzi).

Lemat 1. Niech $i \in \{1, 2, \ldots, n-1\}$ oraz niech $k \in \{1, 2, \ldots, R[i]\}$. Wówczas:

a) jeżeli $R[i-k] \neq R[i]-k$, to $R[i+k] = \min(R[i-k],\ R[i]-k)$,

b) a jeżeli $R[i-k] = R[i]-k$, to $R[i+k] \geqslant R[i-k]$.

Co prawda wyprowadzenie Lematu 1 kosztowało nas trochę wysiłku, jednak na jego podstawie możemy od razu zaproponować algorytm wyznaczania promieni antypalindromów. W algorytmie będziemy próbowali korzystać z części a) Lematu, a jeżeli kiedykolwiek będziemy zmuszeni do użycia części b) lub wyjdziemy poza obręb największego antypalindromu o środku na danej pozycji, to dla tej kłopotliwej pozycji wyznaczymy promień antypalindromu nachalnie (cyferka po cyferce). Po tym znów będziemy próbowali używać Lematu 1, korzystając z promienia obliczonego dla nowej pozycji itd.

W implementacji dodamy na początku i końcu słowa s tzw. strażników: symbole `$` i `#`, niewystępujące poza tym w słowie s, których negacje $\overline{\texttt{\$}}$ oraz $\overline{\texttt{\#}}$ będą jeszcze innymi symbolami. W ten sposób poradzimy sobie z przypadkami brzegowymi — nigdy nie będziemy przedłużać antypalindromów poza słowo.

```
1:  Algorytm Manachera
2:  s[0] := '$';  s[n+1] := '#';
3:  i := 1;  j := 0;
4:  while i ⩽ n-1 do begin
5:     while s[i-j] = s̄[i+j+1] do j := j+1;
6:     R[i] := j;
7:     k := 1;
8:     while (k ⩽ R[i]) and (R[i-k] ≠ R[i]-k) do begin
9:        R[i+k] := min(R[i-k], R[i]-k);
10:       k := k+1;
11:    end
12:    j := max(j-k, 0);  i := i+k;
13: end
```

Dokładniejsze sprawdzenie, że powyższa implementacja działa zgodnie z wcześniejszym opisem słownym, pozostawiamy Czytelnikowi. Zastanówmy się natomiast, dlaczego ten algorytm jest efektywny czasowo — pokażemy, że wbrew pewnym pozorom, ma on złożoność czasową $O(n)$.

Owe pozory to zagnieżdżenie pętli **while**. Chcemy pokazać, że łącznie pętle te wykonają co najwyżej $O(n)$ obrotów. Standardowe podejście w takim przypadku polega na znalezieniu jednego parametru (potencjalnie pewnej funkcji od zmiennych występujących w algorytmie), który wzrasta w każdym obrocie wewnętrznych pętli

i ma przedział wartości rozmiaru $O(n)$. Tutaj też tak można, ale wygodniej będzie dokonać interpretacji algorytmu, biorąc pod uwagę to, co on tak naprawdę oblicza.

Otóż cały czas utrzymujemy przedział $[i-j+1\,..\,i+j]$ oznaczający (właśnie obliczany bądź już obliczony) największy antypalindrom o środku na aktualnej pozycji i. Pierwsza wewnętrzna pętla **while** (wiersz 5 pseudokodu) przesuwa prawy koniec tego przedziału o jeden, za to środka przedziału (zmienna i) nie rusza. Z kolei druga z rozważanych, wewnętrznych pętli w każdym obrocie przesuwa środek przedziału o jeden w prawo $(i+1, i+2, \ldots)$, nie ruszając prawego końca. To oznacza, że każda z tych pętli z osobna wykonuje co najwyżej n obrotów, gdyż zarówno środek, jak i prawy koniec aktualnego przedziału przyjmują wartości ze zbioru $\{1, 2, \ldots, n+1\}$.

Wszystkich pozostałych operacji jest w algorytmie wykonywanych $O(n)$. To oznacza, że rzeczywiście jego złożoność czasowa jest liniowa względem n.

Podsumowanie

Przypomnijmy: w rozwiązaniu wzorcowym najpierw wyznaczamy promienie antypalindromów parzystych za pomocą zmodyfikowanego algorytmu Manachera, a następnie obliczamy wynik ze wzoru (1). Każdy z tych kroków ma złożoność czasową $O(n)$.

Rozwiązanie wzorcowe zostało zaimplementowane w plikach `ant.cpp`, `ant1.pas` i `ant2.cpp`.

Inne rozwiązania

Rozwiązanie wzorcowe jest bardzo eleganckie i zarazem efektywne, jednakże bez znajomości algorytmu Manachera wydaje się dosyć trudne do wymyślenia. Okazuje się, że istnieje cała gama innych rozwiązań tego zadania, a wśród nich i takie, które pozwalały zdobyć na zawodach maksymalną punktację. Poza najmniej efektywnymi, które pozostawiamy do rozważenia Czytelnikowi — najprostsze rozwiązanie o złożoności czasowej $O(n^3)$ oraz trochę lepsze rozwiązanie o złożoności czasowej i pamięciowej $O(n^2)$, oparte na metodzie programowania dynamicznego (porównaj opis rozwiązania zadania *Palindromy* z II Olimpiady Informatycznej [2]) — wszystkie one były oparte na wyznaczaniu promieni antypalindromów i obliczały wynik ze wzoru (1).

W najprostszym z tych rozwiązań obliczamy żądane promienie antypalindromów, każdorazowo poszerzając podsłowo o środku na danej pozycji i o kolejne cyfry, dopóki jest ono antypalindromem. W ten sposób otrzymujemy rozwiązanie o złożoności czasowej $O(n^2)$ i pamięciowej $O(n)$, którego pseudokod zamieszczamy poniżej. Dodajmy jeszcze, że koszt czasowy tego rozwiązania zależy tak naprawdę od sumy promieni antypalindromów, czyli lepszym oszacowaniem na złożoność czasową jest $O(n+m)$, przy czym m to maksymalny wynik. Niestety, pesymistycznie $m = \Theta(n^2)$, o czym łatwo przekonać się, rozważając słowo $010101\ldots01$.

```
1: Algorytm O(n+m)
2: s[0] := '$';  s[n+1] := '#';
3: for i := 1 to n-1 do begin
4:     R[i] := 0;
5:     while s_{i-R[i]} = ¬s_{i+R[i]+1} do R[i] := R[i]+1;
```

```
6: end
```

Implementację rozwiązania korzystającego z tej metody można znaleźć w pliku `ants3.cpp`. Na zawodach takie rozwiązania uzyskiwały około 50 punktów.

Do wyznaczania promieni $R[i]$ możemy także użyć wyszukiwania binarnego. W tym celu potrzebna nam będzie efektywna funkcja sprawdzająca, czy dane podsłowo $s[i..j]$ słowa s jest antypalindromem:

```
1: Algorytm z wyszukiwaniem binarnym
2: for i := 1 to n − 1 do begin
3:    a := 0;  b := min(i, n − i);
4:    while a < b do begin
5:       c := (a + b) div 2;
6:       if czy_antypalindrom(s[i − c + 1..i + c]) then a := c
7:       else b := c − 1;
8:    end
9:    R[i] := a;
10: end
```

Na pierwszy rzut oka takie sprawdzenie nie wydaje się proste do wykonania w efektywny sposób, tj. szybciej niż liniowo względem długości podsłowa. Okazuje się jednak, że można tak zmodyfikować wyjściowe słowo s, aby sprawdzanie, czy dane podsłowo s jest antypalindromem, sprowadzało się do sprawdzenia *równości* pewnych dwóch podsłów zmodyfikowanego słowa s. Tym zmodyfikowanym słowem będzie $t = s\overline{s}^R$.

Obserwacja 1. Podsłowo $s[i..j]$ jest antypalindromem wtedy i tylko wtedy, gdy

$$t[i..j] \;=\; t[(2n+1-j)..(2n+1-i)].$$

Dowód: Oznaczmy $w = s[i..j]$. Wystarczy zauważyć, że:

$$t[i..j] \;=\; w, \quad t[(2n+1-j)..(2n+1-i)] \;=\; \overline{w}^R.$$

Kto nie wierzy, niech sprawdzi (co najmniej na przykładzie). ■

Sprowadziliśmy zatem wyjściowy problem do problemu sprawdzania równości zadanych podsłów słowa t. A na to jest już wiele gotowych algorytmów.

Jednym z nich jest *słownik podsłów bazowych*, który przypisuje całkowite dodatnie identyfikatory podsłowom słowa t o długościach będących potęgami dwójki, co następnie pozwala wyznaczać jednoznaczne identyfikatory dowolnych podsłów t. Więcej o tej metodzie można przeczytać w książce [19] (algorytm Karpa-Millera-Rosenberga, KMR) lub w opisie rozwiązania zadania *Powtórzenia* z VII Olimpiady Informatycznej [7]. Złożoność czasowa wstępnych obliczeń to w tej metodzie $O(n \log^2 n)$ lub $O(n \log n)$, w zależności od jakości implementacji, przy złożoności pamięciowej rzędu $O(n \log n)$. Po tych obliczeniach sprawdzanie równości podsłów jest już wykonywane w czasie stałym.

Rozwiązania używające tej metody mają łączną złożoność czasową $O(n \log^2 n)$ lub $O(n \log n)$, a pamięciową $O(n \log n)$. Różne implementacje można znaleźć w plikach

`ant[s4-s7].[cpp|pas]`. Ich głównym mankamentem jest duża złożoność pamięciowa (jak na limit pamięciowy 32 MB), przez co zdobywały na zawodach około 50 punktów.

Lubiący odrobinę ryzyka mogą do sprawdzania równości podsłów słowa t użyć metody *haszowania*. W tej metodzie cyfry występujące w danym podsłowie traktujemy jako kolejne współczynniki wielomianu i przydzielamy temu podsłowu identyfikator będący wartością tego wielomianu w jakimś punkcie całkowitym, $x = q$, wziętą modulo pewna liczba całkowita dodatnia (pierwsza) p. Niestety, tak przydzielone identyfikatory nie muszą być różne dla różnych podsłów (w przypadku zajścia takiej równości mówimy o tzw. *kolizji*). Dobierając odpowiednio dużą wartość p i losując wartość q, możemy jednak mieć nadzieję, że dla żadnych podsłów, które musimy akurat porównać, nie napotkamy na kolizję. I rzeczywiście, prawdopodobieństwo kolizji jest małe, ale mimo wszystko nie da się jej wykluczyć.

Więcej o tej metodzie można przeczytać chociażby w książkach [19] i [21] (algorytm Karpa-Rabina, KR), a także — w praktycznym ujęciu — w artykule *Drobne oszustwo* w numerze 11/2009 czasopisma *Delta*[1]. Można ją zaimplementować bardzo efektywnie: złożoność czasowa i pamięciowa wstępnych obliczeń jest rzędu $O(n)$, a na zapytania o równość podsłów możemy odpowiadać w czasie stałym.

Rozwiązania używające tej metody mają — ze względu na wyszukiwanie binarne — złożoność czasową $O(n \log n)$, jednak złożoność pamięciowa $O(n)$ jest lepsza niż w przypadku słownika podsłów bazowych. Różne implementacje (przykłady: `antb1.cpp`, `antb2.cpp`) uzyskiwały na zawodach punktację zbliżoną do maksymalnej bądź maksymalną, w zależności od szczęścia — a, jak mówi znane przysłowie, szczęście sprzyja lepszym, czyli w tym przypadku tym, którzy rozsądnie dobierają parametry używane w haszowaniu.

Rozwiązania błędne

Wśród rozwiązań błędnych możemy wyróżnić głównie niepoprawne implementacje opisanych powyżej rozwiązań. Przede wszystkim należało zwrócić uwagę na to, że wynik mógł nie mieścić się w typie całkowitym 32-bitowym, ale wymagać użycia zmiennych 64-bitowych — rozwiązanie podobne do wzorcowego, lecz nieuwzględniające tego faktu, uzyskiwało na zawodach około 70 punktów.

Testy

Do oceny rozwiązań tego zadania użyto 14 grup testów. Poniżej zamieszczona jest tabela zawierająca podstawowe statystyki poszczególnych testów (n to długość wyjściowego słowa, a m to wynik, czyli liczba podsłów słowa będących antypalindromami).

Nazwa	n	m
ant1a.in	1	0
ant1b.in	1	0
ant1c.in	2	1
ant1d.in	10	11

Nazwa	n	m
ant1e.in	2	0
ant2a.in	45	0
ant2b.in	50	8
ant2c.in	50	50

[1] Artykuł dostępny także na stronie internetowej czasopisma: `http://www.mimuw.edu.pl/delta/`

Nazwa	n	m
ant2d.in	100	146
ant3a.in	61	900
ant3b.in	422	657
ant3c.in	600	832
ant3d.in	800	152 122
ant4a.in	1 002	1 555
ant4b.in	1 030	995
ant4c.in	1 198	4 303
ant4d.in	1 601	361 141
ant5a.in	6 656	235 444
ant5b.in	6 231	247 829
ant5c.in	7 053	3 177 100
ant5d.in	7 668	13 319
ant6a.in	20 480	2 755 601
ant6b.in	18 334	1 073 433
ant6c.in	20 000	100 000 000
ant6d.in	38 100	39 782
ant7a.in	37 008	17 179 122
ant7b.in	100 000	149 818
ant7c.in	106 822	53 411
ant7d.in	499 993	0
ant8a.in	158 304	2 088 399 136
ant8b.in	200 000	687 393
ant8c.in	200 000	940 530
ant8d.in	221 260	332 150

Nazwa	n	m
ant9a.in	400 004	602 414
ant9b.in	407 701	1 783 070 527
ant9c.in	409 600	839 278 590
ant9d.in	400 000	597 956
ant10a.in	500 000	750 062
ant10b.in	500 000	499 417
ant10c.in	500 000	2 084 116 658
ant10d.in	500 000	1 746 350
ant11a.in	500 000	499 915
ant11b.in	500 000	5 208 749 996
ant11c.in	500 000	752 483
ant11d.in	500 000	600 178
ant12a.in	500 000	8 929 285 709
ant12b.in	500 000	10 416 916 665
ant12c.in	500 000	15 625 125 000
ant12d.in	500 000	52 900 520 440
ant13a.in	500 000	20 000 251 084
ant13b.in	500 000	746 684
ant13c.in	500 000	32 034 926 466
ant13d.in	500 000	1 606 456
ant14a.in	499 998	20 833 333 333
ant14b.in	491 520	304 094 082
ant14c.in	500 000	500 006
ant14d.in	500 000	62 500 000 000

Jakub Radoszewski, Wojciech Rytter — Treść zadania, Opracowanie
Piotr Niedźwiedź — Program

Dostępna pamięć: 64 MB. *OI, Etap II, dzień pierwszy, 10.02.2010*

Chomiki

Bajtazar prowadzi hodowlę chomików. Każdy chomik ma unikalne imię, złożone z małych liter alfabetu angielskiego. Chomiki mają obszerną i komfortową klatkę. Bajtazar chce umieścić pod klatką wyświetlacz, który będzie wyświetlał imiona jego chomików. Wyświetlacz będzie miał postać ciągu liter, z których każda może być zapalona lub zgaszona. Naraz będzie wyświetlane tylko jedno imię chomika. Zapalone litery tworzące to imię muszą znajdować się obok siebie.

Bajtazar chce, aby wyświetlacz mógł wyświetlać imiona chomików przynajmniej w m różnych miejscach. Dopuszcza on, aby to samo imię mogło być wyświetlane w kilku różnych miejscach, i nie wymaga, aby każde imię mogło być wyświetlane na wyświetlaczu. Zauważ, że wystąpienia imion na wyświetlaczu mogą dowolnie na siebie nachodzić. Możesz założyć, że imię żadnego z chomików nie występuje (jako spójny fragment) w imieniu żadnego innego chomika. Bajtazar poprosił Cię o pomoc w wyznaczeniu najmniejszej liczby liter, z jakich musi składać się wyświetlacz.

Inaczej mówiąc, należy wyznaczyć minimalną długość napisu (złożonego z małych liter alfabetu angielskiego), w którym łączna liczba wystąpień imion chomików jest nie mniejsza niż m. (Mówimy, że słowo s występuje w napisie t, jeżeli s stanowi spójny fragment t).

Wejście

Pierwszy wiersz standardowego wejścia zawiera dwie liczby całkowite n oraz m ($1 \leqslant n \leqslant 200$, $1 \leqslant m \leqslant 10^9$), oddzielone pojedynczym odstępem i oznaczające, odpowiednio, liczbę chomików Bajtazara i minimalną liczbę wystąpień imion chomików na wyświetlaczu. Każdy z kolejnych n wierszy zawiera niepusty napis złożony z małych liter alfabetu angielskiego będący imieniem chomika. Sumaryczna długość wszystkich imion nie przekracza 100 000 liter.

Wyjście

Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą — minimalną liczbę liter, z których musi być zbudowany wyświetlacz.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*

`4 5`
`monika`
`tomek`
`szymon`
`bernard`
	`23`

Najkrótszy wyświetlacz może mieć, na przykład, postać: `szymonikatomekszymonika`. *Zawiera on łącznie 5 wystąpień imion chomików:* `szymon` *i* `monika` *występują dwukrotnie,* `tomek` *raz, a* `bernard` *ani razu.*

Rozwiązanie

Trochę formalizmów

Na początku opisu wprowadźmy dla wygody kilka oznaczeń. W tym zadaniu interesować nas będą słowa złożone z małych liter alfabetu angielskiego. Jeśli u jest takim słowem, to przez $|u|$ oznaczymy jego długość, czyli liczbę liter, z których się składa. Powiemy, że słowo u jest *prefiksem* (*sufiksem*, *podsłowem*) słowa v, jeżeli słowo u stanowi początkowy fragment (odpowiednio końcowy fragment albo dowolny spójny fragment) słowa v. Przez $\#wyst(u, v)$ oznaczymy łączną liczbę wystąpień słowa u w postaci podsłów słowa v. Przykładowo, słowo `aba` jest zarazem prefiksem i sufiksem słowa `abababaaba` oraz $\#wyst($`aba`, `abababaaba`$) = 3$.

Niech $S = \{s_1, s_2, \ldots, s_n\}$ będzie zbiorem n słów reprezentujących imiona chomików Bajtazara. W tym zadaniu poszukujemy najkrótszego słowa t, w którym łączna liczba wystąpień imion chomików jest nie mniejsza niż m, co przy wprowadzonych oznaczeniach możemy zapisać jako:

$$\#wyst(s_1, t) + \#wyst(s_2, t) + \ldots + \#wyst(s_n, t) \geqslant m. \tag{1}$$

Warto dodać, że parametr m może być rzędu 10^9, czyli możemy być zmuszeni do poszukiwania naprawdę długiego słowa t. To oznacza, że w rozwiązaniu powinniśmy ograniczyć się do wyznaczenia długości słowa t, bez konstruowania go.

Zakładamy dalej, że słowa ze zbioru S są niepuste ($|s_i| > 0$) i mają łącznie długość L, tzn. $L = |s_1| + |s_2| + \ldots + |s_n|$. W zadaniu zaszyte jest jeszcze jedno, dosyć tajemnicze ograniczenie, że żadne ze słów s_i nie stanowi podsłowa żadnego innego, tj.:

$$\#wyst(s_i, s_j) = 0 \quad \text{dla } i \neq j. \tag{2}$$

Jak wygląda rozwiązanie?

Spróbujmy przyjrzeć się temu, jaką strukturę powinno mieć wynikowe słowo t. Niech $s_{i_1}, s_{i_2}, \ldots, s_{i_m}$ będzie sekwencją reprezentującą pierwsze (od lewej) m wystąpień słów ze zbioru S w słowie t. Dodajmy dla jasności, że na każdej pozycji słowa t może rozpoczynać się co najwyżej jedno wystąpienie jakiegoś ze słów s_{i_j}, gdyż w przeciwnym przypadku któreś ze słów ze zbioru S byłoby prefiksem któregoś innego, co nie jest możliwe wobec warunku (2).

Zacznijmy od tego, że słowo s_{i_1} musi być prefiksem słowa t. Faktycznie, gdyby tak nie było, to słowo powstałe z t przez wyrzucenie pierwszej litery nadal spełniałoby warunek (1), a więc t nie byłoby najkrótsze.

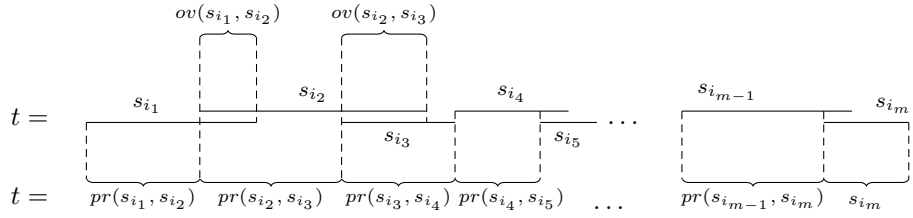
Następne w słowie t występuje s_{i_2}. Łatwo widać, że jego wystąpienie musi zaczynać się w ramach pierwszych $|s_{i_1}| + 1$ liter słowa t, gdyż w przeciwnym przypadku znów moglibyśmy wyciąć jedną literę słowa t, nie zaburzając warunku (1). Z drugiej strony, koniec wystąpienia słowa s_{i_2} musi następować za końcem wystąpienia s_{i_1}, jako że odwrotna sytuacja oznaczałaby, iż s_{i_2} byłoby podsłowem s_{i_1}, co, jak wiemy, nie jest możliwe.

Konstruując słowo t, możemy więc albo ustawić s_{i_2} tuż za s_{i_1}, albo próbować dopasować te słowa z nakładką, tzn. nałożyć pewien prefiks słowa s_{i_2} na identyczny sufiks słowa s_{i_1}. Ponieważ słowo t ma być najkrótsze możliwe, więc opłaca nam się wybrać nakładkę tak długą, jak tylko się da. Zauważmy, że to, jak długą nakładkę wybierzemy, nie ma żadnego znaczenia dla konfiguracji słów $s_{i_3}, \ldots, s_{i_m}$, ponieważ i tak słowo s_{i_2} wystaje poza słowo s_{i_1}, a każde z tych kolejnych wystąpień następuje dalej niż s_{i_2}.

Zauważmy, że całe to rozumowanie można powtórzyć dla słów $s_{i_3}, \ldots, s_{i_m}$, otrzymując ostateczny wniosek, że słowo t musi zaczynać się od s_{i_1}, kończyć się na s_{i_m} oraz że należy zawsze wybierać najdłuższą możliwą nakładkę między $s_{i_{j-1}}$ a s_{i_j}. Spróbujmy zapisać ten wniosek bardziej formalnie.

Dla danych słów u i v, *najdłuższą nakładką* tych słów (ang. *overlap*, oznaczenie: $ov(u, v)$) nazwiemy najdłuższe takie słowo y, że $u = xy$ i $v = yz$ dla pewnych niepustych słów x, z. W szczególności, słowo y może być puste. Z kolei przez *prefiks* $pr(u, v)$ *słowa* u *względem słowa* v będziemy rozumieli słowo x z definicji najdłuższej nakładki, tzn. $u = pr(u, v)ov(u, v)$. Dla przykładu,

$$ov(\texttt{aababab},\ \texttt{babbaa}) = \texttt{bab}, \quad pr(\texttt{aababab},\ \texttt{babbaa}) = \texttt{aaba},$$
$$ov(\texttt{abaab},\ \texttt{abaab}) = \texttt{ab}, \quad pr(\texttt{abaab},\ \texttt{abaab}) = \texttt{aba}.$$



Rys. 1: Struktura szukanego słowa t.

Korzystając z wprowadzonych właśnie oznaczeń, możemy zapisać słowo t jako:

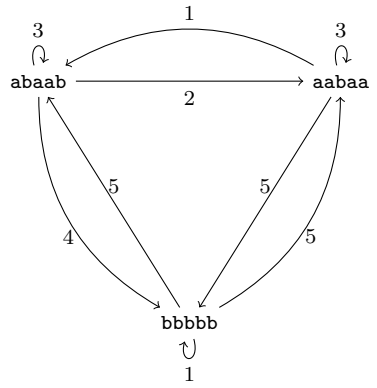
$$t = pr(s_{i_1}, s_{i_2}) \cdot pr(s_{i_2}, s_{i_3}) \cdot \ldots \cdot pr(s_{i_{m-1}}, s_{i_m}) \cdot s_{i_m}, \tag{3}$$

patrz także rys. 1. To oznacza, że gdybyśmy wiedzieli, jak dobrać kolejne słowa s_{i_j}, to umielibyśmy rozwiązać nasze zadanie. Tego jednak póki co nie wiemy, więc będziemy musieli jeszcze trochę pokombinować.

Graf prefiksów

Okazuje się, że łatwiej rozwiązać nasze zadanie, jeśli dane zinterpretujemy jako graf. Dla danego zbioru S skonstruujemy ważony, pełny graf skierowany $G = (V, E)$, który nazwiemy *grafem prefiksów*. Zbiór wierzchołków grafu $V = \{1, 2, \ldots, n\}$ utożsamiamy ze zbiorem słów S. Między każdą parą wierzchołków (i, j) występuje krawędź skierowana, której wagę ustalamy jako $|pr(s_i, s_j)|$. Zauważmy, że w przypadku $i = j$ mamy

w grafie pętlę, czyli formalnie G jest multigrafem. Przykładowy graf prefiksów jest przedstawiony na rys. 2.



Rys. 2: Ilustracja grafu prefiksów dla zbioru słów $S = \{\texttt{abaab}, \texttt{aabaa}, \texttt{bbbbb}\}$.

Jeżeli przypomnimy sobie, co w poprzedniej sekcji udało nam się ustalić odnośnie do postaci wynikowego słowa t, to łatwo zauważymy, że graf prefiksów ma istotny związek z naszym wyjściowym problemem. Faktycznie, zapis (3) możemy odczytać następująco: *długość słowa t jest równa długości najkrótszej ścieżki o m wierzchołkach, łączącej pewne wierzchołki $i, j \in V$, powiększonej o $|s_j|$*. Słowo „najkrótsza" odnosi się tu, oczywiście, do sumy wag na krawędziach. Ta przyjemna własność sprowadza rozwiązanie naszego problemu do poszukiwania najkrótszych (w jakimś sensie) ścieżek w grafie prefiksów. Zanim zaczniemy kontynuować tę myśl, zastanówmy się jeszcze, w jaki sposób efektywnie wyznaczyć graf prefiksów dla danego zbioru S.

Do konstrukcji grafu G wystarczą nam długości prefiksów postaci $|pr(s_i, s_j)|$. Będziemy je wyznaczać osobno dla każdej pary słów (s_i, s_j). Przypomnijmy, że w tym celu wystarczy umieć obliczać najdłuższe nakładki postaci $|ov(s_i, s_j)|$ (czy Czytelnik pamięta dlaczego?). Najprostsze podejście, polegające na sprawdzaniu kolejnych możliwych długości nakładek, pozwala obliczyć każdą taką najdłuższą nakładkę w złożoności czasowej $O\left(\min(|s_i|, |s_j|)^2\right)$.

Aby poprawić ten wynik, zauważmy, że w przypadku, gdy $i \neq j$, $ov(s_i, s_j)$ jest równe długości najdłuższego prefiksu słowa s_j, który jest zarazem sufiksem słowa s_i, czyli jest równe długości najdłuższego właściwego *prefikso-sufiksu* słowa $s_j\#s_i$ (właściwego, czyli krótszego od całego tego słowa). Przy tym # jest symbolem, który (ewidentnie) nie należy do alfabetu angielskiego, a umieściliśmy go w środku po to, aby na pewno wynikowy prefikso-sufiks nie był dłuższy niż s_i lub s_j. W przypadku szczególnym $i = j$ najdłuższa nakładka s_i i s_i odpowiada po prostu najdłuższemu właściwemu prefikso-sufiksowi samego słowa s_i.

Sprowadziliśmy zatem problem wyznaczania najdłuższych nakładek do obliczania najdłuższych właściwych prefikso-sufiksów pewnych słów. W tym miejscu warto przypomnieć, że długość najdłuższego prefikso-sufiksu dowolnego słowa można obliczyć w czasie proporcjonalnym do długości tego słowa, korzystając z funkcji prefiksowej z algorytmu Knutha-Morrisa-Pratta (patrz np. książki [19, 21]). To pokazuje, że $|ov(s_i, s_j)|$ możemy obliczyć (w każdym przypadku) w czasie $O(|s_i| + |s_j|)$, co daje

następujący łączny koszt czasowy obliczenia wszystkich takich nakładek:

$$O\left(\sum_{i=1}^{n}\sum_{j=1}^{n}(|s_i|+|s_j|)\right) = O\left(\sum_{i=1}^{n}(n\cdot|s_i|+L)\right) = O\left(n\cdot\sum_{i=1}^{n}|s_i|+nL\right) = O(nL).$$

W ten właśnie sposób obliczano najdłuższe nakładki, a więc także i względne prefiksy, w rozwiązaniu wzorcowym. Warto w tym miejscu dodać, że istnieją inne metody wykonania tego podzadania. Jedna z nich polega na zastosowaniu drobnego oszustwa, a mianowicie haszowania: dla każdej pary (s_i, s_j) przeglądamy kolejne możliwe długości nakładki i za pomocą haszy sprawdzamy w czasie stałym, czy odpowiedni prefiks i sufiks są równe. W ten sposób otrzymujemy inne rozwiązanie o złożoności czasowej $O(n \cdot L)$, które jest jednakże obarczone pewnym ryzykiem błędu. Więcej o metodzie haszowania można przeczytać w opracowaniu zadania *Antysymetria* w tej książeczce oraz w umieszczonych tam odnośnikach.

Na koniec dodajmy, że wszystkie nakładki można także obliczyć w optymalnej złożoności czasowej $O(n^2 + L)$, co wymaga użycia bardziej skomplikowanych technik — zainteresowanych czytelników odsyłamy do artykułu [37], niestety dostępnego tylko w języku angielskim.

Macierze

Sformułujmy pozostały do rozwiązania problem grafowy w sposób abstrakcyjny. Dany jest ważony, pełny graf skierowany $G = (V, E)$, $V = \{1, 2, \ldots, n\}$, oraz liczba m. Dla każdej pary wierzchołków (i, j) chcemy obliczyć długość najkrótszej ścieżki z i do j przechodzącej przez dokładnie $m - 1$ krawędzi. Zauważmy, że rzeczywiście tyle nam wystarczy, aby rozwiązać wyjściowy problem.

Oznaczmy przez $T(m-1)$ poszukiwaną tabelkę odległości, rozmiaru $n \times n$. Dla utrudnienia, takie tabelki będziemy w dalszej części tekstu nazywać *macierzami*.

Macierz $T(0)$ reprezentuje najkrótsze możliwe ścieżki — jednowierzchołkowe — więc ma całkiem nieskomplikowaną postać: $T(0)_{i,i} = 0$ dla każdego wierzchołka i, a poza tym $T(0)_{i,j} = \infty$ dla $i \neq j$, co wiąże się z tym, że takich ścieżek po prostu nie ma. Macierz $T(1)$ także wypełniamy całkiem łatwo — wystarczy w pole $T(1)_{i,j}$ wpisać wagę krawędzi prowadzącej z wierzchołka i do wierzchołka j. Innymi słowy, $T(1)$ jest *macierzą sąsiedztwa* grafu G. Aby opisać to, co dzieje się dalej, przyjrzyjmy się ogólnej sytuacji: jak obliczyć $T(a + b)$ na podstawie $T(a)$ oraz $T(b)$? Odpowiedź na to pytanie daje następujący lemat.

Lemat 1. Dla dowolnych $a, b \geqslant 0$ oraz $i, j \in V$,

$$T(a+b)_{i,j} = \min\{T(a)_{i,k} + T(b)_{k,j} \ : \ k \in V\}.$$

Dowód: Żądaną równość łatwo uzasadnić, jeśli odpowiednio zinterpretuje się występujące w niej komórki macierzy. Zauważmy mianowicie, że ścieżkę reprezentującą $T(a+b)_{i,j}$ możemy podzielić na dwie części: pierwsze a krawędzi i ostatnie b krawędzi. Niech k_0 będzie wierzchołkiem, w którym spotykają się te części. Wówczas pierwsza

część naszej ścieżki na pewno musi stanowić najkrótszą a-krawędziową ścieżkę z wierzchołka i do wierzchołka k_0, podobnie druga część stanowi najkrótszą ścieżkę z k_0 do j. Stąd

$$T(a+b)_{i,j} = T(a)_{i,k_0} + T(b)_{k_0,j}.$$

Aby zakończyć dowód, wystarczy dodać, że dla wszystkich pozostałych wierzchołków k musi zachodzić

$$T(a)_{i,k} + T(b)_{k,j} \geqslant T(a)_{i,k_0} + T(b)_{k_0,j},$$

gdyż w przeciwnym przypadku sklejając ścieżki odpowiadające $T(a)_{i,k}$ oraz $T(b)_{k,j}$, otrzymalibyśmy ścieżkę z i do j złożoną z $a+b$ krawędzi, ale krótszą niż $T(a+b)_{i,j}$, co nie jest możliwe. To pokazuje, że $T(a+b)_{i,j}$ jest równe owemu minimum, które występuje w tezie lematu. ■

Zauważmy teraz, że jeśli dla dowolnych macierzy M, N rozmiaru $n \times n$ zdefiniujemy działanie $\oplus$ jako[1]:

$$(M \oplus N)_{i,j} \stackrel{\text{def}}{=} \min\{M_{i,k} + N_{k,j} \ : \ k = 1, 2, \ldots, n\},$$

to wzór zawarty w Lemacie 1 możemy zapisać w bardzo prostej postaci:

$$T(a+b) = T(a) \oplus T(b) \quad \text{dla } a, b \geqslant 0. \tag{4}$$

Korzystając z powyższego wzoru, bardzo łatwo skonstruować *jakieś* rozwiązanie naszego zadania: aby obliczyć $T(m-1)$, wystarczy zastosować tenże wzór $m-2$ razy, za każdym razem podstawiając $a = 1$. Przypomnijmy jednak, że parametr m może być naprawdę duży, więc takim rozwiązaniem nie możemy się jeszcze zadowolić.

Okazuje się, że wzór (4) stanowi klucz również do bardziej efektywnej metody obliczania $T(m-1)$. Możemy mianowicie zastosować postępowanie analogiczne do szybkiego potęgowania binarnego[2], czyli korzystające z następujących wzorów rekurencyjnych:

$$T(2k) = T(k) \oplus T(k), \qquad T(2k+1) = T(1) \oplus T(2k).$$

W ten sposób możemy obliczyć $T(m-1)$, wykonując działanie $\oplus$ zaledwie $O(\log m)$ razy. Dla macierzy $n \times n$ koszt czasowy wykonania działania $\oplus$ to $O(n^3)$, ponieważ musimy obliczyć n^2 wartości, z których każda to minimum z n sum par liczb. To oznacza, że umiemy obliczyć $T(m-1)$ w złożoności czasowej $O(n^3 \log m)$.

Rozwiązanie wzorcowe

W rozwiązaniu wzorcowym najpierw, w czasie $O(n \cdot L)$, obliczamy macierz sąsiedztwa grafu prefiksów G, a następnie podstawiamy ją za $T(1)$ i obliczamy $T(m-1)$ za

[1] Czytelnicy zaznajomieni z mnożeniem macierzy mogą zauważyć podobieństwo działania $\oplus$ do mnożenia macierzy — gdyby operację „min” zastąpić przez sumę, a dodawanie przez mnożenie, to otrzymalibyśmy dokładnie standardowe mnożenie macierzy.

[2] Można pokazać, że działanie $\oplus$ (nazywane z ang. *min-plus product*) jest łączne i w związku z tym macierz $T(m-1)$ można tak naprawdę zapisać jako $T(1)^{m-1}$, czyli $(m-1)$-szą potęgę $T(1)$ względem działania $\oplus$.

pomocą opisanej metody binarnej, w czasie $O(n^3 \log m)$. Korzystając z komórek tej macierzy, obliczamy wynik ze wzoru:

$$\min \{T(m-1)_{i,j} + |s_j| \ : \ i,j = 1,2,\ldots,n\}.$$

Złożoność czasowa rozwiązania wzorcowego to $O(n \cdot (n^2 \log m + L))$, a pamięciowa $O(n^2 + L)$. Jego implementacje można znaleźć w plikach `cho.cpp`, `cho1.pas` oraz `cho2.cpp`.

Inne rozwiązania

W tym zadaniu rozwiązania powolne to przede wszystkim nieefektywne implementacje pomysłów zawartych w rozwiązaniu wzorcowym. Wyróżniamy tu rozwiązania obliczające graf prefiksów metodą siłową — złożoność czasowa $O(L^2 + n^3 \log m)$, implementacje w plikach `chos1.cpp` i `chos2.pas`, rozwiązania obliczające $T(m-1)$ za pomocą $(m-2)$-krotnego wykonywania działania $\oplus$ — złożoność czasowa $O(n \cdot L + n^3 \cdot m)$, implementacje w plikach `chos3.cpp` i `chos4.pas`, oraz rozwiązania mające obie te wady — złożoność czasowa $O(L^2 + n^3 \cdot m)$, pliki `chos5.cpp` i `chos6.pas`. Rozwiązania pierwszego typu otrzymywały na zawodach około 80 punktów, natomiast pozostałe od około 30 do 50 punktów.

Wśród rozwiązań niepoprawnych warto zwrócić uwagę na różne heurystyki oparte na konstruowaniu optymalnej ścieżki jedynie za pomocą pojedynczych cykli w grafie G (pliki `chob1.cpp`, `chob2.cpp`, `chob5.cpp`, zdobywały do 40 punktów) oraz implementacje rozwiązania wzorcowego nieużywające zmiennych całkowitych 64-bitowych (`chob3.cpp`, zdobywało 80 punktów).

Testy

Do sprawdzenia rozwiązań użyto 10 grup testów. Poza specyficznymi testami *1d*, *1e*, *6d* i *8d*, poszczególne testy w grupach były następujących typów:

typ a: testy losowe,

typ b: testy, w których kolejne imiona chomików konstruowano, wybierając losowy sufiks poprzedniego imienia i używając go jako prefiksu nowego imienia (np. `abcd`, `bcdefgh`, `ghi`),

typ c: testy podobne do testów typu *b*, w których dodatkowo do wybranych imion doklejano losowe sufiksy, będące prefiksami już skonstruowanych imion, tworząc pewnego rodzaju cykle (np. `abcd`, `cdefgh`, `fghabc`),

typ d: testy wydajnościowe, w których imiona chomików mają specjalną strukturę, wymuszającą wykorzystanie efektywnych metod do obliczania najdłuższych nakładek.

Na następnej stronie znajduje się tabela z podstawowymi statystykami wykorzystanych testów (n — liczba imion chomików, L — suma długości imion, m — dolne ograniczenie na liczbę wystąpień imion chomików w wynikowym słowie).

Nazwa	n	L	m
cho1a.in	7	134	45
cho1b.in	9	126	32
cho1c.in	7	100	37
cho1d.in	2	4	1
cho1e.in	8	64	50
cho2a.in	16	452	87
cho2b.in	17	536	78
cho2c.in	24	596	86
cho3a.in	42	464	650
cho3b.in	35	508	744
cho3c.in	47	873	994
cho4a.in	61	2 394	5 071
cho4b.in	94	2 652	4 781
cho4c.in	55	3 005	4 016
cho5a.in	76	3 652	38 980
cho5b.in	73	3 946	32 521
cho5c.in	79	5 780	30 944
cho6a.in	90	6 664	160 907
cho6b.in	85	8 258	174 302
cho6c.in	104	7 152	168 395
cho6d.in	101	10 201	524 288

Nazwa	n	L	m
cho7a.in	94	13 706	795 662
cho7b.in	93	18 636	985 466
cho7c.in	113	13 960	806 575
cho8a.in	134	16 124	9 161 573
cho8b.in	155	25 282	8 508 759
cho8c.in	92	24 784	9 180 994
cho8d.in	141	19 881	7 654 210
cho9a.in	104	28 488	52 909 188
cho9b.in	114	37 786	98 565 618
cho9c.in	98	37 144	67 504 299
cho9d.in	100	59 850	717 226 983
cho10a.in	7	95 991	889 592 424
cho10b.in	187	54 081	523 267 787
cho10c.in	185	64 190	925 285 703
cho10d.in	100	99 850	811 650 835

Jacek Tomasiewicz	Michał Pilipczuk Jacek Tomasiewicz	Michał Pilipczuk
Treść zadania	Opracowanie	Program

Dostępna pamięć: 64 MB. *OI, Etap II, dzień pierwszy, 10.02.2010*

Klocki

Bajtek dostał na urodziny komplet drewnianych klocków. Klocki są nierozróżnialne, mają kształt jednakowej wielkości sześcianów. Bajtek układa klocki jeden na drugim, tworząc w ten sposób słupki. Zbudował cały rządek takich słupków, jeden obok drugiego, w linii prostej. Słupki mogą mieć różne wysokości.

Tata Bajtka, Bajtazar, zadał mu zagadkę. Podał mu liczbę k i poprosił, żeby tak poprzestawiał klocki, aby jak najwięcej kolejnych słupków miało wysokość przynajmniej k klocków. Przy tym, klocki można przekładać tylko w określony sposób: klocek można wziąć tylko ze słupka, którego wysokość przekracza k, i przełożyć na sąsiedni słupek. Podczas przekładania nie można tworzyć nowych słupków, klocki wolno przekładać tylko pomiędzy już istniejącymi.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie liczby całkowite oddzielone pojedynczym odstępem: n ($1 \leqslant n \leqslant 1\,000\,000$), oznaczająca liczbę słupków, oraz m ($1 \leqslant m \leqslant 50$), oznaczająca liczbę pytań Bajtazara. Słupki są ponumerowane od 1 do n. W drugim wierszu znajduje się n liczb całkowitych $x_1, x_2, \ldots, x_n$ pooddzielanych pojedynczymi odstępami ($1 \leqslant x_i \leqslant 1\,000\,000\,000$). Liczba x_i oznacza wysokość i-tego słupka. W trzecim wierszu znajduje się m liczb całkowitych $k_1, k_2, \ldots, k_m$ pooddzielanych pojedynczymi odstępami ($1 \leqslant k_i \leqslant 1\,000\,000\,000$). Są to kolejne liczby k, dla których należy rozwiązać zagadkę, czyli wyznaczyć największą możliwą liczbę kolejnych słupków o wysokości co najmniej k, jakie można uzyskać za pomocą poprawnych przestawień przy tej wartości parametru k.

Wyjście

Twój program powinien wypisać na standardowe wyjście m liczb całkowitych pooddzielanych pojedynczymi odstępami — i-ta z tych liczb powinna być odpowiedzią na zagadkę dla zadanego zestawu słupków oraz parametru k_i.

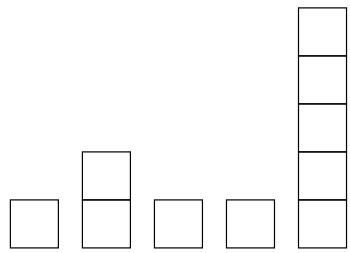
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
5 5 2 1 1 0
```



Rozwiązanie

Uproszczenie zadania

Rozważmy rządek słupków o wysokościach $x_1, x_2, \ldots, x_n$ oraz jedną, konkretną wartość parametru k. *Fragmentem* ciągu x nazwiemy jego spójny podciąg, czyli podciąg złożony z pewnej liczby kolejnych elementów ciągu x. Powiemy, że fragment $x_i, x_{i+1}, \ldots, x_j$ jest *poprawnie rozmieszczony*, jeżeli każdy słupek w tym fragmencie ma wysokość co najmniej k (takie fragmenty chcemy umieć uzyskiwać). Dalej, fragment $x_i, x_{i+1}, \ldots, x_j$ nazwiemy *dobrym*, jeżeli możemy go sprowadzić do fragmentu poprawnie rozmieszczonego, wykonując (wielokrotnie) w ciągu x operację polegającą na przestawieniu pojedynczego klocka ze słupka o wysokości przekraczającej k na sąsiedni słupek (czyli zgodnie z treścią zadania). W tym zadaniu interesuje nas długość najdłuższego dobrego fragmentu ciągu x.

Niestety, nie jest wcale łatwo wyszukiwać w ciągu dobre fragmenty, gdyż podczas przekształcania takich fragmentów w poprawnie rozmieszczone możemy używać klocków pochodzących z bardzo odległych miejsc w ciągu x. Dlatego też wprowadzimy pojęcie *bardzo dobrego* fragmentu, czyli fragmentu, który można przekształcić w poprawnie rozmieszczony za pomocą operacji wykonywanych wyłącznie na słupkach *z tego fragmentu*. Szczęśliwie okazuje się, że aby rozwiązać zadanie, wystarczy zajmować się bardzo dobrymi fragmentami wyjściowego ciągu, patrz poniższe twierdzenie.

Twierdzenie 1. *Najdłuższy dobry fragment ciągu x jest zarazem najdłuższym bardzo dobrym fragmentem tego ciągu.*

Dowód: Każdy bardzo dobry fragment ciągu jest oczywiście także dobrym fragmentem ciągu. Odwrotna zależność niestety nie zachodzi, jednakże pokażemy, że każdy *najdłuższy* dobry fragment ciągu musi być bardzo dobry.

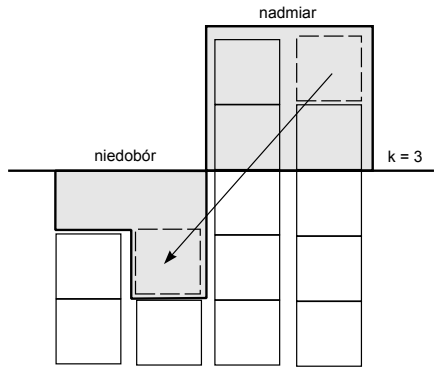
Załóżmy przez sprzeczność, że najdłuższy dobry fragment $x_i, x_{i+1}, \ldots, x_j$ nie jest bardzo dobry. To oznacza, że podczas wykonywania operacji na ciągu x prowadzących do przekształcenia tego fragmentu w poprawnie rozmieszczony, trafił do niego jakiś klocek spoza tego fragmentu. Załóżmy, że był to klocek pochodzący ze słupka x_u dla $u < i$ (przypadek $u > j$ rozpatruje się analogicznie). Twierdzimy, że wówczas po wykonaniu wszystkich operacji wszystkie słupki $x_u, x_{u+1}, \ldots, x_{i-1}$ mają wysokość co najmniej k, co oznacza, że fragment $x_i, x_{i+1}, \ldots, x_j$ bynajmniej nie był najdłuższym dobrym, gdyż dobry jest także fragment $x_u, x_{u+1}, \ldots, x_i, \ldots, x_j$.

No dobrze, a dlaczego wszystkie te słupki mają wysokość co najmniej k? Aby ów klocek ze słupka x_u mógł dostać się do rozważanego fragmentu, z każdego ze słupków $x_u, x_{u+1}, \ldots, x_{i-1}$ musiał zostać co najmniej raz przestawiony na słupek sąsiadujący z nim po prawej. To oznacza, że każdy z tych słupków musiał mieć wówczas wysokość przekraczającą k. Aby zakończyć dowód, wystarczy teraz zauważyć, że w wyniku wykonywania opisanych w zadaniu operacji nie da się obniżyć żadnego słupka o wysokości co najmniej k do wysokości mniejszej niż k. ■

Wprowadźmy dwa następujące parametry związane z fragmentami $x_i, x_{i+1}, \ldots, x_j$ ciągu x:

- liczba brakujących klocków (*niedobór*) — jest to minimalna liczba klocków, jakie musielibyśmy postawić na pewnych słupkach rozważanego fragmentu, aby wszystkie słupki tego fragmentu miały wysokość co najmniej k,
- liczba nadmiarowych klocków (*nadmiar*) — jest to liczba klocków, które znajdują się w rozważanym fragmencie powyżej wysokości k.

Oznaczmy je odpowiednio przez $p(x_i, \ldots, x_j;\ k)$ oraz $q(x_i, \ldots, x_j;\ k)$.



Rys. 1: Rysunek przedstawiający nadmiar q oraz niedobór p klocków. Zobrazowana jest także operacja przeniesienia jednego klocka z nadmiaru do najbliższego słupka o wysokości mniejszej niż k.

Następujące twierdzenie dostarcza wygodnej charakteryzacji bardzo dobrych fragmentów ciągu x, używając wprowadzonych pojęć niedoboru i nadmiaru.

Twierdzenie 2. *Fragment $x_i, x_{i+1}, \ldots, x_j$ ciągu słupków jest bardzo dobry wtedy i tylko wtedy, gdy nadmiar w tym fragmencie jest nie mniejszy od niedoboru, czyli*

$$q(x_i, \ldots, x_j;\ k) \geqslant p(x_i, \ldots, x_j;\ k).$$

Dowód: Jedna strona równoważności zawartej w tezie twierdzenia jest całkiem oczywista: jeśli fragment ciągu jest bardzo dobry, to niedobór nie może w nim przekraczać nadmiaru, gdyż wówczas w żaden sposób nie udałoby się zapełnić w tym fragmencie całego niedoboru.

Udowodnimy teraz implikację w drugą stronę: jeżeli nadmiar (q) jest nie mniejszy od niedoboru (p), to fragment $x_i, x_{i+1}, \ldots, x_j$ jest bardzo dobry. W tym celu pokażemy sekwencję operacji, która sprowadza nasz fragment do poprawnie rozmieszczonego.

Jeżeli $p = 0$, to wiemy, że wszystkie słupki fragmentu mają wysokość co najmniej k, więc nie musimy nic robić. Jeśli nie, to możemy przejść do pewnego stanu wysokości słupków $x'_i, x'_{i+1}, \ldots, x'_j$, w którym p jest mniejsze o 1. Wystarczy, że weźmiemy jakikolwiek słupek, którego wysokość jest większa niż k (wiemy, że taki słupek istnieje, ponieważ $q \geqslant p$), i przeniesiemy jeden klocek do najbliższego słupka, którego wysokość jest mniejsza od k. Ponieważ jest on najbliższy, więc wszystkie słupki pomiędzy nimi mają wysokość co najmniej k, a zatem możemy kolejno przenosić klocek pomiędzy tymi słupkami. W ten sposób zarówno p jak i q zmniejszyły się o 1 (ponieważ jeden

nadmiarowy klocek uzupełnił lukę w brakujących klockach). Warunek $q \geqslant p$ nie zmienił się. Schemat ten możemy powtórzyć p razy, dzięki czemu liczba brakujących klocków spadnie do 0. Oznaczać to będzie, że wszystkie słupki będą miały wysokość co najmniej k. ■

Następująca obserwacja jest wnioskiem z dwóch powyższych twierdzeń.

Wniosek 1. Aby rozwiązać zadanie, wystarczy dla każdego zapytania k znaleźć największą długość fragmentu ciągu wysokości słupków x, w którym nadmiar jest co najmniej taki jak niedobór.

Narzucające się rozwiązania

Po uproszczeniu zadania możemy z łatwością wskazać nieoptymalne rozwiązania naszego problemu.

Pierwsze – wolniejsze

Możemy przejrzeć każdy fragment ciągu słupków i wybrać najdłuższy z tych, w których nadmiar jest nie mniejszy od niedoboru. Wartość nadmiaru i niedoboru obliczamy liniowo dla każdego z fragmentów. Fragmentów jest $\frac{n(n+1)}{2}$, zapytań m, więc daje nam to sumaryczny czas $O(mn^3)$.

Za tak rozwiązane zadanie uzyskiwało się na zawodach 20 punktów. Implementacja znajduje się w pliku `klos1.cpp`.

Drugie – szybsze

Możemy poprawić trochę złożoność powyższego rozwiązania. Zauważmy, że nie interesuje nas dokładna wartość nadmiaru oraz niedoboru. Wystarczyłoby nam znać wartość różnicy nadmiar minus niedobór, czyli „$q - p$”. Jeśli różnica ta byłaby nieujemna, to liczba klocków nadmiarowych byłaby nie mniejsza od liczby klocków brakujących.

Przyjmijmy, że liczby brakujących klocków trzymamy jako liczby ujemne, a nadmiarowych — jako dodatnie. Wtedy dla pojedynczego, i-tego słupka możemy obliczyć wartość $w_i = x_i - k$, która oznacza różnicę „$q - p$” dla tego właśnie słupka.

A jak to wygląda dla dłuższych fragmentów $(x_i, x_{i+1}, \ldots, x_j)$? Otóż wystarczy zauważyć, że każdy ze słupków zawartych w takim fragmencie dostarcza albo tylko nadmiaru, albo tylko niedoboru (albo ma wysokość dokładnie k, a wówczas możemy przyjąć optymistycznie, że ma nadmiar równy 0). To oznacza, że różnicę „$q - p$” dla fragmentu możemy wyznaczyć, sumując nadmiary z nadmiarowych słupków i odejmując od tego sumę niedoborów z pozostałych słupków. Zauważmy jednak, że dokładnie taki sam wynik uzyskamy, sumując po prostu odpowiednie elementy ciągu w:

$$q(x_i, \ldots, x_j;\ k) - p(x_i, \ldots, x_j;\ k) \ = \ w_i + w_{i+1} + \ldots + w_j. \tag{1}$$

Aby móc efektywnie obliczać takie sumy, wyznaczymy ciąg sum częściowych ciągu w. Przypomnijmy, że jest on zdefiniowany jako $a_i = w_1 + w_2 + \ldots + w_i$ i możemy go łatwo obliczyć w koszcie czasowym $O(n)$:

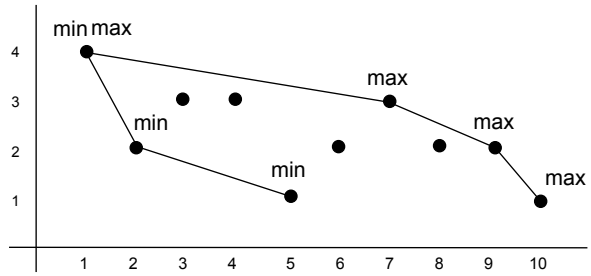
```
1: a[0] := 0;
2: for i := 1 to n do a[i] := a[i − 1] + w[i];
```

W ten oto sposób wartość (1) możemy wyznaczyć w czasie stałym ze wzoru $a_j - a_{i-1}$. Dzięki temu złożoność czasowa rozwiązania spada do $O(mn^2)$. Za takie rozwiązanie można było uzyskać około 40 punktów. Implementacja znajduje się w pliku `klos2.cpp`.

W poszukiwaniu lepszego algorytmu

W powyższym, drugim rozwiązaniu nieefektywnym sprowadziliśmy wyjściowy problem do następującego: Mamy dany ciąg liczb $a_0, a_1, \ldots, a_n$ i poszukujemy dwóch indeksów l i r, dla których $a_r \geqslant a_l$ oraz $r - l$ jest największe możliwe. Odtąd skupimy się już tylko na tym właśnie sformułowaniu problemu.

Definicja 1. *Minimum prefiksowym* w ciągu a nazwiemy taki indeks l, dla którego wartości $a_0, a_1, \ldots, a_{l-1}$ są większe niż a_l. Analogicznie *maksimum sufiksowym* w ciągu a nazwiemy taki indeks r, dla którego wartości $a_{r+1}, a_{r+2}, \ldots, a_n$ są mniejsze niż a_r.



Rys. 2: Rysunek z minimami prefiksowymi (min) i maksimami sufiksowymi (max) w przykładowym ciągu $a = (4,\ 2,\ 3,\ 3,\ 1,\ 2,\ 3,\ 2,\ 2,\ 1)$.

Zauważmy, że optymalne indeksy l i r muszą być odpowiednio minimum prefiksowym i maksimum sufiksowym. Dlaczego? Gdyby l nie było minimum prefiksowym, to można byłoby je zamienić na największe $l' < l$ będące minimum prefiksowym. Ponieważ $a_{l'}$ byłoby nie większe od a_l, więc warunek $a_r \geqslant a_{l'}$ byłby tym bardziej spełniony, a wartość $r - l'$ byłaby jeszcze większa. Analogicznie wygląda sytuacja z r i maksimami sufiksowymi.

Łatwo zauważyć, że wartości elementów ciągu a odpowiadających minimom prefiksowym i maksimom sufiksowym tworzą ciągi malejące. Jeśli nie jest to dla Czytelnika oczywiste, polecamy ponowne przeczytanie definicji tych pojęć.

Zastanówmy się nad wyznaczaniem minimów prefiksowych. Otóż wystarczy liniowo przeglądać ciąg sum częściowych (a_i), trzymając minimum z aktualnie przejrzanych elementów (akt_min). Jeśli przeglądana wartość jest mniejsza niż akt_min, to aktualny indeks jest minimum prefiksowym i za jego pomocą uaktualniamy akt_min.

W przeciwnym razie dla danego elementu nie wykonujemy żadnych czynności. W algorytmie dla maksimów sufiksowych analogicznie przeglądamy ciąg (a_i) od końca, utrzymując aktualne maksimum akt_max.

Oto pseudokod znajdujący minima prefiksowe oraz maksima sufiksowe i umieszczający je odpowiednio w tablicach $pref$ ($pref[1]$ to pierwsze minimum prefiksowe w ciągu) oraz suf ($suf[1]$ to ostatnie maksimum sufiksowe w ciągu):

```
1:  akt_min := ∞;  il_pref := 0;
2:  for i := 0 to n do
3:    if a[i] < akt_min then begin
4:      akt_min := a[i];
5:      il_pref := il_pref + 1;
6:      pref[il_pref] := i;
7:    end
8:
9:  akt_max := −∞;  il_suf := 0;
10: for i := n downto 0 do
11:   if a[i] > akt_max then begin
12:     akt_max := a[i];
13:     il_suf := il_suf + 1;
14:     suf[il_suf] := i;
15:   end
```

Przykładowo, dla ciągu (4, 2, 3, 3, 1, 2, 3, 2, 2, 1) przedstawionego graficznie na rys. 2 tablice minimów prefiksowych i maksimów sufiksowych mają postać:

$$pref = [1,\ 2,\ 5], \quad suf = [10,\ 9,\ 7,\ 1]. \tag{2}$$

Rozwiązanie wzorcowe

Pokażemy teraz, jak w czasie liniowym dla każdego z minimów prefiksowych znaleźć najdalsze, nie mniejsze co do wartości maksimum sufiksowe.

Wykonujemy algorytm stosowy. Ustawiamy maksima sufiksowe w stos, mający na szczycie najwyższy indeks, i po kolei przeglądamy minima prefiksowe od największych indeksów. Dla danego minimum prefiksowego zdejmujemy ze stosu maksima sufiksowe, aż wartość w maksimum na szczycie będzie nie mniejsza niż wartość w przeglądanym minimum. Znalezione maksimum jest najdalszym, dla przeglądanego minimum, elementem spełniającym $a_l \leqslant a_r$ — wszystkie dalsze maksima zostały zdjęte, więc były mniejsze co do wartości niż przeglądane minimum. Jednocześnie przeglądamy minima w porządku rosnących wartości, więc zdjęte ze stosu maksima nigdy już nie będą przydatne.

Poniżej implementacja tego podejścia, w której stos jest ukryty w tablicy suf.

```
1: wsk_suf := 1;
2: wynik := 0;
3: for wsk_pref := il_pref downto 1 do begin
```

```
4:    while a[pref[wsk_pref]] > a[suf[wsk_suf]] do
5:      wsk_suf := wsk_suf + 1;
6:    wynik := max(wynik, suf[wsk_suf] − pref[wsk_pref]);
7:  end
8:  return wynik;
```

Polecamy Czytelnikowi uzasadnienie poprawności tej implementacji. Pomocne jest w tym zauważenie, że wartość w największym minimum prefiksowym nie może przekroczyć wartości w największym maksimum sufiksowym.

Aby oszacować złożoność czasową powyższego algorytmu, należy zwrócić uwagę na łączną liczbę obrotów pętli **while**. Zauważmy mianowicie, że każdy obrót powoduje zwiększenie wartości zmiennej wsk_suf o jeden. Na początku zmienna ta ma wartość 1 i w żadnym momencie nie może przekroczyć il_suf, które jest nie większe niż n. Ponieważ wszystkich operacji poza pętlą **while** jest wykonywanych $O(n)$, więc pokazaliśmy, że złożoność czasowa powyższego algorytmu to właśnie $O(n)$.

Dla naszego przykładowego ciągu (rys. 2 i tablice minimów prefiksowych i maksimów sufiksowych (2)), powyższy algorytm wyznaczy następujące przypisania: dla minimum z pozycji 5 maksimum z pozycji 10, dla minimum z pozycji 2 maksimum z pozycji 9 (ta para daje największą różnicę indeksów), dla minimum z pozycji 1 maksimum także z pozycji 1.

Podsumowanie

Warto podsumować cały algorytm wzorcowy. Przebiega on w następujących krokach:

1. wczytaj tablicę wejściową $[x_1, x_2, \ldots, x_n]$
2. dla kolejnych zapytań k:
3. oblicz tablicę $[a_0, a_1, \ldots, a_n]$
4. oblicz minima prefiksowe dla tablicy $[a_0, a_1, \ldots, a_n]$
5. oblicz maksima sufiksowe dla tablicy $[a_0, a_1, \ldots, a_n]$
6. wykonaj algorytm stosowy
7. wypisz największą odległość wśród par obliczonych w punkcie 6.

Złożoność czasowa algorytmu wynosi $O(nm)$, ponieważ „pętla” w punkcie 2 obraca się m razy, a obliczanie tablicy w punkcie 3 oraz wyznaczanie minimów prefiksowych i maksimów sufiksowych działają w czasie $O(n)$. Pokazaliśmy, że liniowo działa również algorytm stosowy. Złożoność pamięciowa algorytmu wynosi $O(n)$.

Implementacja rozwiązania wzorcowego znajduje się w plikach `klo.cpp` oraz `klo1.pas`.

Rozwiązania trochę wolniejsze

Podczas rozwiązywania zadania niektórzy zawodnicy stwierdzali, że rozwiązania nieznacznie gorsze, czyli o złożoności czasowej $O(mn \log n)$, są wystarczające do uzyskania maksymalnej punktacji, mimo że limity na dane wejściowe zdawały się sugerować wymaganie rozwiązania o złożoności liniowej. Takie rozwiązanie można otrzymać na co najmniej dwa sposoby.

Sortowanie

Tak jak w rozwiązaniu wzorcowym, w liniowej złożoności czasowej sprowadzamy wyjściowy problem do znalezienia najbardziej oddalonych indeksów l, r, dla których $a_r \geqslant a_l$. Następnie sortujemy pary (wartość, indeks): (a_i, i). Będziemy je przeglądać w kolejności od największych a_i, a w przypadku remisu od największych i.

W każdym momencie pamiętamy najbardziej wysunięty na prawo indeks r wśród już przetworzonych. W trakcie przeglądania kolejnych par znajdujemy największą odległość pomiędzy właśnie przeglądanym indeksem l a bieżącym indeksem r. Para l, r indeksów realizujących tę właśnie największą odległość jest właśnie szukaną parą — warunek $a_l \leqslant a_r$ jest zagwarantowany przez kolejność przeglądania par (wartość, indeks), zaś optymalność dzięki temu, że r jest indeksem wysuniętym najbardziej na prawo. Złożoność czasowa tego algorytmu wynosi $O(mn \log n)$.

Za takie rozwiązanie można było otrzymać około 70-80 punktów. Implementacja znajduje się w pliku `klos5.cpp`.

Wyszukiwanie binarne

Tak jak w rozwiązaniu wzorcowym obliczamy minima prefiksowe i maksima sufiksowe. Następnie, zamiast liniowego przeszukiwania dwoma wskaźnikami, dla każdego minimum prefiksowego wyszukujemy binarnie odpowiednie maksimum sufiksowe, korzystając z tego, że minima i maksima tworzą ciągi malejące. Łączna złożoność czasowa $m \cdot n$ takich wyszukiwań binarnych wynosi $O(mn \log n)$.

Za takie rozwiązanie można było otrzymać około 80-90 punktów. Implementacja znajduje się w pliku `klos6.cpp`.

O czym jeszcze należało pamiętać

Podczas rozwiązywania zadania należało pamiętać, aby używać zmiennych całkowitych 64-bitowych, chociażby do przechowywania wyrazów ciągu a. Algorytmy używające tylko zmiennych 32-bitowych uzyskiwały maksymalnie 70 punktów.

Testy

W zestawie były dwa typy testów. W testach pierwszego typu ciąg x składa się z grup wysokich słupków pooddzielanych fragmentami o mniejszej wysokości (testy 1, 2, 3, 4, 5, 9). Testy drugiego typu były generowane poprzez zadanie odpowiednich ciągów

sum częściowych (a_i), tak aby charakteryzowały się żądanej postaci ciągami minimów prefiksowych i maksimów sufiksowych (testy 6, 7, 8, 10).

Nazwa	n	m	Opis
klo1.in	25	26	mały test poprawnościowy
klo2.in	70	25	mały test poprawnościowy
klo3.in	799	50	mały test poprawnościowy
klo4.in	1 543	49	mały test poprawnościowy
klo5.in	6 464	50	średni test poprawnościowo-wydajnościowy
klo6.in	50 003	50	średni test poprawnościowo-wydajnościowy
klo7.in	150 000	50	duży test poprawnościowo-wydajnościowy
klo8.in	299 999	50	duży test poprawnościowo-wydajnościowy
klo9.in	919 200	50	duży test poprawnościowo-wydajnościowy
klo10.in	1 000 000	50	duży test wydajnościowy

Michał Włodarczyk
Treść zadania, Opracowanie

Adam Gawarkiewicz
Program

Dostępna pamięć: 32 MB.

OI, Etap II, dzień drugi, 11.02.2010

Owce

Mieszkańcy Wyżyny Bajtockiej z dziada-pradziada trudnią się hodowlą owiec. Każdy szanujący się hodowca posiada ogrodzone pastwisko w kształcie wielokąta wypukłego[1]*, na którym wypasa swoje owce. Każda szanująca się owca ma swoje ulubione miejsce na pastwisku, gdzie codziennie skubie trawę. Od czasu do czasu owce lubią się też pobawić. Ponieważ owce bawią się parami, każdy pastuch posiada parzystą liczbę owiec, tak żeby każda owca miała zawsze partnera do zabawy.*

Niepokój pastuchów wzbudziło rozporządzenie wydane przez bajtogrodzkiego komisarza ds. rolnictwa. Głosi ono, że od nowego roku owce można wypasać tylko na pastwiskach w kształcie trójkątów. Każdy hodowca, którego pastwisko jest n-kątem dla $n > 3$, musi je podzielić na trójkąty, stawiając na jego terenie $n - 3$ płotów. Pojedynczy płot musi mieć kształt odcinka łączącego dwa wierzchołki wielokąta stanowiącego całe pastwisko. Płoty nie mogą przecinać się poza wierzchołkami wielokąta. Bez spełnienia warunku komisarza hodowcy nie otrzymają dopłaty do hodowli posiadanych owiec.

Bajtazar, który jest hodowcą owiec, musi zadecydować, jak podzielić swoje pastwisko. Głowi się teraz, ile jest w ogóle sposobów, na jakie można podzielić jego pastwisko tak, aby żaden płot nie przechodził przez ulubione miejsce żadnej owcy i wszystkie owce nadal mogły się bawić, czyli aby w każdym trójkącie znajdowały się ulubione miejsca wypasu parzystej liczby owiec. Pomóż mu i napisz odpowiedni program!

Wejście

Pierwszy wiersz standardowego wejścia zawiera trzy liczby całkowite n, k oraz m ($4 \leqslant n \leqslant 600$, $2 \leqslant k \leqslant 20\,000$, $2 \mid k$, $2 \leqslant m \leqslant 20\,000$) pooddzielane pojedynczymi odstępami, oznaczające odpowiednio: liczbę wierzchołków wielokąta stanowiącego pastwisko, liczbę owiec oraz pewną dodatnią liczbę całkowitą m. W kolejnych n wierszach znajdują się po dwie liczby całkowite x_i i y_i ($-15\,000 \leqslant x_i, y_i \leqslant 15\,000$) oddzielone pojedynczym odstępem, oznaczające współrzędne i-tego wierzchołka pastwiska. Wierzchołki te są podane w kolejności zgodnej z ruchem wskazówek zegara. W kolejnych k wierszach znajdują się po dwie liczby całkowite p_j, q_j ($-15\,000 \leqslant p_j, q_j \leqslant 15\,000$) oddzielone pojedynczym odstępem, oznaczające współrzędne ulubionego miejsca j-tej owcy. Ulubione miejsce każdej owcy znajduje się wewnątrz (tj. nie na brzegu) pastwiska.

Wyjście

Twój program powinien wypisać na standardowe wyjście jedną liczbę całkowitą, oznaczającą resztę z dzielenia przez m liczby takich podziałów pastwiska na trójkąty, aby żaden płot nie przechodził przez ulubione miejsce żadnej owcy i w każdym powstałym trójkącie znajdowały się ulubione miejsca parzystej liczby owiec.

[1] Wielokąt wypukły to taki wielokąt prosty (bez samoprzecięć), w którym każdy kąt wewnętrzny jest mniejszy niż 180°.

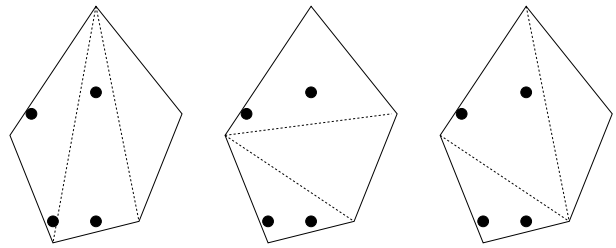
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
3
```



Na rysunku przedstawiono trzy możliwe sposoby podziału pastwiska na trójkąty. Kropkami zaznaczono ulubione miejsca owiec.

Rozwiązanie

Wprowadzenie

W zadaniu mamy do czynienia z problemem triangulacji wielokąta wypukłego — musimy ustalić liczbę jego podziałów na trójkąty, które spełniają określone warunki. Zanim zaczniemy myśleć o naszym problemie, spróbujmy zastanowić się nad czymś znacznie prostszym: ile jest podziałów pastwiska, na którym nie ma owiec?

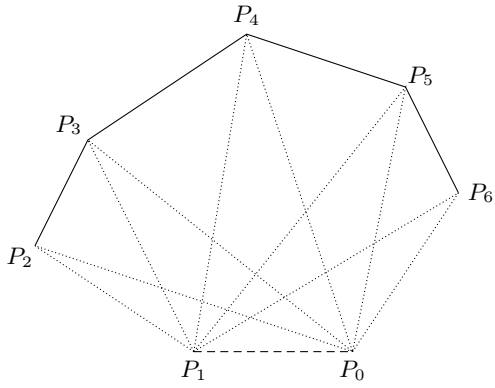
Łatwo zauważyć, że liczba ta nie zależy od kształtu pastwiska, a jedynie od liczby jego boków — wszelkie deformacje pastwiska zachowujące jego wypukłość zachowują także poprawne podziały. Oznaczmy odpowiedź na powyższe pytanie przez A_n, gdzie n to liczba boków wielokąta. Oczywiście $A_n = 1$ dla $n \leqslant 3$ (podział pusty jest w pełni poprawnym podziałem). Dla większych n wyróżniamy jeden bok n-kąta; bok ten musi stanowić bok pewnego trójkąta z podziału. Trzeci wierzchołek takiego trójkąta możemy wybrać na $n-2$ sposobów. Każdy z nich daje podział typu wielokąt–trójkąt–wielokąt, gdzie „wielokąt" może być w szczególności trójkątem lub nawet odcinkiem.

Daje to następujący wzór rekurencyjny:

$$A_n = \sum_{i=2}^{n-1} A_i \cdot A_{n+1-i}.$$

Okazuje się, że $A_n = C_{n-2}$, gdzie C_n to n-ta liczba Catalana[1]. Przy okazji można dostrzec, że przetwarzane w zadaniu liczby mogą rosnąć wykładniczo. Będziemy zatem musieli pamiętać, by w trakcie obliczeń przechowywać tylko reszty z dzielenia

[1]Istnieje wzór jawny na C_n, lecz nie będzie nam potrzebny. Zainteresowanych odsyłamy do [23].



Rys. 1: Możliwe trójkąty dla wyróżnionego boku $[P_0, P_1]$.

modulo m, aby nie wyjść poza zakres typów całkowitych 64-bitowych (jeśli będziemy zawsze wykonywali działania modulo m, to wystarczą nam tak naprawdę typy całkowite 32-bitowe).

Pierwsze rozwiązanie

Sposób, w jaki otrzymaliśmy wzór na A_n, daje się łatwo zaadaptować do potrzeb naszego zadania w pełnej jego ogólności. Tym razem dla pewnego wielokąta zawartego w pastwisku, zamiast sumować po wszystkich możliwych trójkątach o wyróżnionym boku, ograniczamy się do tych, które zawierają parzystą liczbę owiec i których boki nie przecinają owczych pozycji.

Będziemy chcieli oprzeć nasze rozwiązanie o programowanie dynamiczne, więc zastanówmy się, dla których wielokątów musimy znać wynik. Wydawać by się mogło, że dla wszystkich, których wierzchołki są także wierzchołkami pastwiska. Nie byłaby to dobra wiadomość, gdyż jest ich $\Theta(2^n)$. Zauważmy jednak, że wielokąty, których używamy bezpośrednio do obliczenia wyniku dla całego pastwiska, mają tylko jeden bok będący przekątną pastwiska. Co więcej, jeśli przy rozważaniu takich wielokątów ów właśnie bok potraktujemy jako wyróżniony, to dalej wielokąty, które otrzymamy w wyniku podziałów, będą miały tylko jeden bok stanowiący przekątną pastwiska. Wystarczy zatem rozważać tego rodzaju wielokąty. Jest ich oczywiście kwadratowo wiele.

Niech $P_0, P_1, \ldots, P_{n-1}$ oznaczają wierzchołki wyjściowego wielokąta. Oznaczmy przez $S_{x,y}$ wielokąt utworzony przez przecięcie pastwiska prostą przechodzącą przez wierzchołki o numerach x oraz y. Jako że takie wielokąty są zazwyczaj dwa, sprecyzujmy definicję. Jeśli $x < y$, to $S_{x,y}$ składa się z wierzchołków $\{P_x, P_{x+1}, \ldots, P_y\}$. Jeśli zaś $x > y$, to $S_{x,y}$ zawiera wierzchołki $\{P_x, P_{x+1}, \ldots, P_{n-1}, P_0, \ldots, P_y\}$. Przez $T_{x,y}$ oznaczmy liczbę poprawnych triangulacji wielokąta $S_{x,y}$. W poniższym pseudokodzie zaczynamy od obliczenia $T_{x,y}$ dla $y \equiv x+1 \pmod{n}$ oraz dla $y \equiv x+2 \pmod{n}$, tzn. dla boków wielokąta oraz dla trójkątów, a następnie obliczamy dla coraz większych wielokątów. Dla uproszczenia przyjmujemy, że wierzchołki numerowane są cyklicznie,

czyli $P_x = P_{x+n}$, oraz że dotyczy to także odwołań do tablicy, tzn. zapis $T[x][y]$ oznacza tak naprawdę $T[x \bmod n][y \bmod n]$.

```
1:  { obliczenia dla pojedynczych boków i trójkątów }
2:  for x := 0 to n − 1 do begin
3:    T[x][x + 1] := 1;
4:    if trójkąt Δ(P_x, P_{x+1}, P_{x+2}) zawiera parzyście wiele owiec
5:          and odcinek [P_x, P_{x+2}] nie zawiera żadnej owcy then
6:      T[x][x + 2] := 1
7:    else
8:      T[x][x + 2] := 0;
9:  end
10:
11: { główna pętla }
12: for d := 3 to n − 1 do { wielkość rozważanego aktualnie wielokąta to d + 1 }
13:   for x := 0 to n − 1 do
14:     for y := 1 to d − 1 do
15:       if trójkąt Δ(P_x, P_{x+y}, P_{x+d}) zawiera parzyście wiele owiec
16:             and jego boki nie zawierają żadnej owcy then
17:         T[x][x + d] := (T[x][x + d] + T[x][x + y] · T[x + y][x + d]) mod m;
18: return T[0][n − 1];
```

Nie sprecyzowaliśmy jeszcze, jak zliczać owce w trójkątach, lecz można przypuszczać, że wymagać to będzie co najwyżej $O(k)$ operacji na każdy trójkąt. Powyższy algorytm działa więc w czasie $O(n^3k)$ — przeglądamy $O(n^2)$ wielokątów, w każdym wykonujemy $O(nk)$ operacji.

Odrobina geometrii

Należy jeszcze wyjaśnić jedną rzecz: jak rozstrzygnąć, czy owca znajduje się wewnątrz trójkąta?

Najprostszą metodą jest skorzystanie z *iloczynu wektorowego*, który wektorom $\vec{u}, \vec{v}$ zawartym w przestrzeni trójwymiarowej przypisuje prostopadły do nich wektor $\vec{u} \times \vec{v}$ o długości równej podwojonemu polu trójkąta rozpinanego przez $\vec{u}, \vec{v}$. W naszym przypadku, leżące na płaszczyźnie $\vec{u} = \langle u_1, u_2 \rangle, \vec{v} = \langle v_1, v_2 \rangle$ zanurzamy w przestrzeń trójwymiarową i kojarzymy z wektorami $\langle u_1, u_2, 0 \rangle, \langle v_1, v_2, 0 \rangle$. Wtedy iloczyn wektorowy działa następująco:

$$\langle u_1, u_2, 0 \rangle \times \langle v_1, v_2, 0 \rangle = \langle 0, 0, u_1 v_2 - u_2 v_1 \rangle.$$

Interesującym dla nas faktem jest, że wektor $\vec{u} \times \vec{v}$ jest skierowany do góry (tzn. $u_1 v_2 - u_2 v_1 > 0$), jeśli $\vec{v}$ jest „na lewo" od $\vec{u}$. Oczywiście przeciwna nierówność pociąga odwrotne położenie, zaś $u_1 v_2 - u_2 v_1 = 0$ oznacza współliniowość wektorów.

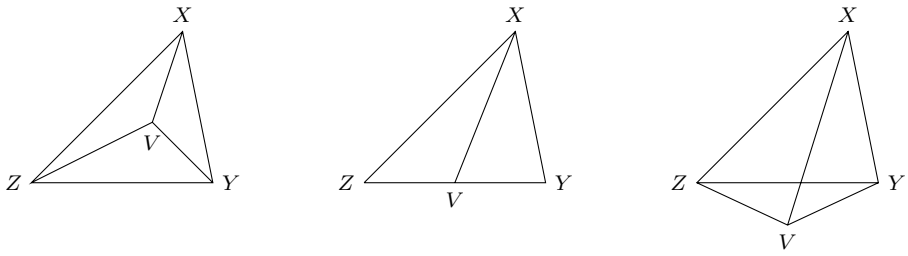
Aby zrozumieć powyższą zależność, połóżmy $\vec{u} = \langle u_1, 0 \rangle$ i niech $u_1 > 0$. Wówczas iloczyn wektorowy $\vec{u} \times \vec{v}$ ma zwrot dodatni wtedy i tylko wtedy, gdy $v_2 > 0$, czyli koniec wektora $\vec{v}$ zaczepionego w 0 znajduje się nad osią X. Ponadto $|\vec{u} \times \vec{v}| = |u_1 v_2|$ to

wysokość trójkąta $(\vec{u}, \vec{v})$ przemnożona przez długość jego podstawy, czyli podwojone pole tego trójkąta. Teraz wystarczy zauważyć, że iloczyn wektorowy nie zmienia się, jeśli oba wektory przekręcimy o dowolny kąt względem 0. Warto sprawdzić to samemu, pamiętając, jak działa obrót o kąt α: $\phi_\alpha(\langle x, y\rangle) = \langle x\cos\alpha - y\sin\alpha,\ x\sin\alpha + y\cos\alpha\rangle$.

Aby zbadać położenie owcy względem trójkąta, możemy zbadać jej położenie względem każdego boku (np. czy jest nie na prawo od żadnego z nich, jeśli będziemy je przeglądać w kierunku przeciwnym do ruchu wskazówek zegara). Można też zauważyć, że punkt V należy do trójkąta $\Delta(X, Y, Z)$, tylko gdy

$$P(X, Y, Z) = P(X, V, Y) + P(Y, V, Z) + P(Z, V, X), \tag{1}$$

gdzie $P(X, Y, Z)$ to pole trójkąta $\Delta(X, Y, Z)$. Pola trójkątów wyznaczamy oczywiście przy pomocy iloczynu wektorowego.



Rys. 2: W dwóch pierwszych przypadkach kryterium (1) da odpowiedź pozytywną, w trzecim zaś nie.

Więcej o iloczynie wektorowym i geometrii obliczeniowej można dowiedzieć się np. z książek [21] oraz [22].

Rozwiązanie wzorcowe

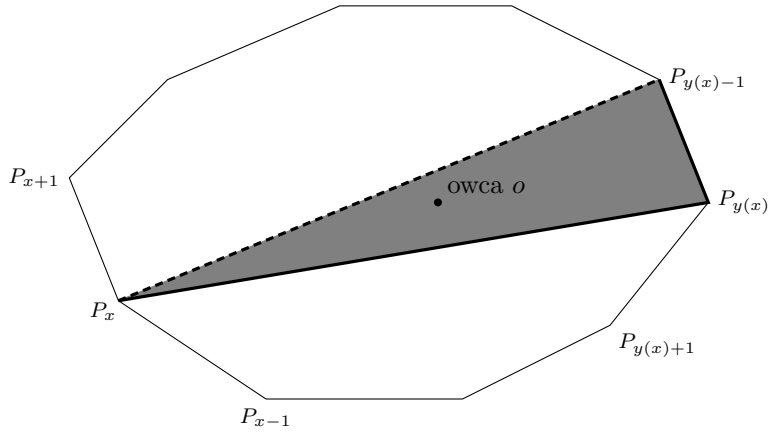
W pierwszym rozwiązaniu widoczne jest miejsce, w którym wiele razy wykonujemy podobne obliczenia. Chodzi o sprawdzanie parzystości liczby owiec w trójkątach i wyszukiwanie „złych" przekątnych. W dalszych rozważaniach trójkąt zawierający parzyście wiele owiec nazwiemy *dobrym* trójkątem. *Dobra* przekątna to zaś taka, która nie zawiera pozycji żadnej owcy.

Zauważmy, że trójkąt $\Delta(P_x, P_y, P_z)$ (gdzie $x < y < z$) jest dobry, gdy jego boki są dobrymi przekątnymi oraz wielokąty $S_{x,y}$, $S_{y,z}$ i $S_{z,x}$ zawierają w sumie parzyście wiele owiec. Istnieją proste sposoby pozwalające w czasie $O(nk \log k)$ znaleźć liczbę owiec we wszystkich wielokątach postaci $S_{x,y}$, polegające na sortowaniu owiec kątowo po kolei względem każdego wierzchołka. Jednak nie będą one dla nas satysfakcjonujące i skorzystamy ze sprytnego rozwiązania działającego w czasie $O(nk)$.

Oznaczmy przez $\Delta_{x,y}$ trójkąt $\Delta(P_x, P_{y-1}, P_y)$, z bokiem $[P_x, P_y]$, ale bez $[P_x, P_{y-1}]$ (przypomnijmy, że o $[P_{y-1}, P_y]$ wiadomo, że nie ma na nim owiec). Dla każdej owcy o i wierzchołka x istnieje dokładnie jeden wierzchołek $y(x)$ taki, że o znajduje się

w $\Delta_{x,y(x)}$. Dla ustalonej owcy obliczymy wartość $y(x)$ dla wszystkich x w łącznym czasie $O(n)$. Kluczowy jest tu następujący fakt, którego formalny dowód pozostawiamy Czytelnikowi.

Fakt 1. *Wierzchołek $P_{y(x+1)}$ nigdy nie leży „na lewo" od wierzchołka $P_{y(x)}$, tzn.* $y(x+1) \in \{y(x), y(x)+1, \ldots, x\}$.



Rys. 3: Ilustracja Faktu 1. Zamalowany trójkąt to $\Delta_{x,y} = \Delta(P_x, P_{y-1}, P_y)$.

Dzięki tej obserwacji wiemy, jak efektywnie zliczać owce w trójkątach. Dla danej owcy o i każdego wierzchołka x znajdujemy wartość $y(x)$ w czasie $O(n)$: najpierw liniowo obliczamy $y(0)$, a dla kolejnych x zwiększamy y (modulo n), aż dojdziemy do $y(x)$. Fakt 1 gwarantuje, że wartość y zmieni się nie więcej niż $2n$ razy.

Dla każdego $\Delta_{x,y}$ wiemy już, ile owiec się w nim znajduje. Możemy teraz w łącznym czasie $O(n^2)$ obliczyć te wartości dla każdego wielokąta $S_{x,y}$ (z brzegiem). W poniższym pseudokodzie spamiętywane są liczby owiec w $\Delta_{x,y}$ i $S_{x,y}$, ale właściwie wystarczy znać tylko ich parzystość. Jednocześnie, przy zliczaniu owiec w trójkątach zostają zapamiętane złe przekątne. Łatwo dostrzec, że każda przekątna zawierająca pozycję pewnej owcy zostanie zauważona. A opracowanie metody sprawdzenia, czy pozycja owcy znajduje się na danej przekątnej, która to metoda jest podobna do tych opisanych w poprzedniej sekcji, pozostawiamy Czytelnikowi.

W pseudokodzie zakładamy, że używane tablice są zainicjowane w następujący sposób: wszystkie komórki tablic ile_owiec_w_trójkącie oraz ile_owiec_w_S są wyzerowane, a wszystkie komórki tablicy dobra_przekątna ustawione na **true**.

```
1: for o := 1 to k do begin
2:   y := 2;
3:   for x := 0 to n − 1 do begin
4:     while owca o nie leży w Δ_{x,y} do
5:       y := (y + 1) mod n;
6:     if owca leży na odcinku [P_x, P_y] then
7:       dobra_przekątna[x][y] := false;
```

```
8:        ile_owiec_w_trójkącie[x][y] := ile_owiec_w_trójkącie[x][y] + 1;
9:     end
10:  end
11:  for x := 0 to n − 1 do
12:     for d := 2 to n − 1 do
13:        ile_owiec_w_S[x][(x + d) mod n] :=
14:              ile_owiec_w_S[x][(x + d − 1) mod n]
15:              +ile_owiec_w_trójkącie[x][(x + d) mod n];
```

Powyższe przetwarzanie niewiele różni się od wstępnych obliczeń w zadaniu *Najazd* z XIII Olimpiady Informatycznej. Warto dodać, że wówczas rozwiązanie wzorcowe używało metod o złożoności czasowej $O(nk \log k)$, a algorytm działający w czasie $O(nk)$ został wymyślony przez zawodników. W książeczce [13] znajduje się wyczerpujące opracowanie zadania *Najazd*, w którym dokładnie opisano także wolniejsze algorytmy zliczania punktów w trójkątach. Sporo miejsca poświęcono również iloczynowi wektorowemu.

Skonstruowane przez nas rozwiązanie działa w czasie $O(nk + n^3)$ (przetwarzanie wstępne oraz przeglądanie n^2 wielokątów) i przechodzi wszystkie testy. Można znaleźć je w plikach `owc.cpp`, `owc0.pas`, ..., `owc4.pas`.

Rozwiązania wolniejsze i błędne

W plikach `owcs1.cpp` oraz `owcs2.pas` zaimplementowano proste rozwiązanie siłowe działające w czasie wykładniczym. W kolejnych parach plików aż do `owcs7.cpp` i `owcs8.pas` można znaleźć kolejne ulepszenia rozwiązania naiwnego aż do algorytmu opisanego na początku opracowania, który zdobywał na zawodach do 60 punktów.

Rozwiązania korzystające z wolniejszego, wymagającego czasu $O(nk \log k)$ zliczania owiec w trójkątach zdobywały co najmniej 80 punktów (często było to ponad 90 punktów, zdarzało się nawet 100). Zaimplementowano je w plikach `owcs[9-14].[cpp|pas]`.

W pliku `owcb1.cpp` zaimplementowano rozwiązanie niesprawdzające, czy dana przekątna jest zła. Otrzymywało ono 10 punktów.

Testy

Wszystkie testy zostały wygenerowane losowo.

Nazwa	n	k	m
owc1.in	14	6	19 387
owc2.in	14	8	13 787
owc3.in	22	12	12 385
owc4.in	50	30	9 987
owc5.in	100	1 000	1 987

Nazwa	n	k	m
owc6.in	200	5 000	12 987
owc7.in	300	5 000	13 581
owc8.in	400	10 000	14 817
owc9.in	500	20 000	19 717
owc10.in	600	20 000	17 679

Jakub Wojtaszczyk
Treść zadania, Opracowanie

Łukasz Bieniasz-Krzywiec
Program

Dostępna pamięć: 64 MB.

OI, Etap II, dzień drugi, 11.02.2010

Teleporty

Król Bajtazar włada całym układem słonecznym, w którym znajduje się n planet. Planety te są ponumerowane od 1 do n. Jego pałac znajduje się na planecie nr 1, zaś jego baza wojskowa na planecie nr 2. Dawno temu Bajtazar wybudował teleport, który w ciągu dwustu pięćdziesięciu minut (czyli nieco ponad czterech godzin) jest w stanie przenieść go z planety nr 1 na planetę nr 2 lub z powrotem.

Obecnie dostępna jest bardziej zaawansowana technologia, która umożliwia tworzenie teleportów zapewniających transport między dwiema planetami w czasie jednej godziny (teleporty takie są z natury dwukierunkowe). Niektóre planety układu są już połączone takimi teleportami. Teleporty te umożliwiają przebycie trasy pomiędzy planetami 1 i 2, na szczęście nie szybciej niż prywatny teleport Bajtazara (co oczywiście zagrażałoby jego bezpieczeństwu).

Obecnie każda para planet, która nie jest jeszcze bezpośrednio połączona teleportem nowej generacji, złożyła wniosek o utworzenie takiego teleportu łączącego te planety. Bajtazar chce zgodzić się na jak największą liczbę takich wniosków, ale tak, aby ciągle nie dało się pokonać trasy z planety nr 1 do planety nr 2 szybciej niż przy pomocy jego prywatnego teleportu. Pomóż mu określić, na ile nowych teleportów może się zgodzić.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie liczby całkowite n oraz m ($2 \leqslant n \leqslant 40\,000$, $0 \leqslant m \leqslant 1\,000\,000$), oddzielone pojedynczym odstępem, oznaczające odpowiednio liczbę planet w królestwie Bajtazara oraz liczbę istniejących teleportów nowej generacji. W kolejnych m wierszach opisane są już istniejące teleporty. W każdym z tych wierszy znajdują się po dwie liczby całkowite a_i i b_i ($1 \leqslant a_i < b_i \leqslant n$) oddzielone pojedynczym odstępem, oznaczające, że istnieje teleport łączący planety a_i i b_i. Każda para liczb występuje na wejściu co najwyżej raz. Możesz założyć, że istniejące teleporty umożliwiają przedostanie się z planety nr 1 na planetę nr 2, ale nie w czasie krótszym niż 250 minut.

Wyjście

Twój program powinien wypisać w pierwszym wierszu standardowego wyjścia jedną liczbę całkowitą, oznaczającą maksymalną liczbę nowych teleportów, na utworzenie których może zgodzić się Bajtazar.

Przykład

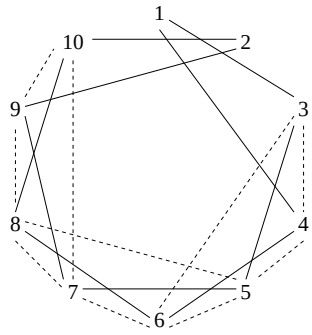
Dla danych wejściowych:

```
10 10
1 3
3 5
5 7
```

```
7 9
2 9
1 4
4 6
6 8
8 10
2 10
```

poprawnym wynikiem jest:

```
10
```



Linią ciągłą na rysunku zaznaczono istniejące teleporty, a przerywaną te, na budowę których Bajtazar może pozwolić.

Rozwiązanie

Wprowadzenie

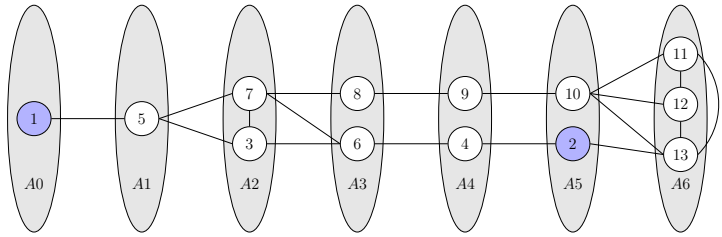
Postawiony przed nami problem można łatwo wyrazić w terminologii grafowej. Mamy dany graf nieskierowany $G = (V, E)$, którego wierzchołki odpowiadają planetom w układzie słonecznym. Dwa wierzchołki są połączone krawędzią, gdy między odpowiadającymi im planetami istnieje teleport nowej generacji. Nasz graf, zgodnie z treścią zadania, ma n wierzchołków ($n \geqslant 2$) i m krawędzi. Graf ten będziemy nazywać grafem *wejściowym*. Wiadomo, że wierzchołki o numerach 1 i 2 są połączone ścieżką, bo istniejące teleporty pozwalają na przedostanie się z planety nr 1 na planetę nr 2. Co więcej, odległość wierzchołków 1 i 2 w naszym grafie (czyli długość najkrótszej łączącej je ścieżki) jest równa co najmniej 5.

Powiemy, że krawędź $e \in E$ jest *niebezpieczna*, jeśli przebiega przez nią jakakolwiek ścieżka długości 4 lub mniejszej łącząca wierzchołki 1 i 2 (takich krawędzi mieć nie chcemy). Graf H nazwiemy *nadgrafem* grafu G, jeśli ma ten sam zbiór wierzchołków oraz każda krawędź grafu G jest też krawędzią grafu H. Poszukujemy zatem nadgrafu H wejściowego grafu G, który nie zawiera niebezpiecznych krawędzi i ma największą możliwą liczbę krawędzi. Dowolny nadgraf H grafu wejściowego, który nie zawiera żadnej niebezpiecznej krawędzi, będziemy nazywać grafem *wynikowym*.

Szybko można spostrzec, że z uwagi na możliwy duży rozmiar danych wejściowych sprawdzenie wszystkich opcji (czyli wszystkich możliwych grafów wynikowych) nie wchodzi w grę. Ponadto, nietrudno dojść do wniosku, że kluczowe jest zrozumienie wpierw, które krawędzie można dodawać, a następnie — jeśli mamy wybór — jak wybierać, by otrzymać największą możliwą liczbę krawędzi w grafie wynikowym. Poniżej sformułujemy obserwacje, które odpowiadają na te pytania. Warto podkreślić, że jakkolwiek wszystkie rozwiązania koniec końców musiały wyglądać dość podobnie, to drogi rozumowania do nich prowadzące mogły wyglądać różnie. Podana niżej droga nie powinna być zatem traktowana jako jedyna możliwa.

Podział na warstwy i analiza zadania

Spójrzmy na dowolny graf wynikowy H. Podzielimy jego wierzchołki na *warstwy* wedle odległości od wierzchołka 1. Przez A_k oznaczmy zbiór tych wierzchołków v, dla których najkrótsza ścieżka z 1 do v ma długość dokładnie k. Oczywiście 2 leży w warstwie A_k dla pewnego skończonego $k \geqslant 5$ i wszystkie warstwy od A_0 do A_k są niepuste (zawierają co najmniej po jednym wierzchołku z najkrótszej ścieżki z 1 do 2). Zauważmy, że jeśli w grafie H istnieje wierzchołek v, który nie jest połączony ścieżką z wierzchołkiem 1, to nie jest on też połączony z wierzchołkiem 2 (bo 1 i 2 łączy ścieżka). Możemy zatem dodać krawędź $1v$ i będzie ona bezpieczna (kwestią optymalności tego wyboru zajmiemy się dalej). Będziemy więc rozważać tylko grafy wynikowe H, w których wszystkie wierzchołki są połączone ścieżką z 1, czyli leżą w jakiejś warstwie.



Rys. 1: Przykładowy graf z podziałem na warstwy.

Zaczniemy od prostej obserwacji, która uzasadnia sens podziału na warstwy.

Obserwacja 1. Rozważmy dwa wierzchołki u i v, które leżą w tej samej warstwie lub w dwóch sąsiednich warstwach, lecz nie są połączone krawędzią. Jeżeli dodamy tę krawędź, to podział grafu H na warstwy nie zmieni się.

Dowód: Istotnie — ani odległość od 1 do u, ani do v nie zmieni się (bo różnica tych odległości już była nie większa niż 1, więc nie zyskujemy na dodaniu tej krawędzi). Odległość od 1 do żadnego innego wierzchołka również się nie zmieni, bo początkowy fragment dowolnej ścieżki zawierającej krawędź uv możemy podmienić na nie dłuższą ścieżkę do v (ew. do u, jeśli szliśmy tą krawędzią w drugą stronę) niezawierający uv. ■

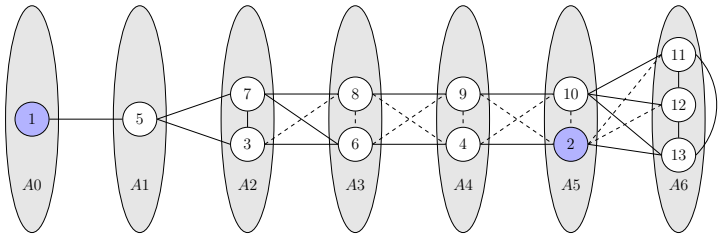
W szczególności, krawędź opisana w Obserwacji 1 jest bezpieczna (bo 2 wciąż leży w warstwie o numerze nie mniejszym niż 5).

Możemy zatem założyć, że w grafie wynikowym każde dwa wierzchołki leżące w tej samej warstwie lub w dwóch sąsiednich warstwach są połączone krawędzią. Z drugiej strony, dwa wierzchołki w warstwach odległych o przynajmniej 2 nie mogą być połączone krawędzią (bo dawałoby to krótszą ścieżkę do wierzchołka z dalszej warstwy). Stąd, znając podział grafu wynikowego na warstwy, możemy obliczyć liczbę

jego krawędzi — będzie to[1]:

$$\frac{1}{2}\sum_{k=0}^{\infty}\big(|A_k|\cdot(|A_{k-1}|+|A_k|+|A_{k+1}|-1)\big). \tag{1}$$

Czynnik 1/2 jest w powyższym wyrażeniu skutkiem tego, że każdą krawędź liczymy dwukrotnie — na jej początku i na jej końcu, zaś odjęcie jedynki w nawiasie odpowiada temu, że wierzchołek nie jest połączony sam ze sobą. Wprowadziliśmy też dla wygody zapisu sztuczne oznaczenie A_{-1}, ten zbiór jest oczywiście pusty. Będziemy teraz szukać takiego nadgrafu grafu wejściowego H, dla którego wartość (1) jest jak największa. Graf, w którym dodaliśmy wszystkie krawędzie, których dodanie umożliwia Obserwacja 1, będziemy nazywać *pełnym grafem wynikowym*.



Rys. 2: Przykładowy, pełny graf wynikowy dla grafu wejściowego z poprzedniego rysunku.

Rozważmy dowolny wierzchołek $v \in A_k$, $k > 1$, w pewnym pełnym grafie wynikowym H. Powiemy, że można go *przesunąć* do warstwy A_{k-1}, jeśli żadna z krawędzi łączących v z wierzchołkami warstwy A_{k+1} nie występowała w grafie wejściowym G. Przesunięcie polega wtedy na usunięciu wszystkich krawędzi łączących v z A_{k+1}, przesunięciu v do A_{k-1} oraz dodaniu krawędzi do każdego wierzchołka z A_{k-2}. Analogicznie definiujemy przesunięcie z A_k do A_{k+1} dla $k > 0$. Przykładowo, na rysunku 2 można przesunąć dowolny z wierzchołków 11, 12, 13 z warstwy A_6 do warstwy A_5 (w sumie to mało ciekawy przykład, ale lepszy rydz niż nic).

Otrzymany w wyniku przesunięcia pojedynczego wierzchołka graf będzie poprawnym grafem wynikowym, o ile nie przesunęliśmy wierzchołka 2 z A_5 do A_4. Faktycznie, utrzymujemy niezmiennik, który mówi, że najkrótsza ścieżka z wierzchołka 1 do dowolnego wierzchołka w warstwie A_k ma długość dokładnie k, a podczas przesuwania nie usuwamy krawędzi pochodzących z grafu wejściowego G.

Rozważmy teraz graf H, który jest jednym z grafów wynikowych o największej możliwej liczbie krawędzi, oraz jego podział na warstwy. Spójrzmy, w jakich warstwach mogą leżeć jego wierzchołki.

[1]Suma (1) formalnie jest nieskończona, jednak zawsze będzie zawierać jedynie skończenie wiele niezerowych składników.

Obserwacja 2. W grafie wynikowym o największej możliwej liczbie krawędzi warstwy A_k dla $k \geqslant 6$ są puste, zaś warstwy A_0 i A_5 składają się z dokładnie jednego wierzchołka (odpowiednio 1 i 2).

Dowód: Załóżmy, że A_k to ostatnia niepusta warstwa dla pewnego $k \geqslant 6$ i $v \in A_k$. Jako że w H istnieje ścieżka z wierzchołka 1 do dowolnego innego wierzchołka oraz $k \geqslant 6$, to warstwy A_{k-1} oraz A_{k-2} są niepuste. Z drugiej strony, warstwa A_{k+1} jest pusta z założenia, zatem możemy przesunąć v z A_k do A_{k-1}. Przy tym przesunięciu nie usuwamy żadnych krawędzi (bo A_{k+1} jest pusta), zaś dodamy przynajmniej jedną krawędź z v do A_{k-2}. Zatem otrzymany graf wynikowy ma więcej krawędzi niż H, co przeczy założeniu o H.

Oczywiście, w warstwie A_0 może być, z definicji, tylko wierzchołek 1. Jeśli w warstwie A_5 jest jakiś wierzchołek oprócz 2, to, podobnie jak powyżej, dowolny z nich możemy przesunąć do A_4. ■

Wykonanie tych przesunięć na grafie z rysunku 2 spowoduje przesunięcie wierzchołków 10, 11, 12 i 13 do warstwy A_4.

Obserwacja 3. Istnieje graf wynikowy o największej możliwej liczbie krawędzi, w którym w warstwie A_1 leżą tylko te wierzchołki, które sąsiadowały z wierzchołkiem 1 w grafie wejściowym G, zaś w A_4 tylko te, które sąsiadowały z 2 w G.

Dowód: Załóżmy, że mamy jakiś „dodatkowy" wierzchołek w warstwie A_1. Jeżeli przesuniemy go do A_2 (możemy, bo z założenia nie sąsiadował on z 1 w grafie G, a 1 to jedyny wierzchołek w warstwie A_0), to stracimy jedną krawędź do A_0, a zyskamy przynajmniej jedną krawędź do A_3 (warstwa A_3 jest niepusta, bo przechodzi przez nią ścieżka z wierzchołka 1 do wierzchołka 2). Zatem graf po takim przesunięciu będzie miał nie mniej krawędzi. Podobnie możemy przesunąć wszystkie „dodatkowe" wierzchołki z A_4 do A_3. ■

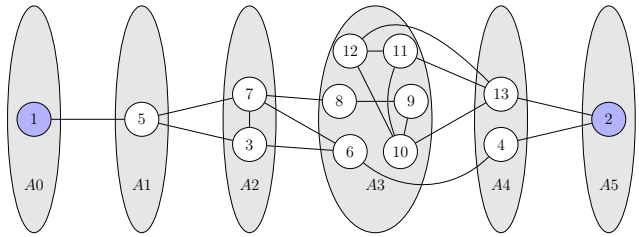
Zastosowanie tych przesunięć na naszym przykładowym grafie przepchnie wierzchołki 9, 10, 11 i 12 do warstwy A_3.

Znamy już więc dokładnie zawartość wszystkich warstw grafu H poza A_2 i A_3. Co więcej wierzchołki, które w G sąsiadowały z jakimkolwiek wierzchołkiem z warstwy A_1 (czyli — na przykładowym rysunku — wierzchołki 7 i 3), muszą leżeć w A_2, zaś wierzchołki, które sąsiadowały z jakimkolwiek wierzchołkiem, który umieścimy, na mocy Obserwacji 3, w warstwie A_4 (czyli, na przykładowym rysunku, wierzchołki 6, 10, 11 i 12, sąsiadujące z 4 lub 13), muszą leżeć w A_3. Niech U oznacza zbiór pozostałych wierzchołków (na przykładowym rysunku $U = \{8, 9\}$).

Obserwacja 4. Niech H będzie grafem z Obserwacji 3. Jeżeli $|A_1| \geqslant |A_4|$, to można założyć, że wierzchołki z U leżą w A_2. W przeciwnym razie można założyć, że $U \subset A_3$.

Dowód jest identyczny jak poprzednio — przeniesienie wierzchołka z A_2 do A_3 usuwa $|A_1|$ krawędzi, zaś dodaje $|A_4|$.

Wiemy zatem, jak będzie wyglądał optymalny graf, i jesteśmy gotowi do implementacji rozwiązania.



Rys. 3: Optymalny podział grafu z rysunku 1 na warstwy.

Rozwiązanie wzorcowe

Zauważmy, że skoro graf G ma do 40 000 wierzchołków, to graf wynikowy H może mieć nawet 800 000 000 krawędzi, czyli więcej, aniżeli chcemy trzymać w pamięci i konstruować. Szczęśliwie, jeżeli wyznaczymy optymalny podział na warstwy, to używając wzoru (1), będziemy w stanie wyznaczyć liczbę krawędzi wynikowego grafu. Zatem zadanie sprowadza się do wyznaczenia optymalnego podziału wierzchołków grafu G na warstwy.

Wiemy, że warstwy powyżej A_5 będą puste, zaś warstwy A_0 i A_5 będą składać się z wierzchołków o numerach odpowiednio 1 i 2. Wierzchołki, które muszą znaleźć się w warstwach A_1, A_2, A_3 i A_4 przed zastosowaniem Obserwacji 4, można wyznaczyć, przeszukując graf wszerz[2]. Pozostaje nam pewna liczba „wolnych" wierzchołków, które możemy przydzielić do warstwy A_2 lub A_3 i które przydzielamy w zależności od liczności warstw A_1 i A_4, zgodnie z Obserwacją 4.

Przykładowy algorytm realizujący to zadanie wygląda następująco:

1: **Algorytm wzorcowy**
2: *Wejście:* Graf nieskierowany G w postaci list sąsiedztwa.
3: *Wyjście:* Rozmiar maksymalnego rozszerzenia grafu G.
4: $n := |V|;\ m := |E|;\ a := 0;\ b := 0;\ c := 0;\ d := 0;$
5: { $t_1[i]$ to odległość wierzchołka i-tego od wierzchołka numer 1 }
6: $t_1 := BFS(G, 1);$
7: { $t_2[i]$ to odległość wierzchołka i-tego od wierzchołka numer 2 }
8: $t_2 := BFS(G, 2);$
9: **for** $i := 1$ **to** n **do begin**
10: **if** $t_1[i] = 1$ **then** $a := a + 1$
11: **else if** $t_1[i] = 2$ **then** $b := b + 1;$
12: **if** $t_2[i] = 1$ **then** $c := c + 1$
13: **else if** $t_2[i] = 2$ **then** $d := d + 1;$
14: **end**
15: $e := n - 1 - a - b - 1 - c - d;$
16: { $a = |A_1|,\ b = |A_2|,\ c = |A_4|,\ d = |A_3|,\ e = |U|$ }

[2]Algorytm BFS, jego opis można znaleźć np. w książce [21].

17: **return** $n(n-1)/2-m-(n-1-a)-a(1+c+d)-b(1+c)-d-e(1+\min(a,c))$;

Złożoność czasowa i pamięciowa tego algorytmu wynosi $O(n+m)$.

Warto zauważyć, że w powyższym algorytmie można zastosować drobne usprawnienie. W procedurze BFS nie musimy przeszukiwać całego grafu. Wystarczy, że zatrzymamy się na wierzchołkach odległych o co najwyżej 2 od źródła.

Implementacje rozwiązania wzorcowego wraz z przedstawionym usprawnieniem znajdują się w plikach `tel.cpp` oraz `tel0.pas`.

Inne rozwiązania

Rozwiązania nieefektywne

Najprostszym siłowym rozwiązaniem tego zadania jest wyznaczenie wszystkich możliwych wyborów par planet, dla których zezwolimy na zbudowanie teleportu. Dla każdego takiego wyboru sprawdzamy, czy jest dopuszczalny (tj. czy nie istnieje zbyt krótka trasa z planety 1 na planetę 2). Jeśli tak, to porównujemy jego liczność z najlepszym dotychczas znalezionym i zapamiętujemy maksimum.

Powyższy algorytm działa w czasie $O(n^2 \cdot 2^{n^2/2})$ i zużywa $O(n^2)$ pamięci, a zatem jest bez szans na rozwiązanie zadania wobec ograniczeń podanych w treści. Można starać się go poprawiać różnymi sposobami stwierdzania, jakich krawędzi na pewno nie warto dodawać (przykładowo, sprawdzać poprawność po każdej dodanej krawędzi, a nie na końcu), natomiast należy oczekiwać, że takie rozwiązania wciąż będą zbyt wolne, a także zużyją zbyt dużo pamięci.

Implementacje tego rozwiązania znajdują się w plikach `tels0.cpp` i `tels1.pas`. W plikach `tels2.cpp` i `tels3.pas` znajdują się rozwiązania wzbogacone o heurystykę pozwalającą na odrzucanie części krawędzi. Rozwiązania te przechodziły 1 na 12 testów.

Ciekawe rozwiązanie bardzo niepoprawne

Na problem można też spojrzeć z nieco innej strony. Odwróćmy pytanie i zastanówmy się, ile najmniej krawędzi trzeba usunąć z grafu pełnego, aby wyeliminować wszystkie ścieżki między wierzchołkami 1 i 2 o długościach mniejszych niż 5.

Niech S będzie zbiorem wszystkich ścieżek prostych między wierzchołkami 1 i 2 o długościach mniejszych niż 5 w grafie pełnym $G_f = (V, E_f)$. Z każdą krawędzią $e \in E_f \setminus E$ wiążemy zbiór $S_e \subseteq S$ wszystkich ścieżek, które przez nią przechodzą. Podzbiory te oczywiście nie muszą być rozłączne. Rozwiązanie polega na znalezieniu najmniejszego (w sensie liczności) zbioru $F \subseteq E_f \setminus E$, takiego, że $S = \bigcup_{e \in F} S_e$. Wówczas maksymalne rozszerzenie grafu G to $E_f \setminus (E \cup F)$.

W ten sposób sprowadziliśmy nasze zadanie do problemu pokrycia zbioru (ang. *Set Cover*), który jest problemem NP-zupełnym. Pierwsze rozwiązanie niepoprawne wykorzystuje to podejście i implementuje następujący algorytm heurystyczny.

1: **Heurystyczne rozwiązanie problemu pokrycia zbioru**
2: *Wejście:* Graf nieskierowany G w postaci macierzy sąsiedztwa.
3: *Wyjście:* Moc maksymalnego rozszerzenia grafu G.

```
4:  Wyznacz zbiór S oraz zbiory S_e dla e ∈ E_f \ E.
5:  f := 0;
6:  while S ≠ ∅ do begin
7:    A := najliczniejszy ze zbiorów S_e;
8:    S := S \ A;
9:    foreach e ∈ E_f \ E do
10:     S_e := S_e \ A;
11:   f := f + 1;
12: end
13: return n(n − 1)/2 − m − f;
```

Można go zaimplementować, tak aby działał w pesymistycznym czasie $O(n^4)$ i zużywał $O(n^3)$ pamięci. Jest to algorytm $H_s = \sum_{k=1}^{s} \frac{1}{k} \leqslant \ln(s) + 1$ aproksymacyjny, gdzie s to moc najliczniejszego ze zbiorów S_e. To oznacza, że wybrane przez niego pokrycie jest co najwyżej $\ln(s) + 1$ razy zbyt liczne. Dowód tego faktu i bardziej szczegółowy opis tego algorytmu Czytelnik może znaleźć np. w książce [21].

Implementacja tego rozwiązania znajduje się w pliku `telb1.cpp`. Nie przechodziło ono żadnego z zestawów testów.

Ulepszone rozwiązanie niepoprawne

Poprzednie rozwiązanie można w łatwy sposób trochę ulepszyć. Zmiana polega na tym, że w pierwszym kroku wybieramy do pokrycia wszystkie zbiory odpowiadające krawędziom, których na pewno nie można dołączyć do grafu (nawet pojedynczo). To usprawnienie nie zmienia pesymistycznej złożoności algorytmu, ale powoduje, że rozwiązanie jest bardzo szybkie dla niektórych klas grafów. Znacznie poprawia również dokładność udzielanych przez algorytm odpowiedzi.

To rozwiązanie przechodziło jeden zestaw testów (przykładowa implementacja w pliku `telb2.cpp`).

Jeszcze inne rozwiązanie niepoprawne

W tym rozwiązaniu do generowanego rozszerzenia nie dodajemy jedynie krawędzi, które (dodane pojedynczo) powodują zmniejszenie odległości między wierzchołkami 1 i 2 poniżej 5. Rozwiązanie to jest niepoprawne, gdyż łatwo może się zdarzyć, że dodanie dwóch krawędzi spowoduje powstanie ścieżki długości 2 pomiędzy wierzchołkami 1 i 2, pomimo iż każda z krawędzi z osobna nie skracała odległości z 1 do 2.

Implementacja tego rozwiązania znajduje się w pliku `telb3.cpp`. Ten program nie przechodził żadnego zestawu testów.

Testy

Programy zgłoszone przez uczestników były sprawdzane na dwunastu zestawach testowych. Każdy zestaw składał się z kilku testów. W poniższym zestawieniu przyjęliśmy oznaczenia zgodne z opisem rozwiązania, tzn.:

- n — liczba wierzchołków grafu,
- m — liczba krawędzi grafu,
- a — liczba wierzchołków w odległości 1 od wierzchołka 1,
- b — liczba wierzchołków w odległości 2 od wierzchołka 1,
- c — liczba wierzchołków w odległości 1 od wierzchołka 2,
- d — liczba wierzchołków w odległości 2 od wierzchołka 2.

Wszystkie testy poza `tel1c.in` i testami z odpowiedzią 0 były losowymi grafami o opisanych wyżej parametrach.

Zestawienie testów:

Nazwa	n	m	a	b	c	d	Opis
tel1a.in	6	5	1	1	1	1	poprawnościowy
tel1b.in	7	5	1	1	1	1	poprawnościowy
tel1c.in	7	6	1	1	1	1	poprawnościowy
tel1d.in	7	7	1	1	1	1	poprawnościowy
tel1e.in	8	9	2	1	2	1	poprawnościowy
tel2a.in	43	101	5	7	1	1	poprawnościowy
tel2b.in	54	300	10	17	3	3	poprawnościowy
tel2c.in	70	231	2	1	3	1	poprawnościowy
tel3a.in	80	79	1	1	1	1	poprawnościowy
tel3b.in	80	700	30	10	20	7	poprawnościowy
tel4a.in	102	3 120	25	25	25	25	wydajnościowy
tel4b.in	102	3 125	25	25	25	25	wydajnościowy; odpowiedź 0
tel4c.in	150	2 000	40	1	41	10	wydajnościowy
tel5a.in	202	12 538	51	49	48	52	wydajnościowy
tel5b.in	202	12 547	51	49	48	52	wydajnościowy; odpowiedź 0
tel5c.in	500	12 345	123	20	123	20	wydajnościowy
tel6a.in	1 000	100 000	1	300	23	77	wydajnościowy
tel6b.in	5 000	300 000	30	10	3 000	29	wydajnościowy
tel7a.in	10 000	200 000	10	3 000	2 300	770	wydajnościowy
tel7b.in	12 000	500 001	10 000	1	2	1	wydajnościowy
tel8a.in	15 000	300 000	3 001	3 003	3 002	2 999	wydajnościowy
tel8b.in	19 999	700 543	4	2	3	1	wydajnościowy
tel9a.in	25 000	600 000	5 007	4 988	1	10 000	wydajnościowy

Nazwa	n	m	a	b	c	d	Opis
tel9b.in	25 001	400 010	3	1	10 000	1	wydajnościowy
tel10a.in	30 000	700 000	5 007	4 988	1	10 000	wydajnościowy
tel10b.in	35 000	800 001	100	2	10 000	99	wydajnościowy
tel11a.in	40 000	1 000 000	7 690	8 030	8 008	7 777	wydajnościowy
tel11b.in	40 000	39 999	1	1	1	1	wydajnościowy
tel11c.in	39 999	999 999	20 000	100	101	99	wydajnościowy
tel12a.in	40 000	1 000 000	123	256	199	203	wydajnościowy
tel12b.in	40 000	1 000 000	5 000	4 000	9 000	303	wydajnościowy

Zawody III stopnia

opracowania zadań

Marian M. Kędzierski
Treść zadania

Marian M. Kędzierski
Piotr Niedźwiedź
Opracowanie

Piotr Niedźwiedź
Program

Dostępna pamięć: 256 MB. *OI, Etap III, dzień próbny, 13.04.2010*

Monotoniczność

Schematem monotoniczności *ciągu liczb całkowitych $a_1, a_2, \ldots, a_n$ nazwiemy ciąg $s_1, s_2, \ldots, s_{n-1}$ złożony ze znaków $<$, $>$ lub $=$. Znak s_i reprezentuje relację pomiędzy liczbami a_i i a_{i+1}. Na przykład, schematem monotoniczności ciągu $2, 4, 3, 3, 5, 3$ jest $<, >, =, <, >$.*

Powiemy, że ciąg liczb $b_1, b_2, \ldots, b_{n+1}$, o schemacie monotoniczności $s_1, s_2, \ldots, s_n$, **realizuje** *pewien schemat monotoniczności $s'_1, s'_2, \ldots, s'_k$, jeżeli dla każdego całkowitego $i = 1, 2, \ldots, n$ zachodzi $s_i = s'_{((i-1) \bmod k)+1}$. Innymi słowy, ciąg $s_1, s_2, \ldots, s_n$ uzyskujemy, powtarzając odpowiednio długo ciąg $s'_1, s'_2, \ldots, s'_k$ i ewentualnie odrzucając kilka ostatnich wyrazów tego powtórzenia. Na przykład, ciąg $2, 4, 3, 3, 5, 3$ realizuje następujące schematy monotoniczności:*

- $<, >, =$
- $<, >, =, <, >$
- $<, >, =, <, >, <, <, =$
- $<, >, =, <, >, =, >, >$

i wiele innych.

Dany jest ciąg liczb całkowitych $a_1, a_2, \ldots, a_n$. Twoim zadaniem jest znalezienie najdłuższego jego podciągu $a_{i_1}, a_{i_2}, \ldots, a_{i_m}$ ($1 \leqslant i_1 < i_2 < \ldots < i_m \leqslant n$) realizującego pewien zadany schemat monotoniczności $s_1, s_2, \ldots, s_k$.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie liczby całkowite n oraz k ($1 \leqslant n \leqslant 20\,000$, $1 \leqslant k \leqslant 100$), oddzielone pojedynczym odstępem i oznaczające odpowiednio długość ciągu (a_i) oraz długość schematu monotoniczności (s_j).

W drugim wierszu znajduje się n liczb całkowitych a_i pooddzielanych pojedynczymi odstępami, oznaczających wyrazy badanego ciągu ($1 \leqslant a_i \leqslant 1\,000\,000$).

W trzecim wierszu znajduje się k znaków s_j postaci `<`, `>` lub `=`, pooddzielanych pojedynczymi odstępami, oznaczających kolejne wyrazy schematu monotoniczności.

Wyjście

W pierwszym wierszu standardowego wyjścia Twój program powinien wypisać jedną liczbę całkowitą m oznaczającą maksymalną długość podciągu ciągu $a_1, a_2, \ldots, a_n$ realizującego schemat monotoniczności $s_1, s_2, \ldots, s_k$.

W drugim wierszu Twój program powinien wypisać dowolny przykład takiego podciągu $a_{i_1}, a_{i_2}, \ldots, a_{i_m}$, oddzielając jego wyrazy pojedynczymi odstępami.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`7 3`	`6`
`2 4 3 1 3 5 3`	`2 4 3 3 5 3`
`< > =`	

Rozwiązanie

Szczególny przypadek

Treść zadania niestety nie należy do najprostszych, więc na początek zajmijmy się analizą pewnego szczególnego przypadku. Schemat „`<`” reprezentuje znany problem szukania najdłuższego podciągu rosnącego zadanego ciągu. My natomiast zajmiemy się schematem „`<>`”, od którego zresztą zaczęła się historia tego zadania. Można ten przypadek rozpatrzyć w znacznie prostszy sposób niż oryginalne zadanie, jednak pomoże on dojść do rozwiązania ogólnej wersji.

Mamy zatem znaleźć najdłuższy podciąg, którego drugi wyraz jest większy od pierwszego, a potem każdy następny jest na zmianę mniejszy i większy od poprzedniego. Dla prostoty notacji, nazwijmy każdy taki podciąg *antymonotonicznym*. W rozwiązaniu wykorzystamy technikę programowania dynamicznego. Chcielibyśmy dla każdej pozycji i w ciągu zapamiętać, jako $tab[i]$, maksymalną długość podciągu antymonotonicznego kończącego się na pozycji i. Jest to najczęściej spotykane podejście w programowaniu dynamicznym, jednak w tym wypadku nie widać, jak do obliczenia wartości $tab[i]$ wykorzystać tylko wartości w tablicy tab pod indeksami od 1 do $i-1$.[1] Zilustrujmy to na przykładzie: jeżeli $a_i = 10$, $a_{i-1} = 20$ oraz $tab[i-1] = 5$, to podciągu reprezentowanego przez $tab[i-1]$ nie możemy przedłużyć o wyraz a_i, ponieważ zakłóciłoby to schemat monotoniczności tego podciągu. Jeżeli a_i miałby być antymonotonicznym przedłużeniem podciągu kończącego się na a_{i-1}, to długość tego podciągu musiałaby być parzysta. Jednak wartość $tab[i-1]$ nie mówi nic o maksymalnej **parzystej** długości podciągu antymonotonicznego kończącego się na a_{i-1}.

Jak poradzić sobie z tym ograniczeniem? Otóż zamiast jednej tablicy tab, użyjemy dwóch:

- $tab1$ — reprezentującej maksymalny podciąg antymonotoniczny *nieparzystej* długości kończący się na zadanej pozycji,
- $tab2$ — reprezentującej maksymalny podciąg antymonotoniczny *parzystej* długości kończący się na zadanej pozycji.

Obliczenie wartości dla pozycji i będzie przebiegało następująco[2]:

$$tab1[i] = \max(1,\ 1 + \max\{tab2[j]\ :\ j < i \text{ oraz } a_j > a_i\}), \tag{1a}$$

$$tab2[i] = 1 + \max\{tab1[j]\ :\ j < i \text{ oraz } a_j < a_i\}. \tag{1b}$$

[1] W rzeczywistości (jak okaże się później) do obliczenia $tab[i]$ wystarczy znajomość wartości $tab[1], tab[2], \ldots, tab[i-1]$; jest to jednak wysoce nieoczywiste.

[2] Przyjmujemy standardowo $\max \emptyset = -\infty$.

Po obliczeniu wartości we wszystkich komórkach obydwu tablic szukana optymalna długość podciągu antymonotonicznego będzie równa maksimum z wszystkich elementów tych tablic. Dodatkowo, aby można było odtworzyć optymalny podciąg, potrzebne będą kolejne dwie tablice, w których zapisywać będziemy liczby j, dla których realizowane są maksima w każdym ze wzorów (1ab).

Łatwo widać, że złożoność czasowa takiego algorytmu to $O(n^2)$.

Przypadek ogólny

Będziemy odtąd zakładać, że zapis „m mod k" oznacza $((m-1) \bmod k)+1$. Innymi słowy, będziemy operować na $\{1, 2, \ldots, k\}$ jako na zbiorze reszt modulo k, co jest wygodne, jeśli wszystkie ciągi w zadaniu są numerowane od jedynki. Dodatkowo, przyjmiemy, że zapis s_m (gdzie s to schemat monotoniczności) oznacza $s_{(m \bmod k)}$.

Aby uogólnić poprzedni algorytm do oryginalnego zadania, użyjemy k tablic: $tab[1..k][1..n]$, a wzór na wartości w ich komórkach będzie następujący:

$$tab[(m+1) \bmod k][i] = 1 + \max\{tab[m][j] \ : \ j < i \text{ oraz } a_j \langle s_m \rangle a_i\}, \qquad (2)$$

z wyjątkiem przypadku $m = k$, w którym dodatkowo bierzemy maksimum z tej wartości i z jedynki. Użyty powyżej zapis $a_j \langle s_m \rangle a_i$ oznacza:

$a_j < a_i$ jeśli $s_m =$ „<", $\quad a_j = a_i$ jeśli $s_m =$ „=", $\quad a_j > a_i$ jeśli $s_m =$ „>".

W tym przypadku obliczamy zawartość już nie dwóch, lecz k tablic, toteż złożoność czasowa wzrasta do $O(n^2k)$. Przykładowa implementacja powyższego algorytmu znajduje się w plikach `mons1.cpp` oraz `mons2.pas`. Na zawodach podobne rozwiązania otrzymywały ok. 40-50 pkt.

Rozwiązanie wzorcowe

Powyższy algorytm jest już tylko o krok od wzorcowego. Wprawny zawodnik szybko zauważy pole do optymalizacji. Otóż jest nim obliczanie maksimum w powyższych wzorach — można do tego wykorzystać drzewa przedziałowe (zwane także licznikowymi). Jest to struktura danych zbudowana nad ciągiem, pozwalająca (w podstawowej wersji) efektywnie obliczać wyniki dowolnego działania łącznego (na przykład sumy, minimum, maksimum) na zadanym spójnym fragmencie tego ciągu oraz równie szybko aktualizować jego wyrazy. Złożoność pamięciowa drzewa przedziałowego jest liniowa względem długości ciągu, zaś każda operacja zajmuje czas logarytmiczny[3].

Zbudowanie drzew przedziałowych bezpośrednio nad każdą z k tablic $tab[m][1..n]$ niestety nie rozwiązuje naszego problemu. Wówczas bylibyśmy w stanie obliczać maksima ze spójnych fragmentów odpowiedniej tablicy $tab[m]$. My jednak potrzebujemy wyznaczać je dla zbiorów indeksów postaci:

- $\{j$ takich, że $j < i$ oraz $a_j < a_i\}$,

[3]Więcej na temat drzew przedziałowych można znaleźć np. w opracowaniu zadania *Latarnia* w tej książeczce oraz w podanych tam odnośnikach.

- $\{j$ takich, że $j < i$ oraz $a_j = a_i\}$,
- $\{j$ takich, że $j < i$ oraz $a_j > a_i\}$.

Zauważmy jednak, że są to spójne podzbiory zbioru $\{1, 2, \ldots, i-1\}$ w sensie porządku na wartościach ciągu a. Należy więc na początku posortować wszystkie liczby a_i, otrzymując ciąg $a_{r_1}, a_{r_2}, \ldots, a_{r_n}$. Wtedy m-te drzewo przedziałowe będzie budowane nad ciągiem $tab[m][r_1], tab[m][r_2], \ldots, tab[m][r_n]$.

Przykład. Dla ciągu $a = (2, 4, 3, 1, 3, 5, 3)$ z przykładu z treści zadania, ciąg r może mieć postać np. $(4, 1, 3, 5, 7, 2, 6)$ albo $(4, 1, 5, 7, 3, 2, 6)$, czyli indeksy odpowiadające równym elementom ciągu a mogą być ustawione w dowolnej kolejności.

Zauważmy, że spójne fragmenty ciągu $r_1, r_2, \ldots, r_n$ odpowiadają spójnym przedziałom wartości w ciągu a_j. Dla każdego elementu a_i ciągu a, będziemy pamiętali indeks p_i jego wystąpienia w ciągu r (tzn. $r_{p_i} = i$), a także dwa indeksy l_i i u_i, oznaczające indeksy skrajnych elementów ciągu r odpowiadających elementom ciągu a równym a_i, tzn.:

$$a_i = a_{r_{l_i}} = a_{r_{l_i+1}} = \ldots = a_{r_{u_i}},$$

natomiast elementy $a_{r_{l_i-1}}$ oraz $a_{r_{u_i+1}}$ nie istnieją lub są odpowiednio mniejszy i większy niż a_i. Dzięki temu będziemy w stanie wyznaczać maksimum w równaniu (2) w czasie $O(\log n)$, obliczając, na drzewie przedziałowym, wartość ze wzoru[4]:

$$tab[(m+1) \bmod k][i] = \begin{cases} 1 + \max\{tab[m][r_j] \ : \ j \in [1,\ l_i - 1]\} & \text{dla } s_m = \text{„<”}, \\ 1 + \max\{tab[m][r_j] \ : \ j \in [u_i + 1,\ n]\} & \text{dla } s_m = \text{„>”}, \\ 1 + \max\{tab[m][r_j] \ : \ j \in [l_i,\ u_i] \setminus \{p_i\}\} & \text{dla } s_m = \text{„=”}. \end{cases}$$

Aby obliczanie maksimów było poprawne, wartości *tab* muszą być inicjowane np. na $-\infty$. W ten sposób egzekwujemy warunek $j < i$ z równania (2).

Również w czasie $O(\log n)$ możemy aktualizować odpowiednie drzewo przedziałowe po nadpisaniu wartości $tab[m][i]$. Przeprowadzona optymalizacja pozwala zatem otrzymać algorytm o sumarycznej złożoności czasowej $O(kn \log n)$ i pamięciowej $O(kn)$. Jest to algorytm wzorcowy, jego przykładowa implementacja znajduje się w plikach `mon.cpp` oraz `mon1.pas`.

Rozwiązanie alternatywne

Uważny Czytelnik mógł dostrzec, że rozwiązanie wzorcowe używa drzew przedziałowych podobnie, jak to ma miejsce w znanym algorytmie wyznaczania najdłuższego podciągu rosnącego. Okazuje się, że bazując na innym, klasycznym algorytmie rozwiązującym ten problem, także można skonstruować dość efektywne rozwiązanie zadania. Przedstawmy najpierw ów algorytm[5]. Przeglądamy kolejne wyrazy ciągu, przechowując przy tym odpowiedzi na pytania postaci: ile wynosi najmniejsza liczba, jaką może

[4] W trzecim z poniższych przypadków należy zauważyć, że $[l_i,\ u_i] \setminus \{p_i\} = [l_i,\ p_i - 1] \cup [p_i + 1,\ u_i]$.

[5] Nieco dokładniejszy opis tego algorytmu można znaleźć w opracowaniu zadania *Egzamin na prawo jazdy* z XIV Olimpiady Informatycznej, patrz książeczka [14].

kończyć się podciąg rosnący długości m (oczywiście chodzi o podciągi dotychczas przejrzanego fragmentu ciągu). Uznajemy, że jeśli nie ma podciągów długości m, to szukana liczba wynosi $+\infty$. Wartości te przechowujemy w tablicy $\ell[1..n]$, wygodnie nam będzie przyjąć $\ell[0] = 0$.

Zastanówmy się, jak aktualizować tę tablicę. Zauważmy, że liczby w niej zawarte tworzą ciąg niemalejący. Każdy koniec podciągu rosnącego długości m jest też bowiem końcem krótszych podciągów rosnących. Aby pewien ciąg rosnący długości m mógł kończyć się liczbą a_i, musi zachodzić $\ell[m-1] < a_i$. Wówczas będziemy chcieli dokonać przypisania $\ell[m] := \min(\ell[m], a_i)$. Aby ta operacja cokolwiek zmieniła, musimy mieć $a_i < \ell[m]$. Podsumowując, wystarczy znaleźć wszystkie takie m, że $\ell[m-1] < a_i < \ell[m]$. Ciąg ℓ jest niemalejący, więc szukana wartość będzie co najwyżej jedna, a można ją znaleźć np. przez wyszukiwanie binarne.

Spróbujmy zaadaptować przedstawiony algorytm do potrzeb zadania. Na razie załóżmy, że w schemacie s występują tylko znaki „<” i „>”. Aby uprościć dalszy opis, podciąg długości m dotychczas przejrzanego fragmentu ciągu a, kończący się wyrazem $a_i = p$ i realizujący schemat s nazwijmy *dobrym m-podciągiem o końcu p*. Jeśli $s_m =$ „<”, to w polu $\ell[m]$ będziemy przechowywać najmniejszy możliwy koniec dobrego m-podciągu, jeśli zaś $s_m =$ „>”, to największy. Pola tablicy ℓ inicjujemy wówczas odpowiednio na $+\infty$ i $-\infty$. Warunek aktualizacji wartości $\ell[m]$ jest podobny jak poprzednio (dla $m = 1$ przyjmujemy pierwszą część za trywialnie prawdziwą):

$$\ell[m-1] \,\langle s_{m-1}\rangle\, a_i \quad \text{oraz} \quad a_i \,\langle s_m\rangle\, \ell[m]. \tag{3}$$

Jego uzasadnienie także nie ulega zmianie. Tym razem liczby w tablicy ℓ nie tworzą ciągu monotonicznego, więc będziemy po prostu sprawdzać wszystkie możliwe wartości m.

Pozostaje jeszcze do rozwiązania problem znaków „=” w schemacie s. Tu użyjemy rozwiązania podobnego jak w algorytmie wzorcowym: dla każdej możliwej wartości v w ciągu a_i i każdego $j \in \{1, 2, \ldots, k\}$, takiego że $s_j =$ „=”, będziemy pamiętać długość najdłuższego dobrego podciągu o końcu v i o długości przystającej do j modulo k. Jeśli na samym początku przenumerujemy wartości ciągu a_i, wystarczy nam do tego zwykła tablica $T[j][v]$.

Teraz aktualizacja naszych struktur dla danego a_i, gdzie $i > 1$, będzie przebiegać następująco.

```
1:  for m := i downto 1 do begin
2:    { war określa, czy istnieje dobry m-podciąg o końcu a_i }
3:    if s_{m-1} ≠ „=” then war := ( ℓ[m-1] ⟨s_{m-1}⟩ a_i )
4:    else war := ( T[(m-1) mod k][a_i] ⩾ m-1 );
5:    if not war then continue;
6:    if (s_m ≠ „=”) and a_i ⟨s_m⟩ ℓ[m] then
7:      ℓ[m] := a_i
8:    else if s_m = „=” then
9:      T[m mod k][a_i] := max(T[m mod k][a_i], m);
10: end
```

Przypomnijmy, że dla $m = 1$ pomijamy sprawdzenie warunku *war*, wiedząc, że jest spełniony. Dokładne uzasadnienie poprawności przedstawionego algorytmu pozosta-

wiamy Czytelnikowi jako sympatyczne ćwiczenie (w szczególności, dlaczego indeks m w pętli **for** musi maleć, a nie może np. rosnąć?).

Umiemy już wyznaczyć długość najdłuższego dobrego podciągu. Jest to większa z dwóch wartości: największego m, dla którego $\ell[m] \neq \pm\infty$, oraz największej z wartości w tablicy T. Zastanówmy się teraz, jak odzyskać szukany podciąg. W rozwiązaniu wzorcowym było to łatwe — wystarczyło w tablicy (nazwijmy ją F) pamiętać indeks j realizujący maksimum we wzorze (2). Tu zapamiętamy dokładnie te same wartości. Aby było to możliwe, musimy jeszcze nieco wzbogacić nasze informacje. Elementy tablic ℓ i T łączy fakt, że informują o istnieniu pewnego dobrego podciągu. Dołączymy zatem do nich dodatkowe tablice przechowujące *indeks* ostatniego wyrazu tego podciągu.

Jak przy ich pomocy obliczyć $F[m \bmod k][i]$? Otóż w przedstawionym wyżej pseudokodzie dla danego i i każdego m stwierdzaliśmy, czy istnieje dobry m-podciąg o końcu a_i. Korzystając z naszych pomocniczych tablic, umiemy wskazać jego poprzedni wyraz. Dzięki temu możemy bezpośrednio wskazać poprzedni wyraz najdłuższego spośród takich podciągów dla m dającego ustaloną resztę z dzielenia przez k, a właśnie tę wartość chcemy zapamiętać w tablicy F.

Złożoność czasowa powyższego algorytmu wynosi $O(n^2)$, zaś pamięciowa — $O(nk)$. W plikach `mon2.cpp` i `mon3.pas` znajdują się programy, które są nieco usprawnionymi wersjami przedstawionego algorytmu. W praktyce są nawet 10 razy szybsze niż rozwiązanie wzorcowe, jako że stała ukryta w notacji O jest tu stosunkowo nieduża.

Rozwiązanie o złożoności $O(n \log n)$

Okazuje się, że nasze zadanie można rozwiązać jeszcze szybciej. Poniższe rozwiązanie jest dosyć zaskakujące: otóż algorytm wzorcowy pozostaje poprawny, jeśli zamiast k tablic $tab[1][1..n], tab[2][1..n], \ldots, tab[k][1..n]$ użyjemy jednej — $f[1..n]$, reprezentującej ich maksimum. Krok obliczenia wartości $f[i]$ wygląda wtedy następująco:

$$f[i] = 1 + \max\{f[j] \text{ dla } j < i \text{ takich, że } a_j \langle s_{f[j]} \rangle a_i\}. \tag{4}$$

Okazuje się, że wartość $f[i]$ reprezentuje wówczas długość najdłuższego podciągu zgodnego ze schematem monotoniczności, kończącego się na pozycji i.

Twierdzenie 1. *Każdy najdłuższy podciąg zadanego ciągu $(a_i)_{i=1}^n$, spełniający zadany schemat monotoniczności s oraz kończący się na pozycji j (czyli na elemencie a_j), nazwiemy* optymalnym podciągiem dla pozycji j. *Niech $f(x)$ oznacza długość najdłuższego optymalnego podciągu dla pozycji $x \in \{1, \ldots, n\}$. Wtedy*

$$f(x) = 1 + \max\{f(y) \ : \ y < x \ \wedge \ a_y \langle s_{f(y)} \rangle a_x\}.$$

Dowód: Dowód przeprowadzimy nie wprost. Załóżmy, że $a_1, a_2, \ldots, a_n$ jest najkrótszym takim ciągiem, dla którego teza twierdzenia nie zachodzi. Wówczas zachodzi ono dla ciągu $a_1, a_2, \ldots, a_{n-1}$ (bo inaczej wybrany ciąg nie byłby najkrótszy). To znaczy, że pozycja n jest jedyną, dla której optymalny podciąg (odpowiadający $f(n)$) nie jest przedłużeniem o pozycję n jakiegoś innego optymalnego podciągu.

Niech $f(n) = k$. Istnieje zatem optymalny podciąg o długości k dla pozycji n. Ponadto, skoro twierdzenie nie zachodzi dla n, to $k > 1$. Niech więc $m < n$ oznacza pozycję $(k-1)$-szego wyrazu tego optymalnego podciągu. Wiemy, że $f(m) \geqslant k-1$, a gdyby $f(m) = k-1$, to twierdzenie zachodziłoby dla pozycji n. Zatem $f(m) > k-1$. Spójrzmy na pewien optymalny podciąg dla pozycji m. Jest on wyznaczony przez ciąg pozycji $b_1, b_2, \ldots, b_{f(m)}$, gdzie $b_{f(m)} = m$. Należy zwrócić uwagę, że $f(b_i) = i$ dla każdego $i = 1, 2, \ldots, m$ (ponieważ twierdzenie zachodzi dla wszystkich $x < n$).

Pokażemy, że można usunąć pewną liczbę wyrazów z końca ciągu b_i, a w zamian dostawić liczbę n, tak aby wyrazy otrzymanego ciągu spełniały schemat monotoniczności oraz aby jego długość była nie mniejsza niż k, co będzie stanowić żądaną sprzeczność.

W tym celu rozważmy kilka przypadków:

- $a_m = a_n$, czyli $s_{k-1} =$ „=”. Nowy ciąg ma postać $b_1, b_2, \ldots, b_{f(m)-1}, n$ — zastępując a_m przez a_n, nie zmieniamy podciągu, więc ten pozostanie dobry.
- $a_m < a_n$, czyli $s_{k-1} =$ „<”. Podzielmy ten przypadek na dwa podprzypadki:
  - $a_{b_{k-1}} < a_n$. Nowy ciąg ma postać $b_1, \ldots, b_{k-1}, n$ — dalej $(k-1)$-szy wyraz jest mniejszy od k-tego.
  - $a_{b_{k-1}} \geqslant a_n$. Tu jest nieco trudniej. Wiemy, że $a_{b_{k-1}} < a_{b_k}$. Stąd wnioskujemy, że musi istnieć takie $w \geqslant k$, że $a_{b_w} > a_{b_{w+1}}$ (czyli $s_w =$ „>”) i $a_{b_w} > a_n$. Wtedy nowy ciąg ma następującą postać: $b_1, \ldots, b_{k-1}, \ldots, b_w, n$.
- $a_m > a_n$, czyli $s_{k-1} =$ „>”. Ten przypadek jest symetryczny względem poprzedniego.

Tak więc twierdzenie zachodzi również dla $x = n$, co jest sprzeczne z założeniem. Otrzymujemy więc, że teza twierdzenia jest prawdziwa dla dowolnego ciągu a, schematu monotoniczności s oraz pozycji x. ■

Jak możemy zaimplementować niniejsze rozwiązanie? Wyznaczanie tablicy f z definicji, z liniowym obliczaniem maksimów, zajmie nam czas $\Theta(n^2)$. Chcielibyśmy znów skorzystać z drzew przedziałowych. Możemy, podobnie jak w rozwiązaniu wzorcowym, zbudować takie drzewo nad tablicą $f[r_i]$. Jednak tym razem nasz problem wcale nie sprowadza się do szukania maksimum w przedziale — to, czy szukamy liczb a_j większych, równych, czy też mniejszych od a_i, zależy od j. Jednak możemy zbudować trzy tablice: $f_<$, $f_=$ i $f_>$ zdefiniowane następująco (dla $q \in \{$„<”, „=”, „>”$\}$):

$$f_q[j] = \begin{cases} f[j] & \text{jeżeli } s_{f[j]} = q \\ 0 & \text{w przeciwnym przypadku.} \end{cases}$$

Dzięki temu możemy nad każdą z tablic $f_q[r_j]$ zbudować drzewo przedziałowe, w których to drzewach będziemy szukać maksimów w spójnych przedziałach. Złożoność czasowa algorytmu wyniesie $O(n \log n)$, zaś pamięciowa $O(n)$. Jest on więc równie szybki, co najlepsze znane rozwiązanie problemu obliczania najdłuższego podciągu rosnącego, który to problem jest, rzecz jasna, szczególnym przypadkiem rozważanego zadania. Implementację tego rozwiązania można znaleźć w pliku `mon5.cpp`.

Rozwiązania powolne

Najprostszym i zarazem najmniej efektywnym rozwiązaniem zadania jest metoda przeszukiwania z nawrotami (brutalna), pesymistycznie wymagająca zbadania wszystkich $\Theta(2^n)$ możliwych podciągów. Takie rozwiązanie zaimplementowano w pliku `monb2.cpp`. Na zawodach otrzymywało ok. 20 punktów.

Innym rozwiązaniem poprawnym, ale zużywającym dużą ilość pamięci, jest algorytm, który nie wykorzystuje przenumerowania indeksów przy budowaniu drzewa przedziałowego i w związku z tym buduje je nad indeksami z przedziału od 1 do $\max_i a_i$. Zaimplementowano je w pliku `monb3.cpp`, otrzymywało ono ok. 30 punktów.

Testy

Zadanie było sprawdzane na 10 zestawach danych testowych. Testy dzielą się na trzy rodzaje: testy *losowe*, testy zawierające tylko jeden rodzaj monotoniczności (*monotoniczne*) oraz testy *skokowe*, czyli składające się z większych, monotonicznych grup.

Nazwa	n	k	Wynik	Opis
mon1a.in	20	5	12	test losowy
mon1b.in	1	1	1	test losowy
mon2.in	50	3	25	test losowy
mon3.in	100	10	100	test losowy
mon4.in	500	30	300	test losowy
mon5a.in	4 000	50	2 000	test losowy
mon5b.in	4 000	30	7	test losowy
mon6a.in	10 000	100	17	test losowy
mon6b.in	3 000	2	1 522	test monotoniczny
mon7a.in	12 000	100	11	test losowy
mon7b.in	5 000	3	2 575	test monotoniczny
mon8a.in	15 000	100	12 346	test losowy
mon8b.in	10 000	2	5 072	test monotoniczny
mon8c.in	15 000	100	17	test skokowy
mon9a.in	18 000	100	15 320	test losowy
mon9b.in	20 000	3	10 074	test monotoniczny
mon9c.in	20 000	100	17	test skokowy
mon10a.in	20 000	100	15 700	test losowy
mon10b.in	20 000	1	10 120	test monotoniczny
mon10c.in	20 000	100	15	test skokowy

Jakub Wojtaszczyk
Treść zadania, Opracowanie

Jakub Łącki
Program

Dostępna pamięć: 32 MB.

OI, Etap III, dzień pierwszy, 14.04.2010

Gra w minima

Asia i Basia poznały niedawno grę w minima, która bardzo im się spodobała. Zasady tej gry są następujące. Na stole leży pewna liczba kart, na których napisane są dodatnie liczby całkowite. Gracze wykonują ruchy na przemian (pierwszy ruch należy do Asi). W swoim ruchu gracz wybiera dowolną liczbę kart (ale co najmniej jedną) i zabiera je ze stołu. Za taki ruch gracz otrzymuje liczbę punktów równą najmniejszej z liczb zapisanych na zabranych przez niego kartach. Gra kończy się, gdy ze stołu zostanie wzięta ostatnia karta. Każdemu graczowi zależy na tym, aby różnica między jego końcowym wynikiem a wynikiem przeciwnika była jak największa.

Asia i Basia szybko spostrzegły, że w tej grze musi istnieć strategia optymalna. Poprosiły Cię więc o napisanie programu, który dla danego zestawu kart wyznaczy, jaki będzie wynik gry, przy założeniu, że obaj gracze grają optymalnie.

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita n ($1 \leqslant n \leqslant 1\,000\,000$), oznaczająca liczbę kart. W drugim wierszu znajduje się n dodatnich liczb całkowitych $k_1, k_2, \ldots, k_n$ ($1 \leqslant k_i \leqslant 10^9$), pooddzielanych pojedynczymi odstępami, oznaczających liczby zapisane na kartach.

Wyjście

Twój program powinien wypisać na standardowe wyjście jeden wiersz zawierający jedną liczbę całkowitą — liczbę punktów, o które Asia wygra z Basią, pod warunkiem, że obie będą grać optymalnie (jeżeli więcej punktów uzyska Basia, to należy wypisać liczbę ujemną).

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`3` `1 3 1`	`2`

Wyjaśnienie do przykładu: *Asia zabiera ze stołu kartę z liczbą 3, za co otrzymuje trzy punkty. Basia bierze obie pozostałe karty, za co otrzymuje jeden punkt. Gra skończy się wynikiem trzy do jednego, a zatem Asia wygra dwoma punktami.*

Rozwiązanie

Wprowadzenie

Zastanówmy się na początek, w jaki sposób opisać dowolną sytuację, która może pojawić się na stole w trakcie rozgrywki. Nietrudno dostrzec, że to, ile punktów ma aktualnie każdy z graczy, nie wpływa na ich strategie działania. Liczy się jedynie aktualny zestaw kart na stole.

Możemy więc dla każdego układu kart K określić wartość $wynik(K)$ jako największą możliwą różnicę punktów pomiędzy graczem wykonującym pierwszy ruch a jego (optymalnie grającym) przeciwnikiem. Na K można patrzeć jak na zbiór liczb, a dokładniej mówiąc na *multizbiór*, czyli strukturę podobną do zbioru, w której dopuszczamy powtórzenia elementów.

W przypadku, gdy na stole jest tylko jedna karta, na której zapisana jest liczba n, gracz nie ma wyboru — musi ją zabrać, tym samym osiągając n punktów przewagi nad przeciwnikiem. Tę własność można zapisać jako $wynik(\{n\}) = n$.

W przypadku, gdy kart jest więcej, wzór na najlepszy możliwy do osiągnięcia wynik przedstawia się następująco:

$$wynik(K) = \max_{S \subseteq K,\ S \neq \emptyset} (\min(S) - wynik(K \setminus S)). \tag{1}$$

W danej sytuacji gracz może zdjąć ze stołu dowolny niepusty podzbiór kart (oznaczony powyżej jako S). Za ten ruch otrzymuje liczbę punktów równą minimum tego multizbioru. Następnie rozpoczyna się kolejka jego przeciwnika, o którym wiemy, że gra optymalnie. Zatem, napotkawszy na stole sytuację $K \setminus S$, zdobędzie on $wynik(K \setminus S)$ punktów przewagi nad aktualnym graczem, co wyjaśnia drugą część powyższego wzoru.

Wzór ten tłumaczy się wprost na algorytm działający w czasie wykładniczym. Używa on techniki programowania dynamicznego i oblicza wszystkie wartości $wynik(K)$, korzystając z przedstawionej przed chwilą własności. Musi przejrzeć wszystkie podzbiory każdego z podzbiorów kart z wejścia, co przy użyciu masek bitowych i kilku sztuczek programistycznych daje się zrealizować w czasie $O(3^n)$. Po szczegóły odsyłamy do programów `grab4.cpp` i `grab5.pas`. Tego typu rozwiązania uzyskiwały na zawodach 20 punktów. Działanie tych programów jest niezdefiniowane dla dużych danych wejściowych, w szczególności czasem udzielają błędnych odpowiedzi, przez co muszą być zakwalifikowane do rozwiązań niepoprawnych.

Rozwiązanie wzorcowe

Rozwiązanie wzorcowe jest prostsze aniżeli rozwiązanie przedstawione powyżej, ale wyprowadzenie go wymaga zrozumienia, jak wyglądają optymalne strategie w grze w minima. Do wniosków przedstawionych poniżej najprościej dojść, rozgrywając (chociażby samemu) kilka partii gry i badając, które posunięcia działają, a które są ewidentnie bez sensu. Zacznijmy od następującej obserwacji:

Obserwacja 1. Istnieje optymalna strategia gry, która w każdym ruchu wybiera multizbiór zawierający kartę o największej wartości spośród dostępnych.

Dowód: Niech K będzie aktualnym układem kart, a S będzie podzbiorem kart wybranych przy tym układzie w pewnej optymalnej strategii. Niech dalej $x = \min(K)$. Jeśli $x \in S$, to nie ma czego dowodzić. Przyjmijmy zatem, że $x \notin S$. Będziemy porównywali sytuację po ruchach polegających na wzięciu odpowiednio S i $S \cup \{x\}$. Zauważmy, że obydwa dają nam tyle samo punktów. Jeżeli na nasz ruch polegający na wzięciu $S \cup \{x\}$ nasz przeciwnik reaguje, biorąc pewien podzbiór T, to po ruchu S nasz przeciwnik jest w stanie doprowadzić do identycznej sytuacji, biorąc $T \cup \{x\}$ (i zyskując identyczną liczbę punktów). Zatem przeciwnik ma po ruchu S strategię, która kończy się nie gorzej niż jego strategia po ruchu $S \cup \{x\}$ — to oznacza, że ruch S nie jest lepszy od ruchu $S \cup \{x\}$. ■

Załóżmy teraz, że mamy przed sobą karty o wartościach $k_1 \leqslant k_2 \leqslant \ldots \leqslant k_m$. Wiemy już, że na pewno wśród kart, które wybierzemy, będzie karta k_m. Jedną z opcji jest, że będzie to jedyna karta, którą weźmiemy. Wtedy przeciwnik będzie wykonywał ruch, mając do dyspozycji karty $k_1, k_2, \ldots, k_{m-1}$, i — z definicji — końcowy wynik będzie równy $k_m - wynik(\{k_1, k_2, \ldots, k_{m-1}\})$. Drugą opcją jest, że wybierzemy jeszcze jakieś karty.

Obserwacja 2. Optymalna spośród strategii, w których w pierwszym ruchu wybieramy przynajmniej dwie karty, w tym k_m, daje dokładnie $wynik(\{k_1, k_2, \ldots, k_{m-1}\})$ punktów.

Dowód: Wpierw pokażmy, jak uzyskać taką liczbę punktów. Rozważmy strategię, która daje tyle punktów w grze z kartami $k_1, k_2, \ldots, k_{m-1}$. Ta strategia każe w pierwszym ruchu wziąć multizbiór S. Zatem w oryginalnej grze możemy w pierwszym ruchu wziąć $S \cup \{k_m\}$, a potem kontynuować wedle naszej strategii, jako że sytuacja po pierwszym ruchu jest identyczna.

Analogicznie rozważmy sytuację, w której w oryginalnej grze pierwszy gracz wziął multizbiór $S \cup \{k_m\}$, przy czym S to niepusty podzbiór $\{k_1, k_2, \ldots, k_{m-1}\}$, i dostał za to $\min(S)$ punktów. Sytuacja jest więc taka sama, jak gdyby w grze na kartach $\{k_1, k_2, \ldots, k_{m-1}\}$ gracz pierwszy wybrał multizbiór S. Gracz drugi może zatem grać tak, żeby łączny wynik gry nie przekroczył $wynik(\{k_1, k_2, \ldots, k_{m-1}\})$. ■

Oczywiście, optymalną strategią dla m kart jest wybranie lepszej z dwóch powyższych strategii. To pozwala nam usprawnić wzór (1). Niech $k_1 \leqslant k_2 \leqslant \ldots \leqslant k_n$ będzie uporządkowanym, wyjściowym układem kart. Wprowadźmy oznaczenie

$$wynik(m) \stackrel{\text{def}}{=} wynik(\{k_1, k_2, \ldots, k_m\}).$$

Na mocy powyższych obserwacji otrzymujemy następującą zależność, która jest podstawą działania rozwiązań wzorcowych:

$$wynik(m) = \max\{wynik(m-1),\ k_m - wynik(m-1)\} \quad \text{dla } m \geqslant 2. \tag{2}$$

Wszystkie wartości $wynik(m)$ można obliczyć w złożoności $O(n)$, jednak konieczne jest uprzednie posortowanie danych wejściowych. Z tego powodu czas działania programu wzorcowego to $O(n \log n)$. Znaleźć go można w plikach `gra.cpp`, `gra1.cpp` (wersja autora) oraz `gra2.pas`.

Warto zauważyć jeszcze, że choć na wejściu znajduje się do miliona liczb nieprzekraczających 10^9, końcowy wynik będzie nie większy niż największa z tych liczb, co można pokazać za pomocą prostego argumentu indukcyjnego. Co więcej, wynik nigdy nie będzie ujemny, gdyż pierwszy gracz może w pierwszym ruchu zdjąć ze stołu wszystkie karty i zagwarantować sobie dodatni rezultat. Dzięki tym spostrzeżeniom zbędne staje się korzystanie z typów całkowitych 64-bitowych.

Rozwiązania niepoprawne

Jak zwykle w grach, można znaleźć wiele niepoprawnych strategii, które na pierwszy rzut oka mogą wydawać się ciekawe — np. „weź wszystkie karty", „zawsze bierz największą", „weź wszystkie karty o największej wartości", czy szereg innych. Przykładowe błędne rozwiązania są zaimplementowane w plikach `gra[b1-b8].[cpp|pas]`. Na zawodach zdobywały one od 0 do 30 punktów.

Testy

Dla każdego testu określamy dwa parametry: n oznacza liczbę kart, a m maksymalną wartość zapisaną na kartach. Przygotowane testy można podzielić na 5 kategorii:

- **losowy:** losowe wartości kart.
- **pierwiastek:** wartości kart rosną od 1 do m proporcjonalnie do pierwiastka z numeru karty.
- **wykładniczy:** wartości kart rosną proporcjonalnie do funkcji wykładniczej od numeru karty.
- **hiperbola:** wartość na i-tej karcie to $\lfloor \frac{m}{i} \rfloor$.
- **grupy:** ustalamy pewną liczbę rozłącznych, oddalonych od siebie przedziałów i następnie losujemy wartości kart należące do tych przedziałów.

Ponadto, w niektórych testach do wartości kart dodawane były małe, losowe liczby całkowite.

Nazwa	n	m	Opis
gra1a.in	1	1	jedna karta
gra1b.in	2	3	dwie karty
gra1c.in	2	5	dwie karty
gra1d.in	6	86	test wygenerowany ręcznie
gra2.in	10	9	pierwiastek
gra3.in	343	322	pierwiastek
gra4.in	3 843	8 582	grupy

Nazwa	n	m	Opis
gra5.in	81 231	999 955 097	losowy
gra6.in	103 243	9 998 828	wykładniczy
gra7.in	213 243	1 000 000	hiperbola
gra8.in	871 843	502 119 250	grupy
gra9.in	1 000 000	999 984 457	wykładniczy
gra10.in	1 000 000	1 000 000 000	hiperbola

Jakub Wojtaszczyk
Treść zadania, Opracowanie

Wojciech Śmietanka
Program

Dostępna pamięć: 256 MB.

OI, Etap III, dzień pierwszy, 14.04.2010

Latarnia

Przy wejściu do budynku, w którym mieszka Bajtazar, Bitocy w środku nocy włączył latarnię. Światło latarni nie daje spać Bajtazarowi. Co prawda, latarnia nie świeci bezpośrednio w okno Bajtazara, ale jej światło odbija się od innych okien i wpada w okno Bajtazara. Bajtazar nie może zasnąć i denerwuje się. Żeby czymś zająć umysł, zamiast się denerwować, Bajtazar wygląda przez okno i zastanawia się, w które jeszcze okna świeci latarnia. Problem ten wciągnął go na tyle, że zapomniał o śnie i zadzwonił do Ciebie, prosząc o pomoc. Znasz Bajtazara nie od dziś i wiesz, że póki nie napiszesz dla niego programu rozwiązującego ten problem, sam(a) też nie pójdziesz spać.

Bajtazar mieszka w budynku B, w którym jest n okien. Latarnia znajduje się na samym dole budynku, na ścianie. Naprzeciwko budynku B stoi budynek C, oddalony o 10 metrów. Jego ściana jest równoległa do ściany budynku Bajtazara. Budynek C ma m okien.

Światło latarni rozchodzi się po liniach prostych, chyba że trafi w okno — wtedy mówimy, że **latarnia świeci w to okno***, a światło latarni odbija się od niego (zgodnie z zasadą „kąt padania równa się kątowi odbicia").*

Na ścianach budynków mamy określone układy współrzędnych — oś X jest pozioma, oś Y jest pionowa, osie na obu ścianach mają zgodne zwroty, a punkty $(0, 0)$ na obu ścianach leżą dokładnie naprzeciwko siebie. Okna to prostokąty położone na ścianach o bokach równoległych do osi układu współrzędnych. Światło nie odbija się od brzegów okien. Ponadto wiadomo, że okna w obrębie jednego budynku mają parami rozłączne wnętrza. Latarnia znajduje się na ścianie budynku B w punkcie $(0, 0)$ i nie znajduje się we wnętrzu ani na brzegu żadnego z okien.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie liczby całkowite n oraz m ($1 \leqslant n, m \leqslant 600$), oddzielone pojedynczym odstępem, oznaczające liczbę okien w pierwszym i w drugim budynku. W kolejnych n wierszach opisane są okna w budynku Bajtazara (budynek B), po jednym w wierszu.

W $(i + 1)$-szym wierszu (dla $1 \leqslant i \leqslant n$) znajdują się cztery liczby całkowite $x_{1,i}$, $y_{1,i}$, $x_{2,i}$, $y_{2,i}$ ($-1\,000 \leqslant x_{1,i} < x_{2,i} \leqslant 1\,000$, $0 \leqslant y_{1,i} < y_{2,i} \leqslant 1\,000$), pooddzielane pojedynczymi odstępami. Taka czwórka oznacza, że i-te okno znajdujące się w budynku B stanowi prostokąt na ścianie budynku, którego lewy dolny i prawy górny róg mają współrzędne $(x_{1,i}, y_{1,i})$ i $(x_{2,i}, y_{2,i})$, wyrażone w metrach.

Następnie w kolejnych m wierszach, w tym samym formacie, opisane są okna budynku C.

Wyjście

Twój program powinien wypisać w pierwszym wierszu standardowego wyjścia liczbę okien budynku B, w które świeci latarnia. Możesz założyć, że dla każdych danych wejściowych będzie istniało co najmniej jedno takie okno.

W drugim wierszu należy wypisać numery tych okien, podane w kolejności rosnącej, poodzielane pojedynczymi odstępami (okna numerujemy od 1).

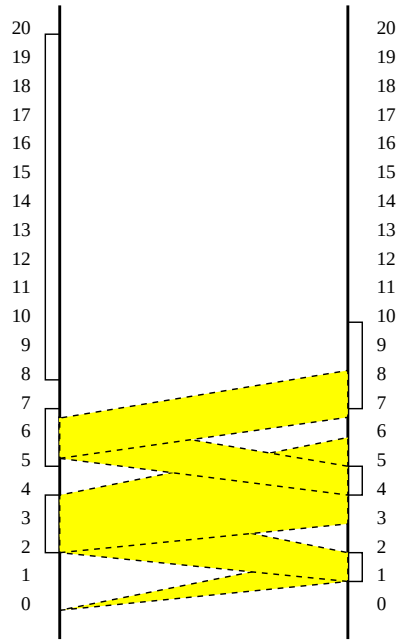
Przykład

Dla danych wejściowych:

```
3 3
-1 2 1 4
-1 5 1 7
-3 8 -2 20
-1 1 1 2
-1 4 1 5
-1 7 1 10
```

poprawnym wynikiem jest:

```
2
1 2
```



Wyjaśnienie do przykładu: *Do pierwszego okna budynku B światło dolatuje, odbiwszy się od pierwszego okna w budynku C (np. odbija się w punkcie $(0, 1.5)$ budynku C i trafia w punkt $(0, 3)$ budynku B). Żeby trafić w drugie okno budynku B, światło latarni musi odbić się trzy razy — ten sam promień trafia w punkt $(0, 4.5)$ w drugim oknie budynku C, a potem w $(0, 6)$ w drugim oknie budynku B. Światło nie trafia w trzecie okno budynku B, tylko obok niego. Należy dodać, że światło latarni oświetla cały budynek C, na rysunku natomiast zaznaczono tylko te promienie świetlne, które po odbiciu oświetlają okna budynku B.*

Rozwiązanie

Wprowadzenie

Powyższe zadanie jest dość trudne (czego dowód stanowi choćby fakt, że podczas zawodów nikt nie otrzymał za nie maksymalnej liczby punktów). Dlatego będziemy do rozwiązania przymierzać się krok po kroku, po drodze analizując i odrzucając różne możliwości.

Oznaczmy $M = \max(n, m)$, czyli M to maksymalna liczba okien na jednej ścianie budynku. Zaczniemy od zrozumienia, jak zachowuje się światło odbite od pojedynczego punktu na budynku C. Zauważmy, że jeśli światło latarni pada na budynek C w punkcie (x, y) zawartym w jakimś oknie, to następnie odbija się i pada na budynek B w punkcie $(2x, 2y)$. Jeśli tam trafi w okno, to odbija się dalej i trafia w budynek C w punkcie $(3x, 3y)$, i ogólnie, odbija się, trafiając w punkty (kx, ky), aż w końcu nie trafi w żadne okno. Aby uprościć nieco notację, dla dowolnego punktu $P = (x, y)$ oraz dowolnej liczby c przez cP oznaczymy punkt (cx, cy); podobnie, dla dowolnego prostokąta R o narożnikach A i B przez cR oznaczymy prostokąt o narożnikach cA i cB.

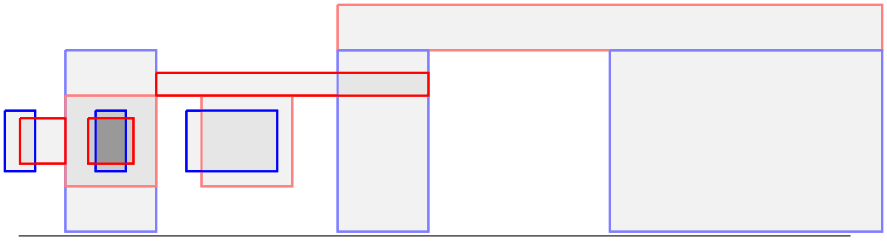
Widzimy teraz, że światło trafiające w okno opisane prostokątem O na budynku C po odbiciu oświetli na budynku B prostokąt $2O$. Jeśli teraz pewne okno na budynku B przecina się z prostokątem $2O$, dając prostokąt O', to po odbiciu od prostokąta O' światło oświetli na budynku C prostokąt $\frac{3}{2}O'$.

To oczywiście nasuwa nam pierwszy pomysł na rozwiązanie — dla kolejnych k pamiętamy prostokąty oświetlone na odpowiednim budynku po k odbiciach, przenosimy je na przeciwległy budynek, przecinamy z oknami, które tam są, dostajemy nowe prostokąty, i tak aż do skutku (tj. aż w końcu, po pewnej liczbie odbić, już nic nie będzie oświetlone). To rozwiązanie jednak zapewne będzie zbyt wolne — jeśli, przykładowo, na budynku B znajduje się 500 okien poziomych o wymiarach 2000×1, a na budynku C jest 600 okien pionowych o wymiarach 1×1000, to po dwóch odbiciach na ścianie budynku C oświetlamy już około 300 000 różnych prostokątów — a to dopiero dwa pierwsze odbicia! Teraz musielibyśmy przeciąć otrzymane prostokąty (po powiększeniu o czynnik 3/2) z 600 prostokątami na ścianie budynku C, i taką operację powtarzać tak długo, jak długo jakikolwiek promień będzie odbijał się od okien. Nawet jeśli uwierzymy, że liczba prostokątów nie będzie przyrastać ponad 300 000, to ciągle — wykonując przecięcia metodą „każdy z każdym" — dostajemy $180\,000\,000\,R$ operacji przecięcia prostokąta z prostokątem, przy czym R to maksymalna liczba odbić. Takie rozwiązanie było zatem zbyt wolne, by uzyskać na zawodach akceptowalną liczbę punktów.

Przy okazji nasuwa się pytanie: jak wiele razy może odbić się promień, zanim trafi w ścianę (nie w okno) budynku? (albo, innymi słowy, jak duże może być R z poprzedniego szacowania?) Otóż zauważmy, że aby odbijać się dalej, w drugim odbiciu musi trafić w okno na budynku B. Tymczasem każdy punkt znajdujący się wewnątrz okna na budynku B ma albo odciętą, albo rzędną różną od zera (bowiem w treści zadania znajduje się informacja, że latarnia nie leży na brzegu ani we wnętrzu żadnego okna, a współrzędne okien są całkowitoliczbowe). Zatem po dwóch odbiciach promień przesunął się przynajmniej o 1 wzdłuż którejś współrzędnej, czyli po $2k$ odbiciach przesunie się wzdłuż którejś współrzędnej przynajmniej o k. Stąd — skoro współrzędne okien są ograniczone co do wartości bezwzględnej przez 1000 — po co najwyżej 2001 odbiciach promień na pewno trafi w ścianę. Ta informacja będzie nam potrzebna do badania jakości konstruowanych rozwiązań.

Przecinanie prostokątów

Teraz zajmiemy się szybszym sprawdzaniem, które okna są oświetlone po k odbiciach.



Rys. 1: Ilustracja poniższego rozumowania dla testu przykładowego i $k = 4$. Najciemniejszym kolorem został zaznaczony obszar, od którego światło odbija się wciąż po czterech odbiciach — jest to prostokąt o narożnikach $(-1, 16/3)$ i $(1, 20/3)$.

Załóżmy, że chcemy wiedzieć, czy pewien punkt P jest oświetlony po k odbiciach. Jest tylko jeden promień, który po k odbiciach ma szansę trafić w ten punkt — to promień, który bezpośrednio na ścianę budynku C trafia w punkcie $\frac{1}{k+1}P$. Aby tam się odbił, w punkcie $\frac{1}{k+1}P$ na ścianie budynku C musi być okno. Dalej, aby promień nie zginął na ścianie budynku B po pierwszym odbiciu, w punkcie $\frac{2}{k+1}P$ na ścianie budynku B także musi być okno. I dalej, okna muszą być w punktach postaci $\frac{2r}{k+1}P$ na ścianie budynku B oraz w punktach postaci $\frac{2r+1}{k+1}P$ na ścianie budynku C.

Zauważmy, że można te wszystkie warunki przenieść na wspólną płaszczyznę. Zamiast mówić, że punkt αP leży na którymś z budynków w oknie O, można równoważnie powiedzieć, że punkt P leży w prostokącie $\frac{1}{\alpha}O$. Wykonajmy zatem następujący rysunek: dla każdej liczby całkowitej r, $1 \leqslant r < (k+1)/2$, oraz dla każdego okna O na budynku B rysujemy prostokąt $\frac{k+1}{2r}O$. Podobnie, dla każdego r', $0 \leqslant r' < k/2$, oraz każdego okna O' na budynku C rysujemy prostokąt $\frac{k+1}{2r'+1}O'$. Punkty, które są zawarte w którymś z prostokątów dla każdego r oraz dla każdego r', to dokładnie te punkty, które są oświetlone po k odbiciach (na odpowiednim budynku — w zależności od parzystości liczby k). Zauważmy jeszcze, że skoro okna na ścianie każdego z budynków były rozłączne, to prostokąty rysowane dla ustalonego r lub r' są rozłączne — a zatem zamiast sprawdzać, które punkty są zawarte w jakimś prostokącie dla każdego r i r', wystarczy sprawdzić, które punkty są zawarte w dokładnie k prostokątach. Dodajmy jeszcze, że chcemy tylko wiedzieć, które okna na budynku B będą oświetlone — czyli, w szczególności, wystarczy rozpatrywać problem tylko dla nieparzystych liczb k, gdyż dla parzystych k promień po k odbiciach trafia w budynek C.

Zamiatanie

Mamy zatem do rozwiązania następujący podproblem: dane jest k zbiorów prostokątów (każdy z tych zbiorów to przeskalowane, ze wspólną skalą, okna z któregoś z budynków). Każdy zbiór składa się z M parami rozłącznych prostokątów. Mamy dodatkowo M parami rozłącznych wyróżnionych prostokątów — to są okna na budynku B. Chcemy dowiedzieć się, które z tych M prostokątów przecinają się w jakimś punkcie z przynajmniej k innymi prostokątami. W naszym wypadku $M \leqslant 600$ oraz $k \leqslant 2000$.

Zauważmy, że jeśli znajdziemy pewien punkt, w którym przecina się dokładnie $k+1$ prostokątów, to każdy z nich pochodzi z innego zbioru, a zatem jest wśród nich wyróżniony prostokąt. Możemy zatem usunąć ten wyróżniony prostokąt ze zbioru (zaznaczając, że jest on częścią wyniku) i ponownie szukać punktu, w którym przecina się $k+1$ prostokątów. Jako że wyróżnione prostokąty nie przecinają się wzajemnie, usuwając już trafione prostokąty, nie psujemy wyniku dla tych, których jeszcze nie znaleźliśmy.

Chcemy zatem znaleźć punkt, w którym przecina się przynajmniej $k+1$ prostokątów. Do tego możemy użyć techniki znanej jako *zamiatanie*[1]. Wyobraźmy sobie prostą poziomą (będziemy ją nazywać *miotłą*) leżącą poniżej wszystkich prostokątów, którą następnie będziemy powoli przesuwać w górę. W każdym momencie chcemy dla każdego punktu miotły wiedzieć, ile prostokątów przecina się w tym punkcie. Za każdym razem, gdy w pewnym punkcie miotły przecina się $k+1$ prostokątów, znajdujemy wśród nich wyróżniony i usuwamy go (zmniejszając odpowiednie wartości na miotle oraz dodając usunięty prostokąt do rozwiązania), a następnie kontynuujemy zamiatanie. Technika zamiatania jest bardzo pożyteczna w wielu zadaniach geometrycznych, dlatego też dokładnie omówimy jej zastosowanie w tym zadaniu.

Prosta struktura danych

Przy zamiataniu zazwyczaj przechowujemy pewną strukturę danych, która opisuje obecny stan miotły (czyli, w naszym wypadku, ile prostokątów przecina który punkt miotły), oraz *zdarzenia*, czyli informacje o tym, w których momentach stan miotły będzie się zmieniał.

Wpierw poświęćmy chwilę uwagi strukturze danych. Przez *wartość* punktu będziemy rozumieć liczbę prostokątów, we wnętrzu których ten punkt jest zawarty. Oczywiście, skoro punktów na prostej jest nieskończenie wiele, nie chcemy przechowywać wartości wszystkich punktów. Zauważmy jednak, że wartość ta może zmieniać się tylko w punktach, których odcięta jest równa odciętej jakiegoś boku prostokąta — te punkty miotły nazwiemy *punktami kluczowymi*. Wystarczy zatem, dla ustalonego odcinka między dwoma sąsiednimi punktami kluczowymi, pamiętać wartość dowolnego punktu z tego odcinka, przykładowo środka (tu uwaga: w ogólnym przypadku należałoby jeszcze pamiętać wartości w samych punktach kluczowych, ale w tym konkretnym zadaniu przecinamy tylko wnętrza prostokątów, przez co w żadnym punkcie kluczowym nie przecina się $k+1$ prostokątów). Możemy, wobec tego, zacząć od posortowania wszystkich punktów kluczowych i wyznaczenia środków odcinków między sąsiednimi punktami — te środki odcinków nazwiemy *punktami znaczącymi*. Najprostszą strukturą danych, której możemy tu użyć, jest zwykła tablica, w której na i-tej pozycji trzymamy wartość i-tego punktu znaczącego. Ta struktura okaże się niedługo zbyt powolna, ale na chwilę obecną zależy nam na zrozumieniu samej idei zamiatania — gdy już skonstruujemy algorytm zamiatający, będziemy poprawiać tę strukturę.

[1] O zamiataniu można poczytać w książkach o geometrii obliczeniowej, np. [22], a także posłuchać na stronie `http://was.zaa.mimuw.edu.pl?q=node/37`. Technika ta była wykorzystywana w rozwiązaniach wielu zadań olimpijskich, jak np. *Straż pożarna* z XVI Olimpiady Informatycznej [16] lub *Trójkąty* z XV Olimpiady Informatycznej [15].

Zdarzenia i miotła

Na początku nasza struktura danych zawiera same zera — faktycznie, dla dostatecznie małych y (np. ujemnych) miotła nie przecina żadnych prostokątów. Sytuacja na miotle zmienia się dla wartości y, które są rzędnymi pewnego wierzchołka prostokąta. Jeżeli na wysokości y znajduje się dolny bok pewnego prostokąta o narożnikach (x_1, y_1) i (x_2, y_2), to wartości punktów znaczących zawartych na odcinku (x_1, x_2) zwiększamy o 1. Jeśli na wysokości y znajduje się górny bok, to wartości punktów na odcinku (x_1, x_2) zmniejszamy o 1.

Tworzymy zatem *zdarzenia* — czwórki (y, x_1, x_2, z), gdzie y to wysokość, na której zdarzenie ma miejsce, x_1 i x_2 to końce odcinka, którego zdarzenie dotyczy, zaś z to znak, czyli plus lub minus jedynka, w zależności od typu zdarzenia. Nasz algorytm zamiatający wygląda następująco:

```
1:  Algorytm miotły
2:  Wejście: k+1 zbiorów po M rozłącznych prostokątów każdy.
3:  Wyjście: w tablicy T na pozycji i znajduje się jedynka, jeśli i-ty wyróżniony
4:           prostokąt zawiera punkt, w którym przecina się k+1 prostokątów.
5:    for i := 1 to M do
6:      for j := 0 to k do begin
7:        kluczowe.wstaw(x1[i][j]);
8:        { x1[i][j] to odpowiednia współrzędna i-tego prostokąta z j-tego zbioru }
9:        kluczowe.wstaw(x2[i][j]);
10:     end
11:   kluczowe.sortuj();
12:   kluczowe.usunPowtorzenia();
13:   for x := 1 to rozmiar(kluczowe) - 1 do
14:     znaczace[x] := (kluczowe[x] + kluczowe[x + 1])/2;
15:   for x := 1 to rozmiar(znaczace) do wartosc[i] := 0;
16:   { Pole wartosc[i] będzie przechowywać wartość punktu znaczace[i]. }
17:   for i := 1 to M do
18:     for j := 0 to k do begin
19:       zdarzenia.wstaw(y1[i][j], x1[i][j], x2[i][j], +1);
20:       zdarzenia.wstaw(y2[i][j], x1[i][j], x2[i][j], −1);
21:     end
22:   zdarzenia.sortuj();  { leksykograficznie po współrzędnych }
23:   for j := 1 to rozmiar(zdarzenia) do begin
24:     for i := 1 to rozmiar(znaczace) do
25:       if (znaczace[i] > zdarzenia[j].x1) and
26:          (znaczace[i] < zdarzenia[j].x2) then
27:       begin
28:         if zdarzenia[j].z = +1 then wartosc[i] := wartosc[i] + 1
29:         else wartosc[i] := wartosc[i] − 1;
30:       end
31:     for i := 1 to rozmiar(znaczace) do
32:       if wartosc[i] = k + 1 then begin
33:         O := prostokąt wyróżniony zawierający punkt znaczace[i];
```

```
34:          T[O.numer] := 1;
35:          for z := 1 to rozmiar(znaczace) do
36:            if (O.x1 < znaczace[z]) and (znaczace[z] < O.x2) then
37:              wartosc[z] := wartosc[z] − 1;
38:          Usuń z listy zdarzeń zdarzenie związane z górnym brzegiem O;
39:        end
40:    end
41:    return T;
```

W powyższym algorytmie nie jest sprecyzowany sposób znajdowania wyróżnionego prostokąta zawierającego dany punkt oraz usuwania związanych z nim zdarzeń. To jednak nie jest istotnym problemem. Aby usuwać zdarzenia, wystarczy, przykładowo, dla każdego zdarzenia pamiętać, od którego prostokąta pochodzi, i przed obsłużeniem sprawdzać, czy ten prostokąt nie został już usunięty. Ze znajdowaniem prostokątów jest jeszcze prościej: otóż wystarczy nachalnie przejrzeć wszystkie, których jeszcze nie usunęliśmy, co dodaje do kosztu czasowego składnik $O(M^2)$.

Złożoność czasowa tego algorytmu to $O(k^2M^2)$ — dla każdego z $(k+1)M$ prostokątów przeglądamy listę wszystkich punktów znaczących, których jest $O(kM)$. Jest to zatem algorytm zbyt wolny. Widać, że winna jest tu mało wyrafinowana struktura danych reprezentująca miotłę. Poszukamy zatem czegoś lepszego.

Efektywna implementacja miotły: drzewo przedziałowe

Poszukiwania odpowiedniej struktury danych zacznijmy, metodycznie, od wypisania listy operacji, jakich wykonywanie musi ona umożliwiać. Załóżmy, że miotła zawiera N punktów znaczących, ponumerowanych od 1 do N. Wówczas każdemu punktowi znaczącemu i odpowiada parametr $wartosc[i]$. Musimy umieć obsłużyć pięć operacji:

(a) Ustaw wszystkie wartości na zero.

(b) Dodaj 1 do wszystkich wartości na przedziale $[a, b]$.

(c) Odejmij 1 na przedziale $[a, b]$, do którego wcześniej dodano 1.

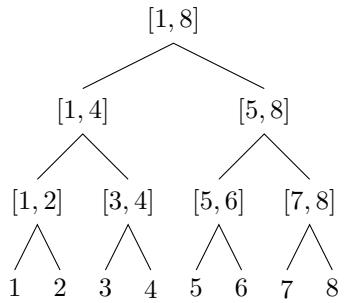
(d) Wyznacz największą wartość w miotle.

(e) Znajdź dowolne wystąpienie tej wartości.

Jeżeli będziemy umieli wykonać każdą z tych operacji w czasie $O(\log N)$, to będziemy w stanie zaimplementować algorytm miotły w czasie $O(kM \log(kM) + M^2)$. Faktycznie, wszystko poza powyższymi operacjami (w tym dwukrotne sortowanie) możemy wykonać w takim właśnie czasie, natomiast do wykonania operacji (b) i (c) potrzeba dodatkowo możliwości przeliczania współrzędnych punktów na miotle na numery najbliższych punktów znaczących, co można wykonywać w czasie $O(\log(kM))$ za pomocą wyszukiwania binarnego w tablicy *znaczace*.

Operacje (a)-(e) można zrealizować efektywnie, wykorzystując odpowiednio dostosowane *drzewo przedziałowe*[2]. Załóżmy, że N jest potęgą dwójki o całkowitym wykładniku (jeśli tak nie jest, to zwiększmy N tak, aby stało się potęgą dwójki). Wówczas *drzewo przedziałowe* to pełne drzewo binarne, którego liście odpowiadają liczbom od 1 do N, a węzły wewnętrzne reprezentują niektóre podprzedziały przedziału $[1, N]$, zwane *przedziałami bazowymi*, patrz rys. 2. W przedziałach interesują nas tylko liczby całkowite w nich zawarte. Przypomnijmy podstawowe własności tej struktury danych:

- Drzewo przedziałowe zawiera $O(N)$ węzłów, ma głębokość $O(\log N)$ i można je zaimplementować za pomocą jednej tablicy, tak aby dało się poruszać wzdłuż jego krawędzi w czasie stałym.
- Wszystkie przedziały bazowe zawierające dane $i \in \{1, 2, \ldots, N\}$ stanowią ścieżkę od liścia odpowiadającego i do korzenia.
- Każdy przedział $[a, b]$ można rozłożyć na sumę $O(\log N)$ parami rozłącznych przedziałów bazowych. Zarówno złożoność czasowa tego algorytmu, jak i liczba przedziałów bazowych będących przodkami przedziałów z otrzymanego rozkładu są rzędu $O(\log N)$.



Rys. 2: Przykład drzewa przedziałowego dla $N = 8$.

Obsługa operacji (a)-(c) stanowi jedno z najlepiej znanych, a zarazem najprostszych zastosowań drzew przedziałowych. Z każdym węzłem v drzewa wiążemy wartość $w[v]$, początkowo równą zeru. Liczby $w[v]$ mają spełniać tę własność, że dla każdego $i \in \{1, 2, \ldots, N\}$, $val[i]$ to suma wartości $w[v]$ po wszystkich przedziałach bazowych zawierających i (czyli na ścieżce od liścia odpowiadającego i do korzenia). Aby tak było, w operacjach (b) i (c) zwiększamy, odpowiednio o $+1$ lub -1, wartości $w[v]$ we wszystkich przedziałach bazowych z rozkładu przedziału $[a, b]$.

[2]W tym opracowaniu przedstawiamy jedynie skrócony opis drzew przedziałowych. Więcej na ich temat można przeczytać np. w opracowaniu zadania *Tetris 3D* z XIII Olimpiady Informatycznej [13] albo na stronie `http://was.zaa.mimuw.edu.pl/?q=node/8`. Sposób dostosowania (formalnie: wzbogacenia) drzew przedziałowych przedstawiony dalej jest podobny jak w przypadku problemu obliczania pola sumy (teoriomnogościowej) pewnej liczby prostokątów na płaszczyźnie, o którym to problemie można poczytać w książce [27] lub posłuchać na wspomnianej już wcześniej stronie `http://was.zaa.mimuw.edu.pl?q=node/37`. Wzbogacanie drzew przedziałowych pojawiło się także stosunkowo niedawno na Olimpiadzie: zadanie *Łyżwy*, XVI OI [16].

Obsłużenie operacji (d) oraz (e) wymaga dodatkowego *wzbogacenia* drzewa, czyli dodania we wszystkich węzłach jeszcze jednego, pomocniczego parametru, który oznaczymy przez W. Wartość $W[v]$ będzie reprezentować maksymalną wartość liścia w poddrzewie ukorzenionym w v, ale tak, jakby to poddrzewo było osobnym drzewem przedziałowym. Innymi słowy, jest to maksimum z sum wartości w na ścieżkach od v do wszystkich liści w poddrzewie v. Następujące równanie rekurencyjne:

$$W[v] = w[v] + \max(W[lewy\_syn(v)],\ W[prawy\_syn(v)]), \tag{1}$$

zachodzące dla wszystkich węzłów wewnętrznych v drzewa przedziałowego, pozwala w standardowy sposób aktualizować wartości W w drzewie po wykonaniu operacji (b) lub (c). Ów standardowy sposób polega na przejściu przez wszystkie węzły będące przodkami węzłów z rozkładu przedziału $[a, b]$, w kierunku od liści do korzenia, i zastosowaniu tego wzoru do obliczenia $W[v]$ we wszystkich węzłach v niebędących liśćmi. Koszt czasowy wciąż logarytmiczny.

Jeśli mamy do dyspozycji wartości parametru W, operację (d) obsługujemy w czasie stałym, zwracając tę wartość w korzeniu. Z operacją (e) także pójdzie łatwo: wystarczy przejść ścieżką od korzenia drzewa do liścia odpowiadającego wartości zwracanej w operacji (d). W tym celu, będąc w danym momencie w węźle v, przechodzimy do tego spośród jego synów, który realizuje maksimum we wzorze (1). Koszt czasowy takiego przejścia to, oczywiście, $O(\log N)$.

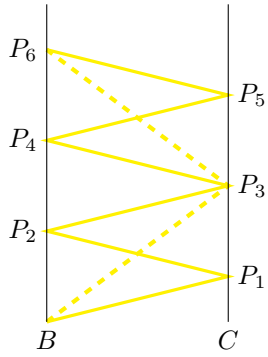
Wykończenie i algorytm wzorcowy

Omówiliśmy już technikę zamiatania wraz z drzewem przedziałowym. Umiemy w czasie $O(kM \log(kM) + M^2)$ odpowiedzieć na pytanie, które z okien na budynku B są oświetlone przez promienie światła odbijające się dokładnie k razy. Na tym etapie rozwiązywania zadania może wydawać się, że już nie musimy analizować naszego problemu, potrzebujemy tylko jakiejś technicznej sztuczki, aby osiągnąć akceptowalną złożoność. W końcu nie widać wyraźnych przeszkód, które uniemożliwiałyby sprytne połączenie, niezależnych w naszym podejściu, obliczeń dla różnych wartości k.

Istnieje jednak inne możliwe źródło ulepszeń. Fakt, że nie jest od razu widoczne, a samo jego dostrzeżenie stanowi tylko połowę sukcesu, stanowi o dużym stopniu trudności zadania. Spróbujmy cofnąć się na sam początek rozważań, do momentu, w którym myśleliśmy jeszcze o dwóch budynkach stojących naprzeciw. W ten sposób możemy dostrzec kluczowy fakt:

Obserwacja 1. Jeżeli promień z latarni może pokonać trasę do pewnego okna w $2^l(2m+1)-1$ odbiciach, to może pokonać tę samą trasę w $2^l - 1$ odbiciach.

Zaiste — rozważmy trasę, którą pokonuje promień, wykonując $2^l(2m+1)-1$ odbić, i przez $P_1, P_2, \ldots, P_{2^l(2m+1)-1}$ oznaczmy kolejne punkty odbić, a przez $P_{2^l(2m+1)}$ — docelowy punkt naszego promienia. Rozważmy teraz promień, który po raz pierwszy trafia w budynek C w punkcie P_{2m+1}. Jego kolejne odbicia wystąpią w punktach $P_{2(2m+1)}$, $P_{3(2m+1)}$, aż do $P_{2^l(2m+1)}$. Mamy gwarancję, że wszystkie punkty odbić są we wnętrzach okien, gdyż są to punkty, w których odbijał się oryginalny promień



Rys. 3: Pięciokrotne odbicie można zastąpić jednokrotnym.

(łatwo sprawdzić, że występują na tych samych budynkach). Zatem faktycznie nasz promień dotarł do celu w $2^l - 1$ odbiciach.

Warto tu zwrócić uwagę, że ten argument nie pozwala zejść z $2^l - 1$ odbić do $2^{l-1} - 1$ odbić — parzyste odbicia zawsze występują na ścianie budynku B, i nie gwarantują tego, że symetryczne punkty na budynku C znajdują się wewnątrz okien.

Co daje nam ta obserwacja? Otóż, jeśli jakieś okno będzie oświetlone, to będzie oświetlone w $2^l - 1$ odbiciach dla pewnego l. Wobec tego możemy rozważać istotnie mniej wartości parametru k — zamiast wszystkich liczb nieparzystych z przedziału $[1, 2000]$, wystarczy rozważać liczby postaci $2^l - 1$. Niech L będzie największą liczbą, dla której $2^L \leqslant R$ (w naszym przypadku $L = 10$, ogólnie $L = O(\log R)$). Stosując omówioną już metodę zamiatania, otrzymujemy algorytm o złożoności czasowej

$$O\left(M^2 + \sum_{l=1}^{L}\left(2^l M \log\left(2^l M\right)\right)\right) = O\left(M^2 + M\log\left(2^L M\right)\sum_{l=1}^{L} 2^l\right) =$$
$$= O\left(M\log\left(2^L M\right)2^{L+1} + M^2\right) = O\left(2^L M(L + \log M) + M^2\right).$$

W powyższych przekształceniach wartość M^2 umieściliśmy poza obliczaną sumą, gdyż odpowiada ona identyfikacji okna zawierającego dany punkt znaczący, które wykreślamy wówczas z dalszych rozważań, uznając je za oświetlone. Dla $L = 10$ i $M \leqslant 600$ takie rozwiązanie powinno bez większych problemów zmieścić się w limicie czasowym.

To podejście zostało zaimplementowane w plikach `lat.cpp` i `lat2.pas`.

Inne rozwiązania

Rozwiązania niepoprawne

W pliku `latb1.cpp` zaimplementowano rozwiązanie takie jak wzorcowe, które ogranicza się do liczby odbić nie większej niż 300. Taki program nie dostanie więcej niż 60 pkt. — jest możliwe, że do oświetlenia pewnego okna będą konieczne 1023 odbicia.

W pliku `latb2.cpp` jest rozwiązanie działające niemalże jak program `lat.cpp`. Jedyna różnica polega na obsłudze zdarzenia, w którym przychodzi przedział $[a, b]$

powodujący, że w miotle jest obszar pokryty przez $k+1$ prostokątów. W tym podejściu usuwamy wszystkie wyróżnione prostokąty, które przecinają się niepusto z przedziałem $[a, b]$. Oczywiście, jest to rozwiązanie niepoprawne.

W pliku `latb3.cpp` znajduje się rozwiązanie, które rozważa kilkaset losowych punktów z wnętrza każdego z okien i sprawdza, w które z nich może trafić promień światła. W tym celu przeglądane są kolejne liczby odbić postaci $2^l - 1$. Przy ustalonej liczbie odbić żądane sprawdzenie można wykonać w czasie $O(2^l)$. Wynika to z faktu, że po odpowiednich obliczeniach wstępnych, sprawdzenie, czy punkt (x, y) leży w oknie na danej ścianie, można wykonać w czasie stałym.

W pliku `latb4.cpp` znajduje się rozwiązanie, które działa prawie tak samo jak poprzednie. Różnica polega na tym, że nie korzystamy tutaj z Obserwacji 1 i sprawdzamy kilka losowych punktów dla kolejnych liczb odbić od 2 aż do 1000.

Rozwiązanie nieoptymalne

Po każdym odbiciu utrzymujemy oświetlony obszar, jako zbiór parami rozłącznych prostokątów. Po początkowych odbiciach zbiór prostokątów ma rozmiar rzędu $O(nm)$, co przy próbie przecinania każdego prostokąta z naszego zbioru z każdym napotkanym oknem daje koszt symulacji jednego odbicia $O(nm(n+m))$. Takie rozwiązanie zostało zaimplementowane w pliku `lats1.cpp`. W zależności od jakości implementacji, za tego typu rozwiązania można było uzyskać od 20 do 40 punktów.

Testy

Testy zostały podzielone na pięć grup, w każdej występuje kilka różnych testów.

Nazwa	n + m	Opis
lat1[a-d].in	$[41, 110]$	testy poprawnościowe
lat2[a-h].in	$[4, 320]$	testy poprawnościowe, dużo odbić
lat3[a-h].in	$[2, 580]$	testy poprawnościowe, dużo odbić
lat4[a-g].in	$[630, 960]$	testy wydajnościowe, dużo odbić
lat5[a-h].in	$[1\,050, 1\,200]$	testy wydajnościowe, dużo odbić

Jakub Radoszewski — Treść zadania, Opracowanie
Łukasz Bieniasz-Krzywiec — Program

Dostępna pamięć: 128 MB. *OI, Etap III, dzień pierwszy, 14.04.2010*

Żabka

Wzdłuż bajtockiego strumyczka stoi n kamieni. Są one położone kolejno w odległościach $p_1 < p_2 < \ldots < p_n$ od źródła strumyczka. Na jednym z tych kamieni stoi żabka, która właśnie ćwiczy się w skakaniu. W każdym skoku żabka skacze na k-ty najbliższy kamień (od kamienia, na którym siedzi). Dokładniej, jeżeli żabka w danej chwili siedzi na kamieniu w położeniu p_i, to po wykonaniu skoku będzie siedziała na takim kamieniu p_j, że:

$$\left|\left\{p_a \ : \ |p_a - p_i| < |p_j - p_i|\right\}\right| \leqslant k \qquad \text{oraz} \qquad \left|\left\{p_a \ : \ |p_a - p_i| \leqslant |p_j - p_i|\right\}\right| > k.$$

Pierwsza z powyższych nierówności oznacza, że liczba punktów $p_a \in \{p_1, p_2, \ldots, p_n\}$, których odległość od punktu p_i jest mniejsza niż odległość między punktami p_i a p_j, jest nie większa niż k. Podobnie, druga z nierówności oznacza, że liczba punktów $p_a \in \{p_1, p_2, \ldots, p_n\}$, których odległość od punktu p_i jest nie większa niż odległość między punktami p_i a p_j, jest większa niż k.

W przypadku, gdy istnieje więcej niż jedno takie p_j, żabka wybiera skrajnie lewą spośród takich pozycji. Na którym kamieniu żabka znajdzie się po wykonaniu m skoków, w zależności od tego, z którego kamienia zaczyna?

Wejście

Pierwszy wiersz standardowego wejścia zawiera trzy liczby całkowite n, k oraz m ($1 \leqslant k < n \leqslant 1\,000\,000$, $1 \leqslant m \leqslant 10^{18}$), pooddzielane pojedynczymi odstępami i reprezentujące odpowiednio: liczbę kamieni, parametr skoku żabki oraz liczbę zaplanowanych skoków żabki. Drugi wiersz zawiera n liczb całkowitych p_j ($1 \leqslant p_1 < p_2 < \ldots < p_n \leqslant 10^{18}$), pooddzielanych pojedynczymi odstępami i oznaczających położenia kolejnych kamieni wzdłuż strumyczka.

Wyjście

Twój program powinien wypisać na standardowe wyjście dokładnie jeden wiersz zawierający n liczb całkowitych $r_1, r_2, \ldots, r_n$ z przedziału $[1, n]$, pooddzielanych pojedynczymi odstępami. Liczba r_i oznacza numer kamienia, na którym żabka zakończy skakanie, jeśli rozpocznie je z kamienia o numerze i (zgodnie z kolejnością z wejścia).

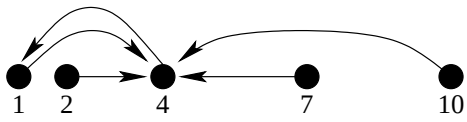
Przykład

Dla danych wejściowych:

```
5 2 4
1 2 4 7 10
```

poprawnym wynikiem jest:

```
1 1 3 1 1
```



Na rysunku pokazano, jak żabka skacze z poszczególnych kamieni (w jednym skoku).

Rozwiązanie

Wprowadzenie

Rozwiązanie niniejszego zadania w naturalny sposób dzieli się na dwie części, z których pierwsza jest związana z parametrem k, a druga z parametrem m. W Fazie I dla każdego z kamieni $p_1, p_2, \ldots, p_n$ znajdujemy k-ty najbliższy mu kamień, co reprezentujemy za pomocą tablicy f:

$$f[i] = j \quad \Leftrightarrow \quad p_j \text{ to } k\text{-ty najbliższy kamień od kamienia } p_i. \tag{1}$$

Tablica f pokazuje zatem, jak żabka skacze z poszczególnych kamieni (w jednym skoku). W Fazie II dla każdego kamienia p_i sprawdzamy, gdzie żabka znajdzie się po wykonaniu m skoków, począwszy od tego kamienia, czyli obliczamy wartość:

$$f^m[i] \;=\; \underbrace{f[f[\ldots f[}_{m \text{ razy}} i] \ldots]]. \tag{2}$$

Widać wyraźnie, że podane fazy możemy rozpatrywać niezależnie. Dla każdej z nich omówimy sposób dojścia od najprostszego (a zarazem najmniej efektywnego) rozwiązania do algorytmu wchodzącego w skład rozwiązania wzorcowego.

W dalszym opisie kamień w położeniu p_i będziemy często nazywali po prostu i-tym kamieniem.

Faza I

Metoda 1a: złożoność czasowa $O(n^2 \log n)$

Aby obliczyć $f[i]$, czyli numer k-tego najbliższego kamienia od i-tego kamienia, możemy na przykład posortować wszystkie kamienie względem odległości od i-tego kamienia, po czym jako $f[i]$ przyjąć numer k-tego kamienia w tym porządku. W ogólności ten pomysł brzmi całkiem rozsądnie, ale musimy poświęcić chwilę na analizę dosyć technicznej definicji *k-tego najbliższego kamienia* z treści zadania, zwracając dodatkowo szczególną uwagę na remisy.

Spróbujmy zacząć od czysto intuicyjnego podejścia, ignorując skomplikowanie wyglądające nierówności z treści zadania. Niech t będzie tablicą złożoną z par postaci:

$$t = ((|p_j - p_i|,\ j) \ :\ j = 1, 2, \ldots, n),$$

reprezentującą odległości wszystkich kamieni od i-tego kamienia. Załóżmy, że t jest uporządkowana rosnąco, przy czym pary są porównywane leksykograficznie po współrzędnych (tzn. porównywane w pierwszej kolejności po pierwszej współrzędnej, a w razie remisu — po drugiej). Zauważmy, że w przypadku równych pierwszych współrzędnych para reprezentująca kamień położony bardziej na lewo znajduje się w tablicy wcześniej. Czy wówczas k-ty element posortowanej tablicy będzie wskazywał na kamień reprezentujący $f[i]$?

Odpowiedź na postawione pytanie jest negatywna, ale zaproponowane rozwiązanie nie jest bardzo odległe od poprawnego — musimy tylko dopracować kilka szczegółów.

Przede wszystkim wykluczmy z rozważań sam i-ty kamień: możemy go usunąć z tablicy t, a także wyeliminować z nierówności z zadania, otrzymując:

$$\begin{aligned} |\{p_a \ : \ a \ \neq i \ \wedge \ |p_a - p_i| < |p_j - p_i|\}| &< k \\ |\{p_a \ : \ a \ \neq i \ \wedge \ |p_a - p_i| \leqslant |p_j - p_i|\}| &\geqslant k. \end{aligned}$$

Kolejne spostrzeżenie jest takie, że w ustalonej odległości $d > 0$ od i-tego kamienia mogą znajdować się co najwyżej dwa inne kamienie: jeden po lewej, a drugi po prawej jego stronie. Zauważmy, że tylko skrajnie lewy spośród tych dwóch kamieni może wyznaczać wartość $f[i]$. W takim razie, wystąpienie skrajnie prawego w t możemy zastąpić wystąpieniem skrajnie lewego.

Niech teraz $t[k] = (|p_j - p_i|, \ j)$ będzie wystąpieniem j-tego kamienia w zmodyfikowanej w opisany sposób tablicy t. Wówczas

$$|\{p_a \ : \ a \ \neq i \ \wedge \ |p_a - p_i| < |p_j - p_i|\}| \in \{k-1, k-2\},$$

natomiast

$$|\{p_a \ : \ a \ \neq i \ \wedge \ |p_a - p_i| \leqslant |p_j - p_i|\}| \in \{k, k+1\},$$

przy czym dokładna wartość w każdym przypadku zależy od tego, czy jest to pierwsze (lub jedyne), czy drugie wystąpienie j-tego kamienia. To pokazuje, że $f[i] = j$.

Poniższy pseudokod ilustruje opisane podejście. Wykorzystujemy w nim procedurę sortującą tablicę t w czasie $O(n \log n)$, patrz np. książka [21]. Dostęp do poszczególnych elementów par w tablicy uzyskujemy za pomocą pól *first* oraz *second*. Zakładamy wreszcie możliwość umieszczania nowych elementów na końcu tablicy t (operacja *insert*), co możemy zaimplementować, używając odpowiednio dużej, statycznej tablicy oraz zmiennej reprezentującej liczbę zajętych początkowych pól. Dodajmy dla jasności, że t jest indeksowana od jedynki.

```
1:  Algorytm 1a
2:    for i := 1 to n do begin
3:      t := ∅;
4:      for j := 1 to n do
5:        if j ≠ i then t.insert((|p_j − p_i|, j));
6:      sort(t);
7:      for j := 2 to n − 1 do
8:        if t[j].first = t[j − 1].first then t[j] := t[j − 1];
9:      f[i] := t[k].second;
10:   end
11:   return f;
```

Złożoność czasowa Algorytmu 1a to $O(n^2 \log n)$.

Metoda 1b: złożoność czasowa $O(n \cdot k)$

Stosunkowo proste usprawnienie Metody 1a możemy uzyskać, jeśli zauważymy, że *im dalej od i-tego kamienia, tym odległości od niego są większe*. To oczywiste stwierdzenie

implikuje, że zawartość tablicy t otrzymujemy w wyniku scalenia dwóch następujących list, uporządkowanych względem odległości od kamienia p_i:

$$L_1 = ((p_i - p_{i-1}, i-1), \ldots, (p_i - p_1, 1)) \quad \text{oraz} \quad L_2 = ((p_{i+1} - p_i, i+1), \ldots, (p_n - p_i, n)).$$

Podczas scalania musimy pamiętać o tym, żeby w przypadku remisu do tablicy t wstawić dwie kopie skrajnie lewego spośród kamieni równoodległych od i-tego kamienia. Jeśli zauważymy, że list L_1 i L_2 nie musimy konstruować explicite (wystarczą nam dwa wskaźniki przemieszczające się po tablicy p wszystkich kamieni) oraz że scalanie możemy przerwać w momencie, gdy wynikowa lista ma już co najmniej k elementów, to otrzymamy algorytm o złożoności czasowej $O(n \cdot k)$.

```
1:  Algorytm 1b
2:    for i := 1 to n do begin
3:      t := ∅;
4:      a := i − 1;  b := i + 1;
5:      while (a ⩾ 1) and (b ⩽ n) and (size(t) < k) do begin
6:        if p_i − p_a < p_b − p_i then begin
7:          { Kamień a jest bliższy. }
8:          t.insert((p_i − p_a, a));  a := a − 1;
9:        end else if p_i − p_a > p_b − p_i then begin
10:         { Kamień b jest bliższy. }
11:         t.insert((p_b − p_i, b));  b := b + 1;
12:       end else begin
13:         { Mamy remis, więc dwukrotnie wstawiamy kamień a. }
14:         t.insert((p_i − p_a, a));  t.insert((p_i − p_a, a));
15:         a := a − 1;  b := b + 1;
16:       end
17:     end
18:     { Po jednej stronie zużyliśmy już wszystkie kamienie, }
19:     { po drugiej jeszcze nie. }
20:     while (a ⩾ 1) and (size(t) < k) do begin
21:       t.insert((p_i − p_a, a));  a := a − 1;
22:     end
23:     while (b ⩽ n) and (size(t) < k) do begin
24:       t.insert((p_b − p_i, b));  b := b + 1;
25:     end
26:     f[i] := t[k].second;
27:   end
28:   return f;
```

Metoda 1c: złożoność czasowa $O(n \log n \log p_n)$

Metoda 1b jest efektywna, jeśli żabka postanawia wykonywać jedynie niezbyt długie skoki, tzn. wartość parametru k jest niewielka. Teraz opiszemy metodę, która spisuje się całkiem dobrze także, gdy k jest duże. Będzie ona oparta na wyszukiwaniu binarnym, i to niejednym.

Możemy mianowicie rozpocząć wyznaczanie $j = f[i]$ od znalezienia wartości $d = |p_j - p_i|$. Innymi słowy, poszukujemy takiej granicznej wartości d, że w odległości nie większej niż d od i-tego kamienia znajduje się co najmniej k innych kamieni, ale w odległości nie większej niż $d-1$ od i-tego kamienia jest już mniej niż k kamieni. Wartość d wyszukamy binarnie w przedziale $[1, p_n]$.

Pozostaje pytanie, jak wyznaczyć liczbę kamieni oddalonych od i-tego kamienia nie więcej niż o d. W tym celu ponownie stosujemy wyszukiwanie binarne (a nawet dwa), aby znaleźć numery skrajnych kamieni położonych po lewej i po prawej stronie i-tego kamienia, których odległości od tego kamienia nie przekraczają d.

Poniżej implementacja tej metody wyznaczania wartości d o złożoności czasowej $O(\log n \log p_n)$.

```
1:  Algorytm 1c, wyznaczanie granicznej odległości d
2:    function na_lewo(i, odl)
3:    begin
4:      lo' := 1; hi' := i - 1;
5:      while lo' < hi' do begin
6:        s' := (lo' + hi') div 2;
7:        if p_i - p_s' ⩽ odl then hi' := s'
8:        else lo' := s' + 1;
9:      end
10:     return lo';
11:   end
12:
13:   function na_prawo(i, odl)
14:   { Analogicznie do na_lewo. }
15:
16:   function oblicz_d(i)
17:   begin
18:     lo := 1; hi := p_n
19:     while lo < hi do begin
20:       s := (lo + hi) div 2;
21:       if na_prawo(i, s) - na_lewo(i, s) ⩾ k then hi := s
22:       else lo := s + 1;
23:     end
24:     return lo;
25:   end
```

Zauważmy na koniec, że znając wartość d, możemy jeszcze raz wywołać funkcje *na_lewo* i *na_prawo*, tym razem już z właściwym argumentem $odl = d$, i ze zwróconych przez nie wyników wybrać ten, który reprezentuje kamień położony dalej od i-tego (lub skrajnie lewy w przypadku remisu). Poprawność tego stwierdzenia wynika stąd, że jeśli odległości skrajnych kamieni od i-tego są różne, to $\text{na\_prawo}(i, d) - \text{na\_lewo}(i, d) = k$, a w przeciwnym przypadku ta różnica jest równa k lub $k+1$. Stosowne dokończenie implementacji poniżej.

```
1:  Algorytm 1c, dokończenie
2:    for i := 1 to n do
```

```
3:  begin
4:    d := oblicz_d(i);
5:    a := na_lewo(i, d);  b := na_prawo(i, d);
6:    if p_i − p_a ⩾ p_b − p_j then f[i] := a
7:    else f[i] := b;
8:  end
9:  return f;
```

Złożoność czasowa Metody 1c to $O(n \log n \log p_n)$. Jest to już całkiem niezły wynik jak na ograniczenia z zadania: $n \leqslant 10^6$, $p_n \leqslant 10^{18}$. Niemniej jednak, także i ten rezultat jeszcze da się poprawić.

Metoda 1d: złożoność czasowa $O(n)$

W dotychczasowych rozwiązaniach wyznaczaliśmy jedynie k-te najbliższe kamienie; tym razem pójdziemy o krok dalej i przyjrzymy się temu, jak wygląda *zbiór* k kamieni najbliższych i-temu kamieniowi (remisy rozstrzygamy zgodnie z regułą skrajnie lewego). Już w Metodzie 1c zauważyliśmy, że jest to przedział, złożony z pewnej liczby kamieni położonych na lewo od i-tego oraz pewnej liczby kamieni leżących na prawo od i-tego. Oznaczmy ten przedział, wraz z samym i-tym kamieniem, przez $I[i] = [a\,..\,a+k]$.

Aby efektywnie wyznaczać przedziały $I[i]$, zastanowimy się, jaka jest zależność pomiędzy $I[i-1]$ a $I[i]$. Zauważmy, że jest to także pewne novum w stosunku do dotychczasowych podejść, w których każdy kamień rozpatrywaliśmy osobno. Okazuje się, że zachodzi następujący, całkiem intuicyjny fakt.

Fakt 1. *Jeśli* $I[i-1] = [a\,..\,a+k]$ *oraz* $I[i] = [b\,..\,b+k]$, *to* $a \leqslant b$.

Dowód: Załóżmy nie wprost, że $b < a$. Wówczas mamy $(b+k+1) \in I[i-1]$, ale $(b+k+1) \notin I[i]$. Korzystając z tego spostrzeżenia, otrzymujemy:

$$|p_{b+k+1} - p_i| \overset{(1)}{<} |p_{b+k+1} - p_{i-1}| \overset{(2)}{<} |p_{a-1} - p_{i-1}| \overset{(3)}{<} |p_{a-1} - p_i| \overset{(4)}{\leqslant} |p_b - p_i|.$$

Powyższe nierówności uzasadniamy następująco:

(1) Ponieważ $i - 1 < i \ < b + k + 1$.

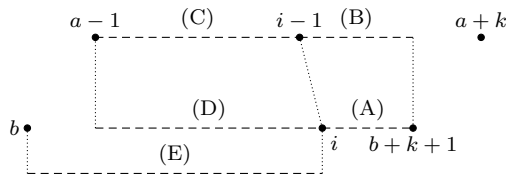
(2) Ponieważ $(b + k + 1) \in I[i-1]$, ale $(a-1) \notin I[i-1]$.

(3) Ponieważ $a - 1 < i - 1 < i$.

(4) Ponieważ $b < a$.

Graficzne przedstawienie tego ciągu nierówności, przy oznaczeniu kolejnych z powyższych wartości bezwzględnych różnic przez (A), (B), (C), (D), (E), przedstawia rys. 1.

Aby zakończyć dowód, wystarczy zauważyć, że uzyskany ciąg nierówności stanowi sprzeczność z faktem, iż $b \in I[i]$, ale $(b + k + 1) \notin I[i]$. ■



Rys. 1: Ilustracja ciągu nierówności: $(A) \overset{(1)}{<} (B) \overset{(2)}{<} (C) \overset{(3)}{<} (D) \overset{(4)}{\leqslant} (E)$.

Musimy jeszcze odpowiedzieć na dwa pytania.

1. *Jak wykorzystać Fakt 1 do efektywnego wyznaczania przedziałów $I[i]$?* W tym celu każdorazowo zaczynamy od $I[i] = I[i-1]$, po czym przesuwamy ten przedział o jeden w prawo, dopóki jest to wskazane. Łatwo widać, że przedział $[a..a+k]$ należy zamienić na $[a+1..a+k+1]$, jeżeli $(a+k+1)$-szy kamień jest położony bliżej i-tego niż znajdujący się aktualnie w przedziale kamień a-ty, czyli:
$$p_{a+k+1} - p_i < p_i - p_a.$$

2. *Po co nam przedziały $I[i]$, czyli jak za ich pomocą obliczać wartości $f[i]$?* Czytelnik zaznajomiony z poprzednimi metodami szybko dostrzeże, że $f[i]$ odpowiada jednemu z końców przedziału $I[i]$. Jeżeli odległości kamieni odpowiadających końcom tego przedziału od kamienia i-tego są różne, to $f[i]$ uzyskujemy, biorąc dalszy z nich. W przeciwnym przypadku, ze względu na kryterium rozstrzygania remisów z treści zadania, jako wynik przyjmujemy kamień skrajnie lewy. Uzasadnienie poprawności tego spostrzeżenia pozostawiamy Czytelnikowi (wbrew pozorom, na ewentualne remisy należy zwrócić uwagę w obydwu przypadkach).

Algorytm 1d otrzymujemy jako bezpośrednią konsekwencję odpowiedzi na powyższe pytania, patrz implementacja poniżej. Na pierwszy rzut oka niepokoić może jego złożoność czasowa: mamy dwie zagnieżdżone pętle, z których każda wykonuje co najwyżej n obrotów. Nietrudno jednak zauważyć, że łączna liczba obrotów tych pętli będzie liniowa względem n, jako że każdy obrót pętli **while** powoduje zwiększenie wartości zmiennej a o jeden, a zmienna ta przyjmuje wartości z przedziału $[1, n-k]$. To rozumowanie, będące bardzo prostym przypadkiem *analizy kosztu zamortyzowanego* (patrz np. książka [21]), pokazuje, że złożoność czasowa poniższego algorytmu to $O(n)$. Dodajmy dla ścisłości, że złożoność pamięciowa tego algorytmu, a także wszystkich wcześniej zaprezentowanych, to $O(n)$.

```
1: Algorytm 1d
2:   a := 1;
3:   f[1] := k + 1;
4:   for i := 2 to n do begin
5:     while (a + k + 1 ⩽ n) and (p_{a+k+1} − p_i < p_i − p_a) do
6:       a := a + 1;
7:     if p_i − p_a ⩾ p_{a+k} − p_i then f[i] := a
8:     else f[i] := a + k;
9:   end
```

```
10:  return f;
```

Porównując Algorytm 1d z mniej efektywnymi metodami, otrzymujemy kolejne potwierdzenie częstego zjawiska, że najefektywniejszy algorytm dla danego problemu cechuje się jednocześnie najprostszą implementacją. Co bynajmniej nie oznacza, że jest najprostszy do wymyślenia!

Faza II

W poprzednich sekcjach przedstawiliśmy kilka różnych algorytmów pozwalających obliczyć $f[i]$ dla $i = 1, 2, \ldots, n$, która to funkcja reprezentuje miejsca docelowe pojedynczych skoków żabki z poszczególnych kamieni. Pod koniec udało nam się już uzyskać całkiem efektywne rozwiązania. Przyszła pora, aby obliczyć miejsca docelowe po seriach skoków długości m, czyli wartości $r[i] := f^m[i]$ dla $i = 1, 2, \ldots, n$.

Metoda 2a: złożoność czasowa $O(n \cdot m)$

Zacznijmy od najprostszego możliwego algorytmu: wartość $f^m[i]$ możemy obliczyć wprost z definicji (2). Złożoność czasowa takiego podejścia to ewidentnie $\Theta(n \cdot m)$ — rzut oka na ograniczenia z zadania pokazuje, że jest to metoda dalece niewystarczająca.

Metoda 2b: złożoność czasowa $O(n^2)$

Przyjrzyjmy się sekwencji kolejnych wartości postaci:

$$i,\ f[i],\ f[f[i]],\ f[f[f[i]]], \ldots$$

Czy można w tym ciągu wychwycić jakąś regularność, która mogłaby pozwolić na szybsze obliczenie $f^m[i]$?

Otóż tak, mianowicie z zasady szufladkowej Dirichleta wynika, że wśród pierwszych $n+1$ wartości $f^j[i]$ ($j = 1, 2, \ldots$) jakaś liczba musi się powtórzyć, gdyż wszystkie te liczby są z zakresu $1..n$. Niech p i q wskazują na pierwsze takie powtórzenie (czyli q najmniejsze możliwe). Wówczas:

$$f^p[i] = f^q[i], \quad f^{p+1}[i] = f^{q+1}[i], \quad f^{p+2}[i] = f^{q+2}[i], \ldots$$

czyli od q-tego elementu fragment $f^p[i], f^{p+1}[i], \ldots, f^{q-1}[i]$ będzie powtarzał się w nieskończoność. To daje prostą receptę na wyznaczenie $f^m[i]$. Znajdujemy wartości p oraz q; jeżeli $m < q$ to przy okazji obliczyliśmy także $f^m[i]$, a w przeciwnym razie możemy odczytać żądaną wartość na podstawie $(m-q) \bmod (q-p)$, czyli traktując m modulo długość cyklu.

Poniższy pseudokod przedstawia implementację tego podejścia w złożoności czasowej $O(n^2)$. Podobnie jak w metodach z Fazy I, wykorzystujemy w nim tablicę indeksowaną od jedynki, do której możemy dokładać elementy na koniec (zmienna t). Łącząc pomysły z Metod 2a oraz 2b, można dokonać kosmetycznej poprawki złożoności czasowej do $O(n \cdot \min(n, m))$, co pozostawiamy Czytelnikowi do przemyślenia.

```
1:  Algorytm 2b
2:    for i := 1 to n do begin
3:      for j := 1 to n do c[j] := −1;
4:      j := i; q := 0;
5:      t := ∅;
6:      while (c[j] = −1) and (q < m) do begin
7:        t.insert(j);
8:        c[j] := q;
9:        j := f[j]; q := q + 1;
10:     end
11:     if q = m then r[i] := j
12:     else begin
13:       p := c[j];
14:       r[i] := t[p + ((m − q) mod (q − p)) + 1];
15:     end
16:   end
17:   return r;
```

Metoda 2c: złożoność czasowa $O(n \log m)$

Redukcja początkowej złożoności czasowej $O(n \cdot m)$ do złożoności czasowej $O(n \log m)$ ma charakter strukturalny. Potraktujmy f jako funkcję, $f : \{1, 2, \ldots, n\} \to \{1, 2, \ldots, n\}$. Interesuje nas m-krotne złożenie tej funkcji, czyli f^m. Tę wartość możemy obliczyć za pomocą szybkiego potęgowania binarnego, np. korzystając ze wzorów:

$$f^{2k+1} = f^{2k} \circ f, \quad f^{2k} = \left(f^k\right)^2.$$

Poniżej implementacja korzystająca z tego samego podejścia, ale nierekurencyjna. Warto zwrócić uwagę na to, że jej złożoność pamięciowa to $O(n)$ — przy koszcie pamięciowym rzędu $\Theta(n \log m)$ moglibyśmy nie zmieścić się w limicie pamięciowym 128 MB.

```
1:  Algorytm 2c
2:    { Oblicza złożenie g∘h funkcji reprezentowanych przez n-elementowe }
3:    { tablice g i h, wynik znajduje się w tablicy w. }
4:    function zlozenie(g, h)
5:    begin
6:      for i := 1 to n do w[i] := g[h[i]];
7:      return w;
8:    end
9:
10:   g := f;   { tablica g będzie przechowywać kolejne potęgi f^(2^a) }
11:   for i := 1 to n do r[i] := i;   { funkcja identycznościowa }
12:   while m > 0 do begin
13:     if m mod 2 = 1 then r := zlozenie(r, g);
14:     m := m div 2;
```

```
15:      g := zlozenie(g, g);
16:    end
17:    return r;
```

Metoda 2d: złożoność czasowa $O(n)$

Patrząc na ograniczenia z zadania ($n \leqslant 10^6$, $m \leqslant 10^{18}$), można zaryzykować hipotezę, że złożoność czasowa i pamięciowa Algorytmu 2c jest wystarczająca. I rzeczywiście, na zawodach rozwiązania korzystające z tego podejścia (oraz Metody 1d) uzyskiwały maksymalną punktację. Pomimo tego musimy wspomnieć o tym, że również Fazę II można wykonać w optymalnej złożoności czasowej i pamięciowej $O(n)$, i taki algorytm został zastosowany w rozwiązaniu wzorcowym. Niestety algorytm ten jest istotnie bardziej skomplikowany od Algorytmu 2c, dlatego też opisujemy go w sposób skrócony. Jest on w pewnym sensie usprawnieniem Metody 2b.

Kluczowym pomysłem jest grafowa interpretacja tablicy (funkcji) f. Niech zbiorem wierzchołków grafu skierowanego $G = (V, E)$ będzie $V = \{1, 2, \ldots, n\}$, natomiast krawędzie G niech będą indukowane przez wartości funkcji f, tzn.:

$$E = \{(i, f[i]) \ : \ i \ \in V\}.$$

Ilustracja takiego grafu znajduje się już w przykładzie w treści zadania. Aby teraz obliczyć $f^m[i]$, musimy odpowiedzieć na pytanie, jaki wierzchołek leży w grafie G na końcu ścieżki o długości m prowadzącej z wierzchołka i.

Nie bez znaczenia jest tutaj szczególna własność grafu G: z każdego wierzchołka wychodzi dokładnie jedna krawędź. Tego typu grafy niejednokrotnie pojawiały się już na zawodach Olimpiady Informatycznej, ostatnio w zadaniach *Mafia* z XV Olimpiady [15], *Skarbonki* z XII Olimpiady [12] oraz *Szpiedzy* z XI Olimpiady [11]. Z tego względu pomijamy dowód faktu, że G ma następującą, specyficzną strukturę: każda jego słabo spójna składowa[1] jest cyklem, do którego doczepiona jest pewna liczba drzew skierowanych (potencjalnie zero). Dokładniej, każde takie drzewo jest skierowane od liści do korzenia, przy czym korzeń jest jednym z wierzchołków cyklu.

Całe rozwiązanie Fazy II sprowadza się do podziału grafu na opisane cykle z podoczepianymi drzewami i rozpatrzeniu osobno jednych i drugich. Oto kolejne kroki tego rozwiązania:

1. Znajdujemy wszystkie cykle w G, stosując algorytm podobny do Algorytmu 2b.
2. W każdym cyklu (reprezentowanym w tablicy, czyli z dostępem swobodnym do elementów[2]) dla każdego wierzchołka i wyznaczamy $f^m[i]$, biorąc modulo długość cyklu.
3. Usuwamy krawędzie cykli z grafu.
4. Odwracamy wszystkie pozostałe krawędzie grafu, otrzymując nowy graf $G' = (V, E')$.

[1] Słabo spójna składowa w grafie skierowanym to zbiór wierzchołków należących do jednej spójnej składowej w grafie po usunięciu skierowań krawędzi.

[2] Można skorzystać z tablic dynamicznych (patrz np. typ `vector` w języku C++), można także tablice odpowiadające wszystkim cyklom umieścić kolejno w jednej, dużej tablicy.

5. Z każdego wierzchołka należącego do jakiegokolwiek cyklu w G uruchamiamy przeszukiwanie w głąb (DFS[3]). W trakcie przeszukiwania w pomocniczej tablicy t przechowujemy wierzchołki ze znajdujących się na stosie wywołań rekurencyjnych. Na podstawie zawartości tej tablicy odpowiadamy na zapytania o $f^m[i]$.

Ostatni z powyższych punktów wymaga dodatkowego wyjaśnienia, które można znaleźć w poniższym pseudokodzie. Korzystamy w nim z pomocniczej funkcji $w\_cyklu(i, p)$, która zwraca numer wierzchołka w cyklu zawierającym wierzchołek i, oddalonego o p krawędzi od i. Dodatkowo, $t.pop\_back()$ oznacza usunięcie ostatniego elementu z tablicy t.

```
1:  Algorytm 2d, końcowe przeszukiwanie.
2:    t := ∅;
3:    procedure dfs(i)
4:    begin
5:      if size(t) ⩾ m then r[i] := t[size(t) − m + 1]
6:      else r[i] := w_cyklu(t[1], m − size(t));
7:      t.insert(i);
8:      foreach (i, j) ∈ E' do dfs(j);
9:      t.pop_back();
10:   end
```

Widzimy, że złożoność czasowa i pamięciowa każdego z powyższych kroków 1-5, a zatem także całego rozwiązania, zależy liniowo od rozmiaru grafu G, czyli jest rzędu $O(n)$.

Rozwiązania

Najefektywniejsze z opisanych rozwiązań poszczególnych faz, czyli Metody 1d i 2d, składają się na rozwiązanie wzorcowe, o złożoności czasowej $O(n)$, zaimplementowane w plikach `zab.cpp`, `zab0.pas` oraz `zab1.cpp`. Jak już wspominaliśmy, jeszcze jedna kombinacja, a mianowicie połączenie Metod 1d oraz 2c, o łącznej złożoności czasowej $O(n \log m)$, uzyskiwała maksymalną punktację. Jej implementacje można znaleźć w plikach `zabs43.cpp` oraz `zabs430.pas`. Poza tym mamy jeszcze 14 innych możliwych kombinacji par metod, z których każda daje jakieś rozwiązanie mniej efektywne. Implementacje wszystkich tych rozwiązań można znaleźć w materiałach do zadania. W zależności od złożoności czasowej uzyskiwały one na zawodach od 10 do około 60 punktów.

Wśród rozwiązań niepoprawnych warto wspomnieć o rozwiązaniu, które błędnie rozstrzyga remisy (`zabb0.cpp`, 0 punktów), oraz o rozwiązaniu nieużywającym zmiennych całkowitych 64-bitowych (`zabb1.cpp`, 40 punktów).

Testy

Wśród przygotowanych testów wyróżniamy kilka kategorii:

[3]Patrz np. książka [21].

- *test losowy* — pozycje kamieni losujemy z rozkładem jednostajnym na pewnym przedziale,
- *długa ścieżka* — funkcja f odpowiada grafowi, w którym jest jeden cykl długości 2 oraz jedna długa ścieżka prowadząca do tego cyklu,
- *długa ścieżka z dowiązaniami* — funkcja f odpowiada grafowi, w którym jest jeden cykl długości 2 oraz jedna długa ścieżka prowadząca do tego cyklu, a do każdego wierzchołka na ścieżce dochodzi ścieżka długości 1,
- *długi cykl* — funkcja f odpowiada grafowi, w którym wszystkie wierzchołki poza dwoma należą do jednego długiego cyklu.

W poniższej tabeli znajdują się opisy testów wykorzystanych na zawodach.

Nazwa	n	k	m	Opis
zab1a.in	2	1	1	przypadek brzegowy
zab1b.in	2	1	1 000 000	przypadek brzegowy
zab1c.in	3	2	1	przypadek brzegowy
zab1d.in	100	2	20	długa ścieżka z dowiązaniami
zab2a.in	2 000	3	904 194	test losowy
zab2b.in	1 500	1	8194	długa ścieżka
zab2c.in	804	3	331097	długi cykl
zab2d.in	800	2	200	długa ścieżka z dowiązaniami
zab3a.in	4 000	20	48 010	test losowy
zab3b.in	4 000	1	667 379	długa ścieżka
zab3c.in	2 832	3	759 352	długi cykl
zab3d.in	4 000	2	100	długa ścieżka z dowiązaniami
zab4a.in	10 000	12	858 448	test losowy
zab4b.in	10 000	1	2 940	długa ścieżka
zab4c.in	5 660	3	684 782	długi cykl
zab4d.in	10 000	2	200	długa ścieżka z dowiązaniami
zab5a.in	20 000	17	115 432	test losowy
zab5b.in	20 000	19 999	976 475	test losowy
zab5c.in	20 000	1	9 767	długa ścieżka
zab5d.in	20 000	2	5 000	długa ścieżka z dowiązaniami
zab6a.in	6 000	7	824 435	test losowy
zab6b.in	80 000	79 999	399 963	test losowy
zab6c.in	80 000	1	39 231	długa ścieżka
zab6d.in	80 000	2	20 000	długa ścieżka z dowiązaniami

Nazwa	n	k	m	Opis
zab7a.in	7 000	1	$\approx 6 \cdot 10^{17}$	test losowy
zab7b.in	200 000	199 999	$\approx 3 \cdot 10^{17}$	test losowy
zab7c.in	300 000	1	94 893	długa ścieżka
zab7d.in	300 000	2	60 000	długa ścieżka z dowiązaniami
zab8a.in	8 000	7	$\approx 3 \cdot 10^{17}$	test losowy
zab8b.in	400 000	399 999	$\approx 8 \cdot 10^{17}$	test losowy
zab8c.in	500 000	1	253 543	długa ścieżka
zab8d.in	600 000	2	149 000	długa ścieżka z dowiązaniami
zab9a.in	9 000	3	$\approx 5 \cdot 10^{17}$	test losowy
zab9b.in	700 000	538 348	10^{18}	test losowy
zab9c.in	834 546	1	377 651	długa ścieżka
zab9d.in	1 000 000	2	150 000	długa ścieżka z dowiązaniami
zab10a.in	10 000	8	$\approx 4 \cdot 10^{17}$	test losowy
zab10b.in	900 000	831 505	10^{18}	test losowy
zab10c.in	1 000 000	1	498 533	długa ścieżka
zab10d.in	1 000 000	2	400 000	długa ścieżka z dowiązaniami

Wojciech Rytter
Treść zadania, Opracowanie

Przemysław Horban
Program

Dostępna pamięć: 128 MB.

OI, Etap III, dzień drugi, 15.04.2010

Jedynki

Niech x będzie ciągiem zer i jedynek. **Skrajnie samotną jedynką** *(w skrócie SKS jedynką) w x jest skrajna (ostatnia lub pierwsza) jedynka, która dodatkowo nie sąsiaduje bezpośrednio z żadną inną jedynką. Na przykład, w ciągu* `10001010` *są dwie SKS jedynki, w ciągu* `1101011000` *nie ma żadnej SKS jedynki, a w ciągu* `1000` *jest tylko jedna SKS jedynka.*

Oznaczmy przez $sks(n)$ sumaryczną liczbę SKS jedynek w reprezentacjach binarnych liczb od 1 do n. Na przykład, $sks(5) = 5$, $sks(64) = 59$, $sks(128) = 122$, $sks(256) = 249$.

Chcemy przetwarzać bardzo duże liczby. Będziemy je więc reprezentować w postaci **zwartej**. *Jeśli x jest dodatnią liczbą całkowitą, $(x)_2$ jest jej zapisem binarnym (zaczynającym się od 1), to zwartą reprezentacją x jest ciąg $REP(x)$ złożony z dodatnich liczb całkowitych, które odpowiadają długościom kolejnych bloków takich samych cyfr. Na przykład:*

$$REP(460\,288) = REP(1110000011000000000_2) = (3,5,2,9)$$
$$REP(408) = REP(110011000_2) = (2,2,2,3)$$

Twoim zadaniem jest napisanie programu, który oblicza ciąg $REP(sks(n))$ na podstawie $REP(n)$.

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą k ($1 \leqslant k \leqslant 1\,000\,000$), oznaczającą długość zwartej reprezentacji dodatniej liczby całkowitej n. Drugi wiersz zawiera k liczb całkowitych $x_1, x_2, \ldots, x_k$ ($0 < x_i \leqslant 1\,000\,000\,000$) pooddzielanych pojedynczymi odstępami. Ciąg liczb $x_1, x_2, \ldots, x_k$ jest zwartą reprezentacją dodatniej liczby całkowitej n. Dodatkowo możesz założyć, że $x_1 + x_2 + \ldots + x_k \leqslant 1\,000\,000\,000$, czyli $0 < n < 2^{1\,000\,000\,000}$.

Wyjście

Twój program powinien wypisać na standardowe wyjście dwa wiersze. W pierwszym z nich powinna znajdować się jedna dodatnia liczba całkowita l. W drugim wierszu powinno znaleźć się l dodatnich liczb całkowitych $y_1, y_2, \ldots, y_l$, pooddzielanych pojedynczymi odstępami. Ciąg $y_1, y_2, \ldots, y_l$ powinien być zwartą reprezentacją liczby $sks(n)$.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`6`	`5`
`1 1 1 1 1 1`	`1 1 2 1 1`

Wyjaśnienie do przykładu: *Ciąg $1, 1, 1, 1, 1, 1$ jest zwartą reprezentacją $101010_2 = 42$, $sks(42) = 45$, natomiast $45 = 101101_2$ ma zwartą reprezentację $1, 1, 2, 1, 1$.*

Rozwiązanie

Wprowadzenie

W tym zadaniu będziemy mieli do czynienia z bardzo elementarną kombinatoryką i nietrywialną algorytmiką, wymuszoną zwartymi reprezentacjami.

Najprościej jest obliczyć osobno liczbę $ls(n)$ liczb nie większych niż n, które mają lewe samotne jedynki, oraz liczbę $ps(n)$ tych, które mają prawe samotne jedynki. Następnie należy odjąć (trywialną do obliczenia) liczbę $cs(n)$ tych liczb nie większych niż n, które mają samotne jedynki będące zarazem lewostronnymi i prawostronnymi (są to wyłącznie liczby mające zapis binarny postaci $(1000\ldots0)_2$). W ten sposób otrzymujemy wzór:

$$wynik \;=\; sks(n) \;=\; ls(n) + ps(n) - cs(n). \tag{1}$$

Jak znaleźć wzory na $ls(n)$ i $ps(n)$? Można zacząć od chwili eksperymentowania, a potem po prostu wpaść na dobre wzory (przecież na zawodach nie trzeba niczego udowadniać). Często, przynajmniej w szczególnych przypadkach, wzory bywają łatwe do dostrzeżenia, wystarczy obliczyć wynik ręcznie dla małych danych i od razu je „widać".

Przykładowo, nietrudno zauważyć (będzie to także szczególnym przypadkiem twierdzeń z następnej sekcji), że zachodzi

$$ls(2^t - 1) \;=\; ps(2^t - 1) \;=\; 2^{t-1}. \tag{2}$$

Wyprowadzenie wzorów

Oznaczmy $t = t(n) = \lfloor \log_2 n \rfloor$. Wówczas zapis binarny liczby n ma $t+1$ bitów. Niech $k = k(n)$ oznacza rozmiar zwartej reprezentacji liczby n.

Zacznijmy od oczywistego wzoru na $cs(n)$.

Twierdzenie 1.

$$cs(n) = t + 1. \tag{3}$$

Liczbę $ls(n)$ także oblicza się całkiem łatwo — jest to liczba tych liczb nieprzekraczających n, których zapis binarny zaczyna się od 10.

Twierdzenie 2. *$ls(1) = 1$, natomiast dla $n \geqslant 2$:*

(a) jeśli zapis binarny liczby n zaczyna się od 11*, to $ls(n) = 2^t$,*

(b) w przeciwnym wypadku $ls(n) = n - 2^{t-1} + 1$.

Dowód: (a) Lewą samotną jedynkę mają wszystkie liczby $(t+1)$-cyfrowe zaczynające się od 10 (takich liczb jest 2^{t-1}) oraz pewne liczby o mniej niż $t+1$ cyfrach, których jest $ls(2^t - 1) = 2^{t-1}$, co można pokazać chociażby przez indukcję. W sumie mamy 2^t liczb.

(b) Tym razem obliczymy, ile liczb nieprzekraczających n *nie ma* lewej samotnej jedynki. Wszystkie takie liczby mają co najwyżej t cyfr, a zatem jest ich dokładnie

$$2^t - 1 - ls(2^t - 1) = 2^{t-1} - 1.$$

Stąd pozostałych liczb jest $n - (2^{t-1} - 1)$. ■

Zdecydowanie najciekawsze jest wyprowadzenie wzoru na liczbę samotnych prawostronnych jedynek. Ta część rozwiązania sprawiła także najwięcej kłopotów zawodnikom, którzy próbując ją wykonać, często grzęźli w żmudnych rachunkach.

Twierdzenie 3.

$$ps(n) = \frac{1}{2}(n + k) \tag{4}$$

(Zauważmy, że równanie (2) *to szczególny przypadek równania* (4) *dla* $k = 1$*).*

Dowód: Do powyższego wzoru można dojść różnymi drogami, jedną z nich jest wielokrotne stosowanie równania (2). Wtedy trzeba posumować pewną liczbę postępów geometrycznych (potęgi dwójki). Pokażemy inną metodę, mniej rachunkową.

Niech $Z(n)$ będzie zbiorem tych x, $x \leqslant n$, które mają samotną prawostronną jedynkę. Oznaczmy $nps(n) = n - ps(n)$; jest to liczba liczb $x \leqslant n$, które nie mają samotnej prawostronnej jedynki. Zamiast równości (4) udowodnimy następującą, równoważną równość:

$$ps(n) - nps(n) \;=\; k. \tag{5}$$

Zdefiniujemy funkcję f, która każdej liczbie mającej samotną prawostronną jedynkę przyporządkowuje liczbę jej nieposiadającą. Działamy na reprezentacjach binarnych (niezwartych). Dodatkowo, dla uproszczenia dalszych rozważań, do reprezentacji wszystkich rozważanych liczb dopiszmy jedno zero wiodące. Niech teraz $(\alpha\ 01\ 0\ldots0)_2$ będzie binarną reprezentacją x (α jest tu, być może pustym, ciągiem zer i jedynek). Wtedy

$$f(x) \;=\; f\big((\alpha 01\underbrace{0\ldots0}_{l})_2\big) \;\stackrel{\text{def}}{=}\; (\alpha 11\underbrace{0\ldots0}_{l})_2.$$

Okazuje się, że f jest bijekcją. Nietrudno dostrzec, że jest różnowartościowa. Aby pokazać, że jest „na", weźmy liczbę y, która nie ma prawej samotnej jedynki. Taka liczba musi być postaci $y = (\beta 110\ldots0)_2$. Stąd oczywiście $y = f(\beta 010\ldots0)_2$.

Niech

$$Z'(n) \;=\; \{x \in Z(n) \;:\; f(x) > n\}.$$

Ponieważ dla każdego x zachodzi $f(x) > x$, więc f odwzorowuje (w sposób różnowartościowy) zbiór $Z(n) \setminus Z'(n)$ na cały zbiór $\{1, 2, \ldots, n\} \setminus Z(n)$. Zatem wystarczy teraz udowodnić, że $|Z'(n)| = k$.

Zauważmy, że:

$$Z'(n) \;=\; \{(\alpha 01\ \underbrace{0\ldots0}_{t(n)-|\alpha|})_2 \;:\; \alpha 01 \text{ lub } \alpha 10 \text{ jest prefiksem } n \text{ w zapisie binarnym}\}.$$

Uzasadnienie powyższej równości nie jest trudne. $Z'(n)$ składa się bowiem z dokładnie takich x, że $x \leqslant n < f(x)$, czyli

$$(\alpha 01 \underbrace{0\ldots0}_{t(n)-|\alpha|})_2 \leqslant n < (\alpha 11 \underbrace{0\ldots0}_{t(n)-|\alpha|})_2.$$

Ile, wobec tego, liczb zawiera $Z'(n)$? Każda z nich odpowiada zapisowi $(n)_2$ w postaci $\alpha 01\gamma$ lub $\alpha 10\gamma$ dla pewnych ciągów zerojedynkowych α i γ [1]. Takich podziałów jest dokładnie tyle, ile bloków w reprezentacji binarnej n (bez zer wiodących), czyli $k(n)$. Stąd

$$|Z'(n)| = k = k(n).$$

Przykład.

$$\begin{aligned} Z'(0111000111000_2) \;=\; \{\; & 0100000000000_2,\; 0111000100000_2, \\ & 0110100000000_2,\; 0111000110100_2 \;\}. \end{aligned}$$

■

Arytmetyka zwartych reprezentacji

Istotnym elementem zadania jest przetwarzanie olbrzymich liczb w postaci zwartej. Ponieważ wszystko jest związane tu jakoś z potęgami dwójki, więc złożoność czasowa operacji na zwartych reprezentacjach będzie liniowa ze względu na ich rozmiar. Zauważmy, że do obliczenia wyniku — wzór (1), do którego podstawiamy równości podane w Twierdzeniach 1, 2 i 3 — potrzebujemy jedynie zaimplementować operacje: dodawania, odejmowania, dzielenia przez 2 i obliczania reprezentacji liczb postaci 2^t. Dwie ostatnie są już naprawdę proste, więc opiszemy jedynie dodawanie i odejmowanie.

Dodawanie zwartych reprezentacji zaimplementujemy, wzorując się na dodawaniu pisemnym. Zapiszmy sumowane liczby a i b jedna pod drugą, wyrównując do prawej, do dłuższej z tych liczb dopiszmy na początku jedno zero wiodące, a do krótszej — tyle zer wiodących, żeby ich długości wyrównały się. *Miejscami podziału* przy sumowaniu nazwiemy wszystkie granice między kolejnymi blokami takich samych cyfr, czy to w ramach liczby a, czy b.

Sumę $a + b$ będziemy obliczać od prawej do lewej, rozpatrując fragmenty dodawanych liczb między kolejnymi miejscami podziału. Zauważmy, że wówczas problem sprowadza nam się do zsumowania dwóch bloków, z których każdy jest złożony tylko z zer lub tylko z jedynek; musimy też wziąć pod uwagę bit (cyfrę) przeniesienia, wynikły z sumowania poprzednich bloków. W wyniku otrzymujemy nowy blok lub ewentualnie dwa bloki, a także bit przeniesienia, którego używamy przy sumowaniu kolejnej pary bloków, patrz rys. 1.

Widać wyraźnie, że stosując powyższy przepis, możemy obliczyć zwartą reprezentację liczby $a + b$ w czasie $O(k(a) + k(b))$, i co więcej, $k(a + b) = O(k(a) + k(b))$. Przykład działania tego algorytmu znajduje się na rys. 2.

[1] Przypomnijmy, że przed pierwszą jedynką w reprezentacji rozważanych liczb dopisaliśmy 0.

blok 1	**blok 2**	**bit**	**wynikowy blok**	**wynikowy bit**
0...00	0...00	0	0...00	0
0...00	0...00	1	0...01	0
0...00	1...11	0	1...11	0
0...00	1...11	1	0...00	1
1...11	1...11	0	1...10	1
1...11	1...11	1	1...11	1

Rys. 1: Tabelka przedstawiająca algorytm sumowania bloków, z których każdy jest złożony z takich samych cyfr.

$$\begin{aligned} a: &\quad 0\ 11\ 111\ 0\ 11\ 11\ 000\ 00\ 1\ 1111\ 00\ 000\ 1 \\ b: &\quad 0\ 00\ 111\ 1\ 11\ 00\ 000\ 11\ 0\ 1111\ 11\ 000\ 0 \\ a+b: &\quad 1\ 00\ 111\ 0\ 10\ 11\ 001\ 00\ 0\ 1110\ 11\ 000\ 1 \end{aligned}$$

Rys. 2: Obliczanie $a + b$ metodą blokową; $k(a) = 7$, $k(b) = 6$, $k(a + b) = 15$. Odstępy reprezentują miejsca podziału.

Jeśli umiemy dodawać zwarte reprezentacje, to odejmowanie możemy zaimplementować za pomocą dodawania. Załóżmy, że chcemy obliczyć $a - b$, przy czym wiemy, że $a - b > 0$ oraz $b > 0$. Niech $T = t(a) + 2$. Zaczynamy od obliczenia $c = a + 2^T - 1 - b$, co robimy w ten sposób, że do liczby a dodajemy liczbę $2^T - 1 - b$; jeśli $REP(b) = (x_1, \ldots, x_k)$, to $REP(2^T - 1 - b) = (T - t(b), x_1, \ldots, x_k)$. Żądanym wynikiem jest teraz $c + 1 - 2^T$. Dodanie 1 wykonujemy jak zwykłe dodawanie. Zauważmy dalej, że $c+1-2^T < a < 2^{T-1}$, więc odjęcie 2^T od $c+1$ wykonujemy poprzez usunięcie początkowej jedynki oraz wszystkich następujących po niej bezpośrednio zer (co najmniej jednego), co odpowiada usunięciu pierwszych dwóch liczb ze zwartej reprezentacji liczby $c + 1$. Kto nie wierzy, niech sprawdzi na przykładzie.

Rozwiązanie wzorcowe, wykorzystujące arytmetykę na zwartych reprezentacjach liczb, jest zaimplementowane w plikach `jed.c`, `jed1.cpp`, `jed2.pas`, `jed3.cpp` oraz, wyjątkowo krótko, w pliku `jed4.cpp`. Wykonuje ono $O(1)$ operacji o koszcie liniowym, więc jego złożoność czasowa wynosi $O(k)$.

Rozwiązania nieoptymalne

Najprostsze rozwiązanie zadania *Jedynki* polega na przejrzeniu wszystkich liczb od 1 do n i zliczeniu napotkanych SKS jedynek. Zostało ono zaimplementowane w pliku `jeds1.pas` i zdobywało na zawodach 20 punktów.

W pliku `jeds0.c` znajduje się implementacja rozwiązania korzystającego z takich samych wzorów co rozwiązanie wzorcowe, ale wykonującego operacje jedynie na zmiennych całkowitych 64-bitowych. Takie rozwiązania zdobywały na zawodach 40 punktów.

Rozwiązanie z pliku `jeds2.cpp` również wykorzystuje opisane wzory, jednakże do ich obliczania używa standardowej (a nie zwartej) arytmetyki dużych liczb. Za tego typu rozwiązania można było zdobyć na zawodach 60-70 punktów.

Testy

Rozwiązania zadania *Jedynki* były sprawdzane na 10 zestawach testowych, opisanych w poniższej tabelce. Parametr n oznacza liczbę reprezentowaną w teście, $k = k(n)$ to długość zwartej reprezentacji n, natomiast S to $t(n)+1$, czyli suma liczb x_i z wejścia. W opisach testów skróty NPJ i PJ oznaczają liczby n rozpoczynające się w zapisie binarnym odpowiednio od 11 i od 10, natomiast DW i MW oznaczają odpowiednio dużą lub małą wariancję liczb występujących w $REP(n)$.

Nazwa	k	n lub S	Opis
jed1a.in	5	$n < 10^6$	NPJ, DW — test losowy
jed1b.in	10	$S = 10$	PJ, MW — liczba $(1010\ldots)_2$
jed1c.in	1	$n = 1$	PJ, MW — przypadek brzegowy
jed2a.in	9	$n < 10^6$	NPJ, MW — test losowy
jed2b.in	2	$n = 2^{15}$	PJ, DW
jed2c.in	1	$n = 1$	PJ, MW — przypadek brzegowy
jed3a.in	10	$S = 55$	PJ, DW — test losowy, posortowany
jed3b.in	5	$S = 31$	NPJ — malejący ciąg potęg 2
jed4a.in	10	$S = 55$	NPJ, MW — test losowy
jed4b.in	3	$n = 2^{59} + 2^{58} - 1$	PJ, DW
jed5a.in	200	$S = 1\,000$	PJ, DW — test losowy, posortowany
jed5b.in	50	$S = 1\,000$	NPJ, MW — test losowy
jed6a.in	300	$S = 10\,000$	PJ, MW — test losowy
jed6b.in	100	$S = 10\,000$	NPJ, DW — test losowy
jed7a.in	500	$S = 10^5$	NPJ, DW — test losowy
jed7b.in	2	$n = 2^{100\,000}$	PJ, DW
jed8a.in	1 000	$S = 10^7$	PJ, MW — test losowy, posortowany
jed8b.in	10 000	$S = 10^7$	NPJ, DW — test losowy
jed9a.in	100 000	$S = 10^8$	PJ, MW — test losowy
jed9b.in	100 000	$S = 10^8$	NPJ, DW — test losowy
jed10a.in	1 000 000	$S = 10^9$	PJ, MW — test losowy
jed10b.in	1 000 000	$S = 10^9$	NPJ, DW — test losowy
jed10c.in	1	$n = 2^{(10^9)} - 1$	NPJ, MW — przypadek brzegowy

Szymon Acedański, Jakub Radoszewski — Treść zadania, Opracowanie
Tomasz Kulczyński — Program

Dostępna pamięć: 64 MB. *OI, Etap III, dzień drugi, 15.04.2010*

Mosty

San Bajcisko to pięknie położone nadmorskie miasto. Składa się ono z n niewielkich, ale gęsto zaludnionych wysp, ponumerowanych od 1 do n. Pewne pary wysp są połączone mostami, którymi poprowadzono dwukierunkowe drogi. Każda para wysp może być połączona co najwyżej jednym mostem. Z każdej wyspy San Bajciska można dotrzeć do dowolnej innej, przemieszczając się pomiędzy wyspami wyłącznie za pomocą mostów.

Bajtazar i Bajtek wybierają się na przejażdżkę rowerową po San Bajcisku. Swoją wycieczkę zaczną na wyspie nr 1. Chcą zwiedzić miasto, przejeżdżając każdym mostem dokładnie raz, i zakończyć podróż na wyspie nr 1. Niestety, w wycieczce będzie im przeszkadzał... wiatr. Otóż na każdym moście strasznie wieje, co **w różnym stopniu** *może utrudniać przejazd rowerowy mostem* **w jedną bądź w drugą stronę**. *Dla uproszczenia zakładamy, że przy ustalonym kierunku przejazdu przez dany most siła wiatru jest stała.*

Pomóż Bajtazarowi i Bajtkowi znaleźć taką trasę spełniającą ich wymagania, po przejechaniu której będą najmniej zmęczeni. Jako miarę tego, na ile trasa jest męcząca, przyjmujemy maksymalną siłę wiatru, z jaką nasi bohaterowi będą musieli zmagać się na trasie.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie liczby całkowite oddzielone pojedynczym odstępem: n oraz m ($2 \leqslant n \leqslant 1\,000$, $1 \leqslant m \leqslant 2\,000$), oznaczające odpowiednio liczbę wysp oraz liczbę mostów w San Bajcisku. Wyspy są ponumerowane od 1 do n, a mosty od 1 do m. W kolejnych m wierszach znajdują się opisy mostów. $(i+1)$-szy wiersz zawiera cztery liczby całkowite a_i, b_i, l_i, p_i ($1 \leqslant a_i, b_i \leqslant n$, $a_i \neq b_i$, $1 \leqslant l_i, p_i \leqslant 1\,000$), pooddzielane pojedynczymi odstępami. Liczby te opisują most nr i łączący wyspy o numerach a_i oraz b_i. Siła wiatru, jaki przeszkadza w trakcie przejazdu rowerem z wyspy a_i do b_i, to l_i; jeżeli zaś przejechać tym samym mostem z wyspy b_i do a_i, to przeszkadzający wiatr będzie miał siłę p_i.

Wyjście

Jeżeli nie istnieje żadna trasa spełniająca wymagania naszych bohaterów, to pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać jedno słowo `NIE`. *W przeciwnym przypadku wyjście powinno składać się z dwóch wierszy, opisujących najmniej męczącą trasę wycieczki po San Bajcisku. Pierwszy z nich powinien zawierać jedną liczbę całkowitą: największą siłę wiatru, jaki będzie wiał w trakcie wycieczki. Właśnie tę liczbę należy zminimalizować. W drugim wierszu powinno znaleźć się m liczb całkowitych, pooddzielanych pojedynczymi odstępami i oznaczających numery mostów, którymi kolejno przejeżdża się na najmniej męczącej trasie.*

Jeśli istnieje więcej niż jedna poprawna odpowiedź, Twój program powinien wypisać dowolną z nich.

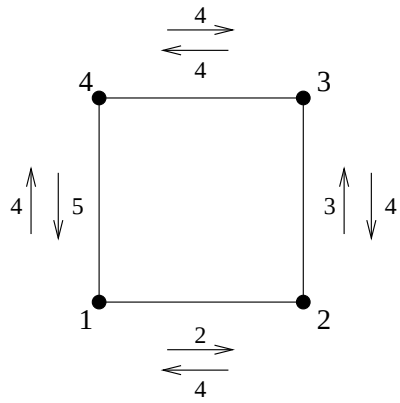
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
4
4 3 2 1
```



Wyjaśnienie do przykładu: *Optymalna trasa Bajtazara i Bajtka ma postać:* $1 \xrightarrow{4} 4 \xrightarrow{4} 3 \xrightarrow{4} 2 \xrightarrow{4} 1$. *Maksymalna siła wiatru na tej trasie wynosi 4.*

Rozwiązanie

Wprowadzenie

Z każdym rokiem Olimpiady Informatycznej dowiadujemy się coraz to nowych rzeczy o Bajtocji, Bajtazarze i jego przyjaciołach. Z niecierpliwością czekamy na czasy, w których już w szkołach podstawowych będzie się recytować na lekcjach ogólnie znane fakty z geografii fizycznej, gospodarczej, a teraz szczególnie z klimatologii Bajtocji.

Z historii wiemy, że za czasów dobrobytu sieć połączeń drogowych w Bajtocji była grafem gęstym, później nieraz starano się zredukować ją do drzewa, a lepiej nie myśleć, co stało się w związku z niedawnym kryzysem na światowych rynkach. Często podawano, że Bajtocja jest położona na wyspie w kształcie wielokąta wypukłego, niejednokrotnie wspominano też coś o estakadach, tunelach i mostach. Analizowany tekst źródłowy umacnia nas w przekonaniu o istnieniu mostów, co więcej, zdradza kolejne tajemnice: że Bajtocja składa się z wielu wysp (!), które w samym San Bajcisku tworzą wraz z mostami graf spójny[1] $G = (V_G, E_G)$, o n wierzchołkach i m krawędziach. Co więcej, Bajtazar postanowił każdej z krawędzi e_i przypisać dwie wagi: l_i, p_i, oznaczające wysiłek potrzebny do pokonania jej rowerem w każdą ze stron. Wraz z Bajtkiem poszukuje on takiego cyklu Eulera[2], żeby maksymalna waga krawędzi napotkanej na tym cyklu, w kierunku przechodzenia po tym cyklu, była możliwie najmniejsza.

[1] Mówimy, że graf nieskierowany jest *spójny*, jeżeli z każdego wierzchołka da się dojść do każdego innego.

[2] *Cykl Eulera* to cykl przechodzący przez każdą krawędź grafu dokładnie raz (ta definicja obowiązuje niezależnie od tego, czy graf jest skierowany, czy nie).

Co prawda z tekstu nie wiemy na 100%, czy San Bajcisko w ogóle jest miastem, ani tym bardziej czy leży w Bajtocji, ale czymże innym mogłoby być!?

Zarys algorytmu wzorcowego

Zacznijmy od przytoczenia klasycznego kryterium istnienia cyklu Eulera w grafach nieskierowanych. Dodajmy na wstępie, iż w całym opracowaniu będziemy zakładać, że wszystkie rozważane grafy nie zawierają wierzchołków izolowanych. Ich usunięcie z grafu nie wpływa bowiem na istnienie cykli Eulera, a ułatwia zwięzłe formułowanie twierdzeń.

Twierdzenie 1 (Eulera dla grafów nieskierowanych). *Graf nieskierowany jest* eulerowski *(tzn. ma cykl Eulera) wtedy i tylko wtedy, gdy jest spójny oraz każdy jego wierzchołek ma parzysty stopień*[3].

A zatem, jeśli wejściowy graf nie spełnia tego kryterium (przypomnijmy, że warunku spójności sprawdzać w zadaniu nie trzeba), możemy od razu być pewni, że za żadne skarby Bajtazar nie przejedzie po San Bajcisku tak, jak to sobie wymarzył. Co jeśli jednak cykl Eulera istnieje? Jak wybrać ten poszukiwany przez Bajtazara? Nazwijmy *k-cyklem* każdy cykl Eulera, w którym na wszystkich mostach wieje nie bardziej niż k (w kierunku przechodzenia po cyklu). Zadanie polega na znalezieniu najmniejszego k takiego, że w danym grafie istnieje k-cykl.

Obserwacja 1. Załóżmy, że w treści pewnego zadania znajduje się jedno z określeń: „znajdź minimum", „maksimum", „najmniejszą" bądź „największą wartość". Wówczas szansa na rozwiązanie tego zadania nie maleje, jeśli w głowie rozwiązującego pojawi się pomysł zastosowania techniki wyszukiwania binarnego po wyniku.

Technika ta była już kilkukrotnie opisywana w książeczkach Olimpiady. Polega w tym przypadku na znalezieniu w ciągu liczb $1, 2, \ldots, 1\,000$ szukanej najmniejszej liczby k przy użyciu wyszukiwania binarnego. Potrzebujemy więc wykonać $O(\log w)$ zapytań o to, czy w grafie istnieje k-cykl, przy czym w jest maksymalną wagą krawędzi na wejściu.

Gdy już znajdziemy najmniejsze k oraz będziemy wiedzieć, w którą stronę należy przejechać każdym z mostów (o tym w kolejnym punkcie), pozostanie wyznaczyć cykl Eulera w grafie skierowanym. W tym miejscu pozwolimy sobie polecić książkę [26], w której można znaleźć więcej informacji na temat cykli Eulera, jak również algorytm ich znajdowania[4] o złożoności czasowej liniowej względem liczby wierzchołków i krawędzi grafu.

Wykonywanie zapytań o k-cykle

Sprowadziliśmy nasze zadanie do problemu ustalenia, czy w grafie istnieje k-cykl dla danego k. Zacznijmy od wyeliminowania prostego przypadku:

[3] *Stopniem* wierzchołka v w grafie nieskierowanym nazywamy liczbę krawędzi incydentnych z tym wierzchołkiem (oznaczenie: $deg(v)$).

[4] Można o tym także posłuchać na stronie `http://was.zaa.mimuw.edu.pl/?q=node/31`

Obserwacja 2. Jeśli k jest mniejsze niż zarówno l_i, jak i p_i dla pewnego i, to w grafie nie istnieje k-cykl.

Jeśli ta obserwacja nie wydaje się Czytelnikowi oczywista, prosimy o ponowne jej przeczytanie, z niniejszym zdaniem włącznie. Skorzystanie z tego spostrzeżenia wyeliminuje nam zbyt małe wartości k, ale nie pozwoli na udzielenie odpowiedzi na zapytanie w każdym przypadku. Rozważmy więc dla danego k częściowo skierowany graf G_k, który otrzymujemy z wyjściowego grafu G, skierowując te krawędzie, przez które można przejechać tylko w jednym kierunku, bo w przeciwnym wieje mocniej niż k. Na rys. 1 po lewej stronie przedstawiono graf G_4 dla przykładowych danych wejściowych z treści zadania. Zauważmy, że w oryginalnym grafie G istnieje k-cykl wtedy i tylko wtedy, gdy da się skierować wszystkie pozostałe krawędzie grafu G_k, tak aby otrzymany graf skierowany był eulerowski. Intuicyjnie, w G_k odrzuciliśmy tylko takie kierunki przechodzenia wzdłuż krawędzi, które nie mogą wystąpić w żadnym k-cyklu.

Definicja 1. *Orientacją* grafu częściowo skierowanego H nazwijmy dowolny graf skierowany, który możemy otrzymać, skierowując nieskierowane krawędzie grafu H.

Odtąd będziemy zastanawiali się już tylko nad tym, jak znaleźć eulerowską orientację G'_k grafu G_k. Chcemy zatem, aby graf G'_k spełniał następujące, również klasyczne kryterium.

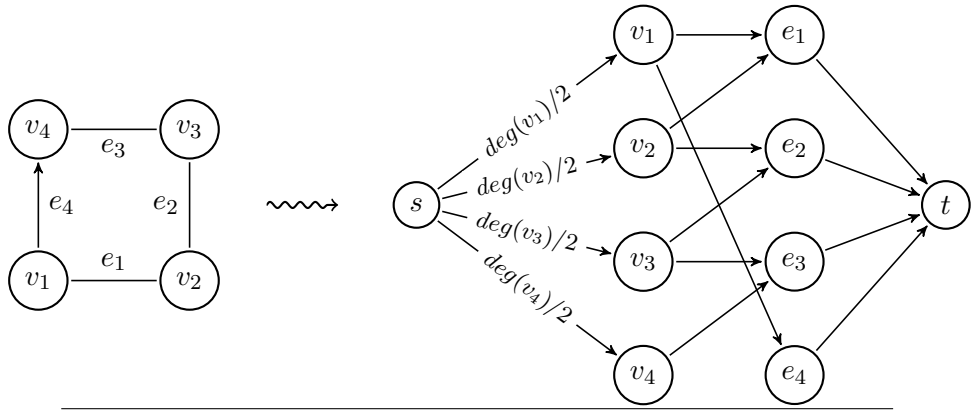
Twierdzenie 2 (Eulera dla grafów skierowanych). *Graf skierowany jest eulerowski wtedy i tylko wtedy, gdy jest silnie spójny*[5] *oraz każdy jego wierzchołek ma stopień wchodzący równy stopniowi wychodzącemu*[6].

W powyższym kryterium pojawiają się dwa warunki, analogiczne do tych z kryterium dla grafów nieskierowanych (Twierdzenie 1). Przypomnijmy jednak, że w oryginalnym grafie warunek spójności mieliśmy zagwarantowany w treści zadania. Na pierwszy rzut oka nie widać, czy pomaga to w jakiś sposób w kwestii silnej spójności wersji skierowanej grafu. Okazuje się jednak, że pomaga, a nawet gwarantuje tę silną spójność, o czym świadczy poniższa obserwacja. W jej dowodzie posługujemy się pojęciem silnie spójnych składowych (patrz np. książka [21]), którego znajomość na pewno ułatwia jego lekturę, ale można spróbować przeczytać ten dowód także bez tej wiedzy.

Obserwacja 3. Załóżmy, że G jest nieskierowanym grafem eulerowskim, a G' jest jego orientacją, w której każdy wierzchołek ma stopień wchodzący równy stopniowi wychodzącemu. Wówczas G' jest silnie spójny, a więc, w szczególności (patrz Twierdzenie 2), G' jest eulerowski.

[5] Mówimy, że graf skierowany jest *silnie spójny*, jeżeli z każdego wierzchołka da się dojść do każdego innego.

[6] *Stopniem wejściowym* (*wyjściowym*) wierzchołka v w grafie skierowanym nazywamy liczbę krawędzi wchodzących do tego wierzchołka (odpowiednio, wychodzących z niego). Oznaczenia: $indeg(v)$ — stopień wejściowy, $outdeg(v)$ — stopień wyjściowy.



Rys. 1: Przykład konstrukcji sieci przepływowej z grafu częściowo skierowanego. Krawędzie nieoznaczone mają przepustowość jednostkową.

Dowód: Załóżmy przez sprzeczność, że G' nie jest silnie spójny. Podzielmy graf G' na tzw. *silnie spójne składowe* — są to maksymalne podgrafy grafu, które są silnie spójne. Silnie spójne składowe każdego grafu są, oczywiście, rozłączne wierzchołkowo. Wiadomo, że w podziale muszą wystąpić co najmniej dwie takie składowe, gdyż w przeciwnym przypadku G' byłby silnie spójny.

Zauważmy, że w G' istnieje jakaś krawędź (u, v) łącząca dwa wierzchołki z różnych silnie spójnych składowych. Faktycznie, gdyby żadna taka krawędź nie istniała, to oryginalny graf G nie byłby spójny.

Skonstruujmy w grafie G' maksymalną, czyli nieprzedłużalną, ścieżkę p, rozpoczynającą się od krawędzi (u, v), która nie przechodzi żadną krawędzią więcej niż raz (przez jeden wierzchołek już może). Ścieżka p musi kończyć się w wierzchołku u, jako że z każdego innego wierzchołka grafu możemy zawsze wyjść jakąś krawędzią, gdyż zawsze mamy tyle samo krawędzi wychodzących co wchodzących. To oznacza, że p jest cyklem, który przechodzi przez wierzchołki u i v z dwóch różnych silnie spójnych składowych. To jednak nie jest możliwe (dlaczego?). Mamy żądaną sprzeczność. ■

Dzięki tej obserwacji możemy skupić się już tylko na warunku dotyczącym stopni wejściowych i wyjściowych wierzchołków w wynikowej orientacji G'_k. Postaramy się tak skierować krawędzie grafu G_k, aby:

- każdej krawędzi przypisać jeden z wierzchołków z nią incydentnych — będzie to jej początek we w pełni skierowanym grafie,
- zapewnić, że każdy z wierzchołków będzie przypisany do tylu krawędzi, ile wynosi połowa jego stopnia w wyjściowym grafie G; w ten sposób każdy z wierzchołków będzie miał stopień wejściowy równy wyjściowemu.

Skonstruujemy w tym celu sieć przepływową[7] H_k, patrz rys. 1. Składa się ona z:

- wierzchołków v_i (dla $i \in \{1, 2, \ldots, n\}$) odpowiadających wierzchołkom G_k,

[7] *Sieć przepływowa* to graf skierowany, w którym każda krawędź ma przypisaną liczbę (zwaną przepustowością) oraz istnieją dwa wyróżnione wierzchołki: *źródło* i *ujście*.

- wierzchołków e_j (dla $j \in \{1, 2, \ldots, m\}$) odpowiadających krawędziom G_k,
- krawędzi $v_i \to e_j$ o jednostkowych przepustowościach, jeśli e_j jest krawędzią nieskierowaną, a v_i jest z nią incydentny,
- krawędzi $v_i \to e_j$ o jednostkowych przepustowościach, jeśli e_j jest krawędzią skierowaną wychodzącą z v_i,
- wierzchołka s (źródła) połączonego z każdym v_i krawędzią o przepustowości równej połowie stopnia v_i w wyjściowym grafie G,
- ujścia t połączonego ze wszystkimi e_j krawędziami jednostkowymi.

Znajdźmy *maksymalny przepływ* w tak skonstruowanej sieci, np. przy użyciu algorytmu Edmondsa-Karpa opisanego w książce [21] [8]. Jeśli w znalezionym przepływie krawędź $v_i \to e_j$ jest nasycona, a e_j jest nieskierowana w G_k, to skierujmy ją tak, by wychodziła z v_i. Jeśli wszystkie krawędzie incydentne ze źródłem (lub, równoważnie, z ujściem) są nasycone, to w grafie G'_k stopnie wejściowe i wyjściowe wszystkich wierzchołków są równe. Wówczas i tylko wówczas (zachęcamy Czytelnika do samodzielnego uzasadnienia tej równoważności!) graf G'_k ma cykl Eulera, czyli początkowy graf posiada k-cykl.

Rozwiązanie wzorcowe – podsumowanie

To kończy opis rozwiązania wzorcowego, zaimplementowanego w plikach `mos.cpp`, `mos1.pas` oraz `mos[3-5].cpp`. Warto zastanowić się jeszcze nad jego złożonością czasową. W tym celu skupimy się przede wszystkim na analizie algorytmu przepływowego.

Zauważmy, że graf $H_k = (V_k, E_k)$ ma $O(n + m)$ wierzchołków i krawędzi, czyli $|V_k| = O(n + m)$ i $|E_k| = O(n + m)$. Podręcznikowa złożoność czasowa algorytmu Edmondsa-Karpa to $O(|V_k| \cdot |E_k|^2)$, czyli całkiem sporo. Ponieważ przepustowości są w naszym przypadku całkowitoliczbowe, więc możemy skorzystać z innego oszacowania złożoności czasowej tego algorytmu, a mianowicie $O((|V_k| + |E_k|) \cdot P)$, przy czym P to wynikowa wartość maksymalnego przepływu. W przypadku grafu H_k mamy oczywiście $P \leqslant m$, co pozwala oszacować koszt czasowy algorytmu przepływowego przez $O(m \cdot (n + m))$.

Przypomnijmy, że w rozwiązaniu wzorcowym wykonujemy $O(\log w)$ kroków, w każdym z których konstruujemy graf H_k (czas $O(n + m)$) i uruchamiamy nasz algorytm przepływowy. Po wykonaniu wszystkich kroków znamy optymalne skierowanie krawędzi, na podstawie którego możemy znaleźć wynikowy cykl Eulera w czasie $O(n+m)$. Stąd, złożoność czasową rozwiązania wzorcowego możemy oszacować przez $O(m \cdot (n + m) \cdot \log w)$. Dodajmy, że złożoność pamięciowa to $O(n + m)$.

Takie rozwiązanie można zaimplementować także w nieco lepszej złożoności czasowej: $O(m \cdot (n + m))$. W tym celu należy zastąpić wyszukiwanie binarne stopniowym dodawaniem do sieci przepływowej krawędzi typu $v_i \to e_j$ odpowiadających

[8] Musimy skorzystać z algorytmu, który dla całkowitych przepustowości konstruuje przepływ o całkowitych wartościach na każdej krawędzi. Tę własność mają jednak wszystkie powszechnie stosowane metody wyznaczania maksymalnego przepływu.

coraz większym wagom (l_i lub p_i). Po każdej dodanej krawędzi próbujemy zwiększać maksymalny przepływ za pomocą ścieżek powiększających w algorytmie Edmondsa-Karpa (zwiększenie przepływu o jednostkę wykonujemy w czasie $O(n + m)$). Krok ten powtarzamy tak długo, aż osiągniemy maksymalny przepływ nasycający wszystkie krawędzie wchodzące do ujścia — wówczas stwierdzamy, że waga ostatnio dodanej krawędzi stanowi koszt optymalnej trasy Bajtazara, a samą trasę odczytujemy ze znalezionego przepływu dokładnie tak jak w rozwiązaniu wzorcowym. Dodajmy tylko, że to usprawnienie nie było wymagane do uzyskania maksymalnej punktacji za zadanie.

Rozwiązanie bez użycia przepływu: skojarzenie

Jeszcze kilka lat temu algorytmy przepływowe nie mieściły się w zakresie kompetencji oczekiwanych od zawodników Olimpiady Informatycznej. Okazuje się, że i tutaj można obejść się bez nich.

Można bowiem zamiast sieci przepływowej rozważyć graf dwudzielny, podobny do przedstawionej sieci, lecz bez źródła i ujścia, oraz z wierzchołkami v_i powielonymi $deg(v_i)/2$ razy. Taki graf dwudzielny ma $O(m)$ wierzchołków oraz

$$O\left(\sum_{v \in V_G} (deg(v))^2\right) \;=\; O\left(\sum_{v \in V_G} (n \cdot deg(v))\right) \;=\; O(n \cdot m)$$

krawędzi. Rozmiar najliczniejszego skojarzenia w tym grafie jest równy maksymalnemu przepływowi w opisywanej sieci przepływowej. Skojarzenie wskazuje również, w jaki sposób należy zorientować poszczególne krawędzie.

Najliczniejsze skojarzenie można wyznaczyć algorytmem Hopcrofta-Karpa (opisanym np. w książce [26]), uzyskując rozwiązanie zadania w czasie $O(m^{1,5} \cdot n \cdot \log w)$, które również powinno zmieścić się w limitach czasowych. Można także użyć algorytmu *turbo-matching* opisanego w opracowaniu zadania *Cło* z XV Olimpiady Informatycznej [15]. Jego wykorzystanie daje co prawda złożoność czasową $O(m^2 \cdot n \cdot \log w)$, lecz w praktyce ten algorytm często działa szybciej od algorytmu Hopcrofta-Karpa. I faktycznie, rozwiązanie naszego zadania oparte na tej metodzie, zaimplementowane w pliku `moss1.cpp`, formalnie wolniejsze, w tym zadaniu na wszystkich testach sprawdza się lepiej od wzorcowego.

Jeszcze inne rozwiązanie

Część czytelników mogłaby nam zarzucić, że przecież skojarzenie w grafie dwudzielnym to prawie to samo co przepływ, więc w powyższym rozwiązaniu *de facto* nie unikamy „algorytmów przepływowych", czymkolwiek właściwie one są. Autorzy tego opracowania poniekąd podzielają tę opinię. Dlatego w tej sekcji opiszemy rozwiązania, które *naprawdę* unikają korzystania z pojęcia przepływu. Skorzystamy w nich z podobnych pomysłów, jak te, które pojawiły się w rozwiązaniu wzorcowym zadania *Kości* z XII Olimpiady Informatycznej [12].

Rozważania rozpocznijmy od tego miejsca w rozwiązaniu wzorcowym, w którym wykoncypowaliśmy już możliwość wykorzystania wyszukiwania binarnego i koncen-

trujemy się właśnie na grafie częściowo skierowanym G_k, dla którego poszukujemy eulerowskiej orientacji G'_k. Wiemy, że wystarczy w tym celu tak skierować nieskierowane krawędzie grafu G_k, aby w wynikowym grafie G'_k każdy wierzchołek miał taki sam stopień wejściowy i wyjściowy. *Zrównoważeniem* wierzchołka v w grafie skierowanym, $\Delta(v)$, nazwijmy różnicę między jego stopniem wyjściowym a wejściowym, tzn.

$$\Delta(v) \stackrel{\text{def}}{=} outdeg(v) - indeg(v).$$

Używając tego pojęcia, możemy po prostu powiedzieć, że w wynikowej orientacji G'_k zrównoważenia wszystkich wierzchołków muszą być zerowe; nazwijmy każdy graf skierowany spełniający ten warunek *zbalansowanym*.

Poszukiwania takiego grafu skierowanego możemy rozpocząć w dość zabawny sposób, a mianowicie, losowo skierowując wszystkie nieskierowane krawędzie G_k. Istnieje wówczas szansa, że nam się poszczęściło i że otrzymany w ten sposób graf G''_k jest zbalansowany. Liczenie na ten przypadek stanowi jednak nadmiar optymizmu: w G''_k mogą wystąpić także wierzchołki o dodatnim bądź ujemnym zrównoważeniu. Możemy jednak założyć, że zrównoważenia wszystkich wierzchołków w G''_k są parzyste — w przeciwnym przypadku od razu wiemy, że G_k nie da się odpowiednio skierować.

Jeśli G''_k nie jest zbalansowany, to możemy spróbować coś poprawić, tak aby stał się zbalansowany. Wykonajmy następujący krok:

(*) *Wybierzmy w G''_k jakikolwiek wierzchołek v^+, taki że $\Delta(v^+) > 0$, i sprawdźmy, czy w G''_k istnieje ścieżka z v^+ do jakiegokolwiek wierzchołka v^- o ujemnym zrównoważeniu, przechodząca jedynie krawędziami, które w oryginalnym grafie G_k były* nieskierowane*. Jeśli taka ścieżka istnieje, to odwróćmy wszystkie krawędzie w niej zawarte.*

W wyniku kroku (*) zmniejszamy zrównoważenie v^+ o 2, zwiększamy zrównoważenie v^- również o 2, natomiast zrównoważenia wszystkich pozostałych wierzchołków pozostają niezmienione. Takie postępowanie ewidentnie przybliża nas do celu, czyli do uzyskania grafu zbalansowanego. Faktycznie, jeśli uda nam się powtórzyć krok (*) odpowiednio dużo razy, a mianowicie:

$$R \stackrel{\text{def}}{=} \sum_{v \,:\, \Delta(v)>0} \frac{\Delta(v)}{2} \tag{1}$$

razy, to otrzymany graf będzie zbalansowany. Pytanie brzmi, czy zachodzi implikacja odwrotna, czyli czy każdy graf niezbalansowany można sprowadzić do zbalansowanego, wykonując krok (*) odpowiednio dużo razy, zakładając oczywiście, że w ogóle istnieje jakakolwiek zbalansowana orientacja. Okazuje się, że ta implikacja zachodzi, co pokazuje następujące twierdzenie (zapisane w sposób dość abstrakcyjny). Od razu zaznaczmy, że wnioskiem z Twierdzenia 3 jest poprawność algorytmu poprawiającego losowo skierowany graf G''_k za pomocą R-krotnego wykonania kroku (*) — złożonością tego algorytmu zajmiemy się zaraz po udowodnieniu twierdzenia.

Twierdzenie 3. *Niech $H = (V_H, E_H)$ będzie grafem częściowo skierowanym oraz niech H'' będzie dowolną niezbalansowaną orientacją grafu H. Niech dalej v^+ będzie dowolnym wierzchołkiem H'' o dodatnim zrównoważeniu. Jeśli H ma jakąś zbalansowaną orientację, to w H'' istnieje ścieżka z v^+ do jakiegoś wierzchołka o ujemnym zrównoważeniu, przechodząca jedynie krawędziami, które w H były nieskierowane.*

Dowód: Niech H' będzie obiecaną, zbalansowaną orientacją grafu H. Niech dalej F będzie zbiorem krawędzi, które są w H'', ale nie ma ich w H'. Zbiór F reprezentuje zatem te krawędzie nieskierowane grafu H, których skierowanie jest inne w H'' niż w H'. Zdefiniujmy nowy graf $H''' = (V_H, F)$.

Ciekawą własnością grafu H''' jest to, że znaki zrównoważeń wierzchołków są w nim takie same jak w H'' (dodatnie zrównoważenia przechodzą na dodatnie, ujemne na ujemne, zerowe na zerowe). Intuicyjnie, zbiór F pokazuje, którym krawędziom z H'' należy odwrócić skierowanie, tak aby ten graf stał się zbalansowany — np. w przypadku wierzchołka o dodatnim zrównoważeniu więcej krawędzi wychodzących trzeba przerobić na wchodzące niż odwrotnie, wchodzących na wychodzące. Formalny dowód tego spostrzeżenia pomijamy.

W szczególności wiemy, że v^+ ma dodatnie zrównoważenie w H'''. Ponadto, jeśli w H''' znajdziemy ścieżkę z v^+ do jakiegoś wierzchołka o ujemnym zrównoważeniu w H''', to będzie ona reprezentować ścieżkę w H'' z v^+ do wierzchołka o ujemnym zrównoważeniu w H'', przechodzącą jedynie krawędziami, które w H są nieskierowane. Znajdując taką ścieżkę, zakończymy dowód. Istnienie takiej ścieżki gwarantuje Obserwacja 4, dla niepoznaki zapisana i udowodniona tuż pod niniejszym dowodem. Używając tej obserwacji, kończymy dowód twierdzenia. ■

Obserwacja 4. Niech H będzie grafem skierowanym oraz niech v^+ będzie wierzchołkiem tego grafu o dodatnim zrównoważeniu. Wówczas z v^+ istnieje ścieżka do jakiegoś wierzchołka o ujemnym zrównoważeniu.

Dowód: W dowodzie tej obserwacji, podobnie jak w dowodzie występującej kilka stron wcześniej Obserwacji 3, skonstruujemy w grafie H maksymalną (nieprzedłużalną) ścieżkę p prowadzącą z wierzchołka v^+, w której żadna krawędź nie powtarza się, choć wierzchołki już mogą. Zastanówmy się, gdzie może się ta ścieżka skończyć.

Na pewno p *nie* może skończyć się w samym wierzchołku v^+, jako że ma on więcej krawędzi wychodzących niż wchodzących, więc po każdym powrocie do niego mamy jeszcze krawędź, którą możemy wyjść. Podobnie uzasadniamy, że ścieżka ta *nie* może skończyć się w żadnym innym wierzchołku o nieujemnym zrównoważeniu. Ponieważ p nie może być nieskończona, więc musi skończyć się w wierzchołku o ujemnym zrównoważeniu. ■

Przyszła pora na obiecaną analizę złożoności czasowej podanego rozwiązania. Konstrukcja losowej orientacji G''_k grafu G_k wymaga, oczywiście, czasu $O(n+m)$. Równie szybko możemy obliczyć zrównoważenia wierzchołków w tej orientacji i sprawdzić, czy przypadkiem któreś z nich nie jest nieparzyste — a jeśli tak, natychmiast zwrócić odpowiedź `NIE`. Krok (*) łatwo zrealizować za pomocą dowolnego algorytmu przeszukiwania grafu (np. BFS, DFS, patrz książka [21]) — daje to także koszt czasowy $O(n+m)$. Krok ten musimy wykonać R razy, przy czym łatwo widać, że $R \leqslant m$. Dokładając do tego zewnętrzne wyszukiwanie binarne, otrzymujemy oszacowanie złożoności czasowej algorytmu: $O(m \cdot (n+m) \cdot \log w)$, czyli takie samo jak w przypadku rozwiązania wzorcowego. Implementację tego rozwiązania można znaleźć w pliku `mos6.cpp`.

Można lepiej

Okazuje się, że pomysły z poprzedniej sekcji również można wykorzystać do konstrukcji rozwiązania o złożoności czasowej $O(m \cdot (n+m))$. Istota tego podejścia jest jednak nieco inna niż w odpowiadającej modyfikacji rozwiązania wzorcowego.

Przedstawimy najpierw nieefektywną realizację tego innego podejścia. Składa się ono z następujących kroków:

1. Zaczynamy standardowo, czyli od sprawdzenia, czy wyjściowy graf G (ten reprezentujący wyspy i mosty pomiędzy nimi) jest eulerowski, patrz Twierdzenie 1. Jeśli nie, to problem z głowy, a jeśli tak, to przechodzimy dalej.

2. Sortujemy krawędzie grafu G *nierosnąco* względem *maksimum* z wag l_i, p_i oznaczających siłę wiatru wiejącego w każdą ze stron. W ten sposób otrzymujemy uporządkowaną listę krawędzi L.

3. Niech G' oznacza graf częściowo skierowany, początkowo równy G (wszystkie krawędzie nieskierowane).

4. Przeglądamy kolejne krawędzie $e \in L$, próbując dla każdej z nich wybrać tańsze z dwóch skierowań, odpowiadające mniejszej sile wiatru (albo dowolne skierowanie, jeśli oba są równie tanie).

   a) W ten właśnie sposób skierowujemy krawędź e w grafie G', po czym sprawdzamy, czy po wykonaniu tego kroku G' ma jakąś orientację eulerowską. To sprawdzenie możemy wykonać na dowolny z trzech przedstawionych w tym opracowaniu sposobów (przepływ, skojarzenie w grafie dwudzielnym albo iteracyjne poprawianie orientacji).

   b) Jeśli tak, to kontynuujemy przeglądanie listy L, a jeśli nie, to kończymy.

Twierdzimy, że ostatnia orientacja eulerowska, jaką otrzymamy — albo po przejrzeniu całej listy L, albo tuż przed skierowaniem jakiejś krawędzi w G', które psuje możliwość znalezienia orientacji eulerowskiej — stanowi szukaną, najtańszą trasę Bajtazara.

Uzasadnienie tego stwierdzenia jest następujące. Niech k_0 oznacza koszt najtańszego cyklu Eulera w grafie G, tzn. maksimum z sił wiatru na krawędziach w kierunku przechodzenia wzdłuż cyklu. Jeśli $l_i > k_0$ dla pewnej, i-tej krawędzi, to musi zachodzić $p_i \leqslant k_0$ i wówczas musimy przejść tą krawędzią w kierunku odpowiadającym p_i. Symetrycznie, jeśli $p_i > k_0$, to i-tą krawędzią trzeba przejść w kierunku odpowiadającym l_i. To oznacza, że po przetworzeniu początkowego fragmentu listy L odpowiadającego krawędziom o maksimum wag większym niż k_0, w grafie G' musi wciąż istnieć orientacja eulerowska — i nasz algorytm ją znajdzie. Dalej będzie być może znajdował jeszcze jakieś inne orientacje, z których żadna nie będzie miała maksymalnej wagi większej niż k_0, gdyż krawędzie umożliwiające uzyskanie wagi przekraczającej k_0 już odpowiednio skierowaliśmy. Stąd, ostatnia znaleziona orientacja ma koszt nie gorszy niż k_0, czyli optymalny, co kończy nasze uzasadnienie.

Wiemy już, że algorytm opisany krokami 1-4 jest poprawny. Co więcej, pozbyliśmy się w nim wyszukiwania binarnego, ale za to przysporzyliśmy sobie dodatkowego

czynnika m, wynikłego z wielokrotnego wykonywania kroku 4a. Czynnika tego pozbędziemy się jednak równie łatwo, jak go wprowadziliśmy: wystarczy, że nie będziemy wyznaczać orientacji eulerowskiej za każdym razem od nowa.

W tym celu będziemy cały czas utrzymywać orientację eulerowską G'' grafu G'. Na początku (krok 3) wyznaczamy ją w czasie $O(n + m)$ za pomocą klasycznego algorytmu na cykl Eulera w grafie nieskierowanym. Musimy teraz pokazać, jak zmieniać orientację eulerowską po wykonaniu skierowania krawędzi e w kroku 4a. Jeśli wykonane skierowanie jest takie samo jak skierowanie e w G'', to nic nie musimy robić. Jeśli jest przeciwne, to musimy odwrócić odpowiadającą krawędź w G'', czym na pewno zaburzymy zbalansowanie orientacji G'': wówczas zrównoważenia końców tej krawędzi będą równe 2 i -2, ale zrównoważenia wszystkich pozostałych wierzchołków pozostaną zerowe. Po tej poprawce graf G'' jest więc niezbalansowaną orientacją nowego grafu G'. Jeśli da się ją poprawić do orientacji zbalansowanej, to na mocy Twierdzenia 3, wystarczy do tego jednokrotne wykonanie kroku (*). To oznacza, że krok 4a możemy wykonać w czasie $O(n + m)$, co pokazuje, że cały algorytm da się w ten sposób zaimplementować w złożoności czasowej $O(m \cdot (n + m))$.

Testy

Rozwiązania tego zadania sprawdzano na 10 zestawach testów, po 3-5 testów w zestawie:

- Testy *a* to testy specyficzne, reprezentujące grafy, które albo są klikami (*3a*, *5a*, *9a*), albo mają mało cykli Eulera (a przynajmniej mało tanich cykli, cokolwiek to znaczy).
- Testy *b* reprezentują losowe grafy mające cykl Eulera.
- Testy *c* mają, w większości, odpowiedź `NIE`.
- Testy *d* (prócz *2d*) reprezentują grafy złożone z cykli prostych o jednym wspólnym wierzchołku. Na każdym z tych cykli w optymalnym rozwiązaniu tylko jedna krawędź ma wymuszone skierowanie.
- W testach *e* oraz w teście *2d* zbiory wag na krawędziach: $\{l_i\}$ oraz $\{p_i\}$, są rozłączne, co wpływa ujemnie na czas działania niektórych rozwiązań powolnych tego zadania.

Nazwa	n	m
mos1a.in	8	14
mos1b.in	8	12
mos1c.in	11	20
mos2a.in	19	28
mos2b.in	20	31
mos2c.in	20	34

Nazwa	n	m
mos2d.in	25	80
mos3a.in	5	10
mos3b.in	41	53
mos3c.in	38	60
mos3d.in	45	55
mos4a.in	35	60

Nazwa	n	m
mos4b.in	78	81
mos4c.in	80	100
mos4d.in	86	102
mos5a.in	25	300
mos5b.in	101	300
mos5c.in	100	250
mos5d.in	127	147
mos6a.in	210	1 567
mos6b.in	200	313
mos6c.in	250	400
mos6d.in	345	387
mos7a.in	500	2 000
mos7b.in	499	1 000
mos7c.in	502	510
mos7d.in	401	480
mos7e.in	45	990

Nazwa	n	m
mos8a.in	750	1 000
mos8b.in	658	800
mos8c.in	800	1 700
mos8d.in	871	900
mos8e.in	800	1 800
mos9a.in	63	1 953
mos9b.in	997	1 157
mos9c.in	1 000	2 000
mos9d.in	601	650
mos9e.in	1 000	1 800
mos10a.in	1 000	1 800
mos10b.in	1 000	2 000
mos10c.in	1 000	1 000
mos10d.in	801	1 000
mos10e.in	1 000	2 000

Piotr Chrząstowski — Treść zadania, Opracowanie
Wojciech Śmietanka — Program

Dostępna pamięć: 64 MB. *OI, Etap III, dzień drugi, 15.04.2010*

Piloci

Piloci w Bajtockim Ośrodku Treningowym przygotowują się do pełnienia misji wymagających niezwykłej precyzji i opanowania. Miernikiem sukcesu jest uzyskanie możliwie długiego lotu wzdłuż zadanej trasy bez zbytnich odchyleń — chodzi o utrzymanie w miarę równego kursu. Jest to zadanie trudne, gdyż symulator jest tak czuły, że rejestruje najmniejsze nawet ruchy wolanta[1]*. Zapisuje przy tym jeden parametr zwany wychyleniem. Przed każdym eksperymentem treningowym piloci mają ustawiony poziom tolerancji t; ich celem jest wykonanie możliwie długiego fragmentu lotu o tej własności, że wszystkie pomiary wychyleń w tym czasie będą różniły się nie więcej niż o t. Innymi słowy, fragment lotu od momentu i do momentu j uznamy za mieszczący się w ramach tolerancji t, jeśli wychylenia, począwszy od i-tego pomiaru a skończywszy na j-tym, tworzą taki ciąg $a_i, \ldots, a_j$, że dla każdych a_k, a_l z tego ciągu zachodzi $|a_k - a_l| \leqslant t$.*

Twoim zadaniem jest napisanie programu, który na podstawie tolerancji t i ciągu pomiarów wychyleń wyznaczy długość najdłuższego fragmentu tego ciągu mieszczącego się w ramach zadanej tolerancji.

Wejście

W pierwszym wierszu standardowego wejścia zapisane są dwie liczby całkowite t oraz n ($0 \leqslant t \leqslant 2\,000\,000\,000$, $1 \leqslant n \leqslant 3\,000\,000$), oddzielone pojedynczym odstępem i określające poziom tolerancji oraz liczbę dokonanych pomiarów wychyleń. W drugim wierszu znajdują się pooddzielane pojedynczymi odstępami wartości zmierzonych wychyleń, będące liczbami całkowitymi z zakresu od 1 do 2 000 000 000.

Wyjście

Twój program powinien wypisać w pierwszym i jedynym wierszu standardowego wyjścia jedną liczbę całkowitą: maksymalną długość fragmentu lotu mieszczącego się w ramach zadanej tolerancji.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`3 9` `5 1 3 5 8 6 6 9 10`	`4`

Wyjaśnienie do przykładu: *Mamy dwa najdłuższe fragmenty, oba długości 4:* `5, 8, 6, 6` *oraz* `8, 6, 6, 9`.

[1] urządzenie sterujące w lotnictwie

Rozwiązanie

Rozwiązanie naiwne o koszcie $O(n^4)$

Oznaczmy przez $a_1, a_2, \ldots, a_n$ ciąg kolejnych wartości zmierzonych wychyleń. Powiemy, że przedział $[i, j]$ jest *t-stabilny*, jeśli dla zadanego parametru t spełnia warunki zadania, czyli dla każdych indeksów i', j' takich, że $i \leqslant i' \leqslant j' \leqslant j$, zachodzi $|a_{i'} - a_{j'}| \leqslant t$. Naszym zadaniem jest wyznaczenie długości najdłuższego t-stabilnego przedziału dla danej stałej t.

Na pierwszy rzut oka narzuca się naiwne rozwiązanie polegające na zbadaniu wszystkich par indeksów i, j i wybraniu takiej, która definiuje najdłuższy stabilny segment.

```
1: rekord := 1; { długość najdłuższego stabilnego segmentu dotąd znalezionego }
2: for i := 1 to n do
3:   for j := i + 1 to n do
4:     if t-stabilny(i, j) then
5:       if j − i + 1 > rekord then rekord := j − i + 1;
```

Jest to wyjątkowo paskudne rozwiązanie. Jego koszt wynosi $O(n^2)$ razy koszt wykonania funkcji *t-stabilny*. Mamy bowiem $\frac{n(n-1)}{2}$ par indeksów (i, j) takich, że $i < j$. Jeśli teraz funkcja *t-stabilny* będzie realizowana dla każdego segmentu od nowa, to przy najprymitywniejszym rozwiązaniu każde jej wywołanie zajmie $O(n^2)$ sprawdzeń, czy któraś z par indeksów badanego segmentu nie zaburza warunku stabilności. To dałoby algorytm o łącznym koszcie $O(n^4)$. Wierząc, że każdy czytelnik tego opracowania, być może z obrzydzeniem, ale poradziłby sobie z tym rozwiązaniem, porzućmy ten przykry temat.

Rozwiązanie trochę mniej naiwne, o koszcie $O(n^3)$

Rzecz jasna, natychmiast można o jeden rząd wielkości ten wynik poprawić. Wystarczy zauważyć, że do stwierdzenia t-stabilności nie jest wymagane przebadanie wszystkich par z danego przedziału, a jedynie stwierdzenie, jaka jest w nim minimalna i maksymalna wartość, i sprawdzenie, czy różnica między nimi nie przekracza t. To da się zrobić w koszcie $2n - 2$, a nawet, jak się trochę postaramy, w koszcie nieprzekraczającym $\frac{3}{2}n - 2$ (o szczegółach optymalnego algorytmu wyznaczającego jednocześnie minimum i maksimum można poczytać w książce [18]). Stosując ten algorytm, obniżymy koszt do $O(n^3)$, ale to nas wcale jeszcze nie zadowala — rozwiązanie tej złożoności otrzymywało na zawodach 20-30 pkt. Jego implementacja znajduje się w plikach `pils[6-9].[cpp|pas]`.

Rozwiązanie przyzwoite $O(n^2)$

Zauważmy, że jeśli różnica $|a_i - a_j|$ dla $i < j$ przekracza t, to ani a_i nie może być częścią stabilnego ciągu zawierającego jakikolwiek element za a_j, ani a_j nie może być częścią

jakiegokolwiek stabilnego ciągu zawierającego element występujący przed a_i. Innymi słowy, jeśli przedział $[i, j]$ nie jest stabilny, a jednocześnie $i' \leqslant i, j \leqslant j'$, to przedział $[i', j']$ też nie jest stabilny. Oznacza to, że jeśli dla danego początku przedziału i będziemy posuwali się do przodu końcem przedziału, czyli indeksem j, i stwierdzimy po raz pierwszy, że przedział $[i, j]$ nie jest stabilny, to możemy zaprzestać dalszej analizy dla tego i: wszystkie dalsze indeksy końca przedziału też dadzą negatywny wynik i na pewno rekordu nie pobiją. Zauważmy też, że dla ustalonego i możemy, idąc z j do przodu, na bieżąco aktualizować wartości minimalną i maksymalną z danego przedziału. Po prostu, zwiększając j, sprawdzamy, czy nowa wartość a_j staje się nowym minimum lub maksimum. W ten sposób otrzymujemy dość prosty algorytm kwadratowy:

```
 1: i := 1;
 2: rekord := 1;
 3: while i ⩽ n − rekord do begin
 4: { nie ma co badać większych i, gdyż wtedy takie zbyt duże i }
 5: { nie może być początkiem rekordowego przedziału }
 6:   min := A[i];   { najmniejsza... }
 7:   max := A[i]; { i największa wartość z badanego przedziału }
 8:   j := i + 1;
 9:   stabilny := true;
10:   while (j ⩽ n) and stabilny do begin
11:     if A[j] > max then max := A[j]
12:     else if A[j] < min then min := A[j];
13:     if max − min > t then stabilny := false
14:     else j := j + 1;
15:   end
16:   if j − i > rekord then rekord := j − i;
17:   { nie musieliśmy tego warunku sprawdzać w każdym obrocie pętli; }
18:   { wszak teraz mamy w ręce najlepsze j dla danego i }
19: end
```

Zauważmy, że tu zawsze kończymy z j o 1 za dużym — jest to najmniejszy indeks, dla którego nie ma przedziału stabilnego $[i, j]$. Zatem różnica $j - i$ daje nam poszukiwaną liczbę elementów najdłuższego segmentu zaczynającego się w i.

Ponieważ mamy tu dwie zagnieżdżone pętle, a każda z nich wykonuje co najwyżej liniową liczbę obrotów, więc koszt tego rozwiązania jest kwadratowy. Można było za nie otrzymać ok. 40 pkt. Przykładowa implementacja w pliku `pils3.pas`.

Rozwiązanie jeszcze przyzwoitsze $O(n \log n)$

Dalsze obniżanie kosztu jest możliwe, ale potrzebne są nieco bardziej zaawansowane techniki. Skoro ważne są tylko elementy maksymalny i minimalny z danego przedziału, to wystarczy, jeśli będziemy umieli je szybko wyznaczać. Możemy zatem iść dwoma wskaźnikami: i oraz j w prawo, i jeśli element a_j psuje stabilność, to pchnąć do przodu i, a jeśli nie, to pchnąć j. W tym celu warto użyć sprytnej struktury danych

do reprezentowania multizbioru (czyli zbioru z możliwością powtórzeń elementów) odpowiadającego danemu przedziałowi $[i, j]$, i to takiej, że wyznaczenie minimum i maksimum w multizbiorze jest szybkie.

Jeśli do reprezentacji multizbioru użyjemy drzew zrównoważonych, np. AVL lub czerwono-czarnych[1], to w każdym węźle będziemy poza wartością przechowywali jej krotność, czyli ile razy pojawia się w reprezentowanym multizbiorze. Drzewa AVL mają logarytmiczną wysokość, a elementy minimalny i maksymalny znajdują się na końcu ścieżek odpowiednio samych lewych i samych prawych synów, idąc od korzenia, więc dostęp do nich jest logarytmiczny.

Pseudokod algorytmu jest dość prosty:

```
 1: i := 1;
 2: TwórzMultizbiórAVL(A[1]); { tworzy drzewo AVL z elementem A[1] }
 3: rekord := 1;
 4: min := A[1];
 5: max := A[1];
 6: j := 2;
 7: while j ⩽ n do begin
 8:    WstawAVL(A[j]);
 9:    if A[j] > max then max := A[j]
10:    else if A[j] < min then min := A[j];
11:    while max − min > t do begin
12:    { jeśli A[j] nie zmieniło min lub max lub jeśli zmieniło, ale nie tak, }
13:    { żeby utracić t-stabilność, to pętla ta nie wykona się ani razu }
14:       UsuńAVL(A[i]);
15:       Aktualizuj(min, max);
16:       i := i + 1;
17:    end
18:    j := j + 1;
19:    if j − i > rekord then rekord := j − i;
20: end
```

Aktualizację *min* i *max* związaną z usuwaniem elementów $A[i]$ można zrobić sprytniej, modyfikując odpowiednio procedury *WstawAVL* i *UsuńAVL*, ale nie będziemy się nad tym specjalnie rozwodzili. Tak czy siak, nawet w takiej wersji taka aktualizacja ma koszt co najwyżej logarytmiczny.

W rezultacie dostaliśmy algorytm o liniowej liczbie obrotów pętli. Indeks i lub j wzrasta o 1 przy każdym obrocie każdej pętli; nieistotne tu jest to, że jedna jest zagnieżdżona w drugiej — łącznie mogą one wykonać się co najwyżej $2n-2$ razy. W każdym obrocie pętli, dzięki zastosowaniu drzew AVL, mamy co najwyżej logarytmiczny koszt wykonania wszystkich znajdujących się w niej operacji, stąd końcowy koszt $O(n \log n)$. Implementację takiego rozwiązania można znaleźć w pliku `pils2.cpp`.

[1] O drzewach zrównoważonych można przeczytać m.in. w książkach [19, 21]. Programujący w C++ mogli użyć na zawodach gotowej implementacji multizbioru za pomocą drzewa czerwono-czarnego, a mianowicie klasy `multiset`.

Warto w tym miejscu dodać, że istnieją także inne rozwiązania niniejszego zadania o złożoności czasowej $O(n \log n)$, które różnią się głównie zastosowaną strukturą danych. Szukana struktura danych musi umożliwiać efektywne wyznaczanie minimów i maksimów w poszczególnych segmentach wyjściowego ciągu. Dla przykładu, bardzo łatwo jest do tego celu zaadaptować statyczne drzewo przedziałowe[2], przechowujące wszystkie elementy wyjściowego ciągu. Szczegóły tego rozwiązania pozostawiamy do dopracowania Czytelnikowi. Dodajmy tylko, że drzewa przedziałowe są nieco bardziej efektywne od opisanej wcześniej reprezentacji multizbioru: mają mniejszą stałą w złożoności czasowej i pamięciowej. Przykłady implementacji w plikach `pils4.cpp` i `pils5.pas`.

Rozwiązania o koszcie $O(n \log n)$ otrzymywały na zawodach od ok. 60 do ponad 90 punktów w zależności od zużycia pamięci oraz stałej ukrytej w notacji O.

Rozwiązania liniowe

Zauważmy, że trzymanie całego multizbioru odpowiadającego przedziałowi $[i, j]$ nie jest konieczne. Tak naprawdę potrzebne jest nam przede wszystkim ustalenie, jakie są wartości minimum i maksimum w zadanym przedziale. Jeśli więc będziemy w stanie zrobić to szybko, to cała reszta wartości multizbioru staje się nieistotna. Podamy teraz dwa różne rozwiązania o liniowym koszcie.

Rozwiązanie liniowe specyficzne dla tego zadania

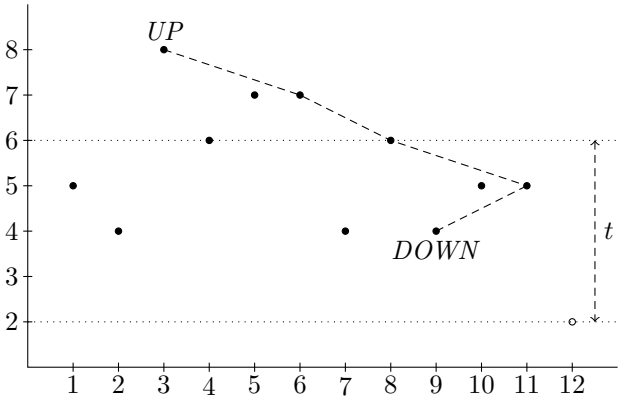
Określmy dla każdego $j \in \{1, \ldots, n\}$ wartość najmniejszego indeksu i, dla którego przedział $[i, j]$ jest t-stabilny. Oznaczmy tę wartość przez $p[j]$. Szukamy najdłuższego przedziału t-stabilnego, więc możliwym rozwiązaniem jest wyznaczenie $p[j]$ dla każdego j i zbadanie, która z par wartości $(p[j], j)$ definiuje najdłuższy przedział, czyli czemu jest równe maksimum z wartości $j - p[j] + 1$. Niech $njm(i, j)$ oraz $njw(i, j)$ oznaczają odpowiednio najmniejszą i największą wartość z przedziału domkniętego $[i, j]$.

Zastanówmy się nad odpowiedzią, myśląc indukcyjnie. To jest sposób, za pomocą którego można dojść do wielu efektywnych rozwiązań. Metoda ta polega w naszym przypadku na przeglądaniu kolejnych elementów ciągu a_j dla $j = 2, \ldots, n$ i opracowaniu schematu przejścia od $j - 1$ do j. Załóżmy indukcyjnie, że znamy wartość $p[j-1]$ i zastanawiamy się, jak wykorzystać tę wiedzę przy wyznaczaniu $p[j]$. Konieczne jest tu dostrzeżenie, dlaczego wartość $p[j]$ miałaby być inna niż $p[j-1]$. Może się tak zdarzyć tylko wtedy, gdy element a_j psuje stabilność, czyli gdy

$$A[j] - njm(p[j-1], j-1) > t \quad \text{albo} \quad njw(p[j-1], j-1) - A[j] > t.$$

Gdy żadna z tych sytuacji nie zachodzi, to $p[j]$ po prostu równa się $p[j-1]$. Natomiast gdy tak dobrze nie jest, powinniśmy znaleźć najwcześniejszy indeks wartości, która nie destabilizuje ciągu kończącego się w j. Rozważmy, wobec tego, trzy przypadki:

[2] Jest to struktura danych zbudowana nad ciągiem, pozwalająca (w podstawowej wersji) w czasie $O(\log n)$ obliczać wyniki dowolnego działania łącznego (na przykład sumy, minimum, maksimum) na zadanym przedziale tego ciągu oraz równie szybko aktualizować jego wyrazy. Więcej na temat drzew przedziałowych można znaleźć np. w opracowaniu zadania *Latarnia* w tej książeczce oraz w podanych tam odnośnikach.



Rys. 1: Ilustracja przykładu (1).

1. Jeżeli $A[j-1] > A[j]$, to zagrożenie jest tylko z góry. Element $A[j]$ może się okazać zbyt mały dla pewnych elementów z segmentu $[p[j-1], j-1]$. Zatem powinniśmy w takim przypadku znaleźć najmniejszy w przedziale $[p[j-1], j-1]$ indeks k taki, że żaden z elementów $A[k], \ldots, A[j-1]$ nie przekracza $A[j]$ o więcej niż t. Dobrze byłoby więc znać indeksy malejących lewostronnych maksimów w tym przedziale, poczynając od wartości największej w całym przedziale (lewostronne maksimum w przedziale to taka wartość, że każda dalsza jest od niej mniejsza).

2. Jeżeli $A[j-1] < A[j]$, to zagrożenie jest tylko z dołu. Tym razem powinniśmy, analogicznie do poprzedniego przypadku, mieć pod ręką indeksy rosnących lewostronnych minimów w tym przedziale, poczynając od najmniejszej wartości w całym przedziale (lewostronne minimum w przedziale to taka wartość, że każda dalsza jest od niej większa).

3. Jeżeli $A[j] = A[j-1]$, to nie musimy nic robić poza sprawdzeniem, czy nie pobiliśmy rekordu — w tym przypadku bowiem zachodzi $p[j] = p[j-1]$.

Widać więc, że kluczem do sukcesu jest tu posiadanie informacji o dwóch specyficznych ciągach: lewostronnych maksimów i lewostronnych minimów w przedziale $[p[j-1], j-1]$. W obu przypadkach, poza samymi wartościami, trzeba też będzie pamiętać ich indeksy, bo wartości występujące w ciągach wcale nie muszą być sąsiadami w oryginalnym ciągu wychyleń.

Przykładowo, dla $t = 4$ i stabilnego dla tej wartości ciągu

$$5, 4, 8, 6, 7, 7, 4, 6, 4, 5, 5 \tag{1}$$

interesujące nas malejące wartości lewostronnych maksimów to $8, 7, 6, 5$ z odpowiadającymi im indeksami $3, 6, 8, 11$, a rosnące lewostronne minima to $4, 5$ z odpowiadającymi im indeksami $9, 11$. Tutaj, rzecz jasna, $p[11] = 1$.

Teraz wydłużenie ciągu o liczbę o indeksie 12 — być może psującą stabilność ciągu — może spowodować modyfikację każdego z tych ciągów. Gdybyśmy, na przykład, do

naszego ciągu wejściowego dodali na końcu 2 na pozycji dwunastej, to w tym konkretnym przypadku pierwszy z ciągów dla nowego, krótszego przedziału kończącego na tej dwójce stanie się równy $6, 5, 2$, o indeksach $8, 11, 12$, zaś drugi stanie się jednoelementowy: 2, o indeksie 12. Dla wydłużonego o dwójkę ciągu wartość $p[12]$ będzie równa 7 — ciąg 4-stabilny kończący się nowo wstawioną dwójką zaczyna się od przedostatniej czwórki. Zauważmy tu ciekawą rzecz. Tak naprawdę nie jest dla nas istotne to, ile wynosi aktualnie $p[j]$. Powinniśmy je wyznaczyć tylko wtedy, gdy przy wzroście j pozostawiamy t-stabilny przedział z tym samym początkiem, co dla $j-1$. Jedynie wtedy możemy pobić rekord.

Dokładnie rzecz ujmując, definiujemy dla każdego j ciąg malejący lewostronnych maksimów UP_j następująco: pierwszym jego elementem jest największa wartość z przedziału $[p[j], j]$ występująca możliwie najpóźniej (czyli, spośród potencjalnie wielu równych największych elementów, ten o największym indeksie). Kolejne elementy tego ciągu tworzą ostatnie wystąpienia wartości mniejszych od poprzednich elementów, o ile spełnią warunek, że aż do końca (czyli do j) wszystkie elementy są od nich mniejsze. Ciąg rosnący lewostronnych minimów $DOWN_j$ definiujemy analogicznie, poczynając od ostatniego wystąpienia elementu najmniejszego z przedziału $[p[j], j]$.

Zauważmy, że oba ciągi, o które dbamy, pozwalają w dość prosty sposób wyznaczyć wartość $p[j]$. W przypadku pierwszego z nich — malejącego, wszystkie jego wartości przekraczające o więcej niż t wartość $A[j]$ odpadają wraz z tymi elementami, które znajdują się przed nimi. Zatem kandydatem na $p[j]$ jest indeks o jeden większy od ostatniego elementu w ciągu UP_j, który psuje stabilność. Jeśli zatem pierwszy element tego ciągu nie jest zbyt duży, to na tym kończymy (tzn. wykorzystamy indeks obliczony w ten sposób dla $j-1$). Jeśli natomiast pierwszy element przerasta $A[j]$ o więcej niż t, to usuwamy z początku ciągu UP_j wszystkie elementy zbyt duże i zatrzymujemy się na pierwszym, którego wartość jest większa od $A[j]$ o co najwyżej t, pamiętając powiększony o jeden indeks ostatnio usuniętego elementu. Może się zdarzyć, że wszystkie elementy ciągu zostaną z niego usunięte.

Analogicznie postępujemy w przypadku drugiego ciągu, wyznaczając drugiego kandydata na $p[j]$. Obaj kandydaci są pierwszymi elementami, które nie protestują z powodu pojawienia się $A[j]$. Są to elementy o indeksach o 1 większych od ostatnio usuniętych wartości. Pierwszy z nich gwarantuje, że jest najmniejszym indeksem takim, że wszystkie wartości pomiędzy nim a j nie przekraczają $A[j]$ o więcej niż t. Natomiast drugi, że jest najmniejszym indeksem takim, że wszystkie wartości między nim a j są mniejsze od $A[j]$ o co najwyżej t. Wartość $p[j]$, a więc nowe i, jest więc równa większemu z tych dwóch kandydatów.

Teraz pora na aktualizację ciągów UP i $DOWN$ z drugiej strony. Nowa wartość $A[j]$ musi stać się ostatnim elementem każdego z tych ciągów. W przypadku ciągu malejącego powinniśmy zatem usunąć z końca wszystkie mniejsze wartości i — jeśli ta wartość jeszcze nie występowała — dodać na końcu wartość $A[j]$, a jeśli występowała, to nic nie dodawać, tylko zaktualizować indeks ostatniego wystąpienia tej wartości. W przypadku drugiego ciągu — analogicznie powinniśmy usunąć wszystkie większe wartości i dodać ją na końcu, jeśli jej jeszcze nie było, a jeśli była, to też zaktualizować indeks ostatniego wystąpienia. Wykonanie tej czynności zagwarantuje nam zachowanie niezmiennika pętli: ciągi UP i $DOWN$ będą spełniały warunki definicji,

a jednocześnie będziemy mieli informację o ostatnim wystąpieniu tej wartości.

Z punktu widzenia jednorodności kodu warto dopiero teraz sprawdzić, czy nowy element $A[j]$ wydłużył najdłuższy do tej pory zauważony ciąg stabilny. Wystarczy w tym celu sprawdzić, czy $j - i$ nie pobiło rekordu.

Nasz algorytm rozbija się zatem na trzy fazy:

1. Aktualizacja ciągów *UP* i *DOWN* z początku.
2. Aktualizacja ciągów *UP* i *DOWN* z końca.
3. Aktualizacja rekordu.

Oba nasze ciągi wygodnie jest zrealizować za pomocą tzw. kolejki dwustronnej, czyli struktury, która pozwala dodawać i usuwać elementy z obu stron. Konkretnie chodzi nam o następujące operacje, przy założeniu, że elementami kolejki będą rekordy o dwóch polach: *val* i *ind*, reprezentujących odpowiednio wartość i indeks danego elementu ciągu:

- Inicjuj(q) — procedura inicjująca kolejkę q na pustą,
- Pusta(q) — funkcja sprawdzająca pustość kolejki q,
- Pierwszy(q) — funkcja przekazująca jako wynik pierwszy element kolejki q,
- UsuńPierwszy(q) — funkcja usuwająca pierwszy element kolejki q i przekazująca go jako wynik,
- Ostatni(q) — funkcja przekazująca jako wynik ostatni element kolejki q,
- UsuńOstatni(q) — funkcja usuwająca ostatni element kolejki q i przekazująca go jako wynik,
- WstawNaKoniec(q, x) — procedura wstawiająca na koniec kolejki q element x.

Można taką strukturę danych zrealizować np. za pomocą listy dwukierunkowej albo w tablicy za pomocą bufora cyklicznego[3]. Każda z podanych operacji dokonuje się w tych implementacjach w czasie stałym.

Pseudokod rozwiązania liniowego

W pseudokodzie zmienna *ost_up* wskazuje na indeks ostatniego elementu ciągu *UP*, który zaburzał stabilność od góry, zaś *ost_down* wskazuje na indeks ostatniego elementu ciągu *DOWN*, który zaburzał stabilność od dołu. Wszystkie wyliczenia złożonych warunków logicznych są leniwe, czyli przerywane, gdy tylko wynik stanie się wiadomy. Tak właśnie dzieje się w programach napisanych m.in. w C++ i Pascalu.

```
1: Inicjuj(UP); { inicjujemy ciąg malejący na pusty }
2: Inicjuj(DOWN); { inicjujemy ciąg rosnący na pusty }
```

[3]Szczegóły implementacji bufora cyklicznego można znaleźć pod adresem `http://wazniak.mimuw.edu.pl/index.php?title=Wst%C4%99p_do_programowania` w rozdziale *Stosy i kolejki*.

```
3:  ost_up := 0; ost_down := 0;
4:  rekord := 1;
5:  for j := 1 to n do begin
6:    x.ind := j;
7:    x.val := A[j];
8:    while (not Pusta(UP)) and (Pierwszy(UP).val − t > A[j]) do
9:      ost_up := UsuńPierwszy(UP).ind;
10:   while (not Pusta(DOWN)) and (Pierwszy(DOWN).val + t < A[j]) do
11:     ost_down := UsuńPierwszy(DOWN).ind;
12:   i := max(ost_up + 1, ost_down + 1);
13:   { koniec fazy pierwszej }
14:   while (not Pusta(UP)) and (Ostatni(UP).val < A[j]) do
15:     UsuńOstatni(UP);
16:   if Pusta(UP) or (Ostatni(UP).val > A[j]) then
17:     WstawNaKoniec(UP, x)
18:   else Ostatni(UP).ind := j;
19:   while (not Pusta(DOWN)) and (Ostatni(DOWN).val > A[j]) do
20:     UsuńOstatni(DOWN);
21:   if Pusta(DOWN) or (Ostatni(DOWN).val < A[j]) then
22:     WstawNaKoniec(DOWN, x)
23:   else Ostatni(DOWN).ind := j;
24:   { koniec fazy drugiej }
25:   if j − i + 1 > rekord then rekord := j − i + 1;
26: end
```

Pozostaje pytanie o złożoność tego rozwiązania. Na pierwszy rzut oka nie widać, żeby to miało być rozwiązanie o liniowym koszcie. Wszak wewnątrz pętli po wszystkich j mamy pętle przetwarzające ciągi *UP* i *DOWN* — oba o potencjalnie liniowej długości. Jednak to przetwarzanie łącznie we wszystkich obrotach pętli po j nie może zająć więcej niż liniowo wiele: przecież wewnętrzne pętle usuwają elementy nadmiarowe, a każdy z nich był wstawiony tylko raz do każdego z tych ciągów. Nie można wielokrotnie usuwać czegoś, co tylko raz się pojawiło. Zatem łączna liczba wszystkich usunięć też jest liniowa, bo liniowo dużo elementów do tych ciągów wstawiliśmy.

Implementacje rozwiązań podobnych do opisanego tutaj można znaleźć w plikach `pil.cpp`, `pil2.pas` i `pil3.cpp`.

Odnośniki

W tym miejscu warto dodać, że zadanie o *Pilotach* sprowadziliśmy w tym opracowaniu tak naprawdę do następującego problemu. Należy na początkowo pustym ciągu liczb wykonać n operacji postaci: wstaw zadany element na początku ciągu lub usuń jeden element z końca ciągu. Po każdej operacji musimy umieć podać minimum (równoważnie: maksimum) z elementów ciągu.

Rozwiązania tak sformułowanego problemu o koszcie czasowym $O(n)$, działające na takiej samej zasadzie jak wyżej opisane, pojawiały się już na zawodach olimpijskich,

patrz opracowanie zadania *BBB* z XV Olimpiady Informatycznej [15] lub opracowanie zadania *Sound* z Bałtyckiej Olimpiady Informatycznej 2007 [4].

Range Minimum Query

Dużo bardziej wyrafinowane rozwiązanie wyjściowego problemu otrzymujemy, stosując tzw. algorytm RMQ. Pozwala on szybko odpowiadać na pytania, jaka jest minimalna/maksymalna wartość z jakiegoś segmentu ciągu. Po wstępnym przetworzeniu w czasie liniowym udziela odpowiedzi w czasie stałym. To rozwiązanie jest dość trudne do zaimplementowania. Jego opis można znaleźć w dwuczęściowym artykule poświęconym problemom RMQ i LCA, w numerach 9/2007 i 11/2007 czasopisma *Delta*[5].

Testy

Ze względu na fakt, że do tego zadania można było wymyślić wiele różnych heurystyk, zestaw testów jest stosunkowo liczny.

1. Testy *a* są losowe.
2. Testy *b* to na zmianę losowo: albo $+1$, albo -1 w stosunku do poprzedniej wartości.
3. Testy *c* generują sinusoidę. Długość okresu różni się w zależności od testu.
4. W testach *d* różnice między kolejnymi wyrazami ciągu są losowane z pewnego przedziału.
5. W testach *e* ciąg generalnie powoli rośnie, lokalnie zdarzają się zaburzenia tego trendu.
6. Testy *f* zawierają przypadki brzegowe.

Poniżej znajduje się tabela z podstawowymi statystykami poszczególnych testów.

Nazwa	n	t
pil1a.in	50	5 000
pil1b.in	50	6
pil1c.in	50	2 500
pil1d.in	50	16
pil1e.in	50	35
pil1f.in	1	1
pil2a.in	200	20 000
pil2b.in	200	8

Nazwa	n	t
pil2c.in	200	10 000
pil2d.in	200	66
pil2e.in	200	280
pil2f.in	5	0
pil3a.in	500	1 000 000 000
pil3b.in	500	9
pil3c.in	500	500 000 000
pil3d.in	500	166

[4] Opracowania zadań z zawodów BOI 2007 można znaleźć — tylko w wersji anglojęzycznej — na stronie internetowej `http://www.boi2007.de/tasks/book.pdf`

[5] Artykuły dostępne także na stronie internetowej czasopisma: `http://www.mimuw.edu.pl/delta/`

Nazwa	n	t
pil3e.in	500	1 100
pil4a.in	10 000	30 000
pil4b.in	10 000	14
pil4c.in	10 000	15 000
pil4d.in	10 000	3 333
pil4e.in	10 000	100 000
pil5a.in	50 000	1 000 000 000
pil5b.in	50 000	16
pil5c.in	50 000	500 000 000
pil5d.in	50 000	16 666
pil5e.in	50 000	1 115 000
pil6a.in	100 000	10 000 000
pil6b.in	100 000	17
pil6c.in	100 000	5 000 000
pil6d.in	100 000	33 333
pil6e.in	100 000	3 000 000
pil7a.in	300 000	300 000
pil7b.in	300 000	19
pil7c.in	300 000	150 000
pil7d.in	300 000	100 000
pil7e.in	300 000	9 000 000

Nazwa	n	t
pil8a.in	1 000 000	1 000 000 000
pil8b.in	1 000 000	20
pil8c.in	1 000 000	500 000 000
pil8d.in	1 000 000	333 333
pil8e.in	1 000 000	30 000 000
pil9a.in	1 500 000	1 000 000 000
pil9b.in	1 500 000	21
pil9c.in	1 500 000	500 000 000
pil9d.in	1 500 000	500 000
pil9e.in	1 500 000	45 000 000
pil10a.in	3 000 000	1 000 000 000
pil10b.in	3 000 000	22
pil10c.in	3 000 000	500 000 000
pil10d.in	3 000 000	1 000 000
pil10e.in	3 000 000	90 000 000
pil10f.in	3 000 000	2 000 000 000

XXII Międzynarodowa Olimpiada Informatyczna,

Waterloo, Kanada 2010

Marcin Kubica, Tomasz Kulczyński, Jakub Radoszewski
Tłumaczenie

IOI 2010, dzień pierwszy, 16.08.2010

Detektyw

Dr Black został zamordowany. Detektyw Jill musi znaleźć mordercę oraz wyznaczyć miejsce zbrodni i narzędzie zbrodni. Jest sześcioro możliwych morderców, ponumerowanych od 1 do 6. Jest dziesięć możliwych miejsc zbrodni, ponumerowanych od 1 do 10. Jest sześć możliwych narzędzi zbrodni, ponumerowanych od 1 do 6.

Poniżej podano przykładowe listy potencjalnych morderców, miejsc zbrodni oraz narzędzi zbrodni. Znajomość tych nazw nie jest potrzebna do rozwiązania zadania.

Morderca	**Miejsce zbrodni**	**Narzędzie zbrodni**
1. Profesor Śliwiński	*1. Sala balowa*	*1. Ołowiany pręt*
2. Panna Szkarłat	*2. Kuchnia*	*2. Sztylet*
3. Pułkownik Musztarda	*3. Oranżeria*	*3. Świecznik*
4. Pani Bielecka	*4. Jadalnia*	*4. Rewolwer*
5. Wielebny Zieliński	*5. Sala bilardowa*	*5. Lina*
6. Pani Pawlakowa	*6. Biblioteka*	*6. Klucz francuski*
	7. Salon	
	8. Korytarz	
	9. Gabinet	
	10. Piwnica	

Jill wykonuje sekwencję prób odgadnięcia prawidłowej kombinacji mordercy, miejsca zbrodni oraz narzędzia zbrodni. Każdą taką próbę nazwiemy **teorią**. *Jill prosi swojego asystenta, Jacka, o potwierdzenie bądź obalenie każdej kolejnej teorii. Jeśli Jack potwierdzi jakąś teorię, Jill ma problem z głowy. Jeśli Jack obali jakąś teorię, powiadamia Jill, która z podanych przez nią informacji — morderca, miejsce zbrodni lub narzędzie zbrodni — jest nieprawidłowa.*

Zaimplementuj procedurę **Solve**, *która odgrywa rolę Jill. Sprawdzaczka wywoła procedurę* **Solve** *wiele razy, każdorazowo z nowym przypadkiem do rozważenia.* **Solve** *powinno wywoływać funkcję* **Theory**(M, L, W), *która jest zaimplementowana w sprawdzaczce, przy czym M, L oraz W to liczby oznaczające konkretną kombinację mordercy, miejsca zbrodni i narzędzia zbrodni.* **Theory**(M, L, W) *zwraca 0, jeśli teoria jest poprawna. Jeśli teoria jest błędna, zwracana jest wartość 1, 2 lub 3. 1 oznacza, że został podany nieprawidłowy morderca; 2 oznacza, że zostało podane nieprawidłowe miejsce zbrodni; 3 oznacza, że zostało podane nieprawidłowe narzędzie zbrodni. Jeśli więcej niż jedna z tych informacji była nieprawidłowa, Jack wybiera jakąkolwiek spośród tych nieprawidłowych (niekoniecznie w ściśle określony sposób). Jak tylko* **Theory**(M, L, W) *zwróci 0, procedura* **Solve** *powinna zakończyć się.*

Przykład

Dla przykładu, załóżmy, że morderstwa dokonała panna Szkarłat (morderca numer 2) w oranżerii (miejsce zbrodni numer 3) za pomocą rewolweru (narzędzie zbrodni numer 4). Poniżej przedstawiona jest sekwencja wywołań funkcji **Theory** *przez procedurę* **Solve**. *Możliwe wyniki zwracane przez te wywołania są wymienione w drugiej kolumnie tabeli.*

Wywołanie	Zwrócona wartość	Wyjaśnienie
Theory$(1,1,1)$	*1 lub 2, lub 3*	*Wszystkie informacje są nieprawidłowe*
Theory$(3,3,3)$	*1 lub 3*	*Tylko miejsce zbrodni jest prawidłowe*
Theory$(5,3,4)$	*1*	*Tylko morderca jest nieprawidłowy*
Theory$(2,3,4)$	*0*	*Wszystkie informacje są prawidłowe*

Podzadanie 1 [50 punktów]

W każdym teście procedura **Solve** *może zostać wywołana co najwyżej 100 razy. Każde z wywołań może odpowiadać innej wynikowej kombinacji mordercy, miejsca zbrodni oraz narzędzia zbrodni. Przy każdym wywołaniu procedury* **Solve**, *musi ona znaleźć poprawną teorię w co najwyżej 360 wywołaniach funkcji* **Theory**(M,L,W). **Koniecznie pamiętaj o inicjalizacji wszystkich zmiennych używanych w procedurze Solve, przy każdym jej wywołaniu.**

Podzadanie 2 [50 punktów]

W każdym teście procedura **Solve** *może zostać wywołana co najwyżej 100 razy. Przy każdym wywołaniu procedury* **Solve**, *musi ona znaleźć poprawną teorię w co najwyżej 20 wywołaniach funkcji* **Theory**(M,L,W). **Koniecznie pamiętaj o inicjalizacji wszystkich zmiennych używanych w procedurze Solve, przy każdym jej wywołaniu.**

Szczegóły implementacyjne

- *Używaj środowiska do programowania i testowania RunC.*
- *Katalog do implementacji:* `/home/ioi2010-contestant/cluedo/` *(prototyp w pliku* `cluedo.zip`*).*
- *Do implementacji przez zawodnika:* `cluedo.c` *lub* `cluedo.cpp` *lub* `cluedo.pas`*.*
- *Interfejs dla zawodnika:* `cluedo.h` *lub* `cluedo.pas`*.*
- *Interfejs sprawdzaczki:* `grader.h` *lub* `graderlib.pas`*.*
- *Przykładowa sprawdzaczka:* `grader.c` *lub* `grader.cpp` *lub* `grader.pas` *oraz* `graderlib.pas`*.*
- *Wejście przykładowej sprawdzaczki:* `grader.in.1`*.*
  Uwaga: Każdy wiersz wejścia zawiera trzy liczby oznaczające mordercę, miejsce zbrodni oraz broń.
- *Oczekiwane wyjście dla przykładowego wejścia sprawdzaczki: jeśli* **Solve** *poprawnie rozwiąże wszystkie przypadki testowe, plik wyjściowy będzie zawierał „*`OK` *t”, przy czym t to maksymalna liczba wywołań funkcji* **Theory** *w jakimkolwiek przypadku testowym.*
- *Kompilacja i uruchamianie (linia komend):* `runc grader.c` *lub* `runc grader.cpp` *lub* `runc grader.pas`*.*

- *Kompilacja i uruchamianie (wtyczka do Gedita): Control-R wciśnięte podczas edycji jakiegokolwiek pliku rozwiązania.*
- *Wysyłanie (linia komend):* `submit grader.c` *lub* `submit grader.cpp` *lub* `submit grader.pas`.
- *Wysyłanie (wtyczka do Gedita): Control-J wciśnięte podczas edycji jakiegokolwiek pliku rozwiązania.*

Marcin Kubica, Tomasz Kulczyński, Jakub Radoszewski
Tłumaczenie

IOI 2010, dzień pierwszy, 16.08.2010

Ciepło-Zimno

Jack i Jill grają w grę o nazwie **Ciepło-Zimno**. *Jill wybrała sobie liczbę między 1 i N, a Jack wykonuje sekwencję prób zgadnięcia tej liczby.*

Każdą próbę Jacka można opisać za pomocą liczby między 1 i N. Odpowiedzią Jill na każdą z prób jest „ciepło", „zimno" lub „tak samo". Po pierwszej próbie Jacka Jill zawsze odpowiada „tak samo". Po każdej kolejnej próbie Jacka Jill odpowiada:

- *„ciepło", jeśli ta próba jest bliżej liczby Jill niż poprzednia próba,*
- *„zimno", jeśli ta próba jest dalej od liczby Jill niż poprzednia próba,*
- *„tak samo", jeśli ta próba nie jest ani bliżej, ani dalej od liczby Jill niż poprzednia próba.*

Zaimplementuj procedurę **HC**(N), *która odgrywa rolę Jacka. Implementacja może wielokrotnie wywoływać* **Guess**(G), *przy czym G jest liczbą między 1 a N.* **Guess**(G) *zwraca 1, jeśli odpowiedzią jest „ciepło", -1, jeśli odpowiedzią jest „zimno", a 0, jeśli odpowiedzią jest „tak samo".* **HC**(N) *powinna zwrócić liczbę Jill.*

Przykład

Dla przykładu, załóżmy, że $N = 5$ i Jill wybrała liczbę 2. Poniżej przedstawiona jest sekwencja wywołań procedury **Guess** *przez procedurę* **HC**. *Wyniki zwracane przez te wywołania są wymienione w drugiej kolumnie tabeli.*

Wywołanie	**Zwrócona wartość**	**Wyjaśnienie**
Guess(5)	*0*	*„tak samo" (pierwsze wywołanie)*
Guess(3)	*1*	*„ciepło"*
Guess(4)	*-1*	*„zimno"*
Guess(1)	*1*	*„ciepło"*
Guess(3)	*0*	*„tak samo"*

W tym momencie Jack już zna odpowiedź, a procedura **HC** *powinna zwrócić 2. Wyznaczenie liczby Jill zajęło Jackowi 5 prób. A da się lepiej.*

Podzadanie 1 [25 punktów]

HC(N) *może wywołać* **Guess**(G) *co najwyżej 500 razy. Będzie co najwyżej 125 250 wywołań* **HC**(N), *dla N między 1 a 500.*

Podzadanie 2 [25 punktów]

HC(N) *może wywołać* **Guess**(G) *co najwyżej 18 razy. Będzie co najwyżej 125 250 wywołań* **HC**(N), *dla N między 1 a 500.*

Podzadanie 3 [25 punktów]

HC(N) *może wywołać* **Guess**(G) *co najwyżej 16 razy. Będzie co najwyżej 125 250 wywołań* **HC**(N)*, dla* N *między 1 a 500.*

Podzadanie 4 [co najwyżej 25 punktów]

Niech W *będzie największą liczbą całkowitą, dla której* $2^W \leqslant 3N$*. Twoje rozwiązanie uzyska za to podzadanie:*

- *0 punktów, jeśli* **HC**(N) *wywoła* **Guess**(G) $2W$ *lub więcej razy,*
- 25α *punktów, przy czym* α *jest największą liczbą rzeczywistą, dla której* $0 < \alpha < 1$ *oraz* **HC**(N) *wywołuje* **Guess**(G) *co najwyżej* $2W - \alpha W$ *razy,*
- *25 punktów, jeśli* **HC**(N) *wywołuje* **Guess**(G) *co najwyżej* W *razy.*

Będzie co najwyżej 1 000 000 wywołań procedury **HC**(N)*, dla* N *między 1 a 500 000 000.*

Koniecznie pamiętaj o inicjalizacji wszystkich zmiennych używanych w procedurze HC, przy każdym jej wywołaniu.

Szczegóły implementacyjne

- *Używaj środowiska do programowania i testowania RunC.*
- *Katalog do implementacji:* `/home/ioi2010-contestant/hottercolder/` *(prototyp w pliku* `hottercolder.zip`*).*
- *Do implementacji przez zawodnika:* `hottercolder.c` *lub* `hottercolder.cpp` *lub* `hottercolder.pas`*.*
- *Interfejs dla zawodnika:* `hottercolder.h` *lub* `hottercolder.pas`*.*
- *Interfejs sprawdzaczki:* `grader.h` *lub* `graderlib.pas`*.*
- *Przykładowa sprawdzaczka:* `grader.c` *lub* `grader.cpp` *lub* `grader.pas` *oraz* `graderlib.pas`*.*
- *Wejście przykładowej sprawdzaczki:* `grader.in.1`, `grader.in.2`*.*
  Uwaga: Plik wejściowy składa się z kilku wierszy, z których każdy zawiera N **oraz liczbę Jill.**
- *Oczekiwane wyjście dla przykładowego wejścia sprawdzaczki: sprawdzaczka stworzy pliki* `grader.out.1`, `grader.out.2` *itd.*
  - *Jeśli program rozwiązuje poprawnie podzadanie 1, jeden wiersz wyjścia będzie zawierał „*`OK 1`*”.*
  - *Jeśli program rozwiązuje poprawnie podzadanie 2, jeden wiersz wyjścia będzie zawierał „*`OK 2`*”.*

  - *Jeśli program rozwiązuje poprawnie podzadanie 3, jeden wiersz wyjścia będzie zawierał* „`OK 3`”.
  - *Jeśli program rozwiązuje poprawnie podzadanie 4, jeden wiersz wyjścia będzie zawierał* „`OK 4 alpha` α”.

- *Kompilacja i uruchamianie (linia komend):* `runc grader.c` *lub* `runc grader.cpp` *lub* `runc grader.pas`.
- *Kompilacja i uruchamianie (wtyczka do Gedita): Control-R wciśnięte podczas edycji jakiegokolwiek pliku rozwiązania.*
- *Wysyłanie (linia komend):* `submit grader.c` *lub* `submit grader.cpp` *lub* `submit grader.pas`.
- *Wysyłanie (wtyczka do Gedita): Control-J wciśnięte podczas edycji jakiegokolwiek pliku rozwiązania.*

Marcin Kubica, Tomasz Kulczyński, Jakub Radoszewski
Tłumaczenie

IOI 2010, dzień pierwszy, 16.08.2010

Jakość życia

Miasta w stanie Alberta mają zazwyczaj kształt prostokątów podzielonych na pola. Pola są oznaczone współrzędną z zakresu od 0 do $R-1$ w kierunku północ-południe oraz drugą współrzędną od 0 do $C-1$ w kierunku zachód-wschód.

Jakość życia wewnątrz każdego z pól jest określona przy pomocy liczby zwanej **poziomem jakości** *(dla każdego pola innej), między 1 i $R \cdot C$, przy czym 1 jest najlepszym poziomem, a $R \cdot C$ najgorszym.*

Miejski wydział planowania chciałby znaleźć prostokątny podzbiór pól o wymiarach H pól w kierunku północ-południe oraz W pól w kierunku zachód-wschód o najlepszej medianie poziomów jakości. H i W są **nieparzystymi** *liczbami nieprzekraczającymi odpowiednio R i C.* **Medianą** *poziomów jakości nieparzystej liczby pól nazywamy taki poziom jakości m jednego z tych pól, że liczba pól o poziomie jakości lepszym od m jest równa liczbie pól o poziomie jakości gorszym od m.*

Twoim zadaniem jest zaimplementowanie funkcji **rectangle**(R, C, H, W, Q)*, przy czym R i C reprezentują całkowity rozmiar miasta, H i W oznaczają wymiary prostokątnego podzbioru pól, a Q jest tablicą wartości, w której $Q[a][b]$ oznacza poziom jakości pola o współrzędnej a w kierunku północ-południe oraz b w kierunku zachód-wschód.*

Twoja funkcja **rectangle** *powinna zwracać liczbę całkowitą: najlepszą (reprezentowaną najmniejszą liczbą) możliwą medianę poziomów jakości prostokątnego podzbioru pól o rozmiarze H na W.*

W każdym z testów funkcja **rectangle** *zostanie wywołana dokładnie raz.*

Przykład 1

$R = 5$, $C = 5$, $H = 3$, $W = 3$, $Q =$

5	11	12	16	25
17	18	**2**	**7**	**10**
4	23	**20**	**3**	**1**
24	21	**19**	**14**	**9**
6	22	8	13	15

W tym przykładzie najlepsza (liczbowo najmniejsza) mediana poziomów jakości dziewięciu pól jest osiągana przez środkowy prawy kwadrat zawarty w Q, wyróżniony pogrubieniem. Stąd, **rectangle**$(R, C, H, W, Q) = 9$.

Przykład 2

$R = 2$, $C = 6$, $H = 1$, $W = 5$, $Q =$

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

Odpowiedzią dla tego przykładu jest 5.

Podzadanie 1 [20 punktów]

Możesz założyć, że R i C nie przekraczają 30.

Podzadanie 2 [20 punktów]

Możesz założyć, że R i C nie przekraczają 100.

Podzadanie 3 [20 punktów]

Możesz założyć, że R i C nie przekraczają 300.

Podzadanie 4 [20 punktów]

Możesz założyć, że R i C nie przekraczają 1 000.

Podzadanie 5 [20 punktów]

Możesz założyć, że R i C nie przekraczają 3 000.

Szczegóły implementacyjne

- *Używaj środowiska do programowania i testowania RunC.*
- *Katalog do implementacji:* `/home/ioi2010-contestant/quality/` *(prototyp w pliku* `quality.zip`*).*
- *Do implementacji przez zawodnika:* `quality.c` *lub* `quality.cpp` *lub* `quality.pas`*.*
- *Interfejs dla zawodnika:* `quality.h` *lub* `quality.pas`*.*
- *Interfejs sprawdzaczki:* **brak**.
- *Przykładowa sprawdzaczka:* `grader.c` *lub* `grader.cpp` *lub* `grader.pas`*.*
- *Wejście przykładowej sprawdzaczki:* `grader.in.1`*,* `grader.in.2` *itd.*
  Uwaga: Pierwszy wiersz wejścia zawiera: R, C, H, W. Kolejne wiersze zawierają elementy tablicy Q, podawane wierszami.
- *Oczekiwane wyjście dla przykładowego wejścia sprawdzaczki:* `grader.expect.1`*,* `grader.expect.2` *itd.*
- *Kompilacja i uruchamianie (linia komend):* `runc grader.c` *lub* `runc grader.cpp` *lub* `runc grader.pas`*.*
- *Kompilacja i uruchamianie (wtyczka do Gedita): Control-R wciśnięte podczas edycji jakiegokolwiek pliku rozwiązania.*

- *Wysyłanie (linia komend):* `submit grader.c` *lub* `submit grader.cpp` *lub* `submit grader.pas`.
- *Wysyłanie (wtyczka do Gedita): Control-J wciśnięte podczas edycji jakiegokolwiek pliku rozwiązania.*

Marcin Kubica, Tomasz Kulczyński, Jakub Radoszewski
Tłumaczenie

IOI 2010, dzień pierwszy, 16.08.2010

Języki

Twoim zadaniem jest napisanie programu, który dostając kolejne wycinki z Wikipedii (patrz przykład poniżej), będzie zgadywał, w jakim są języku. Po każdej próbie Twój program pozna prawidłową odpowiedź, więc będzie mógł uczyć się w miarę zgadywania.

Każdy z języków oznaczony jest liczbą L między 0 a 55. Każdy wycinek ma dokładnie 100 znaków i jest podany jako tablica E zawierająca 100 liczb całkowitych między 1 a 65 535. Liczby te zostały przyporządkowane w dowolny sposób i nie reprezentują żadnego standardowego kodowania.

Zaimplementuj funkcję **excerpt**(E)*, przy czym E jest tablicą 100 liczb reprezentujących wycinek z Wikipedii, jak opisano wyżej. Twoja funkcja musi wywołać* **language**(L) *dokładnie raz, przy czym L jest odgadniętym przez nią numerem języka, odpowiadającym wersji Wikipedii, z której pochodzi dany wycinek E. System oceniający ma zaimplementowaną funkcję* **language**(L)*, która ocenia Twój strzał oraz zwraca poprawny numer języka. Strzał jest zatem dobry, gdy* **language**$(L) = L$.

System oceniający wywołuje **excerpt**(E) *10 000 razy, raz dla każdego wycinka ze swojego pliku wejściowego.* **Dokładnością** *Twojego rozwiązania nazywamy ułamek liczby wycinków, dla których* **excerpt**(E) *dało poprawną odpowiedź.*

Do rozwiązania tego zadania możesz użyć dowolnej strategii. **Metoda Rocchio** *jest podejściem, które zapewni dokładność równą w przybliżeniu 0,4. Metoda ta oblicza podobieństwo E do każdego języka L, który dotąd wystąpił, i wybiera jako swój strzał język, który jest najbardziej podobny. Podobieństwem nazywamy tutaj liczbę różnych znaków w E, które występują gdziekolwiek w dotychczasowych wycinkach języka L.*

Zwróć uwagę, że dane wejściowe pochodzą z prawdziwych stron Wikipedii, więc mogą w nich występować drobne błędy polegające na zniekształceniu znaku lub fragmentu tekstu. Stanowi to część zadania i należy się tego spodziewać.

Przykład

Przykładowy plik wejściowy `grader.in.1` *zawiera 10 000 przykładowych wycinków. Wybranych 56 języków to te, które zostały oznaczone jako język ojczysty przez któryś z krajów uczestniczących w IOI 2010. Język każdego wycinka jest wybrany losowo spośród tych 56 języków, a wycinek jest początkowym fragmentem losowej strony Wikipedii w tym języku. Każdy wiersz tego pliku zawiera:*

- *dwuliterowy kod ISO oznaczający język Wikipedii;*
- *100 liczb między 1 a 65 535, reprezentujących pierwsze 100 znaków pierwszego paragrafu strony, w oryginalnej kolejności;*
- *tekstową reprezentację (UTF-8) tych 100 znaków, którą można wyświetlić w edytorze tekstu lub przeglądarce Firefox. Ta reprezentacja jest dołączona wyłącznie dla Twojej wygody i nie ma być wejściem programu.*

Oficjalny serwer oceniający także wykorzystuje 10 000 różnych wycinków, wybranych w ten sam sposób ze stron Wikipedii w 56 językach. Serwer przydzieli jednak inną liczbę od 0 do 55 każdemu językowi oraz inną liczbę od 1 do 65 535 każdemu ze znaków.

Podzadanie 1 [30 punktów]

Twoje zgłoszenie musi osiągnąć dokładność co najmniej 0,3 na serwerze oceniającym.

Podzadanie 2 [do 80 punktów]

Twoim wynikiem będzie $114 \cdot (\alpha - 0{,}3)$ *zaokrąglone do najbliższej liczby całkowitej, przy czym* α *jest dokładnością Twojego rozwiązania.*

Szczegóły implementacyjne

- *Używaj środowiska do programowania i testowania RunC.*
- *Katalog do implementacji:* `/home/ioi2010-contestant/language/` *(prototyp w pliku* `language.zip`*).*
- *Do implementacji przez zawodnika:* `lang.c` *lub* `lang.cpp` *lub* `lang.pas`*.*
- *Interfejs dla zawodnika:* `lang.h` *lub* `lang.pas`*.*
- *Interfejs sprawdzaczki:* `grader.h` *lub* `graderlib.pas`*.*
- *Przykładowa sprawdzaczka:* `grader.c` *lub* `grader.cpp` *lub* `grader.pas` *oraz* `graderlib.pas`*.*
- *Wejście przykładowej sprawdzaczki:* `grader.in.1`*.*
  Uwaga: Każdy wiersz wejścia zawiera: dwuznakowy kod języka; wycinek jako 100 liczb pooddzielanych pojedynczymi odstępami; reprezentację tekstową wycinka.
- *Oczekiwane wyjście dla przykładowego wejścia sprawdzaczki: Jeśli zaimplementowana funkcja wywoła* **language** *wedle specyfikacji 10 000 razy, przykładowa sprawdzaczka wypisze „*`OK` α*", przy czym* α *jest dokładnością rozwiązania.*
- *Kompilacja i uruchamianie (linia komend):* `runc grader.c` *lub* `runc grader.cpp` *lub* `runc grader.pas`*.*
- *Kompilacja i uruchamianie (wtyczka do Gedita): Control-R wciśnięte podczas edycji jakiegokolwiek pliku rozwiązania.*
- *Wysyłanie (linia komend):* `submit grader.c` *lub* `submit grader.cpp` *lub* `submit grader.pas`*.*
- *Wysyłanie (wtyczka do Gedita): Control-J wciśnięte podczas edycji jakiegokolwiek pliku rozwiązania.*

Marcin Kubica, Tomasz Kulczyński, Jakub Radoszewski
Tłumaczenie

IOI 2010, dzień drugi, 18.08.2010

Gra w pary

W **Grze w pary** *używa się talii złożonej z 50 kart. Każda z kart ma nadrukowaną jedną z liter od* `A` *do* `Y` *(znaki ASCII o numerach od 65 do 89), przy czym każda z liter jest wydrukowana na dokładnie dwóch kartach. Karty są tasowane, a następnie rozkładane na stole literami do dołu.*

Jack gra w tę grę w ten sposób, że odwraca po dwie karty tak, aby zobaczyć znajdujące się na nich litery. Za każdym razem, gdy Jack widzi po raz pierwszy dwie karty z tą samą, konkretną literą, chłopiec dostaje od mamy cukierka. Na przykład, za pierwszym razem, gdy Jack odwróci obie karty, na których jest zapisana litera `M`*, dostanie cukierka. Niezależnie od tego, czy litery były jednakowe, czy nie, Jack odwraca następnie obie karty z powrotem literami do dołu. Gra toczy się do momentu, gdy Jack dostanie 25 cukierków, po jednym za każdą z liter.*

Twoim zadaniem jest zaimplementowanie procedury **play***, która będzie grała w tę grę. Twoja procedura powinna wywoływać funkcję* **faceup***(C), która jest częścią sprawdzaczki. C jest liczbą od 1 do 50 oznaczającą, którą konkretnie kartę chcesz odwrócić. Karta ta nie może się w tym momencie znajdować w pozycji literą do góry. Funkcja* **faceup***(C) zwraca znak, który jest zapisany na karcie o numerze C.*

Po każdych dwóch wywołaniach **faceup***, sprawdzaczka automatycznie odwraca obie karty z powrotem literami do dołu.*

Twoja procedura **play** *może skończyć się dopiero, gdy Jack otrzyma wszystkie 25 cukierków. Dozwolone jest wywoływanie* **faceup***(C), nawet jeśli Jack dostał już ostatni z cukierków.*

Przykład

Poniżej przedstawiono jedną z możliwych sekwencji wywołań, jaką mogłaby wykonać Twoja procedura **play***, wraz z wyjaśnieniami.*

Wywołanie	**Zwrócona wartość**	**Wyjaśnienie**
faceup*(1)*	'B'	*Na karcie numer 1 znajduje się litera* `B`.
faceup*(7)*	'X'	*Na karcie numer 7 znajduje się litera* `X`*. Litery nie są jednakowe.*
Sprawdzaczka automatycznie odwraca karty 1 i 7 literami do dołu.		
faceup*(7)*	'X'	*Na karcie numer 7 znajduje się litera* `X`.
faceup*(15)*	'O'	*Na karcie numer 15 znajduje się litera* `O`*. Litery nie są jednakowe.*
Sprawdzaczka automatycznie odwraca karty 7 i 15 literami do dołu.		
faceup*(50)*	'X'	*Na karcie numer 50 znajduje się litera* `X`.
faceup*(7)*	'X'	*Na karcie numer 7 znajduje się litera* `X`*. Jack dostaje swojego pierwszego cukierka.*
Sprawdzaczka automatycznie odwraca karty 50 i 7 literami do dołu.		
faceup*(7)*	'X'	*Na karcie numer 7 znajduje się litera* `X`.

Wywołanie	**Zwrócona wartość**	**Wyjaśnienie**
faceup(*50*)	'X'	*Na karcie numer 50 znajduje się litera* X. *Litery są jednakowe, ale Jack nie dostaje cukierka.*
Sprawdzaczka automatycznie odwraca karty 7 i 50 literami do dołu.		
faceup(*2*)	'B'	*Na karcie numer 2 znajduje się litera* B.
...		
(niektóre wywołania funkcji zostały pominięte)		
...		
faceup(*1*)	'B'	*Na karcie numer 1 znajduje się litera* B.
faceup(*2*)	'B'	*Na karcie numer 2 znajduje się litera* B. *Jack dostaje swojego 25. cukierka.*

Podzadanie 1 [50 punktów]

Zaimplementuj jakąkolwiek strategię, która działa zgodnie z zasadami gry i kończy się przed upływem limitu czasu.

Przykładowo, istnieje prosta strategia, która zawsze wykonuje dokładnie

$$2 \cdot (49 + 48 + \ldots + 2 + 1) = 2450$$

wywołań funkcji **faceup**(C).

Podzadanie 2 [50 punktów]

Zaimplementuj strategię, która każdą możliwą rozgrywkę zakończy przy użyciu co najwyżej 100 wywołań funkcji **faceup**(C).

Szczegóły implementacyjne

- *Używaj środowiska do programowania i testowania RunC.*
- *Katalog do implementacji:* `/home/ioi2010-contestant/memory/` *(prototyp w pliku* `memory.zip`*).*
- *Do implementacji przez zawodnika:* `memory.c` *lub* `memory.cpp` *lub* `memory.pas`*.*
- *Interfejs dla zawodnika:* `memory.h` *lub* `memory.pas`*.*
- *Interfejs sprawdzaczki:* `grader.h` *lub* `graderlib.pas`*.*
- *Przykładowa sprawdzaczka:* `grader.c` *lub* `grader.cpp` *lub* `grader.pas` *oraz* `graderlib.pas`*.*
- *Wejście przykładowej sprawdzaczki:* `grader.in.1`*.*
  Uwaga: plik wejściowy zawiera jeden wiersz z 50 znakami oznaczającymi litery na kartach, w kolejności od 1 do 50.

- *Oczekiwane wyjście dla wejścia przykładowej sprawdzaczki: Jeśli zaimplementowana funkcja jest poprawna, plik wyjściowy będzie zawierał „*`OK` *n*”*, gdzie n jest liczbą wywołań* **faceup***(C).*
- *Kompilacja i uruchamianie (linia komend):* `runc grader.c` *lub* `runc grader.cpp` *lub* `runc grader.pas`*.*
- *Kompilacja i uruchamianie (wtyczka do Gedita): Control-R wciśnięte podczas edycji jakiegokolwiek pliku rozwiązania.*
- *Wysyłanie (linia komend):* `submit grader.c` *lub* `submit grader.cpp` *lub* `submit grader.pas`*.*
- *Wysyłanie (wtyczka do Gedita): Control-J wciśnięte podczas edycji jakiegokolwiek pliku rozwiązania lub sprawdzaczki.*

Marcin Kubica, Tomasz Kulczyński, Jakub Radoszewski
Tłumaczenie

IOI 2010, dzień drugi, 18.08.2010

Korek drogowy

Mimo że Kanada jest wielkim krajem, duża jej część pozostaje niezamieszkała, a większość mieszkańców skupia się w pobliżu południowej granicy. Autostrada transkanadyjska, której budowę ukończono w 1962 roku, łączy ludzi mieszkających w tych właśnie okolicach, od St. John's na wschodzie aż po Victorię na zachodzie — autostrada ta ma długość 7 821 km.

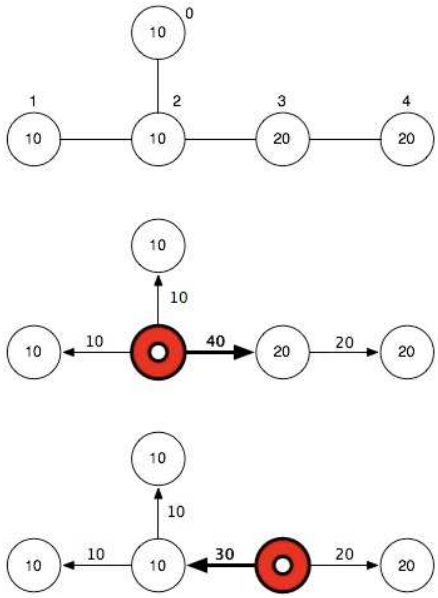
Kanadyjczycy uwielbiają hokej. Po każdym meczu hokejowym tysiące fanów wsiadają do swoich samochodów i wracają do domu, tworząc duże natężenie ruchu na drogach. Bogaty inwestor chciałby kupić zespół hokejowy i wybudować nową halę. Twoim zadaniem jest pomóc mu w wyborze takiego miejsca na budowę tej hali, żeby zminimalizować natężenie ruchu drogowego po meczu.

Państwo składa się z miast połączonych siecią dróg. Wszystkie drogi są dwukierunkowe, a pomiędzy każdą parą miast jest dokładnie jedna **trasa**. **Trasą** *łączącą miasta c_0 i c_k nazywamy ciąg różnych miast $c_0, \ldots, c_k$ taki, że dla każdego i, miasta c_{i-1} oraz c_i są połączone drogą. Nową halę należy zbudować w jednym z miast, które nazwiemy* **centrum hokejowym**. *Po meczu hokejowym wszyscy fani wracają z centrum hokejowego do swoich miast, poza tymi, którzy mieszkają w centrum hokejowym. Natężenie ruchu na każdej z dróg jest proporcjonalne do liczby fanów, którzy będą nią podróżować. Wyznacz takie położenie centrum hokejowego, aby natężenie ruchu na najbardziej obciążonej drodze było jak najmniejsze. Jeśli jest kilka jednakowo dobrych miast, wybierz którekolwiek z nich.*

Twoim zadaniem jest implementacja procedury **LocateCentre**(N, P, S, D). *N jest dodatnią liczbą całkowitą, oznaczającą liczbę miast. Miasta są ponumerowane od 0 do $N-1$. P jest tablicą N liczb całkowitych dodatnich; dla każdego i, $P[i]$ jest liczbą fanów hokeja mieszkających w mieście o numerze i. Łączna liczba fanów hokeja we wszystkich miastach nie przekroczy 2 000 000 000. S i D są tablicami zawierającymi po $N-1$ liczb całkowitych, oznaczającymi umiejscowienie dróg. Dla każdego i istnieje droga łącząca miasta o numerach $S[i]$ oraz $D[i]$. Funkcja powinna zwrócić liczbę całkowitą — numer miasta, w którym należy wybudować halę.*

Przykład

Dla przykładu, rozważmy sieć pięciu miast na najwyższym rysunku na następnej stronie, przy czym miasta 0, 1 i 2 mają po 10 fanów hokeja, a miasta 3 i 4 — po 20 fanów hokeja. Środkowy rysunek przedstawia natężenia ruchu na drogach, gdy halę wybudujemy w mieście numer 2; największe natężenie, równe 40, oznaczono pogrubioną strzałką. Dolny rysunek przedstawia natężenia ruchu na drogach, gdy halę wybudujemy w mieście numer 3; największe natężenie, równe 30, oznaczono pogrubioną strzałką. W takim razie, miasto numer 3 jest lepszym miejscem na budowę nowej hali niż miasto numer 2. Dane dla tego przykładu znajdują się w pliku `grader.in.3a`.



Uwaga

Zauważ, że przy podanych niżej ograniczeniach możliwe jest wysłanie zgłoszenia, które rozwiązuje poprawnie podzadanie 3, ale nie rozwiązuje podzadania 2. Pamiętaj, że Twój ostateczny wynik za całe zadanie pochodzi **wyłącznie z jednego** *spośród Twoich zgłoszeń.*

Podzadanie 1 [25 punktów]

Możesz założyć, że wszystkie miasta leżą w linii prostej od wschodu do zachodu oraz że drogi także leżą wzdłuż tej linii, bez odnóg. Dokładniej, możesz założyć, że dla każdego i, takiego że $0 \leqslant i \leqslant N-2$, $S[i] = i$ oraz $D[i] = i+1$.

Liczba miast wynosi co najwyżej 1000.

Podzadanie 2 [25 punktów]

Możesz przyjąć te same założenia, co w podzadaniu 1, ale liczba miast może wynieść co najwyżej 1 000 000.

Podzadanie 3 [25 punktów]

Założenia z podzadania 1 mogą już nie być zachowane. Liczba miast wynosi co najwyżej 1000.

Podzadanie 4 [25 punktów]

Założenia z podzadania 1 mogą już nie być zachowane. Liczba miast wynosi co najwyżej 1 000 000.

Szczegóły implementacyjne

- *Używaj środowiska do programowania i testowania RunC.*
- *Katalog do implementacji:* `/home/ioi2010-contestant/traffic/` *(prototyp w pliku* `traffic.zip`*).*
- *Do implementacji przez zawodnika:* `traffic.c` *lub* `traffic.cpp` *lub* `traffic.pas`*.*
- *Interfejs dla zawodnika:* `traffic.h` *lub* `traffic.pas`*.*
- *Interfejs sprawdzaczki:* **brak**.
- *Przykładowa sprawdzaczka:* `grader.c` *lub* `grader.cpp` *lub* `grader.pas`*.*
- *Wejście przykładowej sprawdzaczki:* `grader.in.1`, `grader.in.2`.
  Uwaga: Pierwszy wiersz pliku wejściowego zawiera N. Kolejne N wierszy zawiera $P[i]$ dla każdego i od 0 do $N-1$. Kolejne $N-1$ wierszy zawiera pary $S[i]$ $D[i]$ dla i od 0 do $N-2$.
- *Oczekiwane wyjście dla wejścia przykładowej sprawdzaczki:* `grader.expect.1`, `grader.expect.2` *itp.*
- *Kompilacja i uruchamianie (linia komend):* `runc grader.c` *lub* `runc grader.cpp` *lub* `runc grader.pas`*.*
- *Kompilacja i uruchamianie (wtyczka do Gedita): Control-R wciśnięte podczas edycji jakiegokolwiek pliku rozwiązania.*
- *Wysyłanie (linia komend):* `submit grader.c` *lub* `submit grader.cpp` *lub* `submit grader.pas`*.*
- *Wysyłanie (wtyczka do Gedita): Control-J wciśnięte podczas edycji jakiegokolwiek pliku rozwiązania lub sprawdzaczki.*

Marcin Kubica, Tomasz Kulczyński, Jakub Radoszewski
Tłumaczenie

IOI 2010, dzień drugi, 18.08.2010

Labirynt

Wielu farmerów zamieszkałych w południowej części prowincji Ontario tworzy na swoich polach kukurydzy labirynty. Labirynty robi się jesienią, już po zakończeniu żniw. To oznacza, że możesz jeszcze zdążyć zaprojektować najfajniejszy labirynt roku 2010.

Pole jest kratownicą pokrytą łodygami kukurydzy, nie licząc pewnych przeszkód (drzew, budynków i innych), na których kukurydza nie może rosnąć. Łodygi są szalenie wysokie i, podobnie jak przeszkody, mogą tworzyć ściany labiryntu. Ścieżki w labiryncie są tworzone poprzez zgniatanie łodyg na kwadratach jednostkowych o polu równym 1 metrowi kwadratowemu. Pewien kwadrat jednostkowy na brzegu pola stanowi wejście do labiryntu; pewien kwadrat jednostkowy stanowi środek labiryntu.

Jack co roku odwiedza liczne labirynty w kukurydzy, dzięki czemu stał się biegły w szybkim wyszukiwaniu ścieżek od wejścia do środka labiryntu. Projektujesz nowy labirynt. Twoim zadaniem jest wyznaczenie, które łodygi należy zgnieść, tak aby zmaksymalizować liczbę kwadratów jednostkowych, które Jack musi odwiedzić.

Sprawdzaczka określi, który kwadrat stanowi wejście (będzie to jedyny kwadrat na obwodzie), a który stanowi środek labiryntu (będzie to taki kwadrat, który zmusi Jacka do najdłuższego spaceru).

Mapa prostokątnego pola będzie miała reprezentację tekstową; przykładowo, pole o wymiarach 6 m na 10 m zawierające osiem drzew może mieć następującą reprezentację:

```
##X########
###X#######
####X###X##
##########
##XXXX####
##########
```

Symbol `#` *reprezentuje kwadrat jednostkowy z łodygami kukurydzy, a* `X` *reprezentuje kwadrat zawierający przeszkodę (np. drzewo), której nie można zgnieść, projektując ścieżki w labiryncie.*

Pole należy zamienić w labirynt, zgniatając kwadraty zajmowane przez kukurydzę. Jeden zgnieciony kwadrat (wejście) musi znajdować się na brzegu pola. Pozostałe zgniecione kwadraty muszą znajdować się we wnętrzu pola. Chcemy zmaksymalizować długość najkrótszej ścieżki od wejścia do środka labiryntu, zdefiniowaną jako liczba zgniecionych kwadratów, przez które musi przejść Jack, wliczając wejście i środek. Z jednego kwadratu jednostkowego można przejść do drugiego tylko wtedy, gdy oba zostały zgniecione i mają wspólną krawędź.

W swoim zgłoszeniu zgniecione kwadraty oznacz za pomocą kropek (`.`*). Dokładnie jeden ze zgniecionych kwadratów musi być położony na obwodzie. Przykładowo:*

```
#.X########
#.#X#...##
#...X#.X.#
#.#......#
#.XXXX##.#
##########
```

Poniżej zilustrowano poszukiwaną ścieżkę, przy czym wejście oznaczono jako `E`*, środek jako* `C`*, a pozostałą część ścieżki za pomocą znaków* `+`*. Ta ścieżka ma długość 12.*

```
#EX#######
#+#X#C+.##
#+++X#+X.#
#.#++++..#
#.XXXX##.#
##########
```

Katalog `/home/ioi2010-contestant/maze` *zawiera kilka plików tekstowych nazwanych* `field1.txt`*,* `field2.txt` *itd., zawierających mapy pól kukurydzy. Twoim zadaniem jest skopiowanie ich do plików nazwanych* `maze1.txt`*,* `maze2.txt` *itd. i przekształcenie ich w poprawne labirynty poprzez zastąpienie pewnych znaków* `#` *kropkami.*

Uwaga: Przy wstępnym sprawdzeniu (Public Test) serwer sprawdzający przyzna po jednym punkcie za każde podzadanie, do którego dostarczysz poprawne rozwiązanie (niezależnie od długości ścieżki). Wszystkie pozostałe punkty zostaną przyznane przy wyborze rozwiązania do pełnej oceny (Release Test). Łączny wynik zgłoszenia zostanie zaokrąglony do najbliższej liczby całkowitej między 0 a 110.

Podzadanie 1 [co najwyżej 11 punktów]

W pliku `field1.txt` *znajduje się pole opisane w treści zadania (rozmiaru* 6×10*). Stwórz na tym polu labirynt, w którym długość najkrótszej ścieżki od wejścia do środka wynosi* P*, i zapisz go do pliku* `maze1.txt`*. Twoim wynikiem za to podzadanie będzie minimum z 11 i* $10^{P/20}$*. Zauważ, że przykładowe rozwiązanie uzyskuje 3.98 punktu.*

Podzadanie 2 [co najwyżej 11 punktów]

Plik `field2.txt` *zawiera reprezentację pola rozmiaru* 100×100*. Stwórz na tym polu labirynt, w którym długość najkrótszej ścieżki od wejścia do środka wynosi* P*, i zapisz go do pliku* `maze2.txt`*. Twoim wynikiem za to podzadanie będzie minimum z 11 i* $10^{P/4000}$*.*

Podzadanie 3 [co najwyżej 11 punktów]

Plik `field3.txt` *zawiera reprezentację pola rozmiaru* 100×100*. Stwórz na tym polu labirynt, w którym długość najkrótszej ścieżki od wejścia do środka wynosi* P*, i zapisz go do pliku* `maze3.txt`*. Twoim wynikiem za to podzadanie będzie minimum z 11 i* $10^{P/4000}$*.*

Podzadanie 4 [co najwyżej 11 punktów]

Plik `field4.txt` *zawiera reprezentację pola rozmiaru* 100×100*. Stwórz na tym polu labirynt, w którym długość najkrótszej ścieżki od wejścia do środka wynosi* P*, i zapisz go do pliku* `maze4.txt`*. Twoim wynikiem za to podzadanie będzie minimum z 11 i* $10^{P/4000}$*.*

Podzadanie 5 [co najwyżej 11 punktów]

Plik `field5.txt` *zawiera reprezentację pola rozmiaru* 100×100*. Stwórz na tym polu labirynt, w którym długość najkrótszej ścieżki od wejścia do środka wynosi* P*, i zapisz go do pliku* `maze5.txt`*. Twoim wynikiem za to podzadanie będzie minimum z 11 i* $10^{P/5000}$*.*

Podzadanie 6 [co najwyżej 11 punktów]

Plik `field6.txt` *zawiera reprezentację pola rozmiaru* 11×11*. Stwórz na tym polu labirynt, w którym długość najkrótszej ścieżki od wejścia do środka wynosi* P*, i zapisz go do pliku* `maze6.txt`*. Twoim wynikiem za to podzadanie będzie minimum z 11 i* $10^{P/54}$*.*

Podzadanie 7 [co najwyżej 11 punktów]

Plik `field7.txt` *zawiera reprezentację pola rozmiaru* 20×20*. Stwórz na tym polu labirynt, w którym długość najkrótszej ścieżki od wejścia do środka wynosi* P*, i zapisz go do pliku* `maze7.txt`*. Twoim wynikiem za to podzadanie będzie minimum z 11 i* $10^{P/33}$*.*

Podzadanie 8 [co najwyżej 11 punktów]

Plik `field8.txt` *zawiera reprezentację pola rozmiaru* 20×20*. Stwórz na tym polu labirynt, w którym długość najkrótszej ścieżki od wejścia do środka wynosi* P*, i zapisz go do pliku* `maze8.txt`*. Twoim wynikiem za to podzadanie będzie minimum z 11 i* $10^{P/95}$*.*

Podzadanie 9 [co najwyżej 11 punktów]

Plik `field9.txt` *zawiera reprezentację pola rozmiaru* 11×21*. Stwórz na tym polu labirynt, w którym długość najkrótszej ścieżki od wejścia do środka wynosi* P*, i zapisz go do pliku* `maze9.txt`*. Twoim wynikiem za to podzadanie będzie minimum z 11 i* $10^{P/104}$*.*

Podzadanie 10 [co najwyżej 11 punktów]

Plik `fieldA.txt` *zawiera reprezentację pola rozmiaru* 200×200*. Stwórz na tym polu labirynt, w którym długość najkrótszej ścieżki od wejścia do środka wynosi* P*, i zapisz go do pliku* `mazeA.txt`*. Twoim wynikiem za to podzadanie będzie minimum z 11 i* $10^{P/7800}$*.*

Szczegóły implementacyjne

- *W tym zadaniu powinieneś dostarczyć jedynie pliki wyjściowe.*
- *Katalog do implementacji:* `/home/ioi2010-contestant/maze/` *(prototyp w pliku* `maze.zip`*).*
- *Do wysłania przez zawodnika:* `maze1.txt`*,* `maze2.txt`*,* `maze3.txt`*,* `maze4.txt`*,* `maze5.txt`*,* `maze6.txt`*,* `maze7.txt`*,* `maze8.txt`*,* `maze9.txt`*,* `mazeA.txt`*.*

- *Interfejs dla zawodnika:* **brak**.
- *Interfejs sprawdzaczki:* **brak**.
- *Przykładowa sprawdzaczka:* `grader.c` *lub* `grader.cpp` *lub* `grader.pas`.
- *Wejście przykładowej sprawdzaczki:* `grader.in.1`, `grader.in.2` *itd.*
  Uwaga: katalog do implementacji zawiera bardzo proste rozwiązania `maze1.txt`**,** `maze2.txt` **itd. W celu testowania, skopiuj je do plików** `grader.in.1`**,** `grader.in.2` **itd.**
- *Oczekiwane wyjście dla wejścia przykładowej sprawdzaczki: jeśli wejście stanowi poprawny labirynt dla podzadania numer N, przykładowa sprawdzaczka wypisze „*`OK` *N P”, przy czym P jest długością ścieżki.*
- *Kompilacja i uruchamianie (linia komend):* `runc grader.c` *lub* `runc grader.cpp` *lub* `runc grader.pas`.
- *Kompilacja i uruchamianie (wtyczka do Gedita): Control-R wciśnięte podczas edycji pliku sprawdzaczki.*
- *Wysyłanie (linia komend):* `submit maze1.txt` *lub* `submit maze2.txt` *itd.* **Wysłane zostaną wszystkie pliki** `.txt` **z katalogu do implementacji, niezależnie od tego, który plik zostanie podany komendzie** `submit`**.**
- *Wysyłanie (wtyczka do Gedita): Control-J wciśnięte podczas edycji jakiegokolwiek pliku* `.txt`.

Marcin Kubica, Tomasz Kulczyński, Jakub Radoszewski
Tłumaczenie

IOI 2010, dzień drugi, 18.08.2010

Oszczędny kod

Firma kurierska Xetzcop umożliwia przesyłanie paczek między miastami za pomocą poczty lotniczej. W niektórych z tych miast znajdują się sortownie paczek Xetzcopu — miasta takie nazywamy **centralami**. *Xetzcop utrzymuje połączenia lotnicze między miastami, każde połączenie przewozi paczki między dwoma miastami w obie strony.*

Aby paczka została przewieziona z jednego miasta do drugiego, musi ona odbyć ciąg „lotów", przy czym każdy lot polega na przewiezieniu paczki między miastami jednym z połączeń lotniczych Xetzcopu. Co więcej, taki ciąg lotów musi przechodzić przynajmniej przez jedną centralę.

Aby ułatwić przesyłanie paczek, Xetzcop chce na wszystkich paczkach naklejać naklejki z zakodowanymi długościami najkrótszych sekwencji lotów z każdego miasta do każdej centrali. (Długość najkrótszej sekwencji z centrali do niej samej wynosi zero). Oczywiście, potrzebny jest zwięzły sposób zakodowania takiej informacji.

Twoim zadaniem jest zaimplementowanie dwóch procedur: **encode**(N, H, P, A, B) *i* **decode**(N, H). N *oznacza liczbę miast, a* H *oznacza liczbę central. Miasta są ponumerowane od 0 do* $N-1$, *przy czym centrale są w miastach o numerach od 0 do* $H-1$. *Możesz założyć, że* $N \leqslant 1000$ *oraz* $H \leqslant 36$. P *oznacza liczbę par miast, między którymi Xetzcop utrzymuje połączenia lotnicze. Możesz założyć, że wszystkie podane (nieuporządkowane) pary miast będą różne.* A *i* B *to tablice liczb rozmiaru* P, *takie że* $(A[0], B[0])$ *jest pierwszą parą miast, między którymi jest połączenie lotnicze Xetzcopu,* $(A[1], B[1])$ *jest drugą taką parą, itd.*

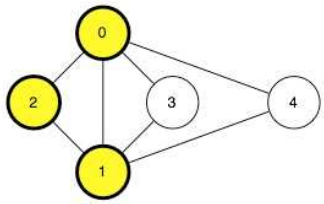
Procedura **encode** *musi skonstruować sekwencję bitów, na podstawie której procedura* **decode** *będzie mogła wyznaczyć liczbę lotów między każdym miastem i każdą centralą. Procedura* **encode** *powinna wysłać tę sekwencję bitów do sprawdzaczki, wywołując wielokrotnie procedurę* **encode_bit**(b), *gdzie b to 0 lub 1. Procedura* **decode** *może odebrać od sprawdzaczki ciąg wysłanych bitów, wywołując wielokrotnie* **decode_bit** *— i-te wywołanie* **decode_bit** *zwróci wartość b z i-tego wywołania* **encode_bit**(b). *Zważ, że procedura* **decode** *może wywoływać* **decode_bit** *co najwyżej tyle razy, ile razy procedura* **encode** *uprzednio wywołała procedurę* **encode_bit**(b).

Po odkodowaniu liczb lotów, procedura **decode** *musi wywołać* **hops**(h, c, d) *dla każdej centrali* h *i każdego miasta* c *(w tym również dla każdej centrali, tzn. również dla* $c = h$*), podając minimalną liczbę lotów* d *potrzebnych do przesłania paczki między* h *i* c. *A zatem procedura* **hops**(h, c, d) *musi zostać wywołana* $N \cdot H$ *razy. Kolejność tych wywołań nie ma znaczenia. Możesz założyć, że zawsze da się przesłać paczkę między każdą centralą i każdym miastem.*

Uwaga: procedury encode i decode mogą komunikować się wyłącznie poprzez podany interfejs. Współdzielenie zmiennych, dostęp do plików czy sieci są zabronione. W C i C++ można zadeklarować zmienne trwałe (persistent) jako `static`, tak aby miała do nich dostęp tylko jedna z procedur encode lub decode, a współdzielenie ich nie było możliwe. W Pascalu można zadeklarować zmienne trwałe (persistent) w części `implementation` plików z rozwiązaniem.

Przykład

Dla przykładu, rozważmy poniższy diagram. Widnieje na nim pięć miast ($N = 5$) i siedem połączeń lotniczych ($P = 7$). W miastach 0, 1 i 2 znajdują się centrale ($H = 3$). Do przesłania paczki między centralą 0 i miastem 3 wystarczy jeden lot, natomiast do przesłania paczki między centralą 2 i miastem 3 potrzebne są 2 loty. Dane przedstawione w tym przykładzie znajdują się w pliku `grader.in.1`.



Poniższa tabela zawiera liczby lotów d, jakie musi przekazać procedura **decode**, *wywołując* **hops**(h, c, d)*:*

d		**Miasto** c				
		0	**1**	**2**	**3**	**4**
Centrala h	**0**	*0*	*1*	*1*	*1*	*1*
	1	*1*	*0*	*1*	*1*	*1*
	2	*1*	*1*	*0*	*2*	*2*

Podzadanie 1 [25 punktów]

Procedura **encode** *nie może wywołać* **encode_bit**(b) *więcej niż 16 000 000 razy.*

Podzadanie 2 [25 punktów]

Procedura **encode** *nie może wywołać* **encode_bit**(b) *więcej niż 360 000 razy.*

Podzadanie 3 [25 punktów]

Procedura **encode** *nie może wywołać* **encode_bit**(b) *więcej niż 80 000 razy.*

Podzadanie 4 [25 punktów]

Procedura **encode** *nie może wywołać* **encode_bit**(b) *więcej niż 70 000 razy.*

Szczegóły implementacyjne

- *Używaj środowiska do programowania i testowania RunC.*

- *Katalog do implementacji:* `/home/ioi2010-contestant/saveit/` *(prototyp w pliku* `saveit.zip`*).*
- *Do implementacji przez zawodnika:*
  - `encoder.c` *lub* `encoder.cpp` *lub* `encoder.pas`*, oraz*
  - `decoder.c` *lub* `decoder.cpp` *lub* `decoder.pas`*.*
- *Interfejs dla zawodnika:*
  - `encoder.h` *lub* `encoder.pas`*, oraz*
  - `decoder.h` *lub* `decoder.pas`*.*
- *Interfejs sprawdzaczki:* `grader.h` *lub* `graderlib.pas`*.*
- *Przykładowa sprawdzaczka:* `grader.c` *lub* `grader.cpp` *lub* `grader.pas` *oraz* `graderlib.pas`*.*
- *Wejście przykładowej sprawdzaczki:* `grader.in.1`*,* `grader.in.2` *itd.*
  Uwaga: Pierwszy wiersz pliku zawiera trzy liczby: N**,** P **i** H**. Kolejne** P **wierszy zawiera pary miast** $A[0]\ B[0]$**,** $A[1]\ B[1]$**, itd. Następne** $H \cdot N$ **wierszy zawiera liczby lotów z każdej centrali do każdego miasta (w tym do siebie samej i wszystkich pozostałych central); dokładniej, liczba lotów z centrali** i **do miasta** j **jest zapisana w (**$i \cdot N + j + 1$**)-szym z tych wierszy.**
- *Oczekiwane wyjście dla wejścia przykładowej sprawdzaczki:*
  - *Jeśli implementacja poprawnie rozwiązuje podzadanie 1, wyjście będzie zawierać* „`OK 1`”*.*
  - *Jeśli implementacja poprawnie rozwiązuje podzadanie 2, wyjście będzie zawierać* „`OK 2`”*.*
  - *Jeśli implementacja poprawnie rozwiązuje podzadanie 3, wyjście będzie zawierać* „`OK 3`”*.*
  - *Jeśli implementacja poprawnie rozwiązuje podzadanie 4, wyjście będzie zawierać* „`OK 4`”*.*
- *Kompilacja i uruchamianie (linia komend):* `runc grader.c` *lub* `runc grader.cpp` *lub* `runc grader.pas`*.*
- *Kompilacja i uruchamianie (wtyczka do Gedita): Control-R wciśnięte podczas edycji jakiegokolwiek pliku rozwiązania.*
- *Wysyłanie (linia komend):* `submit grader.c` *lub* `submit grader.cpp` *lub* `submit grader.pas`*.*
- *Wysyłanie (wtyczka do Gedita): Control-J wciśnięte podczas edycji jakiegokolwiek pliku rozwiązania lub sprawdzaczki.*

XVI Bałtycka Olimpiada Informatyczna,

Tartu, Estonia 2010

Jakub Łącki, Jakub Pawlewicz
Tłumaczenie

BOI 2010, dzień próbny, 30.04.2010

Tabelka

Przyjrzyjmy się poniższej tabelce z literami:

E	R	A	T
A	T	S	R
A	U	T	U

Zauważmy, że słowo TARTU można z niej odczytać na 7 sposobów:

E	**R**	**A**	**T**
A	**T**	S	R
A	**U**	T	U

E	R	**A**	T
A	**T**	S	**R**
A	**U**	**T**	U

E	**R**	**A**	T
A	**T**	S	R
A	**U**	T	U

E	R	A	**T**
A	T	S	**R**
A	U	**T**	**U**

E	**R**	A	T
A	**T**	S	R
A	**U**	T	U

E	R	**A**	T
A	**T**	S	**R**
A	U	**T**	**U**

E	R	**A**	**T**
A	T	S	**R**
A	**U**	**T**	U

Mając daną tabelkę oraz słowo, należy obliczyć, na ile sposobów można odczytać to słowo z tabelki.

Pierwszą literę słowa można odczytać z dowolnej komórki, a po przeczytaniu jakiejś litery, następną można odczytać jedynie z pewnej sąsiedniej komórki (w pionie, poziomie lub po skosie). W trakcie czytania słowa każdej komórki można użyć dowolnie wiele razy.

Wejście

Pierwszy wiersz pliku `grid.in` *zawiera trzy liczby całkowite: H ($1 \leqslant H \leqslant 200$), wysokość tabelki, W ($1 \leqslant W \leqslant 200$), szerokość tabelki, oraz L ($1 \leqslant L \leqslant 100$), długość słowa.*

W każdym z kolejnych H wierszy znajduje się po W liter opisujących tabelkę. Ostatni wiersz zawiera L liter opisujących słowo. Wszystkie litery w tabelce i słowie są wielkimi literami alfabetu angielskiego ($A \ldots Z$).

Wyjście

Pierwszy i jedyny wiersz pliku `grid.out` *powinien zawierać jedną liczbę całkowitą: liczbę możliwych sposobów przeczytania słowa z tabelki. Możesz założyć, że odpowiedź nie będzie większa niż 10^{18}.*

Przykład

Dla pliku wejściowego `grid.in`*:*

```
3 4 5
ERAT
ATSR
AUTU
TARTU
```

poprawnym wynikiem jest plik wyjściowy `grid.out`*:*

```
7
```

natomiast dla pliku wejściowego `grid.in`*:*

```
2 2 10
AA
AA
```

poprawnym wynikiem jest plik wyjściowy `grid.out`*:*

```
78732
```

Jakub Łącki, Jakub Pawlewicz
Tłumaczenie

BOI 2010, dzień pierwszy, 01.05.2010

Lego

Swoją pracę naukową na temat rozpoznawania obrazów postanowiłeś połączyć z układaniem klocków Lego. Stworzyłeś już skomplikowany system rozpoznający obrazy, a teraz musisz napisać osobne narzędzie, by przetestować dotychczasowe wyniki pracy. Napisz program, który dla podanych dwóch rzutów prostopadłych pewnej konstrukcji z klocków Lego, obliczy, na ile różnych sposobów może ona zostać zbudowana.

W tym zadaniu zakładamy, że istnieje tylko jeden rodzaj klocków o rozmiarze 2×2 (patrz rysunek 1), który występuje w trzech kolorach: białym (`W`)*, szarym* (`G`) *lub czarnym* (`B`)*. W każdym kolorze dostępna jest nieskończona liczba klocków. Klocki stawiane są na kwadratowej podstawie rozmiaru 6×6. Każdy klocek należy postawić równolegle do podstawy, tak aby nie wystawał poza nią i opierał się (przynajmniej jednym z czterech zaczepów) na podstawie lub dobrze umocowanym klocku.*

Rys. 1: Po lewej: Dopuszczalne ustawienie klocka na wierzchu innego. Pośrodku: Niedopuszczalne ustawienie (górny klocek unosi się w powietrzu). Po prawej: Jeszcze jedna niepoprawna konfiguracja (górny klocek wystaje poza podstawę).

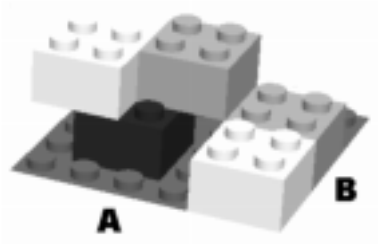
Wejście

Pierwszy wiersz pliku `lego.in` *zawiera liczbę całkowitą H ($1 \leqslant H \leqslant 6$) określającą wysokość konstrukcji. W każdym z kolejnych H wierszy znajduje się po 6 znaków, opisujących rzut konstrukcji z jednej strony (oznaczonej przez A na rysunku 2). Znak numer j w i-tym wierszu opisuje to, co widać w j-tej kolumnie od lewej i i-tym wierszu od góry. Każdy ze znaków to* '`W`'*,* '`G`'*,* '`B`' *lub* '`.`' *i określa kolor klocka* ('`W`', '`G`' *lub* '`B`') *lub dziurę* ('`.`')*. Zauważ, że patrząc na konstrukcję, nie można ocenić głębokości, na jakiej znajduje się widoczny klocek, zatem może on być zarówno w pobliżu przedniej krawędzi konstrukcji, jak i nieco dalej, o ile żaden inny klocek go nie zasłania.*

Następne H wierszy pliku wejściowego opisuje rzut (oznaczony przez B na rysunku 2), jaki widzi obserwator po obróceniu się o 90° przeciwnie do ruchu wskazówek zegara wokół osi budowli.

Wyjście

Do pliku wyjściowego `lego.out` *należy wypisać jeden wiersz z jedną liczbą całkowitą równą liczbie różnych poprawnych konstrukcji z klocków Lego, które są zgodne z rzutami opisanymi*



Rys. 2: Jedna z konstrukcji zgodnych z przykładowym wejściem.

w wejściu. Zauważ, że dwie budowle uznajemy za różne nawet, jeśli jedną można uzyskać z drugiej przez obrót lub symetrię. Dla danych wejściowych wynik zawsze zmieści się w typie 64-bitowym ze znakiem.

Przykład

Dla pliku wejściowego `lego.in`*:*

```
2
WWGG..
.BB.WW
.WGG..
WWGG..
```

poprawnym wynikiem jest plik wyjściowy `lego.out`*:*

```
6
```

Jakub Łącki, Jakub Pawlewicz
Tłumaczenie

BOI 2010, dzień pierwszy, 01.05.2010

Misie

Miasto Bezkres jest podzielone na kwadraty jednostkowe przez nieskończenie wiele nieskończonych, dwukierunkowych ulic prowadzących z południa na północ (tzw. południkowych) i ze wschodu na zachód (równoleżnikowych). Jedną z południkowych ulic oznaczamy numerem 0, zaś numery pozostałych rosną na wschód i maleją na zachód. Podobnie, jedną z równoleżnikowych ulic oznaczamy numerem 0, a numery pozostałych rosną na północ i maleją na południe.

Każde skrzyżowanie ulic oznaczone jest parą liczb, w której pierwsza jest numerem ulicy południkowej, a druga — równoleżnikowej. Niektóre odcinki ulic mają większe znaczenie i nazywamy je głównymi.

Pewnego dnia komisarz Ryba (najbardziej żarliwy stróż prawa w Bezkresie) podczas patrolowania ulic spostrzega na skrzyżowaniu (A, B) samochód z kilkoma członkami sławnego gangu Misiów. Ryba ma cynk, że Misie planują włamać się do miejskiego Miodopoju zlokalizowanego przy skrzyżowaniu $(0, 0)$, więc postanawia ich zatrzymać.

Jak do tej pory Misie nie popełniły żadnego wykroczenia i Ryba nie ma prawa ich aresztować. Może za to zatrzymać swój radiowóz na dowolnym skrzyżowaniu i zablokować nim wjazd w dokładnie jedną z czterech ulic wychodzących z tego skrzyżowania, przy czym nie może to być wjazd w główny odcinek jakiejś ulicy.

Zatem Ryba postanawia napsuć trochę krwi Misiom. Tuż przed tym, jak dotrą oni do jakiegoś skrzyżowania, komisarz może wyprzedzić ich samochód i zablokować jeden wyjazd z tego skrzyżowania. W ten sposób gang Misiów będzie mógł wjechać na to skrzyżowanie, ale nie będzie mógł go opuścić zablokowaną ulicą.

Celem Ryby jest trzymanie gangu jak najdalej od Miodopoju. Znajdź największą odległość D taką, że każde skrzyżowanie (x, y), do którego Misie są w stanie dotrzeć, spełnia warunek $max(|x|, |y|) \geqslant D$.

Wejście

Pierwszy wiersz pliku `bears.in` *składa się z dwóch liczb całkowitych A oraz B ($|A| \leqslant 10^6$, $|B| \leqslant 10^6$) określających punkt startowy Misiów. Drugi wiersz zawiera jedną liczbę całkowitą N ($0 \leqslant N \leqslant 500$) oznaczającą liczbę głównych odcinków ulic. Każdy z kolejnych N wierszy zawiera cztery liczby całkowite X_1, Y_1, X_2, Y_2 ($|X_i| \leqslant 10^6$, $|Y_i| \leqslant 10^6$), które oznaczają, że odcinek pomiędzy skrzyżowaniami (X_1, Y_1) i (X_2, Y_2) jest główny. Zachodzi wówczas równość $X_1 = X_2$ lub $Y_1 = Y_2$.*

Wyjście

Jedyny wiersz pliku `bears.out` *powinien zawierać jedną liczbę całkowitą D — szukaną największą odległość, na jaką Ryba może trzymać Misie od Miodopoju.*

Przykład

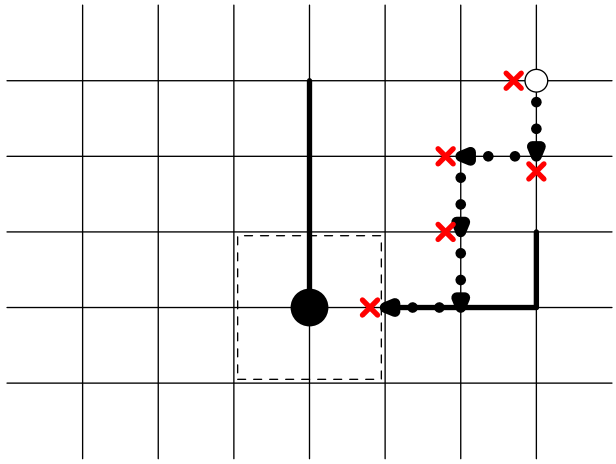
Dla pliku wejściowego `bears.in`*:*

```
3 3
3
1 0 3 0
0 0 0 3
3 0 3 1
```

poprawnym wynikiem jest plik wyjściowy `bears.out`*:*

```
1
```

Poniższy rysunek obrazuje, jak Misie mogą dostać się na odległość 1 od Miodopoju:



Mimo że Misie mogą dalej próbować dostać się do Miodopoju, Ryba nie pozwoli im dotrzeć bliżej skrzyżowania $(0, 0)$.

Jakub Łącki, Jakub Pawlewicz
Tłumaczenie

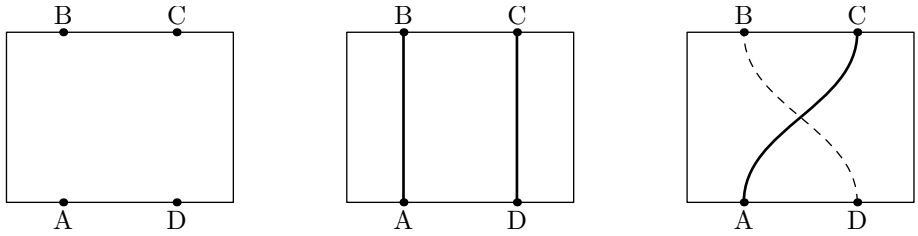
BOI 2010, dzień pierwszy, 01.05.2010

Obwód drukowany

Obwód drukowany to płytka z materiału izolacyjnego, na którą nanosi się połączenia (zwane ścieżkami) przewodzące prąd. Jeśli ścieżki na tej samej warstwie przecinają się, powstaje zwarcie, dlatego czasem używa się obwodów wielowarstwowych i izoluje poszczególne warstwy od siebie. Płytki wielowarstwowe są jednak droższe, dlatego warto szukać takiego podziału ścieżek pomiędzy warstwy, który wymaga minimalnej liczby warstw.

W tym zadaniu zajmujemy się prostokątnymi obwodami, w których każda ścieżka łączy dwa porty na przeciwległych bokach. Naszym celem jest zminimalizowanie kosztu obwodu.

Spójrzmy na skrajnie lewy obwód na poniższym rysunku. Połączenia A z B oraz D z C można wykonać na jednej warstwie, tak jak pokazano na środkowym obwodzie na rysunku. Jednakże nie da się poprowadzić ścieżek z A do C i z D do B w ramach tej samej warstwy, co widać na rysunku po prawej.



Napisz program, który obliczy minimalną liczbę warstw płytki rozmiaru $W \times H$, które są potrzebne do przeprowadzenia N ścieżek, mających połączyć wskazane porty, tak aby ścieżki te nie przecinały się w ramach jednej warstwy.

Możesz założyć, że szerokość przewodów jest zaniedbywalnie mała w porównaniu do odległości pomiędzy portami. Oznacza to, że pomiędzy dowolnymi dwiema ścieżkami zawsze można zmieścić trzecią.

Wejście

Pierwszy wiersz pliku `pcb.in` *zawiera jedną dodatnią liczbę całkowitą N ($N \leqslant 10^5$), określającą liczbę ścieżek. W każdym z kolejnych N wierszy znajdują się dwie liczby całkowite X_{i1} i X_{i2} ($0 \leqslant X_{ij} \leqslant 10^6$) oddzielone odstępem. Oznaczają one, że i-ta ścieżka łączy porty o współrzędnych $(X_{i1}, 0)$ i (X_{i2}, H). Możesz założyć, że wszystkie $2N$ portów wymienionych na wejściu jest parami różnych.*

Wyjście

Pierwszy i jedyny wiersz pliku `pcb.out` *powinien zawierać jedną liczbę całkowitą: minimalną liczbę warstw potrzebną do wykonania podanych połączeń.*

Przykład

Dla pliku wejściowego `pcb.in`*:*

```
2
1 1
3 3
```

poprawnym wynikiem jest plik wyjściowy `pcb.out`*:*

```
1
```

natomiast dla pliku wejściowego `pcb.in`*:*

```
2
1 3
3 1
```

poprawnym wynikiem jest plik wyjściowy `pcb.out`*:*

```
2
```

Jakub Łącki, Jakub Pawlewicz
Tłumaczenie

BOI 2010, dzień drugi, 02.05.2010

Cukierki

Bajtazar pracuje w sklepie ze słodyczami. W sklepie jest N pudełek z cukierkami, w każdym z nich może znajdować się inna liczba cukierków. Gdy zjawia się klient i prosi o K cukierków, Bajtazar idzie na zaplecze i przynosi opakowania, w których łącznie jest dokładnie K cukierków. Jeśli nie może tego zrobić, na przykład jeśli ktoś prosi o 4 cukierki, a na zapleczu jest 5 pudełek z 3 cukierkami w każdym z nich, klient zazwyczaj odchodzi bez kupowania czegokolwiek.

Z tego powodu Bajtazar postanowił obliczyć, ile różnych rozmiarów zamówień może obsłużyć przy użyciu opakowań, które aktualnie posiada. Udało mu się rozwiązać ten problem z łatwością i teraz zastanawia się, co zrobić, aby poprawić ten wynik. Chciałby otworzyć jedno pudełko i zmienić w nim liczbę cukierków, tak aby mógł obsłużyć jak największą liczbę rozmiarów zamówień, o które może poprosić najbliższy klient.

Wejście

Pierwszy wiersz pliku `candies.in` *zawiera jedną liczbę całkowitą* N $(2 \leqslant N \leqslant 100)$ *oznaczającą liczbę pudełek z cukierkami. W drugim wierszu znajduje się ciąg* N *liczb całkowitych* B_i $(1 \leqslant B_i \leqslant 7\,000)$ *pooddzielanych pojedynczymi odstępami. Opisują one liczby cukierków w każdym z pudełek.*

Wyjście

W pierwszym i jedynym wierszu pliku `candies.out` *należy wypisać dwie liczby całkowite* P *i* Q *oddzielone pojedynczym odstępem. Oznaczają one, że Bajtazar powinien zmienić liczbę cukierków w pudełku z* P *cukierkami na* Q*.* P *musi być równe jednemu z* B_i*. Może być wiele optymalnych rozwiązań; należy wybrać to rozwiązanie, w którym* P *jest jak najmniejsze. Wśród wszystkich wyników z minimalnym* P*, wybierz ten z najmniejszym* Q*. Możesz założyć, że poprzez zmianę liczby cukierków w pewnym, dokładnie jednym pudełku Bajtazar może zwiększyć liczbę różnych zamówień, które jest w stanie obsłużyć.*

Przykład

Dla pliku wejściowego `candies.in`*:*

```
4
1 3 4 4
```

poprawnym wynikiem jest plik wyjściowy `candies.out`*:*

```
4 9
```

a dla pliku wejściowego `candies.in`*:*

```
5
3 3 3 3 3
```

poprawnym wynikiem jest plik wyjściowy `candies.out`*:*

```
3 1
```

Przy użyciu pudełek opisanych w pierwszym przykładzie, Bajtazar może obsłużyć zamówienia 9 różnych rozmiarów, konkretnie: 1, 3, 4, 5, 7, 8, 9, 11 i 12. Po otwarciu pudełka z 4 cukierkami i zmianie liczby cukierków na 9, może on obsłużyć zamówienia o rozmiarach 1, 3, 4, 5, 7, 8, 9, 10, 12, 13, 14, 16 oraz 17, czyli łącznie 13 różnych zamówień.

Jakub Łącki, Jakub Pawlewicz
Tłumaczenie

BOI 2010, dzień drugi, 02.05.2010

Kubły

W pewnej fabryce znajduje się całe mnóstwo pustych kubłów ustawionych w jednym rzędzie. Kierownik magazynu porządkuje je przez wkładanie jednych kubłów w drugie, aby zwolnić trochę miejsca po lewej stronie rzędu. Kubły przemieszcza robot, który może podnieść kubeł, przenieść go w prawo, a następnie włożyć do większego kubła.

Warunki bezpieczeństwa nakazują, by każdy kubeł zawierał co najwyżej jeden inny kubeł, który musi być pusty. Poza tym kierownik magazynu ustalił, że po zakończeniu porządków wszystkie kubły zawierające inne kubły powinny znajdować się na lewo od wszystkich pustych kubłów.

Twoim zadaniem jest napisanie programu, który znajdzie największe możliwe K takie, że pierwsze K kubłów z lewej strony rzędu może być przełożone w pewnej kolejności do K bezpośrednio następnych kubłów.

Wejście

Pierwszy wiersz pliku `bins.in` *zawiera dwie liczby całkowite M oraz N ($1 \leqslant M \leqslant 1\,000$, $1 \leqslant N \leqslant 20\,000$) oddzielone pojedynczym odstępem. Oznaczają one odpowiednio rozmiar największego kubła oraz liczbę wszystkich kubłów. Drugi wiersz zawiera N liczb całkowitych A_i ($1 \leqslant A_i \leqslant M$) pooddzielanych pojedynczymi odstępami i określających rozmiary kubłów w kolejności od lewej do prawej.*

Wyjście

Pierwszy i jedyny wiersz pliku `bins.out` *powinien zawierać jedną liczbę całkowitą: największą liczbę K taką, że robot może przełożyć K skrajnie lewych kubłów do następnych K kubłów.*

Przykład

Dla pliku wejściowego `bins.in`*:*

```
5 10
2 2 1 4 3 2 5 4 2 3
```

poprawnym wynikiem jest plik wyjściowy `bins.out`*:*

```
4
```

Jakub Łącki, Jakub Pawlewicz
Tłumaczenie

BOI 2010, dzień drugi, 02.05.2010

Miny

Henio narysował planszę o wymiarach $H \times W$, na której każde pole jest puste bądź zawiera minę. Jędrek, kolega Henia, narysował kolejną planszę $H \times W$. W każdym polu wpisał sumaryczną liczbę min na danym polu i na wszystkich sąsiednich polach (dwa pola uznajemy za sąsiednie, jeżeli stykają się bokiem bądź rogiem), po czym wyrzucił planszę Henia. Pozostała tylko plansza Jędrka z wpisanymi liczbami. Twoim zadaniem jest odtworzenie na jej podstawie oryginalnej planszy Henia.

Możesz założyć, że zawsze istnieje co najmniej jedno rozwiązanie.

Wejście

Pierwszy wiersz pliku wejściowego zawiera dwie liczby całkowite H oraz W ($1 \leqslant H, W \leqslant 600$) określające odpowiednio wysokość i szerokość planszy. Następne H wierszy zawiera po W cyfr opisujących planszę Jędrka.

Wyjście

Do pliku wyjściowego należy wypisać H wierszy, z których każdy powinien zawierać W znaków. Znak '`X`*' oznacza minę, zaś '*`.`*' — puste pole. Jeżeli istnieje wiele poprawnych rozwiązań, wypisz dowolne z nich.*

Przykład

Dla pliku wejściowego `mines.in`*:*

```
3 5
24531
46631
34310
```

poprawnym wynikiem jest plik wyjściowy `mines.out`*:*

```
.XXX.
.XX..
XX...
```

Punktacja

W tym zadaniu udostępniono pliki wejściowe o nazwach od `mines.01.in` *do* `mines.10.in`*. Rozwiązanie polega na wysłaniu odpowiadających plików wyjściowych. Nie należy wysyłać programu użytego do ich wygenerowania.*

XVII Olimpiada Informatyczna Krajów Europy Środkowej,

Koszyce, Słowacja 2010

Tomasz Kulczyński, Jakub Radoszewski
Tłumaczenie

CEOI 2010, dzień pierwszy, 14.07.2010

Ochroniarze

Czy spotkałeś kiedykolwiek członków dziesięciu europejskich rodzin królewskich, reprezentację piłkarską Argentyny wraz z ich trenerem, Diego Maradoną, lub wszystkich laureatów nagrody Turinga i medalu Fieldsa? Otóż na ceremonię zakończenia CEOI 2010 zaprosiliśmy wiele znanych postaci z całego świata. Niestety, bardzo niewielu z nich odpisało na zaproszenie, a ci, którzy to uczynili, grzecznie odmówili. Tak czy inaczej, nie zapomnij wziąć swojego aparatu fotograficznego na ceremonię, bo nigdy nie wiesz, kto może się na niej pojawić!

Łatwo sobie wyobrazić, że bezpieczeństwo gości jest kwestią priorytetową. Problem stanowi rozmieszczenie ich ochroniarzy wśród publiczności, tak aby zagwarantować maksymalny poziom bezpieczeństwa.

W audytorium jest wiele siedzeń, ustawionych na planie dużej kratownicy. Bazując na odpowiednich przepisach, ekspert do spraw bezpieczeństwa wyznaczył niezbędną liczbę ochroniarzy w każdym wierszu i w każdej kolumnie siedzeń.

Zadanie

Masz dane niezbędne liczby ochroniarzy w każdym wierszu i w każdej kolumnie siedzeń w audytorium. Te informacje są podane w sposób skompresowany, wyjaśniony poniżej. Sprawdź, czy jest możliwe rozmieszczenie ochroniarzy w taki sposób, aby w każdym wierszu i w każdej kolumnie znajdowało się dokładnie tylu ochroniarzy, ilu jest niezbędnych.

Zakładamy, że audytorium jest początkowo puste, tzn. możesz rozmieścić ochroniarzy, gdziekolwiek chcesz. Na każdym siedzeniu może znajdować się co najwyżej jeden ochroniarz.

Specyfikacja wejścia

Wejście rozpoczyna się opisem wierszy siedzeń. Pierwsza linia wejścia zawiera jedną liczbę całkowitą R: liczbę grup wierszy. Dalej następuje R linii. Każda z tych linii zawiera dwie liczby całkowite: niezbędną liczbę ochroniarzy w każdym wierszu grupy oraz liczbę wierszy tworzących tę grupę.

Dalej następuje opis grup kolumn. Rozpoczyna się on linią zawierającą jedną liczbę całkowitą C: liczbę grup kolumn. Dalej następuje C linii. Każda z tych linii zawiera dwie liczby całkowite: niezbędną liczbę ochroniarzy w każdej kolumnie grupy oraz liczbę kolumn tworzących tę grupę.

Ograniczenia

Możesz założyć, że łączna liczba ochroniarzy zawarta w ograniczeniach dla wierszy jest taka sama jak łączna liczba ochroniarzy zawarta w ograniczeniach dla kolumn.

Możesz założyć, że ta łączna liczba ochroniarzy nie przekracza 10^{18}.

Możesz założyć, że wszystkie liczby występujące na wejściu są całkowite, dodatnie i nie przekraczają 1 000 000 000.

Możesz założyć, że $1 \leqslant R, C \leqslant 200\,000$.

Pewne grupy testów, warte łącznie 50 punktów, spełniają następujące kryteria:

- *łączna liczba wierszy we wszystkich grupach nie przekracza 2 000*
- *łączna liczba kolumn we wszystkich grupach nie przekracza 2 000*
- *łączna liczba ochroniarzy nie przekracza 1 000 000.*

W grupie testów wartej kolejne 10 punktów, w każdym teście zachodzi $R, C \leqslant 100$.

Specyfikacja wyjścia

Jeśli można spełnić wszystkie ograniczenia, wypisz jedną linię zawierającą liczbę „`1`*", a w przeciwnym przypadku wypisz jedną linię zawierającą liczbę „*`0`*" (bez cudzysłowów).*

Przykłady

Dla danych wejściowych:	*poprawnym wynikiem jest:*

`2`
`2 1`
`1 2`
`1`
`2 2`
	`1`

Mamy dwie grupy wierszy: pierwsza z nich składa się z jednego wiersza, który musi zawierać dwóch ochroniarzy, druga składa się z dwóch wierszy, z których każdy musi zawierać jednego ochroniarza. Jest jedna grupa kolumn: każda z dwóch kolumn musi zawierać dwóch ochroniarzy. Jedno z przykładowych rozmieszczeń ochroniarzy to:

```
XX
X.
.X
```

Dla danych wejściowych:	*poprawnym wynikiem jest:*

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

Dwa wiersze muszą być w całości wypełnione ochroniarzami. Stąd w każdej kolumnie musi być co najmniej dwóch ochroniarzy. Jednakże ostatnia kolumna musi zawierać dokładnie jednego ochroniarza, co stanowi sprzeczność.

Tomasz Kulczyński, Jakub Radoszewski
Tłumaczenie

CEOI 2010, dzień pierwszy, 14.07.2010

Prostokąt arytmetyczny

Po lekcji o ciągach arytmetycznych nauczyciel dał Piotrusiowi zadanie domowe: kartkę papieru z kratownicą o wymiarach $R \times C$, której niektóre pola były wypełnione liczbami. Pewna liczba pól była pusta (potencjalnie zero). Zadaniem Piotrusia jest utworzenie prostokąta arytmetycznego poprzez wypełnienie pustych pól liczbami. W prostokącie arytmetycznym liczby w każdym wierszu i w każdej kolumnie tworzą ciąg arytmetyczny w kolejności występowania w tym wierszu/kolumnie.

Dla przykładu, to jest prostokąt arytmetyczny rozmiaru 3×5:

1	3	5	7	9
2	2	2	2	2
3	1	-1	-3	-5

Piotruś jest zbyt leniwy, by rozwiązywać takie zadania ręcznie. Chciałby mieć program, który zrobi to za niego.

Zadanie

Masz dany prostokąt wypełniony **liczbami całkowitymi** *oraz kropkami. Sprawdź, czy jest możliwe zastąpienie kropek pewnymi* **liczbami wymiernymi**, *tak aby otrzymać prostokąt arytmetyczny. Jeśli istnieje rozwiązanie, wypisz jedno z nich.*

Uwaga: Ciągiem arytmetycznym nazywamy taki ciąg liczb, w którym różnica pomiędzy każdymi dwoma kolejnymi elementami jest stała.

Specyfikacja wejścia

Pierwszy wiersz wejścia zawiera dwie liczby całkowite dodatnie R oraz C: wymiary prostokąta. Potem następuje R wierszy, z których każdy zawiera C elementów pooddzielanych pojedynczymi odstępami. Każdy z tych elementów jest albo kropką (.), albo liczbą całkowitą.

Ograniczenia

Każda z liczb wpisanych już do prostokąta jest między -100 a 100, włącznie. Testy dzielą się na 10 grup, z których każda warta jest 10 punktów. W grupach od 1 do 9 zachodzi $1 \leqslant R, C \leqslant 6$. Własności poszczególnych grup podane są poniżej.

Grupa 1. *Kratownica jest całkowicie wypełniona liczbami.*
Grupa 2. *W każdym teście $R = 1$ lub $C = 1$.*
Grupa 3. *W każdym teście $R = C = 2$.*
Grupa 4. *Każdy test ma dokładnie jedno rozwiązanie, które można znaleźć za pomocą metody opisanej w pierwszym przykładzie (poniżej).*
Grupa 5. *Każdy test ma dokładnie jedno rozwiązanie, które na dodatek składa się tylko z liczb*

całkowitych.
Grupa 6. *Każdy test ma dokładnie jedno rozwiązanie.*
Grupa 7. *Każdy test ma dokładnie jedno rozwiązanie, które składa się tylko z liczb całkowitych, albo nie ma rozwiązania w ogóle.*
Grupa 8. *Każdy test ma dokładnie jedno rozwiązanie albo nie ma rozwiązania w ogóle.*
Grupa 9. *Dowolne testy.*
Grupa 10. *Dowolne testy, w których* $1 \leqslant R, C \leqslant 50$.

Specyfikacja wyjścia

Jeśli nie ma rozwiązania, wypisz jeden wiersz z napisem „`No solution.`" *(bez cudzysłowów). Jeśli jest wiele rozwiązań, wybierz jedno z nich.*

Wypisując rozwiązanie, umieść na wyjściu R *wierszy, każdy zawierający* C *liczb wymiernych pooddzielanych pojedynczymi odstępami.*

Każdą liczbę wymierną należy wypisać w formacie „`N/D`", *przy czym* `D` *jest dodatnie, a* `N` *i* `D` *są względnie pierwsze. Jeśli* `D` *jest równe 1, pomiń część* „`/D`".

Żadna z liczb na wyjściu nie może mieć więcej niż 20 cyfr. (Spełnienie tego wymagania nie powinno być trudne, jego jedynym celem jest ułatwienie sprawdzania odpowiedzi).

Przykłady

Dla danych wejściowych:

```
3 5
. . 3 . 5
. . . 5 .
. . . . 7
```

poprawnym wynikiem jest:

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

Powyższy przykład można rozwiązać następująco: najpierw zauważ, że drugą liczbą w ostatniej kolumnie musi być 6. Potem wypełnij pierwszy wiersz, drugi wiersz, a na koniec kolumny od pierwszej do czwartej.

Dla danych wejściowych:

```
1 6
4 . . 0 . .
```

poprawnym wynikiem jest:

```
4 8/3 4/3 0 -4/3 -8/3
```

Dla danych wejściowych:

```
1 4
1 2 . 2
```

poprawnym wynikiem jest:

```
No solution.
```

Dla danych wejściowych:

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

poprawnym wynikiem jest:

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

To jest jedno z wielu możliwych rozwiązań dla tych danych wejściowych.

Tomasz Kulczyński, Jakub Radoszewski
Tłumaczenie

CEOI 2010, dzień pierwszy, 14.07.2010

Sojusze

W urojonym świecie znajduje się prostokątna wyspa. Ma ona R mil szerokości oraz C mil długości i jest podzielona na $R \times C$ kwadratowych pól. Niektóre pola są niezamieszkałe, a w pozostałych znajdują się wioski różnych istot: elfów, ludzi, krasnoludów oraz hobbitów. Mówimy, że dwie wioski sąsiadują ze sobą, jeśli pola, na których są położone, mają wspólny bok.

Ostatnimi czasy wioski sparaliżował strach przed Bajtozłem. Aby poczuć się bezpieczniej, każda z wiosek chce zawrzeć sojusze militarne z niektórymi ze swych sąsiadów. Sojusz zawierany jest zawsze pomiędzy dwoma sąsiadami i jest porozumieniem obustronnym.

Zależnie od rasy, mieszkańcy wioski nie będą czuli się bezpiecznie, jeśli ich wioska nie zawrze konkretnej konfiguracji sojuszy:

- *Elfy są pewne siebie i potrzebują dokładnie jednego sojuszu.*
- *Ludzie potrzebują sojuszy z dokładnie dwoma sąsiadami.*
  Co więcej, pozostawienie dwóch przeciwległych stron ich wioski bez ochrony jest złe z taktycznego punktu widzenia. W związku z tym, wspomniani dwaj sojusznicy nie mogą być położeni po przeciwległych stronach wioski.
- *Krasnoludy potrzebują sojuszy z dokładnie trzema sąsiadami.*
- *Hobbici są niesłychanie bojaźliwi, więc potrzebują sojuszy ze wszystkimi czterema sąsiadami.*

Innymi słowy, nie licząc ludzi, każda wioska potrzebuje konkretnej liczby sojuszników i nie przejmuje się tym, którzy z sąsiadów będą jej sojusznikami. Ludzie natomiast mają dodatkowe wymaganie: sojusznicy nie mogą znajdować się po przeciwległych stronach wioski.

Warunki te muszą być spełnione niezależnie od położenia wioski na mapie. Przykładowo, wioska krasnoludów potrzebuje trzech sojuszników. Jeśli jest położona na brzegu, oznacza to, że musi mieć sojusze ze wszystkimi trzema sąsiadami. Jeśli znajduje się w rogu wyspy, jej mieszkańcy nigdy nie będą czuli się bezpiecznie.

Zadanie

Masz dany opis wyspy. Twoim zadaniem jest ustalić, czy możliwe jest takie utworzenie sojuszy, aby wszyscy mieszkańcy wyspy czuli się bezpiecznie. Jeśli odpowiedź jest pozytywna, powinieneś także znaleźć jakąś poprawną konfigurację sojuszy. W przypadku wielu możliwych konfiguracji wybierz dowolną z nich.

Specyfikacja wejścia

Pierwszy wiersz wejścia zawiera dwie liczby R i C oznaczające wymiary wyspy. Kolejne R wierszy zawiera opis wyspy. Każdy z wierszy zawiera C pooddzielanych pojedynczymi odstępami liczb od 0 do 4:

- *0 oznacza niezamieszkałe pole,*
- *1 oznacza wioskę elfów,*
- *2 oznacza wioskę ludzką,*
- *3 oznacza wioskę krasnoludów,*
- *4 oznacza wioskę hobbitów.*

(Zauważ, że liczba z wejścia zawsze odpowiada liczbie pożądanych sąsiadów danej wioski).

Ograniczenia

We wszystkich testach zachodzi $1 \leqslant R, C \leqslant 70$.

W testach wartych łącznie 55 punktów zachodzi $\min(R, C) \leqslant 10$. *Wśród nich znajdują się testy warte łącznie 15 punktów, w których zachodzi* $R \cdot C \leqslant 20$.

Kolejna grupa testów warta 10 punktów zawiera mapy, w których występują wyłącznie pola niezamieszkałe oraz wioski ludzkie. (Te testy nie są zawarte w testach wartych 55 punktów, o których mowa wyżej).

Specyfikacja wyjścia

Jeśli nie istnieje poprawne rozwiązanie, wypisz jeden wiersz z napisem „`Impossible!`" *(bez cudzysłowów). W przeciwnym wypadku wypisz jedną z poprawnych map sojuszy w formacie opisanym poniżej.*

Każde pole jest reprezentowane na wyjściu jako macierz 3×3 *znaków. Jeśli jest niezamieszkałe, odpowiadająca macierz powinna być wypełniona znakami* '`.`' *(kropka). W przeciwnym wypadku w środku powinien znaleźć się znak* '`O`' *(wielka litera O) reprezentujący samą wioskę oraz znaki* '`X`' *(wielka litera X) oznaczające konfigurację sojuszników. Pozostała część macierzy* 3×3 *powinna być wypełniona znakami* '`.`' *(kropka).*

Wszystkie możliwe konfiguracje sojuszników dla wiosek poszczególnych ras są przedstawione poniżej.

elves

```
...   .X.   ...   ...
.OX   .O.   XO.   .O.
...   ...   ...   .X.
```

humans

```
.X.   .X.   ...   ...
.OX   XO.   XO.   .OX
...   ...   .X.   .X.
```

dwarves

```
.X.   .X.   .X.   ...
.OX   XOX   XO.   XOX
.X.   ...   .X.   .X.
```

hobbits

```
.X.
XOX
.X.
```

Przykłady

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
............
.OXXOXXO....
.X..X..X....
.X..X..X....
.OXXOXXO....
.X..X..X....
.X..X..X....
.OXXOXXOXXO.
............
```

natomiast dla danych wejściowych:

```
1 2
2 1
```

poprawnym wynikiem jest:

```
Impossible!
```

Tomasz Kulczyński, Jakub Radoszewski
Tłumaczenie

CEOI 2010, dzień drugi, 16.07.2010

Odtwarzacz MP3

Grzegorz dostał właśnie odtwarzacz MP3, który ma wiele ciekawych funkcjonalności, a wśród nich blokadę klawiatury. Jeśli tylko odtwarzacz jest nieaktywny przez ponad T sekund, wszystkie jego przyciski zostają zablokowane. Jeśli blokada klawiatury jest włączona, żaden przycisk nie spełnia swojej normalnej funkcji, a gdy tylko jakikolwiek przycisk zostaje naciśnięty, blokada zostaje wyłączona.

Dla przykładu, załóżmy że $T = 5$ oraz że klawiatura jest aktualnie zablokowana. Grzegorz nacisnął przycisk A, poczekał 3 sekundy, nacisnął przycisk B, poczekał 5 sekund, nacisnął C, poczekał 6 sekund, a następnie nacisnął D. W tym przykładzie jedynie przyciski B oraz C zadziałały standardowo. Zauważ, że między naciśnięciami przycisków C oraz D włączyła się blokada klawiatury.

Głośność odtwarzacza MP3 kontroluje się za pomocą przycisków `+` oraz `-`, które odpowiednio zwiększają albo zmniejszają poziom głośności o 1 jednostkę. Poziom głośności jest liczbą całkowitą między 0 a V_{max}. Naciśnięcie przycisku `+` przy poziomie głośności V_{max} albo przycisku `-` przy poziomie głośności 0 nie zmienia aktualnej głośności.

Zadanie

Grzegorz nie zna wartości parametru T. Postanowił wyznaczyć tę wartość eksperymentalnie. Zaczynając z zablokowaną klawiaturą, wykonał sekwencję N naciśnięć przycisków `+` i/lub `-`. Na końcu eksperymentu odczytał z wyświetlacza odtwarzacza ostateczny poziom głośności. Niestety, zapomniał, jaka była głośność odtwarzacza przed rozpoczęciem eksperymentu. Nieznany, początkowy poziom głośności oznaczmy przez V_1, a znany, ostateczny poziom głośności oznaczmy przez V_2.

Masz daną wartość parametru V_2 oraz listę naciśnięć przycisków w takiej kolejności, w jakiej naciskał je Grzegorz. Dla każdego naciśnięcia znasz typ naciśniętego przycisku (`+` lub `-`) oraz liczbę sekund od początku eksperymentu do chwili, w której przycisk został naciśnięty. Twoim zadaniem jest wyznaczenie największej możliwej, **całkowitej** *wartości parametru T, zgodnej z wynikiem eksperymentu.*

Specyfikacja wejścia

Pierwszy wiersz wejścia zawiera trzy liczby całkowite N, V_{max} oraz V_2 ($0 \leqslant V_2 \leqslant V_{max}$) pooddzielane pojedynczymi odstępami. Każdy z kolejnych N wierszy zawiera opis jednego naciśnięcia w sekwencji: znak „`+`" lub „`-`" oraz liczba całkowita C_i ($0 \leqslant C_i \leqslant 2 \cdot 10^9$), oznaczająca liczbę sekund od chwili rozpoczęcia eksperymentu. Możesz założyć, że chwile naciśnięć są uporządkowane oraz że wszystkie te chwile są różne (tzn. $C_i < C_{i+1}$ dla każdego $1 \leqslant i < N$).

Ograniczenia

Możesz założyć, że $2 \leqslant N \leqslant 100\,000$ oraz $2 \leqslant V_{max} \leqslant 5\,000$.

W testach wartych łącznie 40 punktów zachodzi $N \leqslant 4\ 000$.

W testach wartych łącznie 70 punktów zachodzi $N \cdot V_{max} \leqslant 400\ 000$.

Specyfikacja wyjścia

Jeśli T *może być dowolnie duże, wypisz jeden wiersz zawierający słowo* „`infinity`" *(bez cudzysłowów).*

W przeciwnym przypadku wypisz jeden wiersz zawierający dwie liczby całkowite T *oraz* V_1 *oddzielone pojedynczym odstępem.*

Wypisane wartości muszą spełniać warunek, że wykonawszy eksperyment z czasem włączenia blokady T*, począwszy od poziomu głośności* V_1*, otrzymujemy końcowy poziom głośności* V_2*. Jeśli jest więcej niż jedno rozwiązanie, wybierz to z nich, które maksymalizuje wartość parametru* T*; jeśli wciąż jest więcej niż jedno rozwiązanie, wybierz to z nich, które maksymalizuje wartość parametru* V_1.

(Zauważ, że zawsze istnieje co najmniej jedno rozwiązanie: jeżeli $T = 0$*, to żaden z przycisków nie spełnia swojej normalnej funkcji, więc wystarczy przyjąć* $V_1 = V_2$*).*

Przykłady

Dla danych wejściowych:	*poprawnym wynikiem jest:*

`6 4 3`
`- 0`
`+ 8`
`+ 9`
`+ 13`
`- 19`
`- 24`
	`5 4`

Dla $T = 5$*, przyciski spełniają następujące funkcje: odblokuj, odblokuj,* `+`*,* `+`*, odblokuj,* `-`.

Dla dowolnego $V_1 \in \{2, 3, 4\}$ *otrzymalibyśmy* $V_2 = 3$*. Zauważ, że przykładowe wyjście zawiera największą możliwą wartość parametru* V_1.

Dla $T \geqslant 6$ *ostatnie dwa naciśnięcia klawiszy spełnią swoje normalne funkcje, toteż nie będzie możliwe uzyskanie wartości* $V_2 = 3$.

Dla danych wejściowych:	*poprawnym wynikiem jest:*

`3 10 10`
`+ 1`
`+ 2`
`+ 47`
	`infinity`

Jeżeli $V_1 = 10$*, to dla dowolnego* T *będzie zachodziło* $V_2 = 10$.

Tomasz Kulczyński, Jakub Radoszewski
Tłumaczenie

CEOI 2010, dzień drugi, 16.07.2010

Ogromna wieża

Starożytni Babilończycy postanowili zbudować ogromną wieżę. Wieża będzie składać się z N sześciennych bloków położonych jeden na drugim. Babilończycy zebrali z całego państwa wiele bloków różnych rozmiarów. Z ostatniej, nieudanej próby budowy dowiedzieli się, że jeśli ustawią większy blok bezpośrednio na dużo mniejszym, wieża przewróci się.

Zadanie

Zakładamy, że każde dwa bloki są różne, nawet jeśli są jednakowej wielkości. Dla każdego bloku znasz długość jego boku. Masz również daną liczbę całkowitą D, oznaczającą, że nie wolno położyć bloku A bezpośrednio na bloku B dokładnie wtedy, gdy długość boku bloku A jest ściśle większa niż D plus długość boku bloku B.

Oblicz liczbę różnych sposobów zbudowania wieży ze **wszystkich** *bloków. Ponieważ wynik może być bardzo duży, wypisz resztę z dzielenia go przez $10^9 + 9$.*

Specyfikacja wejścia

Pierwszy wiersz wejścia składa się z dwóch liczb całkowitych dodatnich N oraz D: liczby bloków oraz tolerancji określonej jak wyżej.

Drugi wiersz zawiera N pooddzielanych pojedynczymi odstępami liczb całkowitych; każda z nich jest długością boku jednego z bloków.

Ograniczenia

Wszystkie liczby na wejściu są całkowite, dodatnie i nie przekraczają 10^9.

N jest zawsze równe co najmniej 2.

W testach wartych 70 punktów N nie przekroczy 70.

Wśród nich są testy warte 45 punktów, w których N będzie równe co najwyżej 20.

Wśród tych z kolei są testy warte 10 punktów, w których N będzie równe co najwyżej 10.

W części testów całkowita liczba poprawnych wież będzie nie większa niż 1 000 000. Te testy są warte łącznie 30 punktów.

W ostatnich sześciu testach (wartych 30 punktów) N będzie większe niż 70. Górne ograniczenie na N w tych testach nie jest podane.

Specyfikacja wyjścia

Wypisz jeden wiersz zawierający jedną liczbę całkowitą: liczbę wież, które można zbudować, modulo 1 000 000 009.

Przykłady

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`4 1` `1 2 3 100`	`4`

Pierwsze trzy bloki można uporządkować w dowolnej kolejności poza $2, 1, 3$ *oraz* $1, 3, 2$*. Ostatni blok musi leżeć na spodzie.*

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`6 9` `10 20 20 10 10 20`	`36`

Nie można położyć bloku o boku 20 na bloku o boku 10. Istnieje sześć sposobów uporządkowania bloków o boku 10 i sześć sposobów uporządkowania bloków o boku 20.

Tomasz Kulczyński, Jakub Radoszewski
Tłumaczenie

CEOI 2010, dzień drugi, 16.07.2010

PIN

Marcin został administratorem sieci w dużej firmie. Firma ta nie zmieniała systemu autoryzacji pracowników od lat 80. XX wieku. Każdy pracownik ma przypisany czterocyfrowy numer PIN. Nie funkcjonują natomiast żadne nazwy użytkowników ani hasła, jedynym sposobem logowania jest wpisanie swojego PIN-u. Wraz z rozwojem firmy dodano możliwość używania liter, ale PIN nadal składa się tylko z czterech znaków.

Marcin jest niezadowolony z istniejącej sytuacji. Załóżmy bowiem, że w firmie jest dwoje pracowników, których PIN-y różnią się tylko na jednej pozycji, np. `61ab` *oraz* `62ab`*. Jeśli pierwszy z nich przypadkowo wciśnie* `2` *zamiast* `1`*, system zaloguje go jako tego drugiego pracownika. Marcin chciałby poznać statystyki dotyczące aktualnie używanych w firmie PIN-ów, w szczególności obliczyć, ile jest par PIN-ów, które różnią się odpowiednio na 1, 2, 3 albo 4 pozycjach. Ma nadzieję, że te statystyki będą na tyle alarmujące, że uda się przekonać szefa do wymiany systemu autoryzacji na lepszy.*

Zadanie

Masz daną listę PIN-ów oraz liczbę całkowitą D. Znajdź liczbę par PIN-ów, które różnią się na dokładnie D pozycjach.

Specyfikacja wejścia

Pierwszy wiersz wejścia zawiera dwie liczby całkowite dodatnie N oraz D oddzielone pojedynczym odstępem, przy czym N jest liczbą PIN-ów, a D jest wybraną liczbą różniących się pozycji w PIN-ach. Każdy z kolejnych N wierszy zawiera jeden PIN.

Ograniczenia

Możesz założyć, że we wszystkich testach zachodzi $2 \leqslant N \leqslant 50\,000$ oraz $1 \leqslant D \leqslant 4$.

Każdy PIN ma długość 4, a każdy jego znak jest albo cyfrą, albo małą literą od '`a`' *do* '`z`' *włącznie. Możesz założyć, że wszystkie PIN-y na wejściu są różne.*

W testach wartych 15 punktów zachodzi $N \leqslant 2000$.

W testach wartych 60 punktów zachodzi $D \leqslant 2$. Wśród nich znajdują się testy warte 30 punktów, w których $D = 1$.

W testach wartych 75 punktów każdy PIN składa się jedynie z cyfr lub małych liter od '`a`' *do* '`f`' *włącznie. Może więc być traktowany jako liczba w zapisie szesnastkowym.*

Specyfikacja wyjścia

Wypisz pojedynczy wiersz zawierający jedną liczbę oznaczającą liczbę par PIN-ów, które różnią się na **dokładnie** *D pozycjach.*

Przykłady

Dla danych wejściowych:	*poprawnym wynikiem jest:*

`4 1`
`0000`
`a010`
`0202`
`a0e2`
	`0`

Każda para spośród tych PIN-ów różni się na więcej niż jednej pozycji.

Dla danych wejściowych:	*poprawnym wynikiem jest:*

`4 2`
`0000`
`a010`
`0202`
`a0e2`
	`3`

Są trzy pary PIN-ów różniące się na dokładnie dwóch pozycjach: (`0000`, `a010`), (`0000`, `0202`) *i* (`a010`, `a0e2`).

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] *XVI Olimpiada Informatyczna 2008/2009*. Warszawa, 2009.

[17] A. V. Aho, J. E. Hopcroft, J. D. Ullman. *Projektowanie i analiza algorytmów komputerowych*. PWN, Warszawa, 1983.

[18] L. Banachowski, A. Kreczmar. *Elementy analizy algorytmów*. WNT, Warszawa, 1982.

[19] L. Banachowski, K. Diks, W. Rytter. *Algorytmy i struktury danych*. WNT, Warszawa, 1996.

[20] J. Bentley. *Perełki oprogramowania*. WNT, Warszawa, 1992.

[21] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Wprowadzenie do algorytmów*. WNT, Warszawa, 2004.

[22] M. de Berg, M. van Kreveld, M. Overmars. *Geometria obliczeniowa. Algorytmy i zastosowania*. WNT, Warszawa, 2007.

[23] R. L. Graham, D. E. Knuth, O. Patashnik. *Matematyka konkretna*. PWN, Warszawa, 1996.

[24] D. Harel. *Algorytmika. Rzecz o istocie informatyki*. WNT, Warszawa, 1992.

[25] D. E. Knuth. *Sztuka programowania*. WNT, Warszawa, 2002.

[26] W. Lipski. *Kombinatoryka dla programistów*. WNT, Warszawa, 2004.

[27] F. P. Preparata, M. I. Shamos. *Geometria obliczeniowa. Wprowadzenie*. Helion, Warszawa, 2003.

[28] E. M. Reingold, J. Nievergelt, N. Deo. *Algorytmy kombinatoryczne*. WNT, Warszawa, 1985.

[29] K. A. Ross, C. R. B. Wright. *Matematyka dyskretna*. PWN, Warszawa, 1996.

[30] R. Sedgewick. *Algorytmy w C++. Grafy*. RM, 2003.

[31] S. S. Skiena, M. A. Revilla. *Wyzwania programistyczne*. WSiP, Warszawa, 2004.

[32] P. Stańczyk. *Algorytmika praktyczna. Nie tylko dla mistrzów*. PWN, Warszawa, 2009.

[33] M. M. Sysło. *Algorytmy*. WSiP, Warszawa, 1998.

[34] M. M. Sysło. *Piramidy, szyszki i inne konstrukcje algorytmiczne*. WSiP, Warszawa, 1998.

[35] R. J. Wilson. *Wprowadzenie do teorii grafów*. PWN, Warszawa, 1998.

[36] N. Wirth. *Algorytmy + struktury danych = programy*. WNT, Warszawa, 1999.

[37] D. Gusfield, G. M. Landau, B. Schieber. An efficient algorithm for the all pairs suffix-prefix problem. *Information Processing Letters*, 41(4):181–185, 1992.

Niniejsza publikacja stanowi wyczerpujące źródło informacji o zawodach XVII Olimpiady Informatycznej, przeprowadzonych w roku szkolnym 2009/2010. 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 XXII Międzynarodowej Olimpiady Informatycznej, XVI Bałtyckiej Olimpiady Informatycznej oraz XVII Olimpiady Informatycznej Krajów Europy Środkowej.

XVII 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 978-83-922946-9-6