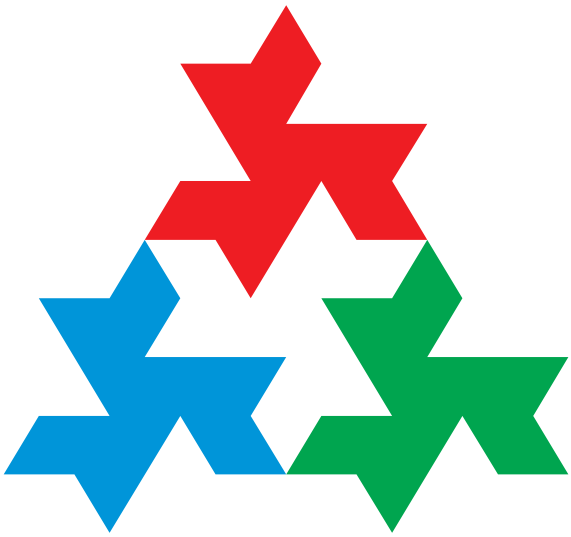
MINISTERSTWO EDUKACJI NARODOWEJ
FUNDACJA ROZWOJU INFORMATYKI
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ



XIX OLIMPIADA INFORMATYCZNA
2011/2012

Olimpiada Informatyczna jest organizowana przy współudziale

asseco
POLAND

WARSZAWA, 2012

Asseco Poland S.A. jest największą spółką informatyczną notowaną na Giełdzie Papierów Wartościowych w Warszawie. Jako międzynarodowy integrator IT, spółka jest ważnym graczem na europejskim rynku producentów oprogramowania. Grupa Asseco znalazła się w pierwszej dziesiątce w rankingu „TOP100 European Software Vendors" (Truffle Capital 2012).

Wychodząc naprzeciw stale rosnącym oczekiwaniom technologicznym i kompetencyjnym stawianym przez klientów, Asseco Poland S.A. skupia się na produkcji i rozwoju innowacyjnego oprogramowania i jako największy software house z polskim kapitałem skutecznie konkuruje z liderami rynku światowego. Jako jedna z nielicznych firm w Polsce, Asseco Poland S.A. buduje i wdraża scentralizowane, kompleksowe systemy informatyczne dla sektora bankowego, z których korzysta ponad połowa banków działających w naszym kraju. Poprzez swoje spółki zależne oferuje również najbardziej zaawansowane rozwiązania dla instytucji ubezpieczeniowych, wdrożone w największych firmach tego sektora na świecie. Stworzyła dedykowane systemy dla administracji publicznej, m.in. dla Zakładu Ubezpieczeń Społecznych, Agencji Restrukturyzacji i Modernizacji Rolnictwa czy Ministerstwa Spraw Wewnętrznych i Administracji. Oferta spółki skierowana jest także do branży energetycznej, telekomunikacji, służby zdrowia, samorządów lokalnych, rolnictwa i służb mundurowych oraz organizacji i instytucji międzynarodowych, takich jak NATO czy UE.

Szerokie i udokumentowane kompetencje w zakresie rozwiązań ERP i Business Intelligence pokrywają wszystkie sektory gospodarki dopełniając ofertę produktową Asseco Poland S.A. i stawiają ją w gronie najbardziej wszechstronnych dostawców IT.

Grupa Asseco na świecie

Asseco Poland S.A. jest liderem międzynarodowej Grupy Asseco, skupiającej rentowne firmy informatyczne z całego świata. Spółka konsekwentnie rozbudowuje holdingi działające w poszczególnych regionach Europy:
Asseco Central Europe (Czechy, Słowacja, Węgry)
Asseco South Eastern Europe (Bałkany, Turcja)
Asseco DACH (Niemcy, Austria i Szwajcaria)
Asseco South Western Europe (Francja, Włochy, Hiszpania, Portugalia)
Asseco Northern Europe (Skandynawia, kraje bałtyckie)

Poprzez swoje spółki z izraelskiej grupy informatycznej Formula Systems, Asseco Poland S.A. wkroczyło także na światowe rynki (m.in. Stany Zjednoczone, Kanada, Japonia, Australia), oferując najbardziej zaawansowane i innowacyjne narzędzia informatyczne. Oferta Matrix IT obejmuje usługi softwarowe, dystrybucję oprogramowania, rozwiązania infrastrukturalne oraz usługi szkoleniowe i wdrożeniowe. Magic Software Enterprises produkuje narzędzia do tworzenia aplikacji oraz rozwiązania do integracji systemów i procesów biznesowych. Sapiens International Corporation jest jednym z wiodących globalnych dostawców autorskich systemów informatycznych dla branży ubezpieczeniowej.

Dzięki temu unikatowemu know-how, spółka skutecznie konkuruje z największymi globalnymi korporacjami na rynku IT, podążając za najnowszymi trendami technologicznymi.

We wrześniu 2012 roku Grupa Asseco zatrudniała ponad 16 000 osób, w tym ok. 4 500 pracowników w Polsce.

MINISTERSTWO EDUKACJI NARODOWEJ
FUNDACJA ROZWOJU INFORMATYKI
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

XIX OLIMPIADA INFORMATYCZNA
2011/2012

WARSZAWA, 2012

Autorzy tekstów:

Igor Adamski
Marek Cygan
Krzysztof Diks
Tomasz Idziaszek
Adam Karczmarz
Marian M. Kędzierski
Alan Kutniewski
Jan Kanty Milczek
Jakub Pachocki
Karol Pokorski
Jakub Radoszewski
Wojciech Rytter
Wojciech Śmietanka
Jacek Tomasiewicz
Michał Włodarczyk
Jakub Wojtaszczyk

Autorzy programów:

Igor Adamski
Mateusz Baranowski
Michał Bejda
Dawid Dąbrowski
Lech Duraj
Tomasz Idziaszek
Alan Kutniewski
Krzysztof Leszczyński
Jan Kanty Milczek
Jakub Pachocki
Zbigniew Wojna
Michał Zgliczyński

Opracowanie i redakcja:

Tomasz Idziaszek
Marcin Kubica
Jakub Radoszewski

Tłumaczenie treści zadań:

Dawid Dąbrowski
Jakub Łącki
Jakub Pachocki
Jakub Pawlewicz
Jakub Radoszewski

Skład:

Jakub Radoszewski

Pozycja dotowana przez Ministerstwo Edukacji Narodowej.

Druk książki został sfinansowany przez

ISBN 978-83-930856-8-2

Spis treści

Sprawozdanie z przebiegu XIX Olimpiady Informatycznej *7*

Regulamin Olimpiady Informatycznej *37*

Zasady organizacji zawodów *45*

Zawody I stopnia — opracowania zadań **53**

Festyn *55*

Litery *63*

Odległość *71*

Randka *77*

Studnia *83*

Zawody II stopnia — opracowania zadań **91**

Tour de Bajtocja *93*

Bony *99*

Szatnia *103*

Okropny wiersz *109*

Rozkład Fibonacciego *115*

Zawody III stopnia — opracowania zadań **123**

Squarki *125*

Licytacja *131*

Pensje *139*

Wyrównywanie terenu *145*

Bezpieczeństwo minimalistyczne *149*

Hurtownia *155*

Prefiksufiks *163*

XXIV Międzynarodowa Olimpiada Informatyczna — treści zadań 173

Skryba dwukierunkowy *175*

Ogniwa *178*

Odometr na kamyki *181*

Turniej rycerski *186*

Idealne miasto *189*

Ostatnia Wieczerza *192*

XVIII Bałtycka Olimpiada Informatyczna — treści zadań 199

Nawiasy *201*

Szczyty *203*

Telefonia komórkowa *206*

Fajerwerki w RightAngleles *207*

Melodia *209*

Tiny *211*

XIX Olimpiada Informatyczna Krajów Europy Środkowej — treści zadań 215

Szeregowanie zleceń *216*

Obwód drukowany *218*

Regaty żeglarskie *220*

Projektowanie autostrad *222*

Sieć *224*

Utylizacja odpadów *226*

Literatura 229

Krzysztof Diks

Wstęp

Drogi Czytelniku!

Po raz dziewiętnasty oddajemy do Twoich rąk „niebieską książeczkę", jak co roku przygotowaną przez wytrawnych znawców tematyki olimpijskiej, algorytmiki i programowania. Książeczka to nie tylko zapis przebiegu XIX Olimpiady Informatycznej, lecz także znakomita pomoc naukowa dla wszystkich, którzy chcą pogłębiać wiedzę i doskonalić umiejętności informatyczne. U podstaw dobrego wykształcenia informatycznego leżą umiejętności algorytmiczne polegające na analizowaniu problemów algorytmicznych, projektowaniu właściwych algorytmów, uzasadnianiu ich poprawności, dobieraniu wydajnych implementacji oraz przeprowadzaniu wszechstronnych testów. Jak widać z powyższego uczestnicy Olimpiady muszą charakteryzować się przymiotami właściwymi zawodowym informatykom. Organizatorzy Olimpiady starają się kształtować te cechy poprzez dobór odpowiednich zadań, których rozwiązania wymagają umiejętności i wiedzy wybiegających daleko poza to, co jest oferowane na regularnych lekcjach informatyki. Dlatego należy docenić każdego, kto rozwiązał choć jedno zadanie olimpijskie.

Najlepsi uczestnicy Olimpiady Informatycznej to ścisła czołówka światowa. Potwierdzeniem tego są wyniki naszych reprezentantów na zawodach międzynarodowych.

Na tegorocznej Międzynarodowej Olimpiadzie Informatycznej Polska zdobyła cztery medale. Złoty medal uzyskał Karol Farbiś. Karol zajął bardzo wysokie, 7. miejsce w klasyfikacji generalnej zawodów. Pozostali polscy reprezentanci, Wojciech Nadara, Wiktor Kuropatwa oraz Bartłomiej Dudek, uzyskali srebrne medale.

Równie dobrze zaprezentowaliśmy się podczas Olimpiady Informatycznej Krajów Europy Środkowej. Mateusz Gołębiewski zajął w łącznej klasyfikacji wysokie drugie miejsce! Wiktor Kuropatwa i Bartłomiej Dudek wywalczyli srebrne medale, a Karol Farbiś medal brązowy. Spektakularnym sukcesem naszych olimpijczyków zakończyła się Bałtycka Olimpiada Informatyczna. Zwycięzcą całych zawodów został Krzysztof Pszeniczny, a pozostałe dwa miejsca na podium zajęli Szymon Stankiewicz oraz Karol Farbiś. Złoty medal uzyskał także Stanisław Dobrowolski. Pozostali nasi zawodnicy — Szymon Łukasz i Przemysław Jakub Kozłowski — uzyskali srebrne medale.

Te wspaniałe sukcesy to zasługa wszystkich uczestników Olimpiady, bo tylko ostra, ale jednocześnie szlachetna rywalizacja pozwala wyłonić tak znakomitą czołówkę. Nie do przecenienia są też zasługi nauczycieli, którzy na co dzień pracują z uczniami. W tym miejscu chciałbym też podziękować wszystkim osobom biorącym udział w organizacji Olimpiady Informatycznej, których talenty, wiedza, umiejętności i zaangażowanie są od lat gwarancją najwyższej jakości tych zawodów.

Krzysztof Diks
Przewodniczący Komitetu Głównego Olimpiady Informatycznej

Sprawozdanie z przebiegu XIX Olimpiady Informatycznej w roku szkolnym 2011/2012

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 XIX 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.

4 października 2011 roku rozesłano do 3344 szkół i zespołów szkół młodzieżowych ponadgimnazjalnych plakaty informujące o rozpoczęciu XIX Olimpiady oraz książki zawierające sprawozdanie i opracowanie rozwiązań zadań z poprzedniej edycji Olimpiady. Zawody I stopnia rozpoczęły się 17 października 2011 roku. Ostatecznym terminem nadsyłania prac konkursowych był 14 listopada 2011 roku.

Zawody II i III stopnia były dwudniowymi sesjami stacjonarnymi, poprzedzonymi jednodniowymi sesjami próbnymi. Zawody II stopnia odbyły się w ośmiu okręgach: Białymstoku, Gliwicach, Krakowie, Poznaniu, Sopocie, Toruniu, Warszawie i Wrocławiu w dniach 7–9 lutego 2012 roku, natomiast zawody III stopnia odbyły się w ośrodku firmy Combidata Poland SA w Sopocie, w dniach 27–30 marca 2012 roku.

Uroczystość zakończenia XIX Olimpiady Informatycznej odbyła się 30 marca 2012 roku w siedzibie firm Combidata Poland SA i Asseco Poland SA w Gdyni przy ul. Podolskiej 21.

SKŁAD OSOBOWY KOMITETÓW OLIMPIADY

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:
dr Jakub Radoszewski (Uniwersytet Warszawski)

kierownik techniczny:
mgr Szymon Acedański (Uniwersytet Warszawski)

kierownik organizacyjny:
Tadeusz Kuran

członkowie:
dr Piotr Chrząstowski-Wachtel (Uniwersytet Warszawski)
prof. dr hab. inż. Zbigniew Czech (Politechnika Śląska)
dr hab. inż. Piotr Formanowicz, prof. PP (Politechnika Poznańska)
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 M. Sysło (Uniwersytet Wrocławski)
dr Maciej Ślusarek (Uniwersytet Jagielloński)
mgr Krzysztof J. Święcicki
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)
dr Jakub Radoszewski (Uniwersytet Warszawski)

Siedzibą Komitetu Okręgowego jest Ośrodek Edukacji Informatycznej i Zastosowań Komputerów w Warszawie, ul. Nowogrodzka 73, Warszawa.

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, Wrocław.

Komitet Okręgowy w Toruniu
przewodniczący:
prof. dr hab. Edward Ochmański (Uniwersytet Mikołaja Kopernika w Toruniu)
zastępca przewodniczącego:
dr Bartosz Ziemkiewicz (Uniwersytet Mikołaja Kopernika w Toruniu)
sekretarz:
mgr Kamila Barylska (Uniwersytet Mikołaja Kopernika w Toruniu)
członkowie:
mgr inż. Rafał Kluszczyński (Uniwersytet Mikołaja Kopernika w Toruniu)
mgr Łukasz Mikulski (Uniwersytet Mikołaja Kopernika w Toruniu)

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

Górnośląski Komitet Okręgowy
przewodniczący:
prof. dr hab. inż. Zbigniew Czech (Politechnika Śląska w Gliwicach)
zastępca przewodniczącego:
dr inż. Krzysztof Simiński (Politechnika Śląska w Gliwicach)
członkowie:
mgr inż. Tomasz Drosik (Politechnika Śląska w Gliwicach)
mgr inż. Dariusz Myszor (Politechnika Śląska w Gliwicach)
dr inż. Jacek Widuch (Politechnika Śląska w Gliwicach)

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

Komitet Okręgowy w Krakowie
przewodniczący:
prof. dr hab. Paweł Idziak (Uniwersytet Jagielloński)
zastępca przewodniczącego:
dr Maciej Ślusarek (Uniwersytet Jagielloński)
sekretarz:
mgr Monika Gillert (Uniwersytet Jagielloński)

członkowie:
- mgr Henryk Białek (emerytowany pracownik Małopolskiego Kuratorium Oświaty)
- dr Iwona Cieślik (Uniwersytet Jagielloński)
- mgr Grzegorz Gutowski (Uniwersytet Jagielloński)
- Marek Wróbel (student Uniwersytetu Jagiellońskiego)

Siedzibą Komitetu Okręgowego jest Katedra Algorytmiki Uniwersytetu Jagiellońskiego, ul. Łojasiewicza 6, Kraków.
(Strona internetowa Komitetu Okręgowego: `www.tcs.uj.edu.pl/OI/`.)

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)
- dr inż. 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 jest Wyższa Szkoła Informatyki i Zarządzania w Rzeszowie, ul. Sucharskiego 2, Rzeszów.

Komitet Okręgowy w Poznaniu
przewodniczący:
- mgr inż. Szymon Wąsik (Politechnika Poznańska)

zastępca przewodniczącego:
- mgr inż. Hanna Ćwiek (Politechnika Poznańska)

sekretarz:
- mgr inż. Bartosz Zgrzeba (Politechnika Poznańska)

członkowie:
- dr inż. Maciej Miłostan (Politechnika Poznańska)
- mgr inż. Andrzej Stroiński (Politechnika Poznańska)

Siedzibą Komitetu Okręgowego jest Instytut Informatyki Politechniki Poznańskiej, ul. Piotrowo 2, Poznań.
(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, prof. UG (Uniwersytet Gdański)

sekretarz:
dr inż. Krzysztof Ocetkiewicz (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, Gdańsk Wrzeszcz.

Jury Olimpiady Informatycznej

W pracach Jury, które nadzorował Krzysztof Diks, a którymi kierowali Szymon Acedański i Jakub Radoszewski, brali udział pracownicy, doktoranci i studenci Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego, Katedry Algorytmiki, Wydziału Matematyki i Informatyki Uniwersytetu Jagiellońskiego, Wydziału Matematyki i Informatyki Uniwersytetu Wrocławskiego, Uniwersytetu w Bergen w Norwegii oraz pracownik firmy Google:

Michał Adamczyk
Igor Adamski
Mateusz Baranowski
Michał Bejda
Arkadiusz Betkier
Maciej Borsz
Dawid Dąbrowski
Maciej Dębski
Lech Duraj
Tomasz Idziaszek
Adam Karczmarz
Tomasz Kulczyński
Alan Kutniewski
Krzysztof Leszczyński
Maciej Matraszek
Mirosław Michalski
Jan Kanty Milczek
Piotr Niedźwiedź
Robert Obryk
Jakub Pachocki
Paweł Parys
Michał Pilipczuk
Karol Pokorski
Adam Polak
Zbigniew Wojna
Jakub Wojtaszczyk
Michał Zgliczyński

ZAWODY I STOPNIA

Zawody I stopnia XIX Olimpiady Informatycznej odbyły się w dniach 17 października – 14 listopada 2011 roku. Wzięło w nich udział 881 zawodników. Decyzją Komitetu Głównego zdyskwalifikowano 22 zawodników. Powodem dyskwalifikacji była niesamodzielność rozwiązań zadań konkursowych. Sklasyfikowano 859 zawodników.

Decyzją Komitetu Głównego do zawodów zostało dopuszczonych 58 uczniów gimnazjów. Pochodzili oni z następujących szkół:

szkoła	miejscowość	liczba uczniów
Gimnazjum nr 24 (Zespół Szkół Ogólnokształcących nr 1)	Gdynia	27
Publiczne Gimnazjum nr 23 (Zespół Szkół Ogólnokształcących nr 6 im. Jana Kochanowskiego)	Radom	7
Gimnazjum nr 50 (Zespół Szkół Ogólnokształcących nr 6)	Bydgoszcz	5
Gimnazjum nr 58 z Oddziałami Dwujęzycznymi im. Króla Władysława IV	Warszawa	4
Gimnazjum nr 16 (Zespół Szkół Ogólnokształcących nr 7)	Szczecin	4
Gimnazjum z Oddziałami Dwujęzycznymi nr 42	Warszawa	3
Społeczne Gimnazjum nr 8 STO	Białystok	1
Gimnazjum nr 1 im. Mikołaja Kopernika	Gdańsk	1
Gimnazjum nr 19 (Zespół Szkół nr 14)	Gdynia	1
Gimnazjum nr 1	Konstantynów Łódzki	1
Gimnazjum nr 2 im. Jana Pawła II	Luboń	1
Gimnazjum nr 1 im. Jana Pawła II	Sejny	1
Zespół Szkół	Ślemień	1
Gimnazjum nr 49 z Oddziałami Dwujęzycznymi (Zespół Szkół nr 14)	Wrocław	1

Kolejność województw pod względem liczby zawodników była następująca:

mazowieckie	152 zawodników	podkarpackie	43
małopolskie	118	lubelskie	28
pomorskie	87	zachodniopomorskie	27
dolnośląskie	74	łódzkie	24
śląskie	74	świętokrzyskie	18
podlaskie	61	warmińsko-mazurskie	16
kujawsko-pomorskie	58	lubuskie	15
wielkopolskie	57	opolskie	7

W zawodach I stopnia najliczniej reprezentowane były szkoły:

szkoła	miejscowość	liczba uczniów
V Liceum Ogólnokształcące im. Augusta Witkowskiego	Kraków	64
XIV Liceum Ogólnokształcące im. Stanisława Staszica	Warszawa	59
I Liceum Ogólnokształcące im. Adama Mickiewicza	Białystok	41
III Liceum Ogólnokształcące im. Marynarki Wojennej RP (Zespół Szkół Ogólnokształcących nr 1)	Gdynia	30

Gimnazjum nr 24 (Zespół Szkół Ogólnokształcących nr 1)	Gdynia	27
VIII Liceum Ogólnokształcące im. Adama Mickiewicza	Poznań	25
XIV Liceum Ogólnokształcące im. Polonii Belgijskiej (Zespół Szkół nr 14)	Wrocław	22
VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich (Zespół Szkół Ogólnokształcących nr 6)	Bydgoszcz	21
I Liceum Ogólnokształcące im. Tadeusza Kościuszki	Legnica	21
Liceum Ogólnokształcące nr III	Wrocław	16
XIII Liceum Ogólnokształcące (Zespół Szkół Ogólnokształcących nr 7)	Szczecin	14
Liceum Akademickie (Zespół Szkół UMK Gimnazjum i Liceum Akademickie)	Toruń	14
VIII Liceum Ogólnokształcące im. Marii Skłodowskiej-Curie	Katowice	10
VIII Liceum Ogólnokształcące im. Króla Władysława IV	Warszawa	10
I Liceum Ogólnokształcące im. Mikołaja Kopernika	Krosno	9
VI Liceum Ogólnokształcące (Zespół Szkół Ogólnokształcących nr 6 im. Jana Kochanowskiego)	Radom	9
Technikum nr 1 im. Stanisława Staszica (Zespół Szkół Technicznych)	Rybnik	9
Zespół Szkół Zawodowych	Brodnica	7
Publiczne Gimnazjum nr 23 (Zespół Szkół Ogólnokształcących nr 6 im. Jana Kochanowskiego)	Radom	7
III Liceum Ogólnokształcące im. Adama Mickiewicza	Tarnów	7
V Liceum Ogólnokształcące	Bielsko-Biała	6
V Liceum Ogólnokształcące im. Stefana Żeromskiego	Gdańsk	6
Technikum nr 7 (Zespół Szkół Informatycznych im. Generała Józefa Hauke Bosaka)	Kielce	5
II Liceum Ogólnokształcące im. Króla Jana III Sobieskiego	Kraków	6
I Liceum Ogólnokształcące im. Bolesława Prusa	Siedlce	6
Gimnazjum nr 50 (Zespół Szkół Ogólnokształcących nr 6)	Bydgoszcz	5
IV Liceum Ogólnokształcące im. Marii Skłodowskiej-Curie (Zespół Szkół Ogólnokształcących nr 2)	Olsztyn	5
Zespół Szkół Elektronicznych	Bydgoszcz	4
I Liceum Ogólnokształcące im. Mikołaja Kopernika	Gdańsk	4
VI Liceum Ogólnokształcące im. Wacława Sierpińskiego	Gdynia	4
Kłodzka Szkoła Przedsiębiorczości	Kłodzko	4
I Liceum Ogólnokształcące im. Stanisława Staszica	Lublin	4

II Liceum Ogólnokształcące im. Hetmana Jana Zamoyskiego	Lublin	4
Zespół Szkół Technicznych	Ostrów Wlkp.	4
Gimnazjum i Liceum im. Jana Pawła II Sióstr Prezentek	Rzeszów	4
I Liceum Ogólnokształcące im. Bolesława Krzywoustego	Słupsk	4
Gimnazjum nr 16 (Zespół Szkół Ogólnokształcących nr 7)	Szczecin	4
Gimnazjum nr 58 z Oddziałami Dwujęzycznymi im. Króla Władysława IV	Warszawa	4
XXVII Liceum Ogólnokształcące im. Tadeusza Czackiego	Warszawa	4
LXXX Liceum Ogólnokształcące (Zespół Szkół Licealnych i Technicznych nr 1)	Warszawa	4
Zespół Szkół Technicznych	Wodzisław Śl.	4
I Liceum Ogólnokształcące im. Edwarda Dembowskiego	Zielona Góra	4
III Liceum Ogólnokształcące im. Krzysztofa Kamila Baczyńskiego	Białystok	3
Zespół Szkół Ogólnokształcących	Dębica	3
I Liceum Ogólnokształcące im. Edwarda Dembowskiego (Zespół Szkół Ogólnokształcących nr 10)	Gliwice	3
Zespół Szkół Techniczno-Informatycznych	Gliwice	3
Zespół Szkół Elektrycznych	Kielce	3
VIII Liceum Ogólnokształcące im. Stanisława Wyspiańskiego	Kraków	3
Liceum Ogólnokształcące im. Władysława Broniewskiego (Zespół Szkół)	Krzepice	3
Zespół Szkół Technicznych	Mielec	3
Publiczne Liceum Ogólnokształcące nr II z Oddziałami Dwujęzycznymi im. Marii Konopnickiej (Zespół Szkół Ogólnokształcących nr II)	Opole	3
Liceum Ogólnokształcące im. Mikołaja Kopernika	Ostrów Maz.	3
Liceum Ogólnokształcące im. Marszałka Stanisława Małachowskiego	Płock	3
Zespół Szkół Komunikacji im. Hipolita Cegielskiego	Poznań	3
III Liceum Ogólnokształcące im. św. Jana Kantego (Zespół Szkół Ogólnokształcących nr 3)	Poznań	3
I Liceum Ogólnokształcące im. Księcia Adama Jerzego Czartoryskiego	Puławy	3
IV Liceum Ogólnokształcące im. Mikołaja Kopernika	Rzeszów	3
Zespół Szkół Elektronicznych	Rzeszów	3

I Liceum Ogólnokształcące im. Kazimierza Jagiellończyka	Sieradz	3
II Liceum Ogólnokształcące (Zespół Szkół Ogólnokształcących nr 2)	Tarnów	3
Gimnazjum z Oddziałami Dwujęzycznymi nr 42	Warszawa	3
I Liceum Ogólnokształcące im. Ziemi Kujawskiej	Włocławek	3
Zespół Szkół Mechaniczno-Elektrycznych	Żywiec	3

Najliczniej reprezentowane były następujące miejscowości:

Warszawa	105 zawodników	Słupsk	6
Kraków	83	Mielec	5
Gdynia	64	Płock	5
Białystok	49	Zielona Góra	5
Wrocław	43	Chełm	4
Poznań	40	Dębica	4
Bydgoszcz	32	Elbląg	4
Legnica	22	Gorzów Wlkp.	4
Szczecin	21	Kłodzko	4
Radom	19	Łomża	4
Toruń	16	Ostrów Wlkp.	4
Lublin	14	Stalowa Wola	4
Rzeszów	14	Wodzisław Śl.	4
Gdańsk	12	Biała Podlaska	3
Kielce	12	Częstochowa	3
Tarnów	12	Gniezno	3
Katowice	11	Końskie	3
Łódź	11	Krzepice	3
Rybnik	10	Opole	3
Bielsko-Biała	9	Ostrów Maz.	3
Gliwice	9	Puławy	3
Krosno	9	Sieradz	3
Siedlce	8	Suwałki	3
Brodnica	7	Włocławek	3
Nowy Sącz	6	Żywiec	3
Olsztyn	6		

Zawodnicy uczęszczali do następujących klas:

do klasy I gimnazjum	2 uczniów
do klasy II gimnazjum	19
do klasy III gimnazjum	37
do klasy I szkoły ponadgimnazjalnej	156
do klasy II szkoły ponadgimnazjalnej	277
do klasy III szkoły ponadgimnazjalnej	320
do klasy IV szkoły ponadgimnazjalnej	48

W zawodach I stopnia zawodnicy mieli do rozwiązania pięć zadań:

- „Festyn” autorstwa Mariana M. Kędzierskiego
- „Litery” autorstwa Mariana M. Kędzierskiego
- „Odległość” autorstwa Wojciecha Śmietanki
- „Randka” autorstwa Alana Kutniewskiego
- „Studnia” autorstwa Michała Włodarczyka

każde oceniane maksymalnie po 100 punktów.

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

- **FES** — Festyn

	FES	
	liczba zawodników	czyli
100 pkt.	19	2,21%
75–99 pkt.	0	0,00%
50–74 pkt.	3	0,35%
1–49 pkt.	22	2,56%
0 pkt.	81	9,43%
brak rozwiązania	734	85,45%

- **LIT** — Litery

	LIT	
	liczba zawodników	czyli
100 pkt.	341	39,70%
75–99 pkt.	9	1,05%
50–74 pkt.	57	6,63%
1–49 pkt.	271	31,55%
0 pkt.	129	15,02%
brak rozwiązania	52	6,05%

- **ODL** — Odległość

	ODL	
	liczba zawodników	czyli
100 pkt.	74	8,62%
75–99 pkt.	41	4,77%
50–74 pkt.	10	1,16%
1–49 pkt.	282	32,83%
0 pkt.	127	14,78%
brak rozwiązania	325	37,84%

- **RAN** — Randka

	RAN	
	liczba zawodników	czyli
100 pkt.	175	20,37%
75–99 pkt.	4	0,47%
50–74 pkt.	36	4,19%
1–49 pkt.	274	31,90%
0 pkt.	33	3,84%
brak rozwiązania	337	39,23%

- **STU** — Studnia

	STU	
	liczba zawodników	czyli
100 pkt.	85	9,90%
75–99 pkt.	22	2,56%
50–74 pkt.	14	1,63%
1–49 pkt.	122	14,20%
0 pkt.	90	10,48%
brak rozwiązania	526	61,23%

W sumie za wszystkie 5 zadań konkursowych:

SUMA	liczba zawodników	czyli
500 pkt.	10	1,16%
375–499 pkt.	46	5,35%
250–374 pkt.	99	11,53%
125–249 pkt.	179	20,84%
1–124 pkt.	393	45,75%
0 pkt.	132	15,37%

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

ZAWODY II STOPNIA

Do zawodów II stopnia, które odbyły się w dniach 7–9 lutego 2012 roku w siedmiu stałych okręgach i Białymstoku, zakwalifikowano 408 zawodników, którzy osiągnęli w zawodach I stopnia wynik nie mniejszy niż 96 pkt.

Zawodnicy zostali przydzieleni do okręgów w następującej liczbie:

Białystok	45 zawodników	Sopot	53
Gliwice	33	Toruń	29
Kraków	98	Warszawa	86
Poznań	20	Wrocław	44

15 zawodników nie stawiło się na zawody, w zawodach wzięło więc udział 393 zawodników.

Zawodnicy uczęszczali do szkół w następujących województwach:

mazowieckie	83 zawodników	podkarpackie	13
małopolskie	74	wielkopolskie	11
podlaskie	40	łódzkie	10
dolnośląskie	36	lubelskie	7
pomorskie	31	warmińsko-mazurskie	6
śląskie	30	lubuskie	4
kujawsko-pomorskie	28	opolskie	3
zachodniopomorskie	14	świętokrzyskie	3

W zawodach II stopnia najliczniej reprezentowane były szkoły:

szkoła	miejscowość	liczba uczniów
V Liceum Ogólnokształcące im. Augusta Witkowskiego	Kraków	53
XIV Liceum Ogólnokształcące im. Stanisława Staszica	Warszawa	45
I Liceum Ogólnokształcące im. Adama Mickiewicza	Białystok	32
III Liceum Ogólnokształcące im. Marynarki Wojennej RP (Zespół Szkół Ogólnokształcących nr 1)	Gdynia	23
XIV Liceum Ogólnokształcące im. Polonii Belgijskiej (Zespół Szkół nr 14)	Wrocław	19
VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich (Zespół Szkół Ogólnokształcących nr 6)	Bydgoszcz	15
XIII Liceum Ogólnokształcące (Zespół Szkół Ogólnokształcących nr 7)	Szczecin	10
Liceum Akademickie (Zespół Szkół UMK Gimnazjum i Liceum Akademickie)	Toruń	10
VIII Liceum Ogólnokształcące im. Króla Władysława IV	Warszawa	9
I Liceum Ogólnokształcące im. Tadeusza Kościuszki	Legnica	8
VI Liceum Ogólnokształcące (Zespół Szkół Ogólnokształcących nr 6 im. Jana Kochanowskiego)	Radom	8
VIII Liceum Ogólnokształcące im. Marii Skłodowskiej-Curie	Katowice	7
Liceum Ogólnokształcące nr III	Wrocław	7
II Liceum Ogólnokształcące im. Króla Jana III Sobieskiego	Kraków	5
V Liceum Ogólnokształcące im. Stefana Żeromskiego	Gdańsk	4
IV Liceum Ogólnokształcące im. Marii Skłodowskiej-Curie (Zespół Szkół Ogólnokształcących nr 2)	Olsztyn	4
VIII Liceum Ogólnokształcące im. Adama Mickiewicza	Poznań	4
Publiczne Gimnazjum nr 23 (Zespół Szkół Ogólnokształcących nr 6 im. Jana Kochanowskiego)	Radom	4
III Liceum Ogólnokształcące im. Adama Mickiewicza	Tarnów	4
V Liceum Ogólnokształcące	Bielsko-Biała	3
Gimnazjum nr 16 (Zespół Szkół Ogólnokształcących nr 7)	Szczecin	3

Najliczniej reprezentowane były miejscowości:

Warszawa	64 zawodników	Wrocław	28
Kraków	60	Gdynia	26
Białystok	35	Bydgoszcz	17

Szczecin	14	Olsztyn	5
Radom	13	Rzeszów	5
Toruń	10	Bielsko-Biała	4
Poznań	9	Gdańsk	4
Katowice	8	Nowy Sącz	4
Legnica	8	Kielce	3
Tarnów	6	Lublin	3
Łódź	5	Zielona Góra	3

7 lutego odbyła się sesja próbna, podczas której zawodnicy rozwiązywali nieliczące się do ogólnej klasyfikacji zadanie „Tour de Bajtocja" autorstwa Marka Cygana. W dniach konkursowych zawodnicy rozwiązywali następujące zadania:

- w pierwszym dniu zawodów (8 lutego):
  - „Bony" autorstwa Jakuba Pachockiego
  - „Szatnia" autorstwa Jana Kantego Milczka
- w drugim dniu zawodów (9 lutego):
  - „Okropny wiersz" autorstwa Igora Adamskiego
  - „Rozkład Fibonacciego" autorstwa Karola Pokorskiego

każde oceniane maksymalnie po 100 punktów.

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

- **TOU — próbne** — Tour de Bajtocja

	TOU — próbne	
	liczba zawodników	czyli
100 pkt.	21	5,34%
75–99 pkt.	9	2,29%
50–74 pkt.	10	2,55%
1–49 pkt.	45	11,45%
0 pkt.	188	47,84%
brak rozwiązania	120	30,53%

- **BON** — Bony

	BON	
	liczba zawodników	czyli
100 pkt.	79	20,10%
75–99 pkt.	9	2,29%
50–74 pkt.	173	44,02%
1–49 pkt.	107	27,23%
0 pkt.	8	2,03%
brak rozwiązania	17	4,33%

- **SZA** — Szatnia

	SZA	
	liczba zawodników	czyli
100 pkt.	30	7,63%
75–99 pkt.	0	0,00%
50–74 pkt.	2	0,51%
1–49 pkt.	222	56,49%
0 pkt.	62	15,78%
brak rozwiązania	77	19,59%

- **OKR** — Okropny wiersz

	OKR	
	liczba zawodników	czyli
100 pkt.	10	2,54%
75–99 pkt.	17	4,33%
50–74 pkt.	12	3,05%
1–49 pkt.	273	69,47%
0 pkt.	35	8,90%
brak rozwiązania	46	11,71%

- **ROZ** — Rozkład Fibonacciego

	ROZ	
	liczba zawodników	czyli
100 pkt.	202	51,40%
75–99 pkt.	52	13,23%
50–74 pkt.	23	5,85%
1–49 pkt.	47	11,96%
0 pkt.	23	5,85%
brak rozwiązania	46	11,71%

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

SUMA	liczba zawodników	czyli
400 pkt.	4	1,02%
300–399 pkt.	17	4,33%
200–299 pkt.	86	21,88%
100–199 pkt.	209	53,18%
1–99 pkt.	70	17,81%
0 pkt.	7	1,78%

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

ZAWODY III STOPNIA

Zawody III stopnia odbyły się w ośrodku firmy Combidata Poland SA w Sopocie, w dniach 27–30 marca 2012 roku. Do zawodów III stopnia zakwalifikowano 100 najlepszych uczestników zawodów II stopnia, którzy uzyskali wynik nie mniejszy niż 206 pkt.

Zawodnicy uczęszczali do szkół w następujących województwach:

mazowieckie	23 zawodników	łódzkie	4
małopolskie	17	lubelskie	3
podlaskie	12	podkarpackie	3
dolnośląskie	11	lubuskie	2
pomorskie	9	zachodniopomorskie	2
śląskie	7	warmińsko-mazurskie	1 zawodnik
kujawsko-pomorskie	5	wielkopolskie	1

Niżej wymienione szkoły miały w finale więcej niż jednego zawodnika:

szkoła	miejscowość	liczba uczniów
V Liceum Ogólnokształcące im. Augusta Witkowskiego	Kraków	17
XIV Liceum Ogólnokształcące im. Stanisława Staszica	Warszawa	10
III Liceum Ogólnokształcące im. Marynarki Wojennej RP (Zespół Szkół Ogólnokształcących nr 1)	Gdynia	9
I Liceum Ogólnokształcące im. Adama Mickiewicza	Białystok	8
XIV Liceum Ogólnokształcące im. Polonii Belgijskiej (Zespół Szkół nr 14)	Wrocław	6
VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich (Zespół Szkół Ogólnokształcących nr 6)	Bydgoszcz	4
VIII Liceum Ogólnokształcące im. Króla Władysława IV	Warszawa	4
Liceum Ogólnokształcące nr III	Wrocław	4
VI Liceum Ogólnokształcące (Zespół Szkół Ogólnokształcących nr 6 im. Jana Kochanowskiego)	Radom	3
I Liceum Ogólnokształcące im. Zygmunta Krasińskiego	Ciechanów	2
I Liceum Ogólnokształcące im. Stanisława Staszica	Lublin	2
XIII Liceum Ogólnokształcące (Zespół Szkół Ogólnokształcących nr 7)	Szczecin	2
I Liceum Ogólnokształcące im. Edwarda Dembowskiego	Zielona Góra	2

27 marca odbyła się sesja próbna, podczas której zawodnicy rozwiązywali nieliczące się do ogólnej klasyfikacji zadanie „Squarki" autorstwa Jakuba Wojtaszczyka. W dniach konkursowych zawodnicy rozwiązywali następujące zadania:

- w pierwszym dniu zawodów (28 marca):
  - „Licytacja" autorstwa Karola Pokorskiego
  - „Pensje" autorstwa Adama Karczmarza
  - „Wyrównywanie terenu" autorstwa Jakuba Pachockiego
- w drugim dniu zawodów (29 marca):
  - „Bezpieczeństwo minimalistyczne" autorstwa Marka Cygana i Wojciecha Ryttera
  - „Hurtownia" autorstwa Tomasza Idziaszka
  - „Prefiksufiks" autorstwa Jacka Tomasiewicza

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:

- **SQU — próbne** — Squarki

	SQU — próbne	
	liczba zawodników	czyli
100 pkt.	19	19,00%
75–99 pkt.	6	6,00%
50–74 pkt.	6	6,00%
1–49 pkt.	23	23,00%
0 pkt.	12	12,00%
brak rozwiązania	34	34,00%

- **LIC** — Licytacja

	LIC	
	liczba zawodników	czyli
100 pkt.	32	32,00%
75–99 pkt.	16	16,00%
50–74 pkt.	5	5,00%
1–49 pkt.	19	19,00%
0 pkt.	11	11,00%
brak rozwiązania	17	17,00%

- **PEN** — Pensje

	PEN	
	liczba zawodników	czyli
100 pkt.	10	10,00%
75–99 pkt.	6	6,00%
50–74 pkt.	2	2,00%
1–49 pkt.	10	10,00%
0 pkt.	51	51,00%
brak rozwiązania	21	21,00%

- **WYR** — Wyrównywanie terenu

	WYR	
	liczba zawodników	czyli
100 pkt.	0	0,00%
75–99 pkt.	1	1,00%
50–74 pkt.	2	2,00%
1–49 pkt.	0	0,00%
0 pkt.	31	31,00%
brak rozwiązania	66	66,00%

- **BEZ** — Bezpieczeństwo minimalistyczne

	BEZ	
	liczba zawodników	czyli
100 pkt.	11	11,00%
75–99 pkt.	0	0,00%
50–74 pkt.	3	3,00%
1–49 pkt.	6	6,00%
0 pkt.	31	31,00%
brak rozwiązania	49	49,00%

- **HUR** — Hurtownia

	HUR	
	liczba zawodników	czyli
100 pkt.	28	28,00%
75–99 pkt.	2	2,00%
50–74 pkt.	7	7,00%
1–49 pkt.	26	26,00%
0 pkt.	31	31,00%
brak rozwiązania	6	6,00%

- **PRE** — Prefiksufiks

	PRE	
	liczba zawodników	czyli
100 pkt.	2	2,00%
75–99 pkt.	0	0,00%
50–74 pkt.	39	39,00%
1–49 pkt.	33	33,00%
0 pkt.	23	23,00%
brak rozwiązania	3	3,00%

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

SUMA	liczba zawodników	czyli
450–600 pkt.	4	4,00%
300–449 pkt.	11	11,00%
150–299 pkt.	31	31,00%
1–149 pkt.	46	46,00%
0 pkt.	8	8,00%

30 marca 2012 roku, w siedzibie firm Asseco Poland SA i Combidata Poland SA w Gdyni, ogłoszono wyniki finału XIX Olimpiady Informatycznej 2011/2012 i rozdano nagrody ufundowane przez: Asseco Poland SA, Ogólnopolską Fundację Edukacji Komputerowej, Olimpiadę Informatyczną, Wydawnictwa Naukowe PWN i Wydawnictwo „Delta".

Poniżej zestawiono listę wszystkich laureatów i wyróżnionych finalistów:

(1) **Karol Farbiś**, 2 klasa, VI Liceum Ogólnokształcące (Zespół Szkół Ogólnokształcących nr 6 im. J. Kochanowskiego), Radom, 464 pkt., laureat I miejsca

(2) **Wojciech Nadara**, 3 klasa, XIV Liceum Ogólnokształcące im. S. Staszica, Warszawa, 461 pkt., laureat I miejsca

(3) **Bartłomiej Dudek**, 3 klasa, XIV Liceum Ogólnokształcące im. Polonii Belgijskiej (Zespół Szkół nr 14), Wrocław, 450 pkt., laureat I miejsca

(3) **Wiktor Kuropatwa**, 3 klasa, V Liceum Ogólnokształcące im. A. Witkowskiego, Kraków, 450 pkt., laureat I miejsca

(5) **Mateusz Gołębiewski**, 3 klasa, XIV Liceum Ogólnokształcące im. Polonii Belgijskiej (Zespół Szkół nr 14), Wrocław, 442 pkt., laureat I miejsca

(6) **Krzysztof Pszeniczny**, 2 klasa, Gimnazjum i Liceum im. Jana Pawła II Sióstr Prezentek, Rzeszów, 430 pkt., laureat I miejsca

(6) **Bartosz Tarnawski**, 3 klasa, Katolickie Liceum Ogólnokształcące (Zespół Katolickich Szkół Ogólnokształcących nr 1 im. bł. ks. E. Szramka), Katowice, 430 pkt., laureat I miejsca

(8) **Marcin Smulewicz**, 3 klasa, Liceum Ogólnokształcące im. B. Prusa, Skierniewice, 426 pkt., laureat I miejsca

(9) **Michał Zając**, 3 klasa, V Liceum Ogólnokształcące im. A. Witkowskiego, Kraków, 367 pkt., laureat II miejsca

(10) **Mateusz Kopeć**, 3 klasa, I Liceum Ogólnokształcące im. A. Mickiewicza, Białystok, 350 pkt., laureat II miejsca

(11) **Michał Łowicki**, 3 klasa, Liceum Ogólnokształcące nr III, Wrocław, 334 pkt., laureat II miejsca

(11) **Szymon Stankiewicz**, 2 klasa, VI Liceum Ogólnokształcące (Zespół Szkół Ogólnokształcących nr 6 im. J. Kochanowskiego), Radom, 334 pkt., laureat II miejsca

(13) **Szymon Łukasz**, 1 klasa, V Liceum Ogólnokształcące im. A. Witkowskiego, Kraków, 310 pkt., laureat II miejsca

(14) **Przemysław Jakub Kozłowski**, 3 klasa gim., Społeczne Gimnazjum nr 8 STO, Białystok, 307 pkt., laureat II miejsca

(15) **Sebastian Daniel Nowak**, 3 klasa, I Liceum Ogólnokształcące im. A. Mickiewicza, Białystok, 301 pkt., laureat II miejsca

(16) **Stanisław Dobrowolski**, 2 klasa, XIV Liceum Ogólnokształcące im. S. Staszica, Warszawa, 286 pkt., laureat II miejsca

(17) **Konrad Paluszek**, 3 klasa gim., Gimnazjum z Oddziałami Dwujęzycznymi nr 42, Warszawa, 282 pkt., laureat II miejsca

(18) **Kamil Żyła**, 2 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP (Zespół Szkół Ogólnokształcących nr 1), Gdynia, 280 pkt., laureat II miejsca

(19) **Jakub Kołodziej**, 3 klasa, I Liceum Ogólnokształcące im. W. Kętrzyńskiego, Giżycko, 278 pkt., laureat II miejsca

(20) **Wojciech Janczewski**, 2 klasa, I Liceum Ogólnokształcące im. T. Kościuszki, Legnica, 250 pkt., laureat III miejsca

(20) **Konrad Kijewski**, 3 klasa, I Liceum Ogólnokształcące im. M. Konopnickiej, Suwałki, 250 pkt., laureat III miejsca

(20) **Błażej Magnowski**, 2 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP (Zespół Szkół Ogólnokształcących nr 1), Gdynia, 250 pkt., laureat III miejsca

(20) **Rafał Stefański**, 2 klasa, XIV Liceum Ogólnokształcące im. S. Staszica, Warszawa, 250 pkt., laureat III miejsca

(24) **Krzysztof Kiewicz**, 3 klasa, VIII Liceum Ogólnokształcące im. Króla Władysława IV, Warszawa, 236 pkt., laureat III miejsca

(25) **Wojciech Szałapski**, 3 klasa, I Liceum Ogólnokształcące im. S. Żeromskiego, Ozorków, 233 pkt., laureat III miejsca

(26) **Aleksander Kramarz**, 3 klasa, VI Liceum Ogólnokształcące im. J. i J. Śniadeckich (Zespół Szkół Ogólnokształcących nr 6), Bydgoszcz, 231 pkt., laureat III miejsca

(27) **Piotr Bejda**, 3 klasa, V Liceum Ogólnokształcące im. A. Witkowskiego, Kraków, 230 pkt., laureat III miejsca

(27) **Paweł Nowak**, 2 klasa, XIII Liceum Ogólnokształcące (Zespół Szkół Ogólnokształcących nr 7), Szczecin, 230 pkt., laureat III miejsca

(29) **Łukasz Majcher**, 3 klasa, IV Liceum Ogólnokształcące im. M. Kopernika, Rzeszów, 225 pkt., laureat III miejsca

(30) **Michał Kowalczyk**, 2 klasa, VI Liceum Ogólnokształcące im. J. i J. Śniadeckich (Zespół Szkół Ogólnokształcących nr 6), Bydgoszcz, 217 pkt., laureat III miejsca

(31) **Krzysztof Kleiner**, 3 klasa, V Liceum Ogólnokształcące im. A. Witkowskiego, Kraków, 208 pkt., finalista z wyróżnieniem

(32) **Piotr Jarosz**, 2 klasa, I Liceum Ogólnokształcące im. Z. Krasińskiego, Ciechanów, 200 pkt., finalista z wyróżnieniem

(32) **Marek Sokołowski**, 1 klasa, I Liceum Ogólnokształcące im. T. Kościuszki, Łomża, 200 pkt., finalista z wyróżnieniem

(34) **Grzegorz Białek**, 3 klasa, VI Liceum Ogólnokształcące im. J. i J. Śniadeckich (Zespół Szkół Ogólnokształcących nr 6), Bydgoszcz, 198 pkt., finalista z wyróżnieniem

(34) **Krzysztof Kulig**, 3 klasa, V Liceum Ogólnokształcące im. A. Witkowskiego, Kraków, 198 pkt., finalista z wyróżnieniem

(36) **Kamil Rychlewicz**, 1 klasa, I Liceum Ogólnokształcące im. M. Kopernika, Łódź, 194 pkt., finalista z wyróżnieniem

(37) **Michał Piekarz**, 3 klasa, V Liceum Ogólnokształcące im. A. Witkowskiego, Kraków, 189 pkt., finalista z wyróżnieniem

(38) **Stanisław Barzowski**, 1 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP (Zespół Szkół Ogólnokształcących nr 1), Gdynia, 183 pkt., finalista z wyróżnieniem

(39) **Grzegorz Świrski**, 2 klasa, V Liceum Ogólnokształcące im. A. Witkowskiego, Kraków, 175 pkt., finalista z wyróżnieniem

(40) **Paweł Tabaszewski**, 3 klasa, XIV Liceum Ogólnokształcące im. S. Staszica, Warszawa, 174 pkt., finalista z wyróżnieniem

(41) **Andrzej Białokozowicz**, 3 klasa, I Liceum Ogólnokształcące im. A. Mickiewicza, Białystok, 168 pkt., finalista z wyróżnieniem

(41) **Leszek Kania**, 2 klasa, V Liceum Ogólnokształcące im. A. Witkowskiego, Kraków, 168 pkt., finalista z wyróżnieniem

(43) **Konrad Cichy**, 3 klasa, Liceum Ogólnokształcące nr III, Wrocław, 166 pkt., finalista z wyróżnieniem

(44) **Michał Kownacki**, 3 klasa, XIV Liceum Ogólnokształcące im. Polonii Belgijskiej (Zespół Szkół nr 14), Wrocław, 161 pkt., finalista z wyróżnieniem

(45) **Maciej Kacprzak**, 2 klasa, VIII Liceum Ogólnokształcące im. Króla Władysława IV, Warszawa, 154 pkt., finalista z wyróżnieniem

(46) **Antoni Zawodny**, 3 klasa, XIV Liceum Ogólnokształcące im. S. Staszica, Warszawa, 150 pkt., finalista z wyróżnieniem

Lista pozostałych finalistów w kolejności alfabetycznej:

- **Mateusz Bałasz**, 3 klasa, I Liceum Ogólnokształcące im. A. Mickiewicza, Białystok
- **Marek Bardoński**, 3 klasa, I Liceum Ogólnokształcące im. Z. Krasińskiego, Ciechanów
- **Robert Błaszkiewicz**, 3 klasa, XIII Liceum Ogólnokształcące (Zespół Szkół Ogólnokształcących nr 7), Szczecin
- **Michał Błaziak**, 2 klasa, I Liceum Ogólnokształcące im. S. Staszica, Lublin

- **Piotr Chabierski**, 2 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP (Zespół Szkół Ogólnokształcących nr 1), Gdynia
- **Mateusz Chołołowicz**, 1 klasa, I Liceum Ogólnokształcące im. A. Mickiewicza, Białystok
- **Patryk Czajka**, 2 klasa, XIV Liceum Ogólnokształcące im. S. Staszica, Warszawa
- **Damian Czapnik**, 2 klasa, VIII Liceum Ogólnokształcące im. Króla Władysława IV, Warszawa
- **Daniel Danielski**, 2 klasa, XIV Liceum Ogólnokształcące im. Polonii Belgijskiej (Zespół Szkół nr 14), Wrocław
- **Anna Dymek**, 3 klasa, V Liceum Ogólnokształcące im. A. Witkowskiego, Kraków
- **Michał Dyrek**, 3 klasa, V Liceum Ogólnokształcące im. A. Witkowskiego, Kraków
- **Piotr Gawryluk**, 2 klasa, I Liceum Ogólnokształcące im. A. Mickiewicza, Białystok
- **Grzegorz Głuch**, 3 klasa, Liceum Ogólnokształcące nr III, Wrocław
- **Norbert Gregorek**, 4 klasa, Zespół Szkół Elektronicznych im. Bohaterów Westerplatte, Radom
- **Albert Gutowski**, 1 klasa, I Liceum Ogólnokształcące im. S. Staszica, Lublin
- **Wojciech Jabłoński**, 1 klasa, XIV Liceum Ogólnokształcące im. S. Staszica, Warszawa
- **Marcin Karpiński**, 1 klasa, I Liceum Ogólnokształcące im. Króla Stanisława Leszczyńskiego, Jasło
- **Michał Kiełbowicz**, 3 klasa, XIV Liceum Ogólnokształcące im. Polonii Belgijskiej (Zespół Szkół nr 14), Wrocław
- **Eryk Kijewski**, 2 klasa gim., Gimnazjum nr 1 im. Jana Pawła II, Sejny
- **Wiktor Klonowski**, 3 klasa, Publiczne Liceum Ogólnokształcące Politechniki Łódzkiej, Łódź
- **Karol Kosik**, 3 klasa, I Liceum Ogólnokształcące im. E. Dembowskiego, Zielona Góra
- **Piotr Kozakowski**, 2 klasa, I Liceum Ogólnokształcące im. E. Dembowskiego, Zielona Góra
- **Ewelina Krakowiak**, 2 klasa, VI Liceum Ogólnokształcące (Zespół Szkół Ogólnokształcących nr 6 im. J. Kochanowskiego), Radom
- **Wojciech Kruk**, 3 klasa, V Liceum Ogólnokształcące im. A. Witkowskiego, Kraków
- **Marcin Kudla**, 2 klasa, Liceum Ogólnokształcące nr 1 im. Noblistów Polskich (Zespół Szkół Ponadgimnazjalnych nr 1), Rydułtowy
- **Paweł Kura**, 3 klasa, VIII Liceum Ogólnokształcące im. M. Skłodowskiej-Curie, Katowice
- **Krzysztof Lewko**, 2 klasa, I Liceum Ogólnokształcące im. A. Mickiewicza, Białystok

- **Adam Lipiński**, 3 klasa, I Liceum Ogólnokształcące im. L. Kruczkowskiego, Tychy
- **Mikołaj Lisik**, 2 klasa, VIII Liceum Ogólnokształcące im. Króla Władysława IV, Warszawa
- **Łukasz Łabęcki**, 2 klasa, VI Liceum Ogólnokształcące im. J. i J. Śniadeckich (Zespół Szkół Ogólnokształcących nr 6), Bydgoszcz
- **Michał Łazowik**, 1 klasa, Liceum Akademickie (Zespół Szkół UMK Gimnazjum i Liceum Akademickie), Toruń
- **Bartosz Łukasiewicz**, 1 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP (Zespół Szkół Ogólnokształcących nr 1), Gdynia
- **Kamil Krzysztof Magryta**, 1 klasa, I Liceum Ogólnokształcące im. Księcia A. J. Czartoryskiego, Puławy
- **Michał Majewski**, 2 klasa, I Liceum Ogólnokształcące im. A. Mickiewicza, Białystok
- **Aleksander Matusiak**, 2 klasa, XIV Liceum Ogólnokształcące im. S. Staszica, Warszawa
- **Krzysztof Mędrela**, 3 klasa, V Liceum Ogólnokształcące im. A. Witkowskiego, Kraków
- **Artur Minorczyk**, 2 klasa, I Liceum Ogólnokształcące im. J. Słowackiego (Akademicki Zespół Szkół Ogólnokształcących), Chorzów
- **Maciej Obuchowski**, 2 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP (Zespół Szkół Ogólnokształcących nr 1), Gdynia
- **Krzysztof Piesiewicz**, 2 klasa gim., Gimnazjum nr 58 z Oddziałami Dwujęzycznymi im. Króla Władysława IV, Warszawa
- **Szymon Policht**, 3 klasa, V Liceum Ogólnokształcące im. A. Witkowskiego, Kraków
- **Krzysztof Potempa**, 3 klasa, V Liceum Ogólnokształcące im. A. Witkowskiego, Kraków
- **Bartosz Prusak**, 2 klasa, VIII Liceum Ogólnokształcące im. A. Mickiewicza, Poznań
- **Radosław Serafin**, 2 klasa, Liceum Ogólnokształcące nr III, Wrocław
- **Michał Seweryn**, 3 klasa, V Liceum Ogólnokształcące im. A. Witkowskiego, Kraków
- **Marek Sommer**, 2 klasa, XIV Liceum Ogólnokształcące im. S. Staszica, Warszawa
- **Jakub Staroń**, 1 klasa, V Liceum Ogólnokształcące, Bielsko-Biała
- **Krzysztof Story**, 3 klasa, XIV Liceum Ogólnokształcące im. Polonii Belgijskiej (Zespół Szkół nr 14), Wrocław
- **Aleksander Szulc**, 2 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP (Zespół Szkół Ogólnokształcących nr 1), Gdynia
- **Jakub Śliwiński**, 3 klasa, I Liceum Ogólnokształcące im. B. Prusa, Siedlce
- **Wiktor Teleżyński**, 3 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP (Zespół Szkół Ogólnokształcących nr 1), Gdynia

- **Adam Trzaskowski**, 1 klasa, XIV Liceum Ogólnokształcące im. S. Staszica, Warszawa
- **Patryk Urbański**, 3 klasa, V Liceum Ogólnokształcące im. A. Witkowskiego, Kraków
- **Paweł Wegner**, 1 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP (Zespół Szkół Ogólnokształcących nr 1), Gdynia
- **Marcel Zięba**, 3 klasa, II Liceum Ogólnokształcące im. Cz. Miłosza, Jaworzno

Komitet Główny Olimpiady Informatycznej przyznał następujące nagrody rzeczowe:

(1) puchar przechodni ufundowany przez Olimpiadę Informatyczną wręczono zwycięzcy XIX Olimpiady Karolowi Farbisiowi,
(2) puchar ufundowany przez Olimpiadę Informatyczną wręczono zwycięzcy XIX Olimpiady Karolowi Farbisiowi,
(3) złote, srebrne i brązowe medale ufundowane przez MEN przyznano odpowiednio laureatom I, II i III miejsca,
(4) laptopy (4 szt.) ufundowane przez Asseco Poland SA przyznano laureatom I miejsca,
(5) tablety (6 szt.) ufundowane przez Asseco Poland SA przyznano kolejnym laureatom I miejsca i laureatom II miejsca,
(6) aparaty fotograficzne (13 szt.) ufundowane przez OFEK przyznano kolejnym laureatom II miejsca i laureatom III miejsca,
(7) zestawy głośników (5 szt.), pamięci USB (2 szt.) oraz odtwarzacze MP3 (7 szt.) ufundowane przez OFEK przyznano kolejnym laureatom III miejsca,
(8) odtwarzacze MP3 (16 szt.) ufundowane przez OFEK przyznano wyróżnionym finalistom,
(9) książki ufundowane przez PWN przyznano wszystkim finalistom,
(10) roczną prenumeratę miesięcznika „Delta” przyznano wszystkim laureatom i wyróżnionym finalistom.

Komitet Główny powołał następujące reprezentacje.

- Na **Międzynarodową Olimpiadę Informatyczną IOI'2012**, która odbędzie się we Włoszech w terminie 23–30 września 2012 roku, oraz **Olimpiadę Informatyczną Krajów Europy Środkowej CEOI'2012**, która odbędzie się na Węgrzech w terminie 7–13 lipca 2012 roku:

  (1) Karol Farbiś
  (2) Wojciech Nadara
  (3) Bartłomiej Dudek
  (4) Wiktor Kuropatwa

  rezerwowi:

  (5) Mateusz Gołębiewski
  (6) Krzysztof Pszeniczny
  (7) Bartosz Tarnawski

- Na **Bałtycką Olimpiadę Informatyczną BOI'2012**, która odbędzie się na Łotwie w terminie 3–7 maja 2012 roku, pojadą zawodnicy, którzy nie uczęszczają do klas maturalnych, w kolejności rankingowej:

  (1) Karol Farbiś
  (2) Krzysztof Pszeniczny
  (3) Szymon Stankiewicz
  (4) Szymon Łukasz
  (5) Przemysław Jakub Kozłowski
  (6) Stanisław Dobrowolski

  rezerwowi:

  (7) Konrad Paluszek
  (8) Kamil Żyła

- W **Obozie Czesko-Polsko-Słowackim**, który odbędzie się w Czechach, wezmą udział reprezentanci na Międzynarodową Olimpiadę Informatyczną, wraz z zawodnikami rezerwowymi.
- W **Obozie Naukowo-Treningowym im. Antoniego Kreczmara** w Krynicy Zdrój wezmą udział reprezentanci na Międzynarodową Olimpiadę Informatyczną, wraz z zawodnikami rezerwowymi, oraz laureaci i finaliści Olimpiady, którzy nie uczęszczają w tym roku szkolnym do programowo najwyższej klasy szkoły ponadgimnazjalnej.

Sekretariat wystawił łącznie 30 zaświadczeń o uzyskaniu tytułu laureata, 16 zaświadczeń o uzyskaniu tytułu wyróżnionego finalisty oraz 54 zaświadczenia o uzyskaniu tytułu finalisty XIX Olimpiady Informatycznej.

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

- Michał Adamczyk (student Uniwersytetu Warszawskiego)
  - Patryk Czajka – finalista
- Marcin Andrychowicz (student Uniwersytetu Warszawskiego)
  - Karol Farbiś – laureat I miejsca
  - Szymon Stankiewicz – laureat II miejsca
  - Norbert Gregorek – finalista
  - Ewelina Krakowiak – finalistka
- Marcin Baczyński (student Uniwersytetu Warszawskiego)
  - Maciej Kacprzak – finalista z wyróżnieniem
- Michał Bejda (student Uniwersytetu Jagiellońskiego)
  - Szymon Policht – finalista
- Maciej Borsz (student Uniwersytetu Warszawskiego)
  - Michał Kowalczyk – laureat III miejsca
- Iwona Bujnowska (I Liceum Ogólnokształcące im. A. Mickiewicza, Białystok)
  - Mateusz Kopeć – laureat II miejsca

  - Przemysław Jakub Kozłowski – laureat II miejsca
  - Sebastian Daniel Nowak – laureat II miejsca
  - Andrzej Białokozowicz – finalista z wyróżnieniem
  - Mateusz Bałasz – finalista
  - Mateusz Chołołowicz – finalista
  - Piotr Gawryluk – finalista
  - Krzysztof Lewko – finalista
  - Michał Majewski – finalista
- Ireneusz Bujnowski (I Liceum Ogólnokształcące im. A. Mickiewicza, Białystok)
  - Mateusz Kopeć – laureat II miejsca
  - Przemysław Jakub Kozłowski – laureat II miejsca
  - Sebastian Daniel Nowak – laureat II miejsca
  - Andrzej Białokozowicz – finalista z wyróżnieniem
  - Marek Sokołowski – finalista z wyróżnieniem
  - Mateusz Bałasz – finalista
  - Mateusz Chołołowicz – finalista
  - Piotr Gawryluk – finalista
  - Krzysztof Lewko – finalista
  - Michał Majewski – finalista
- Ireneusz Chromiński (I Liceum Ogólnokształcące im. B. Prusa, Siedlce)
  - Jakub Śliwiński – finalista
- Czesław Drozdowski (XIII Liceum Ogólnokształcące, Zespół Szkół Ogólnokształcących nr 7, Szczecin)
  - Paweł Nowak – laureat III miejsca
  - Robert Błaszkiewicz – finalista
- Marcin Dublański (student Uniwersytetu Wrocławskiego)
  - Bartłomiej Dudek – laureat I miejsca
- Lech Duraj (Uniwersytet Jagielloński, Kraków)
  - Wiktor Kuropatwa – laureat I miejsca
  - Michał Zając – laureat II miejsca
  - Piotr Bejda – laureat III miejsca
  - Leszek Kania – finalista z wyróżnieniem
  - Krzysztof Kleiner – finalista z wyróżnieniem
  - Krzysztof Kulig – finalista z wyróżnieniem
  - Michał Piekarz – finalista z wyróżnieniem
  - Grzegorz Świrski – finalista z wyróżnieniem
  - Anna Dymek – finalistka
  - Michał Dyrek – finalista
  - Wojciech Kruk – finalista
  - Krzysztof Mędrela – finalista
  - Szymon Policht – finalista
  - Krzysztof Potempa – finalista
  - Michał Seweryn – finalista

  - Patryk Urbański – finalista
- Andrzej Dyrek (V Liceum Ogólnokształcące im. A. Witkowskiego, Kraków)
  - Szymon Łukasz – laureat II miejsca
  - Leszek Kania – finalista z wyróżnieniem
  - Krzysztof Kulig – finalista z wyróżnieniem
  - Michał Piekarz – finalista z wyróżnieniem
  - Grzegorz Świrski – finalista z wyróżnieniem
  - Wojciech Kruk – finalista
  - Szymon Policht – finalista
  - Michał Seweryn – finalista
  - Patryk Urbański – finalista
- Marek Gałaszewski (I Liceum Ogólnokształcące im. M. Konopnickiej, Suwałki)
  - Konrad Kijewski – laureat III miejsca
- Alina Gościniak (VIII Liceum Ogólnokształcące im. A. Mickiewicza, Poznań)
  - Bartosz Prusak – finalista
- Bartosz Górski (student Uniwersytetu Warszawskiego)
  - Piotr Jarosz – finalista z wyróżnieniem
  - Marek Bardoński – finalista
- Zbigniew Griese (I Liceum Ogólnokształcące im. E. Dembowskiego, Zielona Góra)
  - Karol Kosik – finalista
- Eugeniusz Gwóźdź (Liceum Ogólnokształcące nr 1 im. Noblistów Polskich, Zespół Szkół Ponadgimnazjalnych nr 1, Rydułtowy)
  - Marcin Kudla – finalista
- Grzegorz Herman (Uniwersytet Jagielloński)
  - Szymon Łukasz – laureat II miejsca
- Krzysztof Hyżyk (VI Liceum Ogólnokształcące im. J. i J. Śniadeckich, Zespół Szkół Ogólnokształcących nr 6, Bydgoszcz)
  - Aleksander Kramarz – laureat III miejsca
  - Grzegorz Białek – finalista z wyróżnieniem
- Wiktor Janas (XIV Liceum Ogólnokształcące im. Polonii Belgijskiej, Zespół Szkół nr 14, Wrocław)
  - Michał Kownacki – finalista z wyróżnieniem
- Evelyn Jelec (I Liceum Ogólnokształcące im. W. Kętrzyńskiego, Giżycko)
  - Jakub Kołodziej – laureat II miejsca
- Mateusz Jurczyk (Google)
  - Paweł Kura – finalista
- Jerzy Karpiński (PGNiG w Jaśle)
  - Marcin Karpiński – finalista
- Henryk Kawka (Zespół Szkół nr 7, Lublin, i Zespół Szkół im. Z. Chmielewskiego, Nałęczów)
  - Michał Błaziak – finalista

  - Albert Gutowski – finalista
  - Kamil Krzysztof Magryta – finalista
- Karol Konaszyński (student Uniwersytetu Wrocławskiego)
  - Daniel Danielski – finalista
- Sławomir Krzywicki (I Liceum Ogólnokształcące im. E. Dembowskiego, Zielona Góra)
  - Karol Kosik – finalista
  - Piotr Kozakowski – finalista
- Tomasz Kulczyński (student Uniwersytetu Warszawskiego)
  - Piotr Jarosz – finalista z wyróżnieniem
  - Marek Bardoński – finalista
- Wojciech Kwaśny (I Liceum Ogólnokształcące im. J. Słowackiego, Akademicki Zespół Szkół Ogólnokształcących, Chorzów)
  - Artur Minorczyk – finalista
- Anna Beata Kwiatkowska (Zespół Szkół UMK Gimnazjum i Liceum Akademickie, Toruń)
  - Michał Łazowik – finalista
- Romualda Laskowska (I Liceum Ogólnokształcące im. T. Kościuszki, Legnica)
  - Wojciech Janczewski – laureat III miejsca
- Dariusz Lipiński (Telekomunikacja Polska SA)
  - Adam Lipiński – finalista
- Krzysztof Lis (student Uniwersytetu Warszawskiego)
  - Patryk Czajka – finalista
- Krzysztof Loryś (Uniwersytet Wrocławski)
  - Daniel Danielski – finalista
- Paweł Mateja (I Liceum Ogólnokształcące im. M. Kopernika, Łódź)
  - Kamil Rychlewicz – finalista z wyróżnieniem
- Dawid Matla (XIV Liceum Ogólnokształcące im. Polonii Belgijskiej, Zespół Szkół nr 14, Wrocław)
  - Bartłomiej Dudek – laureat I miejsca
  - Mateusz Gołębiewski – laureat I miejsca
  - Michał Kownacki – finalista z wyróżnieniem
  - Michał Kiełbowicz – finalista
- Maciej Matraszek (student Uniwersytetu Warszawskiego)
  - Rafał Stefański – laureat III miejsca
  - Paweł Tabaszewski – finalista z wyróżnieniem
  - Patryk Czajka – finalista
  - Marek Sommer – finalista
- Mirosław Mortka (VI Liceum Ogólnokształcące, Zespół Szkół Ogólnokształcących nr 6 im. J. Kochanowskiego, Radom)
  - Karol Farbiś – laureat I miejsca
  - Szymon Stankiewicz – laureat II miejsca

  - Norbert Gregorek – finalista
  - Ewelina Krakowiak – finalistka
- Rafał Nowak (Uniwersytet Wrocławski)
  - Michał Kiełbowicz – finalista
  - Krzysztof Story – finalista
- Katarzyna Ochmińska (I Liceum Ogólnokształcące im. Księcia A. J. Czartoryskiego, Puławy)
  - Kamil Krzysztof Magryta – finalista
- Grzegorz Owsiany (Wyższa Szkoła Informatyki i Zarządzania, Rzeszów)
  - Krzysztof Pszeniczny – laureat I miejsca
  - Łukasz Majcher – laureat III miejsca
- Marcin Panasiuk (absolwent Uniwersytetu Wrocławskiego)
  - Wojciech Janczewski – laureat III miejsca
- Małgorzata Piekarska (VI Liceum Ogólnokształcące im. J. i J. Śniadeckich, Zespół Szkół Ogólnokształcących nr 6, Bydgoszcz)
  - Michał Kowalczyk – laureat III miejsca
  - Aleksander Kramarz – laureat III miejsca
  - Grzegorz Białek – finalista z wyróżnieniem
  - Łukasz Łabęcki – finalista
- Mirosław Pietrzycki (I Liceum Ogólnokształcące im. S. Staszica, Lublin)
  - Michał Błaziak – finalista
  - Albert Gutowski – finalista
- Karol Pokorski (student Uniwersytetu Wrocławskiego)
  - Paweł Wegner – finalista
- Adam Polak (student Uniwersytetu Jagiellońskiego, Kraków)
  - Wiktor Kuropatwa – laureat I miejsca
  - Piotr Bejda – laureat III miejsca
  - Leszek Kania – finalista z wyróżnieniem
  - Krzysztof Kleiner – finalista z wyróżnieniem
  - Grzegorz Świrski – finalista z wyróżnieniem
  - Anna Dymek – finalistka
  - Wojciech Kruk – finalista
  - Szymon Policht – finalista
  - Patryk Urbański – finalista
- Grzegorz Prusak (student Politechniki Poznańskiej)
  - Bartosz Prusak – finalista
- Damian Rusak (student Uniwersytetu Wrocławskiego)
  - Bartłomiej Dudek – laureat I miejsca
  - Mateusz Gołębiewski – laureat I miejsca
- Antoni Salamon (Katolickie Liceum Ogólnokształcące, Zespół Katolickich Szkół Ogólnokształcących nr 1 im. bł. ks. E. Szramka, Katowice)
  - Bartosz Tarnawski – laureat I miejsca

- Leszek Samluk (I Liceum Ogólnokształcące im. T. Kościuszki, Łomża)
  - Marek Sokołowski – finalista z wyróżnieniem
- Agnieszka Samulska (VIII Liceum Ogólnokształcące i Gimnazjum nr 58 z Oddziałami Dwujęzycznymi im. Króla Władysława IV, Warszawa)
  - Krzysztof Kiewicz – laureat III miejsca
  - Krzysztof Piesiewicz – finalista
- Piotr Sielski (Uniwersytet Łódzki)
  - Kamil Rychlewicz – finalista z wyróżnieniem
- Hanna Stachera (XIV Liceum Ogólnokształcące im. S. Staszica, Warszawa)
  - Paweł Tabaszewski – finalista z wyróżnieniem
  - Patryk Czajka – finalista
  - Wojciech Jabłoński – finalista
- Jakub Sygnowski (student Uniwersytetu Warszawskiego)
  - Patryk Czajka – finalista
- Bartosz Szreder (doktorant Uniwersytetu Warszawskiego)
  - Wojciech Nadara – laureat I miejsca
  - Krzysztof Piesiewicz – finalista
  - Adam Trzaskowski – finalista
- Ryszard Szubartowski (III LO im. Marynarki Wojennej RP, Zespół Szkół Ogólnokształcących nr 1, Gdynia)
  - Kamil Żyła – laureat II miejsca
  - Błażej Magnowski – laureat III miejsca
  - Stanisław Barzowski – finalista z wyróżnieniem
  - Piotr Chabierski – finalista
  - Bartosz Łukasiewicz – finalista
  - Maciej Obuchowski – finalista
  - Aleksander Szulc – finalista
  - Wiktor Teleżyński – finalista
  - Paweł Wegner – finalista
- Michał Śliwiński (Liceum Ogólnokształcące nr III, Wrocław)
  - Michał Łowicki – laureat II miejsca
  - Konrad Cichy – finalista z wyróżnieniem
  - Grzegorz Głuch – finalista
  - Radosław Serafin – finalista
- Joanna Śmigielska (XIV Liceum Ogólnokształcące im. S. Staszica, Warszawa)
  - Stanisław Dobrowolski – laureat II miejsca
  - Rafał Stefański – laureat III miejsca
  - Patryk Czajka – finalista
  - Wojciech Jabłoński – finalista
  - Aleksander Matusiak – finalista
  - Marek Sommer – finalista
  - Adam Trzaskowski – finalista
- Robert Świder (IV Liceum Ogólnokształcące im. M. Kopernika, Rzeszów)

  - Łukasz Majcher – laureat III miejsca
- Jacek Tomasiewicz (student Uniwersytetu Warszawskiego)
  - Mateusz Chołołowicz – finalista
- Justyna Wilińska (VIII Liceum Ogólnokształcące im. Króla Władysława IV, Warszawa)
  - Damian Czapnik – finalista
  - Mikołaj Lisik – finalista

Zgodnie z decyzją Komitetu Głównego z dnia 29 marca 2012 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 XIX Olimpiadzie Informatycznej, zadania konkursowe oraz wzorcowe rozwiązania. W publikacji tej znajdą się także zadania z międzynarodowych zawodów informatycznych.

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

Warszawa, 3 lipca 2012 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 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. Za każde zadanie zawodnik może zdobyć maksymalnie 100 punktów, gdzie 100 jest sumą maksymalnych liczb punktów za poszczególne testy (lub dane z wynikami) dla tego zadania. Oceną rozwiązań zawodnika jest suma punktów za poszczególne zadania. Oceny rozwiązań zawodników 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.

W przypadku uznania przez Komitet Główny pracy za niesamodzielną lub zespołową zawodnicy mogą zostać zdyskwalifikowani.

(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 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.

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

(4) Komitet powołuje i rozwiązuje komitety okręgowe Olimpiady.

(5) 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.

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

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

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

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

(10) Komitet dysponuje funduszem Olimpiady za zgodą Organizatora i za pośrednictwem kierownika organizacyjnego Olimpiady.

(11) Komitet przyjmuje plan finansowy Olimpiady na przyszły rok na ostatnim posiedzeniu w roku poprzedzającym.

(12) Komitet przyjmuje sprawozdanie finansowe z przebiegu Olimpiady na ostatnim posiedzeniu w roku, na dzień 30 listopada.

(13) 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.

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

(15) Kierownik Jury w porozumieniu z przewodniczącym powołuje i odwołuje członków Jury Olimpiady, które jest odpowiedzialne za opracowanie i sprawdzanie zadań.

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

(17) 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.

(18) 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.

(19) 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.

(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 ten 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 może przyznawać 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 Główny 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 XIX Olimpiady Informatycznej w roku szkolnym 2011/2012

§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 400 uczestników, których rozwiązania zadań I stopnia zostaną ocenione najwyżej; do zawodów III stopnia — 80 uczestników, których rozwiązania zadań II stopnia zostaną ocenione najwyżej. Komitet Główny może zmienić podane liczby zakwalifikowanych uczestników co najwyżej o 25%.

(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 — 17 października–14 listopada 2011 r.
  ogłoszenie wyników w witrynie Olimpiady — 9 grudnia 2011 r. godz. 18.00
  rozesłanie pocztą materiałów Olimpiady i Asseco (w tym związanych z zawodami II stopnia) do wszystkich uczestników Olimpiady — 16 grudnia 2011 r.
- zawody II stopnia — 7–9 lutego 2012 r.
  ogłoszenie wyników w witrynie Olimpiady — 17 lutego 2012 r. godz. 18.00
- zawody III stopnia — 27–31 marca 2012 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. W przypadku uznania przez Komitet Główny pracy za niesamodzielną lub zespołową zawodnicy mogą zostać zdyskwalifikowani.

(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 powinny być podane w komentarzu na początku każdego programu.

(5) Nazwy plików z programami w postaci źródłowej muszą mieć następujące rozszerzenia zależne od użytego języka programowania:

Pascal	`pas`
C	`c`
C++	`cpp`

(6) Podczas oceniania skompilowane programy będą wykonywane w wirtualnym środowisku uruchomieniowym modelującym zachowanie 32-bitowego procesora serii Intel Pentium 4, pod kontrolą systemu operacyjnego Linux. Ma to na celu uniezależnienie mierzonego czasu działania programu od modelu komputera, na którym odbywa się sprawdzanie. Daje także zawodnikom możliwość wygodnego

testowania efektywności działania programów w warunkach oceny. Przygotowane środowisko jest dostępne, wraz z opisem działania, w witrynie Olimpiady, na stronie *Środowisko testowe* w dziale „Dla zawodników" (zarówno dla systemu Linux, jak i Windows).

(7) W uzasadnionych przypadkach Komitet Główny zastrzega sobie prawo do oceny rozwiązań w rzeczywistym środowisku systemu operacyjnego Linux.

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

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

§4 ZAWODY I STOPNIA

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

- poprzez System Internetowy Olimpiady, zwany dalej SIO, o adresie: `http://sio.mimuw.edu.pl`, do 14 listopada 2011 r. do godz. 12.00 (południe). Komitet Główny nie ponosi odpowiedzialności za brak możliwości przekazania rozwiązań przez Internet w sytuacji nadmiernego obciążenia lub awarii SIO. 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 dokładnie 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 14 listopada 2011 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 SIO. **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 (CD lub DVD) 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 tej witrynie. 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 SIO dokładnie jedno konto. Zawodnicy korzystający z więcej niż jednego konta mogą zostać zdyskwalifikowani.

(6) Rozwiązanie każdego zadania można zgłosić w SIO co najwyżej 10 razy. Spośród tych zgłoszeń oceniane 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 są proszeni 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) Poprzez SIO udostępniane są narzędzia do sprawdzania rozwiązań pod względem formalnym. Szczegóły dotyczące sposobu postępowania będą dokładnie podane w witrynie.

(9) Od 28.11.2011 r. poprzez SIO każdy zawodnik będzie mógł zapoznać się ze wstępną oceną swojej pracy.

(10) Do 2.12.2011 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 2.12.2011 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. Awarie sprzętu należy zgłaszać dyżurującym członkom Komisji Regulaminowej.

(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 za pomocą skryptu do weryfikacji rozwiązań na komputerze zawodnika.

(15) Rozwiązanie każdego zadania można zgłosić w SIO co najwyżej 10 razy. Spośród tych zgłoszeń ocenianie jest jedynie najpóźniejsze.

(16) Jeżeli awaria systemu SIO bądź połączenia sieciowego uniemożliwiła zawodnikom wysyłanie rozwiązań przed końcem zawodów, wówczas ci zawodnicy mają możliwość pozostawienia swoich rozwiązań na komputerze w katalogu wskazanym przez organizatorów. Zawodnicy, którzy chcą skorzystać z tej możliwości, niezwłocznie po zakończeniu sesji a przed opuszczeniem sali zawodów powinni wręczyć pisemne oświadczenie dyżurującemu w tej sali członkowi Komisji Regulaminowej. Oświadczenie to musi zawierać imię i nazwisko zawodnika, numer stanowiska oraz informację o zadaniach, których rozwiązania powinny zostać pobrane z komputera. Złożenie takiego oświadczenia powoduje, że rozwiązania wskazanych zadań złożone wcześniej w SIO nie będą rozpatrywane.

(17) W sprawach spornych decyzje podejmuje Jury Odwoławcze, złożone z jurora niezaangażowanego w rozważaną kwestię i wyznaczonego członka Komitetu Głównego lub kierownika danego regionu podczas II etapu. 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 XXIV Międzynarodową Olimpiadę Informatyczną w 2012 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 XIII Obozie Naukowo-Treningowym im. Antoniego Kreczmara, który odbędzie się w czasie wakacji 2012 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

Marian M. Kędzierski
Treść zadania, Opracowanie

Michał Bejda
Program

Dostępna pamięć: 64 MB.

OI, etap I, 17.10–14.11.2011

Festyn

W Bajtogrodzie trwa festyn charytatywny, a Ty jesteś jedną z kwestujących osób. Ominęło Cię parę atrakcji, w tym bieg z przeszkodami. Bajtazar, miłośnik zagadek logicznych, obiecał przekazać spory datek, jeżeli rozwiążesz jego zagadkę.

Nie znasz dokładnych wyników wyścigu. Bajtazar zdradził Ci jednak część informacji na ich temat, a teraz pyta Cię, ile maksymalnie różnych wyników w wyścigu mogło mieć miejsce. Dwaj uczestnicy mają różne wyniki, jeśli nie przybiegli do mety w tym samym czasie. Dowiedziałeś się, że każdy uczestnik wyścigu uzyskał czas będący całkowitą liczbą sekund. Poznałeś również związki pomiędzy wynikami niektórych par biegaczy. Bajtazar podał Ci część z tych par biegaczy (A, B), dla których wynik A był dokładnie o jedną sekundę lepszy niż wynik B, oraz część z tych par biegaczy (C, D), dla których wynik C był co najmniej tak samo dobry jak wynik D. Napisz program, który pomoże Ci znaleźć rozwiązanie zagadki Bajtazara.

Wejście

Pierwszy wiersz standardowego wejścia zawiera trzy liczby całkowite nieujemne n, m_1, m_2 ($2 \leqslant n \leqslant 600$, $1 \leqslant m_1 + m_2 \leqslant 100\,000$), pooddzielane pojedynczymi odstępami, oznaczające odpowiednio: liczbę biegaczy, liczbę poznanych par biegaczy pierwszego typu (A, B) oraz liczbę poznanych par biegaczy drugiego typu (C, D). Biegacze są ponumerowani od 1 do n.

W kolejnych m_1 wierszach podane są pary biegaczy pierwszego typu, po jednej parze w wierszu — i-ty z tych wierszy zawiera dwie liczby a_i oraz b_i, $a_i \neq b_i$, oddzielone pojedynczym odstępem, oznaczające, że biegacz a_i miał wynik dokładnie o jedną sekundę lepszy niż biegacz b_i.

W kolejnych m_2 wierszach podane są pary biegaczy drugiego typu, po jednej parze w wierszu — i-ty z tych wierszy zawiera dwie liczby c_i oraz d_i, $c_i \neq d_i$, oddzielone pojedynczym odstępem, oznaczające, że biegacz c_i miał wynik nie gorszy niż biegacz d_i.

Wyjście

W pierwszym i jedynym wierszu standardowego wyjścia Twój program powinien wypisać jedną liczbę całkowitą równą maksymalnej liczbie różnych wyników czasowych, jakie mogli osiągnąć biegacze przy założeniu podanych kryteriów.

Jeśli nie istnieje kolejność zawodników spełniająca ograniczenia Bajtazara, wyjście powinno składać się z jednego wiersza zawierającego słowo „`NIE`" (bez cudzysłowów).

W testach wartych przynajmniej 15% punktów zachodzi dodatkowe ograniczenie $n \leqslant 10$.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*

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

Wyjaśnienie do przykładu: *Bieg mógł zakończyć się na dwa sposoby:*

1. *pierwszy jest biegacz nr 3, na drugim miejscu są ex aequo biegacze nr 1 i 4, a ostatni jest biegacz nr 2;*
2. *na pierwszym miejscu są ex aequo biegacze nr 1 i 3, a na drugim ex aequo biegacze nr 2 i 4.*

W pierwszej z powyższych możliwości mamy najwięcej różnych wyników czasowych, tzn. trzy.

Rozwiązanie

Analiza problemu

W zadaniu rozważamy n zmiennych całkowitych $t_1, t_2, \ldots, t_n$, oznaczających czasy dotarcia na metę kolejnych uczestników biegu. Pomiędzy niektórymi z tych zmiennych znane są pewne zależności, które mogą być dwojakiego typu:

1. $t_a + 1 = t_b$,
2. $t_c \leqslant t_d$.

Przez *wartościowanie* φ zmiennych t_i będziemy rozumieli dowolne przypisanie im wartości całkowitych, które zachowuje podane ograniczenia. *Mocą* wartościowania będziemy nazywali liczbę różnych wartości przypisanych przez to wartościowanie. W zadaniu mamy znaleźć maksymalną moc wartościowania lub stwierdzić, że żadne wartościowanie nie istnieje.

Możemy nieco uprościć sformułowanie zadania przez sprowadzenie dwóch typów zależności do jednego, w którym x_i i y_i oznaczają zmienne, a k_i jest stałą:

$$x_i + k_i \leqslant y_i.$$

Nazwijmy nierówności tej postaci *normalnymi*.

Każdą zależność pierwszego typu możemy zakodować jako dwie nierówności normalne:

$$t_a + 1 \leqslant t_b,$$
$$t_b - 1 \leqslant t_a,$$

a każdą zależność drugiego typu jako:

$$t_c + 0 \leqslant t_d.$$

Graf nierówności

Rozważmy graf $G = (V, E)$, w którym wierzchołkami są zmienne t_i, zaś krawędzie reprezentują nierówności normalne. Każdej nierówności $x_i + k_i \leqslant y_i$ odpowiada krawędź e_i z wierzchołka x_i do wierzchołka y_i o wadze $w(e_i) = k_i$. Graf ten jest tylko nieco inną reprezentacją układu nierówności normalnych i dalej będziemy traktowali te reprezentacje zamiennie.

Niech $p = v_1, v_2, \ldots, v_m$ będzie dowolną ścieżką w grafie nierówności. Oznaczmy kolejne krawędzie na tej ścieżce następująco: $e_i = (v_i, v_{i+1})$ dla $i = 1, 2, \ldots, m-1$. Łącząc odpowiadające im nierówności w ciąg, możemy wywnioskować, że:

$$v_1 \leqslant v_2 - w(e_1) \leqslant v_3 - w(e_1) - w(e_2) \leqslant \ldots \leqslant v_m - w(e_1) - \ldots - w(e_{m-1}), \quad (1)$$

co znaczy, że każde wartościowanie φ musi spełniać zależność $\varphi(v_1) + w(p) \leqslant \varphi(v_m)$, gdzie $w(p) = \sum_{i=1}^{m-1} w(e_i)$ jest wagą ścieżki p.

Widząc to, dla każdej pary wierzchołków $u, v \in V$ od razu chcielibyśmy znaleźć najdłuższą ścieżkę z u do v, gdyż to ona dałaby najsilniejszą zależność spośród wszystkich ścieżek z u do v. Niestety, mogą zdarzyć się takie wierzchołki u, v, dla których nie istnieje żadna ścieżka z u do v. Co więcej, może się zdarzyć, że z pewnego wierzchołka do pewnego innego prowadzą ścieżki dowolnie dużej długości. Prawdopodobnie w większości grafów, które widzieliśmy w życiu, długość ścieżki pomiędzy danymi dwoma wierzchołkami była nieograniczona od góry (zakładając możliwość odwiedzania wierzchołków wielokrotnie). Tutaj sytuacja może być inna, bo w przypadku rozważanego grafu wagi krawędzi często będą ujemne. Niemniej jednak zawsze mogą się w nim zdarzyć ścieżki o nieograniczonej długości.

Aby łatwiej uporządkować te wszystkie przypadki, zajmijmy się najpierw ostatnim. Co oznaczałoby istnienie dowolnie długich ścieżek z pewnego wierzchołka u do pewnego wierzchołka v? Otóż jest to możliwe wtedy i tylko wtedy, gdy graf G zawiera cykl o dodatniej wadze. Jeśli tak jest, to przemierzając taki cykl dowolnie wiele razy, możemy otrzymać dowolnie długą ścieżkę. Z drugiej strony, jeśli graf nie zawierałby cyklu o dodatniej wadze, to z każdej ścieżki można by usunąć wszystkie cykle, nie zmniejszając jej długości, i wtedy otrzymalibyśmy ścieżkę przemierzającą co najwyżej $|V|$ wierzchołków. Jednak ścieżki o ograniczonej liczbie wierzchołków mają też ograniczoną długość, bo wagi pojedynczych krawędzi są ograniczone. Stąd brak dodatniego cyklu oznaczałby, że wszystkie ścieżki w tym grafie mają długości ograniczone z góry.

No dobrze. Co w takim razie zrobić, jeśli graf G zawiera dodatni cykl? Odpowiedź jest bardzo prosta: wypisać `NIE`. Weźmy bowiem dowolny taki cykl p i dowolny wierzchołek v na tym cyklu. W podobny sposób, jak wyżej, możemy wywnioskować, że każde wartościowanie φ musi spełnić zależność $\varphi(v) + w(p) \leqslant \varphi(v)$, czyli $w(p) \leqslant 0$, co jest sprzeczne z założeniem o dodatniej wadze cyklu. Stąd zaś wniosek, że dla zadanych ograniczeń nie istnieje żadne wartościowanie.

Przyjmijmy teraz, że graf nie zawiera dodatniego cyklu. Okazuje się, że wtedy istnieje przynajmniej jedno poprawne wartościowanie. Mówi o tym następujący lemat, który wykorzystamy w dalszej części opisu. Dla dowolnych wierzchołków $u, v \in V$, przez $\mathrm{lp}_G(u, v)$ oznaczmy długość najdłuższej ścieżki z wierzchołka u do wierzchołka v w grafie G, pod warunkiem, że istnieje jakakolwiek ścieżka z u do v w G.

Lemat 1. Układ nierówności normalnych posiada wartościowanie wtedy i tylko wtedy, gdy jego graf nie zawiera cykli o dodatniej wadze.

Dowód: Przed chwilą udowodniliśmy, że istnienie dodatniego cyklu wymusza brak wartościowań, zajmijmy się więc implikacją przeciwną. Załóżmy, że graf G nie zawiera dodatniego cyklu. Skonstruujemy dla niego wartościowanie.

Najpierw dodajmy do grafu dodatkowy wierzchołek $s \notin V$ oraz krawędzie (s, v) o zerowych wagach dla wszystkich wierzchołków $v \in V$, otrzymując w ten sposób

graf G'. Wtedy funkcja $\varphi(v) = \mathrm{lp}_{G'}(s, v)$ jest wartościowaniem dla tak wzbogaconego układu nierówności normalnych. Istotnie, weźmy dowolną nierówność $u + k \leqslant v$ i pokażmy, że jest spełniona. Niech $s \rightsquigarrow u$ będzie najdłuższą ścieżką z s do u. Zatem ścieżka $s \rightsquigarrow u \to v$ ma długość $\varphi(u) + k$, a ponieważ nie może być dłuższa niż najdłuższa ścieżka z s do v, zatem $\varphi(u) + k \leqslant \varphi(v)$. Wobec dowolności wyboru nierówności, funkcja φ jest wartościowaniem dla grafu G', a więc także dla G. ■

Silnie spójne składowe grafu nierówności

Weźmy teraz dwa dowolne wierzchołki $u, v \in V$ i zastanówmy się, jakie są możliwe różnice ich wartości $\varphi(v) - \varphi(u)$ dla różnych wartościowań φ. Jeżeli istnieje ścieżka zarówno z u do v, jak i w drugą stronę, to różnica ta jest ograniczona zarówno z dołu, jak i z góry, mamy wtedy bowiem dwie następujące zależności:

$$\varphi(u) + \mathrm{lp}_G(u, v) \leqslant \varphi(v), \quad \varphi(v) + \mathrm{lp}_G(v, u) \leqslant \varphi(u),$$

które można łącznie przedstawić jako:

$$\mathrm{lp}_G(u, v) \leqslant \varphi(v) - \varphi(u) \leqslant -\mathrm{lp}_G(v, u). \tag{2}$$

Do tego wniosku wrócimy później, ale najpierw zastanówmy się, co w przypadku, gdy nie istnieje ścieżka z u do v lub nie istnieje ścieżka z v do u. Mówimy wtedy, że wierzchołki te leżą w różnych *silnie spójnych składowych* grafu G.

Silnie spójna składowa grafu skierowanego to taki podzbiór jego wierzchołków, że pomiędzy każdymi dwoma wierzchołkami z tej składowej istnieją ścieżki w obie strony, zaś dla dowolnego wierzchołka u z tej składowej i wierzchołka v spoza niej nie istnieje ścieżka przynajmniej w jedną stronę pomiędzy nimi. O silnie spójnych składowych można przeczytać np. w książce [23]. Mają one tę ciekawą własność, że graf silnie spójnych składowych, w którym krawędź ze składowej A prowadzi do składowej B wtedy i tylko wtedy, gdy z każdego (lub równoważnie: dowolnego) wierzchołka $a \in A$ istnieje ścieżka do każdego (lub równoważnie: dowolnego) wierzchołka $b \in B$, jest DAG-iem (ang. *directed acyclic graph*, czyli skierowany graf acykliczny). Przypomnijmy, że każdy DAG można posortować topologicznie, tzn. ustawić jego wierzchołki w ciąg tak, aby każda krawędź prowadziła z lewej strony w prawą.

Korzystając z tych własności, podzielmy graf G na silnie spójne składowe i posortujmy je topologicznie, otrzymując ciąg składowych $V_1, V_2, \ldots, V_s$. Następnie znajdźmy wartościowanie o największej mocy osobno w każdej z nich i oznaczmy te wartościowania odpowiednio przez $\varphi_1, \varphi_2, \ldots, \varphi_s$. Oznaczmy przez K maksymalną wagę krawędzi w grafie: $K = \max(0, \max_{e \in E} w(e))$, a przez d_i (dla $i = 1, 2, \ldots, s$) — najdłuższą ścieżkę w każdej ze składowych:

$$d_i = \max_{u,v \in V_i} \mathrm{lp}_{V_i}(u, v).$$

Zauważmy, że dodanie tej samej liczby do wszystkich zmiennych w wartościowaniu φ_i nie wpływa ani na poprawność tego wartościowania, ani na jego moc. Możemy więc tak „przesunąć” każde z wartościowań φ_i, aby dla każdego $i = 1, 2, \ldots, s$ zachodził warunek:

$$\min_{v \in V_i} \varphi_i(v) = (K + 1) \cdot (i - 1) + (d_1 + \ldots + d_{i-1}).$$

Połączmy wartościowania φ_i w funkcję φ na całym zbiorze zmiennych V następująco:

$$\varphi(v) = \varphi_{c(v)}(v),$$

przy czym $c(v)$ oznacza numer składowej, do której należy wierzchołek v. Okazuje się, że tak utworzona funkcja φ jest wartościowaniem dla całego zbioru, którego moc jest równa sumie mocy wartościowań φ_i.

Po pierwsze: dlaczego funkcja φ jest wartościowaniem? Oczywiście spełnia ona wszystkie zależności pomiędzy parami wierzchołków leżących w tej samej silnie spójnej składowej. Weźmy więc $u \in V_i$ oraz $v \in V_j$, gdzie $i \neq j$. Bez straty ogólności możemy założyć, że $i < j$. Ponieważ składowe są posortowane topologicznie, to nie może istnieć w grafie składowych krawędź z V_j do V_i, a tym samym nie może istnieć w G krawędź z v do u. Może zaś istnieć krawędź z u do v o pewnej wadze k. Jakie wartości może przyjąć $\varphi(v) - \varphi(u)$? Wiemy, że:

- $\varphi(u) = \varphi_i(u) \in [p_i, p_i + d_i]$, gdzie $p_i = (K+1) \cdot (i-1) + (d_1 + \ldots + d_{i-1})$,
- $\varphi(v) = \varphi_j(v) \in [p_j, p_j + d_j]$, gdzie $p_j = (K+1) \cdot (j-1) + (d_1 + \ldots + d_{j-1})$.

Tak więc maksymalna wartość $\varphi(u)$ to $p_i + d_i$, a minimalna wartość $\varphi(v)$ to p_j. Stąd:

$$\varphi(v) - \varphi(u) \geqslant p_j - p_i - d_i = (K+1) \cdot (j-i) + (d_i + \ldots + d_{j-1}) - d_i \geqslant K + 1.$$

Wiemy jednak, że $k < K + 1$, więc nierówność $\varphi(u) + k \leqslant \varphi(v)$ jest spełniona. To pokazuje też, że zbiory wartości poszczególnych wartościowań składowych φ_i są rozłączne, co prowadzi bezpośrednio do wniosku, że moc φ jest równa sumie mocy wszystkich φ_i.

Jaki z tego płynie dla nas wniosek? Bardzo prosty — możemy na początku algorytmu wykonać wstępne obliczenia, polegające na podziale grafu nierówności na silnie spójne składowe, następnie każdą składową zanalizować oddzielnie i zwrócić sumę otrzymanych wyników. Dzielenie grafu na silnie spójne składowe można wykonać przy użyciu dwóch przeszukiwań grafu w głąb (algorytm DFS). Dokładny opis tego algorytmu można znaleźć we wspomnianej już książce [23].

Analiza pojedynczej silnie spójnej składowej

Odtąd możemy już założyć, że rozważany graf nierówności G jest silnie spójny (a więc istnieją w nim ścieżki między każdą parą wierzchołków) oraz nie zawiera dodatnich cykli. Możemy zatem używać oznaczenia $\mathrm{lp}_G(u, v)$ dla dowolnych wierzchołków $u, v \in V$.

Wróćmy teraz do spostrzeżenia (2). Okazuje się, że podane tam ograniczenie na różnicę wartości $\varphi(v) - \varphi(u)$ jest nie tylko prawdziwe, ale też pokazuje wszystkie możliwe wartości, jakie ta różnica może przyjąć.

Twierdzenie 1. *Niech $G = (V, E)$ będzie silnie spójnym grafem nierówności bez dodatnich cykli i niech $u, v \in V$. Niech Φ będzie zbiorem wszystkich wartościowań zbioru V. Wtedy:*

$$\{\varphi(v) - \varphi(u) : \varphi \in \Phi\} = \mathrm{dint}_G(u, v)$$

gdzie zbiór $\mathrm{dint}_G(u, v)$ jest przedziałem liczb całkowitych x, które spełniają $\mathrm{lp}_G(u, v) \leqslant x \leqslant -\mathrm{lp}_G(v, u)$.

Dowód: Wyżej dowiedliśmy już, że $\{\varphi(v) - \varphi(u) : \varphi \in \Phi\} \subseteq \mathrm{dint}_G(u, v)$. Pozostaje wykazać zawieranie odwrotne. Niech k będzie dowolnym elementem przedziału liczb całkowitych $\mathrm{dint}_G(u, v)$. Wykażemy, że istnieje wartościowanie φ, dla którego $\varphi(v) - \varphi(u) = k$.

Dodajmy do grafu G dwie krawędzie odpowiadające dwóm nierównościom normalnym: $u + k \leqslant v$ oraz $v - k \leqslant u$ (razem wyrażającym $v - u = k$); oznaczmy otrzymany graf przez G'. Układ nierówności grafu G posiada wartościowanie φ spełniające $\varphi(v) - \varphi(u) = k$ wtedy i tylko wtedy, gdy G' posiada jakiekolwiek wartościowanie. Z lematu 1 wynika z kolei, że układ nierówności grafu G' posiada wartościowanie wtedy i tylko wtedy, gdy G' nie zawiera dodatniego cyklu. Pozostaje więc jedynie to udowodnić.

Załóżmy przeciwnie, że G' zawiera cykl o dodatniej wadze. Weźmy najmniejszy taki cykl C (ze względu na liczbę wierzchołków). Wtedy musi to być cykl prosty (tzn. wierzchołki na nim nie powtarzają się) i musi zawierać przynajmniej jedną z dwóch dodanych krawędzi, ponieważ graf G nie zawierał żadnego dodatniego cyklu. Jeśli C zawiera obie dodane krawędzie, to musi mieć długość 2, ponieważ jest cyklem prostym. Ale suma wag tych dwóch krawędzi wynosi 0, więc C nie jest wtedy cyklem dodatnim.

Załóżmy więc, że na cyklu C leży dokładnie jedna z dodanych krawędzi. Załóżmy, że jest to (u, v) (przeciwny przypadek rozpatruje się symetrycznie). Niech p będzie ścieżką na cyklu C prowadzącą z v do u. Ponieważ cykl C jest prosty, więc ścieżka p musiała istnieć również w oryginalnym grafie G, skąd $w(p) \leqslant \mathrm{lp}_G(v, u)$. Z założenia zaś $k \leqslant -\mathrm{lp}_G(v, u)$. Stąd $w(C) = k + w(p) \leqslant -\mathrm{lp}_G(v, u) + \mathrm{lp}_G(v, u) = 0$, co jest ponownie sprzeczne z założeniem o dodatniości cyklu C.

Tak więc graf G' nie zawiera cyklu o dodatniej wadze, więc istnieje wartościowanie φ układu nierówności odpowiadającego grafowi G spełniające warunek $\varphi(v) - \varphi(u) = k$. ■

Dzięki twierdzeniu 1 jesteśmy już bardzo blisko rozwiązania. Pozostaje nam tylko sformułowanie następującego wniosku:

Wniosek 1. Niech G będzie grafem jak w twierdzeniu 1. Oznaczmy przez D *maksymalną rozciągłość* wartościowania:

$$D = \max_{u,v \in V} \max\{\varphi(v) - \varphi(u) : \varphi \in \Phi\}.$$

Wtedy $D = \max_{u,v \in V}(-\mathrm{lp}_G(u, v))$. Jeżeli ponadto graf G zawiera jedynie krawędzie o wagach -1, 0 oraz 1, to maksymalną mocą wartościowania tego grafu jest $D + 1$.

Dowód: Bezpośrednio z twierdzenia 1 wynika, że

$$\max\{\varphi(v) - \varphi(u) : \varphi \in \Phi\} = -\mathrm{lp}_G(v, u).$$

Natomiast drugi sformułowany we wniosku fakt wymaga pewnego komentarza.

Niech φ będzie takim wartościowaniem, a u i v takimi wierzchołkami, że $\varphi(v) - \varphi(u) = D$. Niech $u = u_1, u_2, \ldots, u_m = v$ będzie najdłuższą ścieżką z u do v w grafie G. Ponieważ dla każdego $i = 1, \ldots, m - 1$ zachodzi $w(u_i, u_{i+1}) \in \{-1, 0, 1\}$, więc $\varphi(u_{i+1}) - \varphi(u_i) \leqslant 1$, czyli w każdym kroku na tej ścieżce wartościowanie zwiększa

się co najwyżej o jeden. Jednak po przejściu całej ścieżki zwiększa się o D. To znaczy, że każda z wartości od $\varphi(u)$ do $\varphi(v)$ musiała być przyjęta przez φ dla pewnej zmiennej u_i. Dostajemy więc $D+1$ różnych wartości zmiennych. Z drugiej strony nie istnieje wartościowanie o większej mocy, bo wtedy jego rozciągłość $\max_{u,v\in V}(\varphi(v) - \varphi(u))$ musiałaby przekroczyć D. ■

Algorytm

Wiemy zatem, że dla silnie spójnego grafu nierówności G, nie zawierającego cykli dodatnich, maksymalna moc wartościowania wynosi $1 + \max_{u,v\in V}(-\mathrm{lp}_G(u,v))$. Aby znaleźć tę wartość, wystarczy obliczyć długości najdłuższych ścieżek pomiędzy każdą parą wierzchołków w G. Jak znajdować najdłuższe ścieżki? Wystarczy zmienić znak przy wadze każdej krawędzi, otrzymując graf H, a następnie obliczyć w nim (najkrótsze) odległości $\mathrm{dist}_H(u,v)$ między każdą parą wierzchołków u i v. Ponieważ dla każdej pary wierzchołków u i v zachodzi $\mathrm{dist}_H(u,v) = -\mathrm{lp}_G(u,v)$, więc rozwiązaniem jest liczba $1 + \max_{u,v\in V} \mathrm{dist}_H(u,v)$, czyli średnica grafu H powiększona o jeden.

Jest kilka algorytmów, które pozwalają tę średnicę grafu H obliczyć (pamiętajmy tutaj, że graf H może zawierać krawędzie ujemnej długości):

- n powtórzeń algorytmu Bellmana-Forda, co daje złożoność czasową $O(|V|^2 \cdot |E|)$ (takie rozwiązanie pozwalało uzyskać ok. 50% punktów),
- algorytm Floyda-Warshalla o złożoności czasowej $O(|V|^3)$,
- algorytm Johnsona o złożoności czasowej $O(|V| \cdot |E| + |V|^2 \log |E|)$.

Co więcej, każdy z wymienionych algorytmów umożliwia sprawdzenie, czy graf H nie zawiera ujemnych cykli, a zatem czy graf G nie zawiera dodatnich cykli. W rozwiązaniach wzorcowych wykorzystano algorytmy Floyda-Warshalla i Johnsona. O wszystkich trzech algorytmach można przeczytać w książce [23].

Implementację rozwiązań wzorcowych można znaleźć w plikach `fes.cpp`, `fes0.cpp`, `fes1.pas`, `fes2.cpp` i `fes3.cpp`).

Możliwe przyspieszenia

Możliwe jest jeszcze dodatkowe zredukowanie grafu, które wprawdzie może przyspieszyć algorytm, ale nie zmniejsza jego asymptotycznej złożoności.

Otóż na samym początku można zidentyfikować wszystkie grupy zmiennych t_i związanych (bezpośrednio lub pośrednio) zależnościami pierwszego typu, tzn. $t_a + 1 = t_b$. Wtedy wartości zmiennych w jednej takiej grupie są jednoznacznie wyznaczone przez wartość dowolnego jej reprezentanta. Następnie każdą taką grupę można zamienić na pojedynczego reprezentanta lub na dwóch reprezentantów (o największej i najmniejszej wartości w grupie), odpowiednio modyfikując pozostałe nierówności.

To usprawnienie nie było jednak wymagane do uzyskania maksymalnej punktacji.

Testy

Poniższa tabela przedstawia testy wykorzystane do oceny rozwiązań; w szczególności: testy grupy *a* od siódmego włącznie składają się z dużego cyklu i kilku małych składowych sprawdzających poprawność; testy grupy *b* od trzeciego włącznie zostały wygenerowane przez losowe ustawianie zawodników na mecie i ujawnianie losowo zależności pomiędzy nimi; testy grupy *c* to małe testy losowe połączone w DAG; testy grupy *d* to duże testy wydajnościowe wymuszające w niektórych algorytmach dużą liczbę odwiedzin tego samego wierzchołka. Testy z odpowiedzią `NIE` to: *1b*, *4a*, *5a* oraz *6a*.

Liczba *s* podana w opisie testów oznacza liczbę silnie spójnych składowych odpowiedniego grafu nierówności.

Nazwa	n	m_1	m_2	s
fes1a.in	8	3	9	4
fes1b.in	6	3	7	1
fes2a.in	7	2	4	5
fes2b.in	6	2	4	2
fes3a.in	27	12	15	12
fes3b.in	53	71	386	8
fes3c.in	56	40	12	12
fes4a.in	101	20	106	80
fes4b.in	62	41	900	22
fes4c.in	78	60	18	12
fes5a.in	101	20	119	82
fes5b.in	98	17	6 000	49
fes5c.in	250	140	3 129	153
fes6a.in	101	20	144	81
fes6b.in	353	53	6 901	279
fes6c.in	250	155	3 070	159
fes6d.in	600	420	6 365	1
fes7a.in	600	176	397	50
fes7b.in	311	112	3 012	206
fes7c.in	600	718	30 251	290
fes7d.in	600	420	6 365	1

Nazwa	n	m_1	m_2	s
fes8a.in	600	174	390	52
fes8b.in	600	804	5 072	106
fes8c.in	600	763	30 303	283
fes8d.in	600	420	6 365	1
fes9a.in	600	165	404	50
fes9b.in	600	68	50 090	181
fes9c.in	600	795	30 198	279
fes9d.in	600	420	6 365	1
fes10a.in	600	161	416	49
fes10b.in	600	509	50 090	134
fes10c.in	600	816	30 158	281
fes10d.in	600	420	6 365	1
fes11a.in	600	174	390	52
fes11b.in	600	40 139	50 090	30
fes11c.in	600	779	30 198	286
fes11d.in	600	420	6 365	1

Marian M. Kędzierski
Treść zadania, Opracowanie

Zbigniew Wojna
Program

Dostępna pamięć: 128 MB.

OI, etap I, 17.10–14.11.2011

Litery

Mały Jaś ma bardzo długie nazwisko. Nie jest jednak jedyną taką osobą w swoim środowisku. Okazało się bowiem, że jedna z jego koleżanek z przedszkola, Małgosia, ma nazwisko dokładnie tej samej długości, chociaż inne. Co więcej, ich nazwiska zawierają dokładnie tyle samo liter każdego rodzaju — tyle samo liter `A`*, tyle samo liter* `B`*, itd.*

Jaś i Małgosia bardzo się polubili i często bawią się razem. Jedną z ich ulubionych zabaw jest zebranie dużej liczby małych karteczek, napisanie na nich kolejnych liter nazwiska Jasia, a następnie przesuwanie karteczek tak, aby powstało z nich nazwisko Małgosi.

Ponieważ Jaś uwielbia łamigłówki, zaczął zastanawiać się, ile co najmniej zamian sąsiednich liter trzeba wykonać, żeby przekształcić jego nazwisko w nazwisko Małgosi. Nie jest to łatwe zadanie dla kilkuletniego dziecka, dlatego Jaś poprosił Ciebie, głównego programistę w przedszkolu, o napisanie programu, który znajdzie odpowiedź na nurtujące go pytanie.

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita n *(*$2 \leqslant n \leqslant 1\,000\,000$*) oznaczająca liczbę liter w nazwisku Jasia. W drugim wierszu znajduje się* n *kolejnych liter nazwiska Jasia (bez odstępów). W trzecim wierszu znajduje się* n *kolejnych liter nazwiska Małgosi (również bez odstępów). Oba napisy składają się jedynie z wielkich liter alfabetu angielskiego.*

W testach wartych łącznie 30% punktów zachodzi dodatkowy warunek $n \leqslant 1\,000$*.*

Wyjście

Twój program powinien wypisać na standardowe wyjście jedną liczbę całkowitą, oznaczającą minimalną liczbę zamian sąsiednich liter, które przekształcają nazwisko Jasia w nazwisko Małgosi.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*

`3`
`ABC`
`BCA`
	`2`

Rozwiązanie

Analiza problemu

W zadaniu mamy dane dwa słowa $u = u_1u_2 \ldots u_n$ oraz $w = w_1w_2 \ldots w_n$, w których każda litera występuje dokładnie tyle samo razy. Słowa o tej własności określa się czę-

sto mianem *anagramów*. Naszym zadaniem jest znalezienie minimalnej liczby zamian sąsiednich liter pozwalających przekształcić słowo u w słowo w. Takie pojedyncze operacje będziemy odtąd nazywali *przesunięciami*.

Powstaje naturalne pytanie, czy zawsze istnieje sekwencja przesunięć przekształcająca słowo u w słowo w. Na mocy założeń występujących w zadaniu wiemy, że po posortowaniu liter w słowach u oraz w otrzymujemy to samo słowo. A ponieważ przesunięcia pozwalają na posortowanie dowolnego słowa, przykładową sekwencję przesunięć możemy skonstruować w następujący sposób:

- najpierw sortujemy słowo u,
- potem wykonujemy przesunięcia sortujące słowo w, ale w odwrotnej kolejności.

Oczywiście, nie zawsze jest to strategia optymalna (przykładem mogą tu być chociażby słowa „`CBA`" oraz „`BCA`"), jednak jest ona zawsze poprawna.

Dolne ograniczenie

Skoro wiemy już, że przekształcenie słowa u w słowo w przy pomocy przesunięć jest zawsze możliwe, zastanówmy się nad dolnym ograniczeniem na liczbę przesunięć. W tym celu przyda nam się następujące pojęcie:

Definicja 1. Niech u będzie dowolnym anagramem słowa w. *Odległością* słowa u od słowa w będziemy nazywać następującą wartość:

$$d(u,w) = \sum_{a \in A} \sum_{i=1}^{k(a)} |u_{a,i} - w_{a,i}|$$

gdzie $A = \{\mathtt{A}, \mathtt{B}, \ldots, \mathtt{Z}\}$ oznacza alfabet, $k(a)$ oznacza liczbę wystąpień litery a w słowie w (krotność), zaś $u_{a,1}, \ldots, u_{a,k(a)}$ oraz $w_{a,1}, \ldots, w_{a,k(a)}$ to pozycje kolejnych wystąpień tej litery odpowiednio w słowach u oraz w.

Skąd pomysł na taką funkcję? Otóż intuicyjnie ma ona oddawać sumaryczną odległość pomiędzy „odpowiadającymi sobie"[1] literami w słowach u oraz w. Dla pokazania dolnego ograniczenia na wynik nie będzie nam jednak potrzebna wnikliwa analiza funkcji odległości, a jedynie jej następująca prosta własność.

Fakt 1. *Niech u będzie dowolnym anagramem słowa w oraz niech v będzie słowem powstałym z u przez wykonanie dokładnie jednego przesunięcia. Wtedy* $d(u,w) - d(v,w) \in \{-2, 0, 2\}$.

Dowód: Każde przesunięcie zmienia o jeden pozycje dwóch liter, przy czym każdą z nich może albo oddalić od „odpowiadającej jej" litery docelowej, albo ją do niej przybliżyć. Stąd odległość może się zmniejszyć o dwa, zwiększyć o dwa lub pozostać taka sama. ■

[1] Na tym etapie nie jest jeszcze jasne, czy pierwsze wystąpienie litery `A` w słowie u w optymalnej strategii przechodzi na pierwsze wystąpienie litery `A` w słowie w, drugie na drugie, itd. i podobnie dla pozostałych liter.

Prostym wnioskiem z powyższego faktu jest szukane dolne ograniczenie:

Twierdzenie 1. *Minimalna liczba przesunięć przekształcających słowo u w słowo w jest nie mniejsza niż $\frac{1}{2}d(u,w)$.*

Dowód: Wykazaliśmy, że każde przesunięcie zmniejsza odległość słów co najwyżej o dwa. Skoro początkowa odległość słowa u od słowa w wynosi $d(u,w)$, a końcowa — $d(w,w)=0$, to konieczne jest wykonanie co najmniej $\frac{1}{2}d(u,w)$ przesunięć. ■

Rozwiązanie zwracające zawsze wynik $\frac{1}{2}d(u,w)$ okazuje się niepoprawne już dla pary słów „`ABC`" i „`CBA`", ponieważ daje w wyniku 2, podczas gdy poprawnym wynikiem jest 3. Takie rozwiązania na zawodach nie uzyskiwały żadnych punktów (przykładowa implementacja w pliku `litb1.cpp`).

Rozwiązanie poprawne

Tak naprawdę nie udało nam się jeszcze wskazać żadnej konkretnej strategii wykonywania przesunięć: nie wynikła ona ani z analizy dolnego ograniczenia, ani z metody opartej o sortowanie liter w słowach, w której nie widać od razu, jaka jest minimalna liczba przesunięć potrzebnych do posortowania słowa. Czas więc od rozważań czysto teoretycznych przejść do znalezienia pierwszej konkretnej strategii. Narzucającym się pomysłem jest następujące podejście zachłanne.

Niech $a = w_1$ i niech $u_{a,1}$ będzie pierwszym wystąpieniem litery a w słowie u. Wykonajmy $u_{a,1} - 1$ przesunięć w słowie u przemieszczających pierwsze wystąpienie litery a na początek słowa. Następnie odetnijmy pierwsze litery tak zmodyfikowanego słowa u oraz słowa w i całą operację powtórzmy $n-1$ razy.

Czy taka strategia jest optymalna? Nie zawsze algorytmy zachłanne prowadzą do poprawnych albo optymalnych rozwiązań, jednak w tym przypadku odpowiedź jest twierdząca. Musimy to jednak uzasadnić.

Po pierwsze: dlaczego to właśnie litera z pozycji $u_{a,1}$, a nie jakieś inne wystąpienie tej litery w słowie u, ma przejść na pierwszą literę słowa w? Załóżmy przeciwnie, że litera odpowiadająca w_1 w pewnej hipotetycznej strategii optymalnej znajduje się w u na pozycji p, $p > u_{a,1}$. Wtedy w czasie przekształcania słowa u w słowo w musiałby nastąpić moment, w którym litera pochodząca z pozycji $u_{a,1}$ poprzedza bezpośrednio literę z pozycji p, a kolejne przesunięcie zamienia te litery miejscami. Jednak takie przesunięcie nie zmienia w ogóle układu liter w słowie. Rezygnacja z tego jednego przesunięcia da nam zatem strategię lepszą od optymalnej — sprzeczność. Tak więc pierwsza litera słowa w w każdej optymalnej strategii rzeczywiście odpowiada literze z pozycji $u_{a,1}$ w słowie u.

Nazwijmy teraz literę a pochodzącą z pozycji $u_{a,1}$ literą *wyróżnioną*. Przyjrzyjmy się strategii optymalnej pod kątem dwóch niezależnie zmieniających się stanów:

- położenia wyróżnionej litery,
- postaci *reszty* słowa u, tzn. kolejności wszystkich pozostałych liter.

Dlaczego te stany są niezależne? Zauważmy, że mamy dwa rodzaje przesunięć:

1. zmieniające położenie wyróżnionej litery (wtedy kolejność liter w reszcie słowa nie zmienia się),

2. zmieniające resztę słowa (wtedy pozycja wyróżnionej litery nie zmienia się).

Niezależność tych dwóch stanów pozwala nam zmienić kolejność wykonywania przesunięć tak, aby najpierw przeprowadzić wszystkie przesunięcia pierwszego typu, a następnie drugiego. W ten sposób otrzymamy strategię o tej samej liczbie przesunięć, a więc również optymalną. Zarazem pierwsze kroki, przemieszczające literę a z pozycji $u_{a,1}$ na początek, będą identyczne jak w naszym algorytmie zachłannym.

Powtarzając to rozumowanie dla kolejnych liter, pokażemy, że litery uznane wyżej za „odpowiadające sobie" rzeczywiście odpowiadają sobie w każdej strategii optymalnej, a zaproponowany algorytm zachłanny stanowi strategię optymalną.

Bezpośrednia implementacja powyższego algorytmu (patrz pliki `lits1.cpp` i `lits2.pas`) ma złożoność czasową $O(n^2)$. Można ją jednak usprawnić...

Rozwiązanie wzorcowe

Przyspieszenie algorytmu zachłannego wymaga od nas znalezienia struktury danych, która pozwoli efektywnie wyznaczać pierwszą niewykorzystaną pozycję danej litery w zmieniającym się słowie u, tzn. takie wystąpienie tej litery, które nie zostało jeszcze przemieszczone na początek, a następnie odcięte.

Gdyby słowo u nie ulegało zmianom, łatwo byłoby znajdować pierwsze niewykorzystane pozycje poszczególnych liter. Wystarczyłaby do tego tablica stosów *stosy*[] indeksowana literami alfabetu, która pod indeksem $a \in A$ przechowywałaby pozycje kolejnych wystąpień litery a w słowie u, od lewej do prawej. Taką tablicę stosów łatwo zbudować. Wystarczy przejść w pętli po słowie u od końca i każdą napotkaną literę wrzucać na szczyt odpowiadającego jej stosu. Późniejsze korzystanie z tej tablicy również byłoby proste — w każdym kroku algorytmu szukana pozycja litery byłaby na szczycie stosu odpowiadającego tej literze.

Niestety słowo u zmienia się w trakcie działania algorytmu, potrzebujemy więc jakiegoś mechanizmu, który aktualizowałby pozycje zawarte w tablicy *stosy*[] zgodnie ze zmianami kolejności liter. W każdym momencie możemy skupić się jedynie na niewykorzystanych dotąd literach, bo pozycji liter już przesuniętych na początek nie będziemy później badać. A jak zmienia się pozycja niewykorzystanej litery? Otóż w każdym kroku algorytmu będzie ona mniejsza od początkowej o liczbę tych liter, które na początku znajdowały się wcześniej niż rozważana i zostały już wykorzystane.

Musimy zatem zaproponować mechanizm pozwalający szybko znajdować liczbę wykorzystanych liter znajdujących się początkowo w słowie u na pozycji wcześniejszej niż zadana. Do tego służyć może struktura danych zwana *drzewem licznikowym* lub *przedziałowym*[2]. Jest to statyczne drzewo binarne zbudowane nad tablicą liczb $t[1..n]$; pozwala ono na wykonywanie w czasie $O(\log n)$ następujących operacji:

ustaw(i, x) — przypisz $t[i] := x$;

[2]Drzewa przedziałowe pojawiały się już wielokrotnie w rozwiązaniach zadań z Olimpiady Informatycznej, nie będziemy więc ich tutaj szczegółowo opisywać. Można o nich przeczytać np. w opracowaniu zadania Tetris 3D z XIII Olimpiady Informatycznej [13].

$suma(l, r)$ — zwróć sumę $t[l] + t[l + 1] + \ldots + t[r]$, przy czym $l \leqslant r$.

Mając do dyspozycji drzewo licznikowe, możemy już bardzo łatwo utrzymywać informacje o przesunięciach liter w słowie u. Wystarczy na początku, przy budowaniu drzewa, wyzerować tablicę $t[]$, a potem, przesuwając literę z jej początkowej pozycji i na początek słowa, wykonywać operację $ustaw(i, 1)$. Wtedy w dowolnym momencie działania algorytmu wiemy, że pozycja niewykorzystanej litery, która początkowo znajdowała się na pozycji i, od początku algorytmu zmalała o $suma(1, i-1)$. Algorytm wygląda więc następująco:

```
1: wynik := 0;
2: for k := 1 to n do begin
3:    a := m[k];
4:    i := stosy[a].pop();
5:    ustaw(i, 1);
6:    i := i − suma(1, i − 1);
7:    wynik := wynik + i − 1;
8: end
9: return wynik;
```

W ten sposób otrzymujemy rozwiązanie wzorcowe, zaimplementowane w plikach `lit.cpp` i `lit1.pas`.

Nieco inne spojrzenie na zadanie

Pokazaliśmy wcześniej, że dolnym ograniczeniem na wynik jest odległość słów, ale w ogólności konieczna liczba przesunięć może być większa. Istnieje jednak inna miara odległości pomiędzy słowami, którą także można łatwo wyznaczyć, bez symulowania żadnej konkretnej strategii, a która jest już równa szukanemu wynikowi. Aby ją poznać, przyjrzymy się permutacji[3] liter, jaką należy wykonać, by przekształcić słowo u w słowo w. Permutację tę nazwijmy *permutacją przeprowadzającą u na w*.

Wiemy już, że i-te wystąpienie litery `A` w słowie u odpowiada i-temu wystąpieniu litery `A` w słowie w, i-te wystąpienie litery `B` w słowie u odpowiada i-temu wystąpieniu litery `B` w słowie w itd. To pozwala skonstruować permutację przeprowadzającą u na w: otóż jeśli literą odpowiadającą literze u_i w słowie w jest w_k, to mamy $p_i = k$, co oznacza tyle co: „chcemy przestawić literę z pozycji i na pozycję k". Przykładowo, permutacją przeprowadzającą słowo $u =$ `ABAAB` na słowo $w =$ `BABAA` jest $p = (2, 1, 4, 5, 3)$.

Nie jest trudno napisać algorytm konstruujący permutację przeprowadzającą u na w przy użyciu tablicy stosów, wspomnianej w poprzednim rozdziale. Kiedy już będziemy znali tę permutację, możemy obliczyć w niej liczbę inwersji.

Definicja 2. Niech p będzie permutacją n liczb. *Inwersją* w permutacji p nazywamy dowolną parę indeksów $1 \leqslant a < b \leqslant n$, taką że $p_a > p_b$.

[3] *Permutacją* nazywamy operację zmieniającą kolejność liter zadanego słowa. Dowolną permutację n-literowego słowa możemy przedstawić w postaci ciągu $(p_i)_{i=1}^{n}$ liczb całkowitych z zakresu od 1 do n, w którym każda z tych liczb występuje dokładnie raz. Zapis ten oznacza, że permutacja p dla każdego i przemieszcza literę z pozycji i na pozycję p_i.

Innymi słowy, inwersję stanowi każda para elementów permutacji, która jest ustawiona w niewłaściwej kolejności w stosunku do permutacji identycznościowej (posortowanej rosnąco). I tak, na przykład, permutacja $p = (1, 2, \ldots, n)$ nie ma żadnych inwersji, a permutacja $p = (n, (n-1), \ldots, 1)$ ma ich $\frac{n(n-1)}{2}$.

Możemy już teraz poczynić kluczowe spostrzeżenie: otóż szukany przez nas wynik jest równy właśnie liczbie inwersji w permutacji przeprowadzającej u na w! Dlaczego tak jest? Każde przesunięcie w słowie u powoduje zarazem odpowiadające mu przesunięcie (czyli podobnie jak w przypadku słów — zamianę dwóch sąsiednich liczb) w permutacji p, i odwrotnie. Słowo u zostaje ostatecznie przekształcone w słowo w w momencie, w którym permutacja przeprowadzająca u na w jest posortowana. Ponieważ każde przesunięcie zmienia liczbę inwersji dokładnie o jeden (bo zmienia względne położenie dokładnie jednej pary sąsiednich elementów), więc przy przekształcaniu słowa u w słowo w trzeba wykonać co najmniej tyle przesunięć, ile wynosi liczba inwersji w p. Z drugiej strony, każdą permutację można posortować w następujący sposób: dopóki istnieje jakaś para sąsiednich elementów tworząca inwersję, wykonaj na nich przesunięcie. W momencie, w którym nie będzie już żadnej takiej pary, permutacja będzie posortowana[4]. Tak więc liczba inwersji jest nie tylko dolnym ograniczeniem na minimalną liczbę przesunięć przekształcających słowo u w słowo w, lecz także i górnym, czyli jest równa szukanemu wynikowi.

Jak zliczać inwersje w permutacji?

Znanych jest kilka efektywnych algorytmów pozwalających zliczać inwersje w permutacji. Najpopularniejszym rozwiązaniem w programach zawodników była pewna modyfikacja algorytmu sortowania przez scalanie (Mergesort).

Algorytm Mergesort bazuje na operacji *scalenia* (ang. *merge*) dwóch posortowanych ciągów, która łączy je w ciąg posortowany w czasie liniowym. Z jej użyciem można skonstruować następujący algorytm sortowania ($|ciąg|$ oznacza długość ciągu):

```
1: function mergesort(ciąg)
2: begin
3:   if |ciąg| = 1 then return ciąg;
4:   (ciąg1, ciąg2) := podziel_na_połowy(ciąg);
5:   return merge(mergesort(ciąg1), mergesort(ciąg2));
6: end
```

Funkcja *podziel_na_połowy* w powyższym pseudokodzie „łamie ciąg na pół", tzn. długości zwracanych przez nią ciągów różnią się co najwyżej o 1. Przy założeniu, że operacja $merge(ciąg1, ciąg2)$ działa w czasie $O(|ciąg1| + |ciąg2|)$, cały algorytm Mergesort ma złożoność czasową $O(n \log n)$, gdzie n to długość sortowanego ciągu.

Okazuje się, że można tak zmodyfikować operację scalania, żeby oprócz łączenia dwóch posortowanych ciągów w nowy posortowany ciąg, obliczała liczbę takich inwersji (a, b), że a należy do pierwszego scalanego ciągu, zaś b do drugiego. Dla każdej pary wyrazów (a, b) wyjściowej permutacji p będzie dokładnie jedno wywołanie funkcji $merge(ciąg1, ciąg2)$ dla argumentów takich, że a należy do ciągu *ciąg1*, zaś b do ciągu *ciąg2* — nastąpi to w tym wywołaniu funkcji *mergesort*, w którym a i b zostaną

[4]Mowa tu o algorytmie sortowania bąbelkowego (Bubblesort).

rozdzielone do dwóch różnych połówek sortowanego ciągu. Jeżeli więc uda nam się wzbogacić funkcję *merge* w żądany sposób, to w efekcie wyznaczy ona liczbę inwersji w oryginalnym ciągu. Oto pseudokod takiej wzbogaconej operacji scalania:

```
1:  function merge(ciąg1, ciąg2)
2:  begin
3:    k1 := |ciąg1|;   k2 := |ciąg2|;
4:    i1 := 1;   i2 := 1;
5:    while i1 ⩽ k1 or i2 ⩽ k2 do
6:      if i1 > k1 then begin
7:        ciąg[i1 + i2 − 1] := ciąg2[i2];
8:        i2 := i2 + 1;
9:      end else if i2 > k2 then begin
10:       inwersje := inwersje + i2 − 1;   { tu zliczamy inwersje! }
11:       ciąg[i1 + i2 − 1] := ciąg1[i1];
12:       i1 := i1 + 1;
13:     end else if ciąg1[i1] < ciąg2[i2] then begin
14:       inwersje := inwersje + i2 − 1;   { tu zliczamy inwersje! }
15:       ciąg[i1 + i2 − 1] := ciąg1[i1];
16:       i1 := i1 + 1;
17:     end else begin
18:       ciąg[i1 + i2 − 1] := ciąg2[i2];
19:       i2 := i2 + 1;
20:     end
21:   return ciąg;
22: end
```

Dowód poprawności powyższej funkcji pozostawiamy Czytelnikowi jako ćwiczenie. W ten sposób otrzymujemy rozwiązanie alternatywne o koszcie czasowym $O(n \log n)$.

Istnieje też inny, podobny algorytm zliczający inwersje w permutacji, który zasugerował nam Marcin Pilipczuk. Tym razem funkcja dzieląca ciąg dzieli go na pół nie według pozycji, ale według wartości, tzn. jeśli elementy dzielonego ciągu należą do przedziału $[x, y]$ (początkowo: $[1, n]$), to pierwszy ciąg powstały w wyniku podziału składa się z liczb z przedziału $[x, z]$, zaś drugi — z przedziału $[z + 1, y]$, gdzie z jest środkiem przedziału $[x, y]$. Poniżej umieszczamy funkcję rekurencyjną implementującą to podejście. Złożoność czasowa tej metody to także $O(n \log n)$.

```
1:  function inwersje(ciąg, [x, y])
2:  begin
3:    if x = y then return 0;
4:    i1 := 0;   i2 := 0;
5:    wynik := 0;   z := (x + y) div 2;
6:    for i := 1 to |ciąg| do
7:      if ciąg[i] ⩽ z then begin
8:        wynik := wynik + i2;
9:        i1 := i1 + 1;
10:       ciąg1[i1] := ciąg[i];
11:     end else begin
```

```
12:         i2 := i2 + 1;
13:         ciąg2[i2] := ciąg[i];
14:       end
15:     return wynik + inwersje(ciąg1, [x, z]) + inwersje(ciąg2, [z + 1, y]);
16: end
```

Jeszcze inne efektywne rozwiązanie problemu zliczania inwersji w permutacji można znaleźć w opisie rozwiązania zadania *Kodowanie permutacji* w książce [38].

Implementacje rozwiązań opartych na poszczególnych metodach zliczania inwersji można znaleźć w plikach `lit2.cpp` – `lit7.pas`.

Warto na koniec zauważyć, że podany we wcześniejszej sekcji algorytm wzorcowy daje zarazem kolejne rozwiązanie problemu zliczania inwersji w permutacji — również w czasie liniowo-logarytmicznym, ale za pomocą drzew przedziałowych.

Testy

Rozwiązania zawodników były sprawdzane z użyciem 10 zestawów testowych. Zestawy 6–10 wymagały reprezentowania wyniku za pomocą 64-bitowej zmiennej całkowitej.

Nazwa	**n**	*inwersje*	**Opis**
lit1.in	4	4	mały test stworzony ręcznie
lit2.in	7	16	mały test stworzony ręcznie
lit3a.in	1 000	44 599	test losowy
lit3b.in	1 000	480 766	test wymagający dużej liczby przesunięć
lit4a.in	10 000	1 499 627	test losowy
lit4b.in	10 000	48 076 920	test wymagający dużej liczby przesunięć
lit5a.in	50 000	17 536 625	test losowy
lit5b.in	50 000	1 201 923 076	test wymagający dużej liczby przesunięć
lit6a.in	100 000	47 167 845	test losowy
lit6b.in	100 000	4 807 692 306	test wymagający dużej liczby przesunięć
lit7a.in	200 000	130 542 635	test losowy
lit7b.in	200 000	19 230 769 228	test wymagający dużej liczby przesunięć
lit8a.in	500 000	580 934 901	test losowy
lit8b.in	500 000	120 192 307 690	test wymagający dużej liczby przesunięć
lit9a.in	1 000 000	1 482 550 867	test losowy
lit9b.in	1 000 000	480 769 230 766	test wymagający dużej liczby przesunięć
lit10a.in	1 000 000	1 819 136 406	test losowy
lit10b.in	1 000 000	480 769 230 766	test wymagający dużej liczby przesunięć

Tomasz Idziaszek

Wojciech Śmietanka	Wojciech Śmietanka	Zbigniew Wojna
Treść zadania	Opracowanie	Program

Dostępna pamięć: 128 MB. *OI, etap I, 17.10–14.11.2011*

Odległość

W tym zadaniu rozważamy **odległość** *między dodatnimi liczbami całkowitymi, zdefiniowaną w następujący sposób. Przez pojedynczą* **operację** *rozumiemy pomnożenie danej liczby przez liczbę pierwszą*[1] *lub podzielenie jej przez liczbę pierwszą (o ile dzieli się ona bez reszty). Odległość między liczbami a i b, oznaczana $d(a,b)$, to minimalna liczba operacji potrzebnych do przekształcenia liczby a w liczbę b. Na przykład, $d(69,42) = 3$.*

Zauważmy, że faktycznie funkcja d ma cechy odległości — dla dowolnych dodatnich liczb całkowitych a, b i c zachodzi:

- *odległość liczby od niej samej wynosi 0: $d(a,a) = 0$,*
- *odległość od a do b jest taka sama, jak od b do a: $d(a,b) = d(b,a)$, oraz*
- *spełniona jest nierówność trójkąta: $d(a,b) + d(b,c) \geqslant d(a,c)$.*

Dany jest ciąg n dodatnich liczb całkowitych $a_1, a_2, \ldots, a_n$. Dla każdej liczby a_i należy wskazać takie j, że $j \neq i$ oraz $d(a_i, a_j)$ jest minimalne.

Wejście

W pierwszym wierszu standardowego wejścia znajduje się liczba całkowita n ($2 \leqslant n \leqslant 100\,000$). W kolejnych wierszach znajdują się liczby całkowite $a_1, a_2, \ldots, a_n$ ($1 \leqslant a_i \leqslant 1\,000\,000$), po jednej w wierszu.

W testach wartych łącznie 30% punktów zachodzi dodatkowy warunek $n \leqslant 1\,000$.

Wyjście

Twój program powinien wypisać na standardowe wyjście dokładnie n wierszy, a w każdym z nich po jednej liczbie całkowitej. W i-tym wierszu należy wypisać **najmniejsze** *takie j, że: $1 \leqslant j \leqslant n$, $j \neq i$ oraz $d(a_i, a_j)$ jest minimalne.*

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
6	2
1	1
2	1
3	2
4	1
5	2
6	

[1] Przypomnijmy, że liczba pierwsza to taka liczba całkowita dodatnia, która ma dokładnie dwa różne dzielniki: jedynkę i siebie samą.

Rozwiązanie

Analiza problemu

Zacznijmy od przejścia na język teorii grafów. Rozważmy nieskończony graf nieskierowany, w którym wierzchołkami są wszystkie liczby naturalne, a krawędzie odpowiadają pomnożeniu (równoważnie, podzieleniu) danej liczby przez liczbę pierwszą. Problem z zadania formułuje się teraz następująco: dla każdego z n zaznaczonych wierzchołków chcemy znaleźć w grafie najbliższy inny zaznaczony wierzchołek.

Wygodnie będzie, jeśli na wstępie wyeliminujemy z podanego na wejściu ciągu liczb wszystkie powtórzenia. Faktycznie, dla każdej z powtarzających się liczb, jako wynik możemy od razu wskazać dowolne inne wystąpienie takiej samej liczby w ciągu. Odtąd będziemy zakładać, że nasz ciąg nie zawiera powtórzeń (wystarczy pozostawić po jednym egzemplarzu każdej wartości z ciągu).

Jak duży jest nasz graf?

Na początek musimy poradzić sobie z faktem, że nasz graf jest nieskończony. Pokażemy, że w istocie interesować nas będzie jedynie skończony fragment tego grafu.

Rozważmy najkrótszy ciąg mnożeń i dzieleń przez liczby pierwsze, który przekształca liczbę a w liczbę b. Zauważmy, że jeśli w tym ciągu wykonalibyśmy zarówno operację pomnożenia przez p, jak i operację podzielenia przez p, to, usuwając te operacje, uzyskalibyśmy krótszy ciąg, który również przekształca a w b, co byłoby sprzeczne z założeniem, że nasz ciąg jest najkrótszy. Możemy zatem założyć, że dla każdej liczby pierwszej p wykonujemy albo operacje mnożenia przez p, albo dzielenia przez p. W tym drugim przypadku widzimy, że liczba a musi być podzielna przez odpowiednią potęgę p. Widać więc, że kolejność wykonywania operacji nie ma znaczenia, zatem wszystkie operacje dzielenia mogą zostać wykonane przed mnożeniami.

To prowadzi nas do wniosku, że pewna najkrótsza ścieżka pomiędzy a i b w grafie nie zawiera wierzchołków większych niż $\max(a, b)$. Jeśli zatem oznaczymy przez M największą liczbę z wejścia, to wystarczą nam wierzchołki dla liczb od 1 do M.

Oszacujemy teraz liczbę krawędzi w naszym grafie. Wydaje się, że może ich być dużo. Najprostsze oszacowanie daje $M \cdot \pi(M)$ krawędzi, gdzie $\pi(M)$ jest liczbą liczb pierwszych mniejszych od M. Zauważmy jednak, że jeśli dla pewnego a mamy k krawędzi reprezentujących podzielenie a przez liczbę pierwszą, to a musi mieć co najmniej k różnych dzielników pierwszych. Przy założeniu, że $M \leqslant 10^6$, mamy $k \leqslant 7$, gdyż rozważając iloczyn ośmiu najmniejszych liczb pierwszych, dostajemy

$$2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 = 9\,699\,690 > 10^6.$$

Czyli wszystkich krawędzi jest co najwyżej $7 \cdot 10^6$, gdyż każda z krawędzi w grafie reprezentuje, w szczególności, podzielenie liczby przez jakąś liczbę pierwszą.

Można również zauważyć, że liczba pierwsza p będzie generowała krawędzie dla co najwyżej $\lfloor M/p \rfloor$ wierzchołków, gdyż każda taka krawędź łączy wierzchołek

$a \in \{1, \ldots, \lfloor M/p \rfloor\}$ z wierzchołkiem $a \cdot p$. Wiedząc, że

$$\sum_{p \leqslant M,\, p \in \mathbb{P}} \frac{M}{p} \approx M \ln \ln M,$$

wnioskujemy, że liczba krawędzi grafu jest rzędu $O(M \ln \ln M)$. To dostatecznie mało, aby można było przejrzeć cały graf.

Wyznaczanie najbliższych wierzchołków w grafie

Gdy już wiemy, że możemy pozwolić sobie na przejrzenie całego grafu, to w dalszych rozważaniach szczególna postać naszego grafu będzie nieistotna. Okazuje się bowiem, że istnieje efektywny algorytm, który wyznacza najbliższe wierzchołki w dowolnym grafie nieskierowanym.

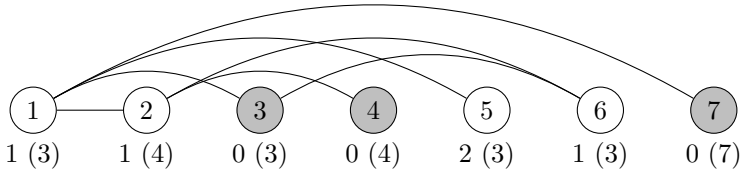
Na początek ustalmy oznaczenia. Mamy dany graf $G = (V, E)$ oraz pewien podzbiór wierzchołków $S \subseteq V$. Dla każdego wierzchołka $v \in S$ chcemy znaleźć inny wierzchołek $w \in S$, taki że odległość między v i w jest jak najmniejsza. Jeśli istnieje więcej niż jeden kandydat na w, wybieramy tego o mniejszym numerze.

Dla ustalonego podzbioru $S \subseteq V$ oraz wierzchołka $v \in V$ oznaczmy przez $d_S[v]$ odległość z v do najbliższego wierzchołka z S (jeśli $v \in S$, to oczywiście $d_S[v] = 0$). Przez $m_S[v]$ oznaczmy taki wierzchołek z S, który realizuje tę odległość. Jeśli istnieje więcej niż jeden kandydat na $m_S[v]$, wybieramy tego o mniejszym numerze.

Najpierw pokażemy, jak obliczyć $d_S[v]$ i $m_S[v]$ dla wszystkich $v \in V$. Można to zrobić, przeszukując graf wszerz (BFS), zaczynając jednocześnie od wszystkich wierzchołków z S. W kolejce Q będziemy trzymać wierzchołki do przetworzenia. Utrzymujemy niezmiennik, że każdy wierzchołek v wrzucany do kolejki ma poprawnie obliczoną wartość $d_S[v]$. Ponadto $m_S[v]$ jest poprawnie obliczone po przetworzeniu wszystkich wierzchołków i, dla których $d_S[i] = d_S[v] - 1$.

```
 1: foreach v ∈ V do d_S[v] := ∞;
 2: foreach v ∈ S do begin
 3:    d_S[v] := 0;
 4:    m_S[v] := v;
 5:    Q.push(v);
 6: end
 7: while not Q.empty do begin
 8:    v := Q.pop();
 9:    foreach (v, w) ∈ E do
10:       if d_S[w] = ∞ then begin
11:          d_S[w] := d_S[v] + 1;
12:          m_S[w] := m_S[v];
13:          Q.push(w);
14:       end
15:       else if d_S[v] + 1 = d_S[w] and m_S[v] < m_S[w] then
16:          m_S[w] := m_S[v];
17: end
```

Przykład 1. Poniższy rysunek przedstawia graf zbudowany dla $M = 7$ i zbioru $S = \{3, 4, 7\}$. Liczby pod wierzchołkiem v oznaczają wartości $d_S[v]$ i $m_S[v]$.



Naszym celem jest obliczenie odległości z każdego wierzchołka $v \in S$ do najbliższego wierzchołka ze zbioru $S \setminus \{v\}$. Oznaczmy tę odległość przez $d_{S\setminus\{v\}}[v]$, a wierzchołek, który ją realizuje, przez $m_{S\setminus\{v\}}[v]$.

Ustalmy $v \in S$ i oznaczmy $w = m_{S\setminus\{v\}}[v]$. Niech $v = v_1, v_2, \ldots, v_k = w$ będzie najkrótszą ścieżką z v do w. Niech i będzie najmniejszą liczbą, taką że $m_S[v_{i-1}] = v$ oraz $m_S[v_i] \neq v$ (taka liczba oczywiście istnieje, bo $m_S[v_1] = v$ i $m_S[v_k] \neq v$). Zauważmy, że najkrótsza odległość z v_i do zbioru S jest realizowana przez pewien wierzchołek z $S \setminus \{v\}$, zatem $m_S[v_i] = m_{S\setminus\{v\}}[v_i]$, czyli $m_S[v_i] = w$ wobec wyboru wierzchołka w.

Oznacza to, że w grafie istnieje taka krawędź $(v', w') \in E$, dla której $m_S[v'] = v$, $m_S[w'] = w$ oraz w jest najbliższym do v wierzchołkiem z $S \setminus \{v\}$, odległym od niego o $d_S[v'] + 1 + d_S[w']$. Nie wiemy, która to krawędź, więc sprawdzimy wszystkie:

```
1:  foreach v ∈ S do
2:     d_{S\{v}}[v] := ∞;
3:  foreach (v', w') ∈ E do begin
4:     v := m_S[v'];
5:     w := m_S[w'];
6:     if v ≠ w then begin
7:        d := d_S[v'] + 1 + d_S[w'];
8:        if d < d_{S\{v}}[v] or (d = d_{S\{v}}[v] and w < m_{S\{v}}[v]) then begin
9:           d_{S\{v}}[v] := d;
10:          m_{S\{v}}[v] := w;
11:       end
12:    end
13: end
```

Przykład 2. W grafie z przykładu 1 dla $v = 4$ mamy $w = 3$ oraz $d_{S\setminus\{v\}}[v] = 3$. Wówczas $v' = 2$ i $w' = 1$ lub $v' = 2$ i $w' = 6$.

Obie fazy algorytmu działają w czasie $O(|V| + |E|)$. Jako ćwiczenie dla Czytelnika proponujemy modyfikację tego algorytmu, aby działał również dla grafów z wagami na krawędziach (należy wtedy użyć algorytmu Dijkstry zamiast BFS).

Złożoność

Złożoność czasowa podanego dwufazowego przeszukiwania grafu jest liniowa względem rozmiaru grafu, czyli rzędu $O(M \ln \ln M)$. Warto zauważyć, że nie trzeba trzymać w pamięci całego grafu G. Wystarczy dla każdej liczby mieć obliczony pewien jej dzielnik pierwszy (do generowania krawędzi odpowiadających dzieleniu) i osobno pamiętać listę wszystkich liczb pierwszych (do generowania krawędzi odpowiadających mnożeniu). Zarówno listę wszystkich liczb pierwszych nieprzekraczających M, jak i przykładowe dzielniki pierwsze poszczególnych liczb od 2 do M można obliczyć w czasie $O(M \ln \ln M)$ za pomocą sita Eratostenesa[1]. Dodajmy dla pełności, że początkowe usuwanie powtórzeń z wyjściowego ciągu możemy wykonać w czasie i pamięci $O(M)$, wykorzystując M-elementową tablicę kubełków (list). Ostatecznie, całe rozwiązanie ma złożoność czasową $O(M \ln \ln M)$, a pamięciową $O(M)$.

Implementacje rozwiązania wzorcowego można znaleźć w plikach `odl.cpp` i `odl1.pas`.

Testy

Przygotowanych zostało 10 zestawów testowych. Jeśli dana grupa składa się z więcej niż jednego testu, to pierwszy test jest testem poprawnościowym, co może np. oznaczać, że odległości do najbliższych liczb są większe aniżeli w losowym teście. W tych testach parametr n jest zazwyczaj stosunkowo niewielki.

Nazwa	n	M	Opis
odl1.in	50	95	test losowy
odl2.in	345	698	potęgi liczb pierwszych
odl3a.in	584	3 499	trochę maksymalnych potęg liczb pierwszych, trochę losowy
odl3b.in	1 000	3 500	losowy test wydajnościowy
odl4a.in	11 884	80 649	liczby o nieparzystej i większej niż 4 sumie wykładników w rozkładzie
odl4b.in	21 234	47 288	losowy test wydajnościowy
odl5a.in	13 459	150 000	kilka liczb, od których jest daleko do najbliższego elementu, reszta losowa
odl5b.in	50 001	150 001	losowy test wydajnościowy
odl6a.in	34 209	399 989	mniejsze liczby: losowe, większe liczby: maksymalne potęgi liczb pierwszych
odl6b.in	73 900	234 564	losowy test wydajnościowy

[1]Przykład takiego wykorzystania sita Eratostenesa można znaleźć w opracowaniu zadania *Zapytania* z XIV Olimpiady Informatycznej [14].

Nazwa	n	M	Opis
odl7a.in	36 118	756 432	dużo liczb, od których odległość do najbliższej to 2, trochę mniej tych, od których odległość to 3
odl7b.in	86 023	803 819	losowy test wydajnościowy
odl8a.in	63 759	999 999	dużo liczb o odległości co najmniej 2, nieliczne o dużo większych odległościach do im najbliższych
odl8b.in	98 837	853 983	losowy test wydajnościowy
odl9a.in	1 482	999 810	losowe odległości między 1 a 5
odl9b.in	99 048	1 000 000	losowy test wydajnościowy
odl10a.in	1 176	1 000 000	potęgi liczb naturalnych o wykładnikach większych niż 1
odl10b.in	100 000	999 996	losowy test wydajnościowy
odl10c.in	100 000	999 999	maksymalne wejście

Alan Kutniewski
Treść zadania, Opracowanie

Zbigniew Wojna
Program

Dostępna pamięć: 128 MB.

OI, etap I, 17.10–14.11.2011

Randka

Bajtazar jest strażnikiem przyrody i pracuje w Jaskini Strzałkowej — znanym miejscu schadzek zakochanych par. Jaskinia ta składa się z n komór połączonych jednokierunkowymi korytarzami. W każdej komorze dokładnie jeden wychodzący z niej korytarz jest oznaczony strzałką. Każdy korytarz prowadzi bezpośrednio z jednej komory do pewnej (niekoniecznie innej) komory.

Notorycznie zdarza się, że zakochane pary, które umawiają się na randki w Jaskini Strzałkowej, zapominają dokładnie ustalić miejsce spotkania i nie mogą się odnaleźć. W przeszłości prowadziło to do wielu nieporozumień i pomyłek... Od czasu, gdy w każdej komorze zainstalowano telefon alarmowy łączący z dyżurnym strażnikiem przyrody, głównym zajęciem strażników stało się pomaganie zakochanym parom w odnajdowaniu się.

Strażnicy wypracowali następującą metodę. Wiedząc, w których komorach znajdują się zakochani, mówią każdemu z nich, ile razy, odpowiednio, powinien przejść z komory do komory korytarzem oznaczonym strzałką, aby oboje mogli się spotkać. Przy tym, zakochani bardzo chcą spotkać się jak najszybciej — zależy im przecież, aby razem miło spędzać czas, a nie samotnie przemierzać korytarze. Strażnicy starają się podawać zakochanym parom takie liczby, aby ich maksima były możliwie jak najmniejsze.

Bajtazar jest już zmęczony ciągłym pomaganiem zakochanym i poprosił Cię o napisanie programu, który by to usprawnił. Program ten, na podstawie opisu Jaskini Strzałkowej oraz aktualnego położenia k zakochanych par, powinien wyznaczyć k par liczb x_i i y_i, takich że:

- *jeżeli i-ta para zakochanych przejdzie odpowiednio: on x_i, a ona y_i korytarzami oznaczonymi strzałkami, to spotkają się w jednej komorze jaskini,*
- *$max(x_i, y_i)$ jest jak najmniejsze,*
- *w drugiej kolejności $min(x_i, y_i)$ jest jak najmniejsze,*
- *jeżeli rozwiązanie wciąż nie jest jednoznaczne, to kobieta powinna pokonywać mniejszy dystans.*

Może się tak zdarzyć, że takie liczby x_i i y_i nie istnieją — wówczas przyjmujemy, że $x_i = y_i = -1$.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie dodatnie liczby całkowite n i k ($1 \leqslant n \leqslant 500\,000$, $1 \leqslant k \leqslant 500\,000$), oddzielone pojedynczym odstępem i określające liczbę komór w Jaskini Strzałkowej i liczbę zakochanych par, które chcą się odnaleźć. Komory są ponumerowane od 1 do n, natomiast zakochane pary są ponumerowane od 1 do k.

W drugim wierszu wejścia znajduje się n liczb całkowitych dodatnich, pooddzielanych pojedynczymi odstępami: i-ta liczba w tym wierszu określa numer komory, do której prowadzi korytarz oznaczony strzałką wychodzący z komory numer i.

W kolejnych k wierszach znajdują się kolejne zapytania zakochanych par. Każde zapytanie składa się z dwóch liczb całkowitych dodatnich oddzielonych pojedynczym odstępem — oznaczają one numery komór, w których znajduje się dana para zakochanych — najpierw on, a potem ona.

W testach wartych łącznie 40% punktów zachodzą dodatkowe warunki $n \leqslant 2\,000$ *oraz* $k \leqslant 2\,000$*.*

Wyjście

Twój program powinien wypisać na standardowe wyjście dokładnie k wierszy, po jednym dla każdej pary zakochanych z wejścia: i-ty wiersz wyjścia powinien opisywać wynik dla i-tej pary z wejścia. Każdy wiersz wyjścia powinien składać się z dwóch liczb całkowitych x_i, y_i, *oddzielonych pojedynczym odstępem.*

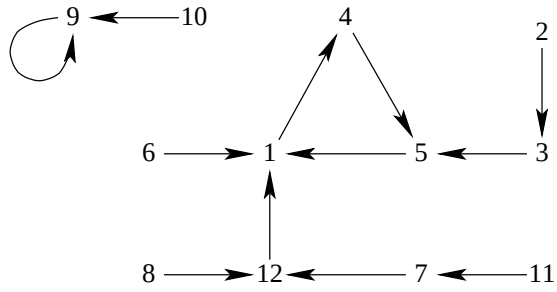
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
2 3
1 2
2 2
0 1
-1 -1
```



Rozwiązanie

Analiza problemu

Na początku zastanówmy się, jaką postać ma graf opisany w tym zadaniu. Jest to graf skierowany, w którym z każdego wierzchołka wychodzi dokładnie jedna krawędź. Nasz graf składa się zatem z pewnej liczby cykli, do których mogą być dołączone drzewa[1].

Powiemy, że dwa wierzchołki należą do tej samej *słabo spójnej składowej* grafu, jeśli można przejść z jednego z nich do drugiego po strzałkach, ignorując ich skierowanie (tzn. możemy iść w przód lub w tył). W każdej składowej znajduje się dokładnie

[1]Ciekawe własności takich grafów były niejednokrotnie wykorzystywane w rozwiązaniach zadań olimpijskich, np. w zadaniu *Mafia* z XV Olimpiady Informatycznej [15] i w zadaniu *Szpiedzy* z XI Olimpiady Informatycznej [11].

jeden cykl. W dalszym opisie przyjmiemy, że każdy wierzchołek cyklu stanowi korzeń pewnego drzewa dołączonego do cyklu — w niektórych przypadkach drzewo to składa się tylko z tego wierzchołka.

Następnie mamy daną listę par wierzchołków w grafie. Dla każdej pary wierzchołków u, v musimy znaleźć taką *optymalną* parę liczb x, y, że po przejściu x kroków z wierzchołka u i y kroków z wierzchołka v dojdziemy do tego samego wierzchołka. Szukana optymalna para x, y powinna przede wszystkim minimalizować $\max(x, y)$, w drugiej kolejności — $\min(x, y)$, a w trzeciej kolejności — liczbę y.

Pomyślmy teraz, gdzie może leżeć wierzchołek, w którym nastąpi spotkanie. Możliwe są trzy przypadki, w zależności od wzajemnego położenia u i v:

1. Wierzchołki leżą na jednym drzewie i na nim też nastąpi spotkanie.
2. Wierzchołki leżą na różnych drzewach, których korzenie znajdują się na jednym cyklu. Wówczas spotkanie nastąpi na tym cyklu.
3. Wierzchołki leżą w różnych składowych grafu, zatem spotkanie jest niemożliwe.

Zauważmy, że łatwo stwierdzić, czy mamy do czynienia z przypadkiem 3, jeśli dla każdego wierzchołka znamy numer składowej grafu, w której się ten wierzchołek znajduje. Takie numery dla wszystkich wierzchołków można wyznaczyć w czasie $O(n)$, przeszukując graf i ignorując skierowanie krawędzi (opisujemy to także w sekcji „Rozwiązanie wzorcowe"). W dalszej części opisu będziemy zatem zakładać, że mamy do czynienia z przypadkiem 1 lub 2.

Rozwiązanie siłowe

Na podstawie powyższych obserwacji możemy skonstruować pierwsze rozwiązanie. Zaznaczamy wszystkie wierzchołki, do których można dojść z wierzchołka u. Następnie, startując z wierzchołka v, idziemy po strzałkach, aż dojdziemy do jednego z wierzchołków, które zaznaczyliśmy. Później robimy to samo, odwracając role. Jako wynik podajemy lepsze z dwóch znalezionych miejsc spotkania (używając zadanego kryterium porównywania wyników).

Dlaczego otrzymany w ten sposób wynik jest optymalny? Wróćmy do określonych wcześniej przypadków.

W pierwszym przypadku, wykonując powyższy algorytm, znajdziemy najbliższego wspólnego przodka wierzchołków u i v w drzewie. Jedynymi innymi potencjalnymi miejscami spotkań są wierzchołki bliższe korzenia lub wierzchołki leżące na cyklu. Jednak żeby do nich dojść, trzeba zwiększyć zarówno x, jak i y, co z oczywistych względów nie da nam lepszego wyniku.

Drugi przypadek jest trochę bardziej skomplikowany. Potencjalne miejsca spotkań leżą na cyklu. Jeśli więc któryś wierzchołek z pary nie leży na cyklu, to musimy przejść z niego w górę drzewa, aż dojdziemy do cyklu. Kiedy już oba wierzchołki znajdą się na cyklu, mamy dwie możliwości: albo przejdziemy po cyklu z pierwszego wierzchołka do drugiego, albo na odwrót. Tym dwóm przypadkom odpowiadają dwa przebiegi naszego algorytmu. Zatem bierzemy pod uwagę dwa potencjalne miejsca spotkań: pierwszy wierzchołek leżący na cyklu należący do ścieżki wychodzącej z u oraz analogiczny wierzchołek należący do ścieżki wychodzącej z v. Każdy inny wierzchołek

cyklu może być tylko gorszym kandydatem na miejsce spotkania, ponieważ ścieżki z u oraz z v do dowolnego niewybranego przez nas wierzchołka na cyklu prowadzą, odpowiednio, przez wybrane przez nas miejsca.

Złożoność czasowa tego rozwiązania to $O(n \cdot k)$. Przykładowe implementacje można znaleźć plikach `rans1.cpp` i `rans2.pas`.

Wnioski z rozwiązania siłowego

Konstruując rozwiązanie siłowe, poczyniliśmy bardzo ważne spostrzeżenia.

1. Jeśli wierzchołki u, v leżą na jednym drzewie, to wynikiem jest ich najniższy wspólny przodek (tzw. *LCA*, ang. *lowest common ancestor*).

2. W przeciwnym razie jedynymi kandydatami na optymalne miejsce spotkania są pierwsze wierzchołki na ścieżkach wychodzących z u i v, które leżą na cyklu.

Rozwiązanie wzorcowe

Optymalne miejsca spotkań zależą od przypadku, z którym mamy do czynienia. Musimy więc umieć szybko go określać. Dla każdego wierzchołka chcemy znaleźć cykl, do którego można z niego dojść, oraz zidentyfikować drzewo, w którym się znajduje. Aby wyznaczyć te wartości dla wszystkich wierzchołków, będziemy w grafie naprzemiennie wyszukiwali cykle i dołączone do nich drzewa.

W pierwszym kroku chcemy znaleźć jakiś cykl. W tym celu wybieramy dowolny wierzchołek w i idziemy po strzałkach. Musimy w końcu dojść do odwiedzonego wcześniej wierzchołka. To będzie oznaczało, że znaleźliśmy cykl; teraz wystarczy przejść po nim, zaznaczając go. Można to zrobić przy pomocy algorytmu przedstawionego poniżej (początkowo dla każdego wierzchołka i mamy $odwiedzony[i] =$ **false**, $cykl[i] = -1$).

```
1:  while not odwiedzony[w] do begin
2:      odwiedzony[w] := true;
3:      w := następny[w];
4:  end
5:  { W tym miejscu w leży na cyklu }
6:  ostatni := w;
7:  do
8:      cykl[w] := nr_cyklu;
9:      w := następny[w];
10: while w ≠ ostatni;
```

W drugim kroku zaznaczamy drzewa, które wchodzą do cyklu znalezionego w pierwszym kroku. Przechodzimy po wszystkich wierzchołkach należących do cyklu. Z każdego z nich idziemy w przeciwną stronę do tej, którą pokazują strzałki, do wierzchołków, które nie mają jeszcze przypisanego cyklu. Innymi słowy, przeszukujemy wszerz (algorytm BFS) graf transponowany. Każdemu wierzchołkowi zapisujemy

numer wierzchołka cyklu, z którego przyszliśmy do tego wierzchołka, jako numer jego drzewa. Wierzchołki leżące na cyklu są korzeniami tych drzew.

Po wykonaniu tych dwóch kroków mamy oznaczony jeden cykl oraz wszystkie jego drzewa. Musimy powtarzać ten algorytm, dopóki istnieją nieoznaczone wierzchołki, ignorując wierzchołki, które już zostały oznaczone.

Wszystkie wstępne obliczenia działają w złożoności $O(n)$. Obliczone za ich pomocą informacje pozwalają łatwo określać, dla każdego zapytania, z którym przypadkiem mamy do czynienia. Odtąd skupimy się na rozwiązywaniu obu przypadków osobno.

Pierwszy przypadek: LCA

Zakładamy, że u i v leżą na jednym drzewie, i szukamy ich najniższego wspólnego przodka (czyli wspomnianego wcześniej LCA). To już jest standardowy problem. Algorytm znajdujący LCA w czasie $O(\log n)$ (po wstępnych obliczeniach o złożoności czasowej i pamięciowej $O(n \log n)$) można znaleźć, na przykład, w opisie rozwiązania zadania *Komiwojażer Bajtazar* z IX Olimpiady Informatycznej [9].

Drugi przypadek: wynik leży na cyklu

W tym przypadku mamy dwóch kandydatów na optymalne miejsce spotkania. Musi nim być korzeń drzewa, na którym leży jeden z danych wierzchołków u, v.

Przede wszystkim, obydwa wierzchołki muszą znać swoją odległość od korzenia. Takie wartości możemy łatwo zapamiętywać podczas wstępnych obliczeń (algorytm BFS).

Teraz musimy jeszcze obliczyć odległość na cyklu z pierwszego korzenia do drugiego, a także z drugiego do pierwszego. Żeby to zrobić, zapamiętujemy długość cyklu oraz numerujemy kolejno jego wierzchołki. Odległość między dwoma korzeniami na cyklu to różnica ich numerów, gdy jest dodatnia, lub długość cyklu pomniejszona o tę różnicę w przeciwnym przypadku.

Podsumowanie

W celu określenia przypadku wykonujemy wstępne obliczenia działające w czasie $O(n)$. Potrzebujemy także czasu $O(n \log n)$ na wstępne obliczenia związane z wyznaczaniem LCA par wierzchołków. Następnie k razy znajdujemy optymalne miejsce spotkania. Obliczanie LCA działa w czasie $O(\log n)$, a obliczanie wyniku na cyklu — w czasie $O(1)$.

Cały algorytm działa więc w czasie $O((n + k) \log n)$ i pamięci $O(n \log n)$. Implementacje rozwiązania wzorcowego można znaleźć w plikach `ran.cpp`, `ran2.pas` i `ran3.cpp`. Natomiast w plikach `rans3.cpp`, `rans4.pas`, `rans5.cpp` i `rans6.pas` znajdują się implementacje rozwiązań wolniejszych, w których poszczególne fazy rozwiązania wzorcowego (obliczanie LCA, obsługa przypadku na cyklu) działają w czasie liniowym.

Testy

Rozwiązania zawodników sprawdzano za pomocą 10 zestawów testowych. Testy 1 i 2 to proste testy generowane ręcznie. Każdy z zestawów 3–10 składał się z trzech testów następujących typów:

- test *a*: jedna bardzo długa gałąź z losowymi zaburzeniami. Na takich testach wolno działają te rozwiązania, w których wyznaczanie LCA jest liniowe.
- test *b*: jeden bardzo duży cykl z losowymi zaburzeniami. Na takich testach wolno działają te rozwiązania, w których wyznaczanie najkrótszej ścieżki na cyklu jest liniowe.
- test *c*: test zawierający różne trudne przypadki. Mały graf i kilka zapytań są wszędzie takie same, reszta wierzchołków i zapytań jest wybrana losowo.

Poniżej znajduje się tabela zawierająca parametry poszczególnych testów.

Nazwa	**n**	**k**
ran1.in	3	5
ran2.in	9	4
ran3abc.in	1 000	1 000
ran4abc.in	2 000	2 000
ran5abc.in	50 000	50 000
ran6abc.in	100 000	100 000
ran7abc.in	200 000	200 000
ran8abc.in	500 000	250 000
ran9abc.in	500 000	500 000
ran10abc.in	500 000	500 000

Michał Włodarczyk
Treść zadania, Opracowanie

Igor Adamski
Program

Dostępna pamięć: 64 MB.

OI, etap I, 17.10–14.11.2011

Studnia

Bajtazar wybrał się na wyprawę wzdłuż Suchej Rzeki, która przecina Pustynię Bajtocką. Niestety Sucha Rzeka wyschła, a Bajtazarowi skończyła się woda. Jedynym ratunkiem dla Bajtazara jest wykopanie studni na dnie wyschniętego koryta rzeki i dokopanie się do wody.

Bajtazar postanowił dobrze przemyśleć, co ma zrobić, zanim weźmie się za kopanie — wie, że jeśli opadnie z sił, a nie dokopie się do wody, to będzie miał skrajnie małe szanse na przetrwanie. Udało mu się określić, na jakiej głębokości pod dnem rzeki zalega woda. Wie też, na ile kopania starczy mu sił. Boi się tylko, żeby w czasie kopania nie osunęła się ziemia, gdyż może go pogrzebać żywcem. Bajtazar przesłał Ci (przez telefon satelitarny) opis topografii koryta rzeki. Poprosił Cię o wyznaczenie planu, gdzie ma kopać, tak aby dokopać się do wody, zanim opadnie z sił, a równocześnie, żeby zbocza w wykopie były jak najłagodniejsze. Bajtazar czeka na Twoją pomoc!

Wejście

W pierwszym wierszu standardowego wejścia są zapisane dwie dodatnie liczby całkowite n oraz m ($1 \leqslant n \leqslant 1\,000\,000$, $1 \leqslant m \leqslant 10^{18}$), oddzielone pojedynczym odstępem. W drugim wierszu znajduje się n dodatnich liczb całkowitych $x_1, x_2, \ldots, x_n$ ($1 \leqslant x_i \leqslant 10^9$), pooddzielanych pojedynczymi odstępami.

Bajtazarowi zostało sił na m ruchów łopatą. Liczby $x_1, x_2, \ldots, x_n$ stanowią opis topografii koryta Suchej Rzeki, zdatnego do kopania studni. Liczby te reprezentują grubość warstwy piasku ponad poziomem wody gruntowej, w kolejnych miejscach, co metr wzdłuż koryta rzeki. Jednym ruchem łopaty Bajtazar może wybrać tyle piachu, aby jedną z liczb x_i zmniejszyć o 1. Jeżeli którakolwiek z liczb x_i, powiedzmy x_k, zmniejszy się do 0, będzie to oznaczać, że Bajtazar dokopał się do wody. Poza dokopaniem się do wody w co najmniej jednym punkcie koryta rzeki, Bajtazarowi zależy na tym, aby na końcu następująca liczba z, charakteryzująca nachylenie piaszczystych zboczy:

$$z = \max_{i=1,2,\ldots,n-1} |x_i - x_{i+1}|,$$

była jak najmniejsza. Jeżeli istnieje wiele poprawnych wartości liczby k, reprezentującej miejsce, w którym Bajtazar powinien dokopać się do poziomu wody, Twój program powinien wypisać dowolną z nich. Możesz przyjąć, że poza miejscami $1, 2, \ldots, n$ na wszystkich głębokościach znajduje się lita skała oraz że Bajtazar zawsze będzie miał wystarczająco dużo siły, żeby w którymś miejscu dokopać się do wody.

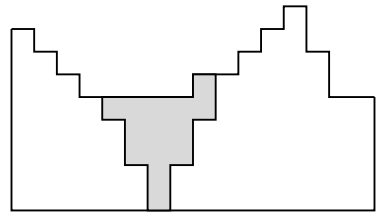
W testach wartych co najmniej 35% punktów zachodzi dodatkowy warunek $n \leqslant 10\,000$.

Wyjście

Twój program powinien wypisać na standardowe wyjście dwie liczby całkowite oddzielone pojedynczym odstępem: miejsce k, w którym Bajtazar powinien dokopać się do wody, oraz najmniejszą możliwą wartość liczby z.

Przykład

Dla danych wejściowych:
```
16 15
8 7 6 5 5 5 5 6 6 7 8 9 7 5 5
```
poprawnym wynikiem jest:
```
7 2
```



Na powyższym rysunku prawidłowy wykop Bajtazara oznaczono szarym kolorem.

Rozwiązanie

Pierwsze podejście

Rozważania rozpoczniemy od skonstruowania najprostszego rozwiązania, które będziemy stopniowo ulepszać, aż dojdziemy do efektywnego algorytmu.

Na początku ustalmy k i z. Chcemy sprawdzić, czy można dokopać się do wody w punkcie k, tak aby nachylenie zbocza nie przekroczyło z. Dla tych parametrów, przez (y_i) oznaczmy ciąg opisujący topografię koryta rzeki po dokopaniu się do wody przy minimalnej liczbie ruchów łopatą (niedługo okaże się, że jest on wyznaczony jednoznacznie). Chcemy stwierdzić, czy:

$$\sum_{i=1}^{n}(x_i - y_i) \leqslant m.$$

Po pierwsze, wiemy, że $y_k = 0$, bo w tym miejscu dokopaliśmy się do wody. Dla $i \neq k$ pozostawmy $y_i = x_i$ i postarajmy się teraz wygładzić teren (tzn. usunąć część piasku tak, aby nachylenie nie przekraczało z). Wyobraźmy sobie, że idziemy w prawo i kiedy natrafiamy na próg o wysokości większej od z, pomniejszamy go. Następnie powtarzamy tę czynność, idąc w lewo. Poniżej przedstawiamy pseudokod tej procedury i dowód poprawności (ciągowi (y_i) odpowiada w kodzie tablica $y[]$):

```
1: for i := 1 to n − 1 do
2:     if y[i + 1] > y[i] + z then
3:         y[i + 1] := y[i] + z;
4: for i := n downto 2 do
5:     if y[i − 1] > y[i] + z then
6:         y[i − 1] := y[i] + z;
```

Lemat 1. Dla ciągu (y_i) po wykonaniu powyższej procedury zachodzi:

$$|y_i - y_{i+1}| \leqslant z \qquad \text{dla } i = 1, \ldots, n-1 \tag{1}$$

i każdy inny ciąg (v_i) o tej własności, $0 \leqslant v_i \leqslant x_i$, $v_k = 0$, spełnia:

$$v_i \leqslant y_i \qquad \text{dla } i = 1, \ldots, n.$$

Dowód: Załóżmy, że po wykonaniu podanego algorytmu nierówność (1) nie jest spełniona, tzn. dla pewnego j zachodzi $y_j > y_{j+1}+z$ lub $y_{j+1} > y_j+z$. Pierwsza możliwość jest wykluczona, ponieważ w drugiej części procedury dla $i = j + 1$ poprawiliśmy y_j, tak aby zachodziło $y_j \leqslant y_{j+1} + z$, a później żadna z tych liczb nie była zmieniana. W drugim przypadku natomiast wiemy, że po przejściu w prawo było $y_{j+1} \leqslant y_j + z$. Idąc w lewo, nie mogliśmy zwiększyć y_{j+1}. Jeśli zaś y_j zostało zmienione, to spełnia równość $y_j = y_{j+1} + z$, co jest sprzeczne z założeniem.

Drugą część tezy dowiedziemy indukcyjnie względem długości ciągów. Dokładniej, pokażemy, że dla dowolnego ciągu (x'_i) (odpowiadającego ciągowi (x_i) z ustawionym $x_k = 0$) i ciągów (y_i) i (v_i), ograniczonych z góry przez ciąg x'_i i spełniających nierówność typu (1), przy czym ciąg y_i jest skonstruowany za pomocą podanego wyżej algorytmu, dla każdego i zachodzi $v_i \leqslant y_i$. Baza indukcji (dla ciągów długości 1) jest trywialna. Przypuśćmy, że teza zachodzi dla wszystkich ciągów długości $n - 1$. Pokażemy, że stąd wynika teza dla ciągów długości n.

Zauważmy, że $v_2, y_2 \leqslant \min(x'_2, x'_1 + z)$. Co więcej, gdybyśmy w podanym wyżej algorytmie wystartowali od ciągu $\min(x'_2, x'_1 + z), x'_3, \ldots, x'_n$, skonstruowalibyśmy właśnie ciąg $y_2, y_3, \ldots, y_n$. Stosując w tym miejscu założenie indukcyjne, dostajemy $v_i \leqslant y_i$ dla $i > 1$. Wreszcie

$$v_1 \leqslant \min(x'_1, v_2 + z) \leqslant \min(x'_1, y_2 + z) = y_1,$$

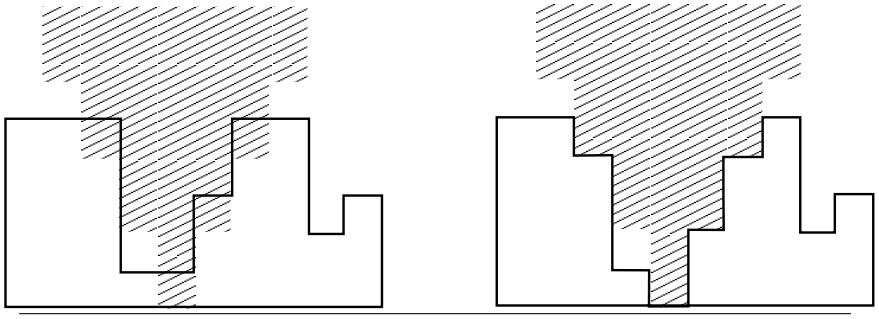
co dowodzi tezy indukcyjnej. ■

Z powyższego lematu wynika, że nasza procedura minimalizuje liczbę ruchów łopatą potrzebnych do wygładzenia terenu. Wiemy zatem, jak sprawdzać w czasie $O(n)$, czy możemy dokopać się do wody dla zadanych k oraz z. Oczywiście, jeśli możemy dokopać się do wody w jakimś punkcie z nachyleniem nie większym niż z, to dla $z+1$ również dostaniemy pozytywną odpowiedź. Ponadto dla $z = \max x_i$ nie trzeba nic wygładzać — wystarczy dokopać się do wody w najniższym punkcie, a w treści zadania zagwarantowano, że jest to możliwe. To pozwala znajdować najmniejsze z przy pomocy wyszukiwania binarnego (w każdym kroku wyszukiwania rozważamy wszystkie możliwe k). Takie rozwiązanie działa w czasie $O(n^2 \log(\max x_i))$ i pozwalało na zawodach zdobyć trochę punktów, ale niestety nie radziło sobie z największymi testami. Implementacje można znaleźć w plikach `stus0.cpp` i `stus1.pas`.

Rozwiązanie wzorcowe

Aby pozbyć się złożoności kwadratowej, musimy sprytniej obliczać liczbę potrzebnych ruchów łopatą. Cofnijmy się do miejsca, w którym przyjęliśmy $y_k = 0$, i załóżmy, że z jest ustalone. Wiemy, że $y_{k-1}, y_{k+1} \leqslant z$, bo inaczej przekroczylibyśmy dozwolone nachylenie. Indukcyjnie pokazujemy, że $y_{k-j}, y_{k+j} \leqslant jz$ dla $j = 1, 2, \ldots$ Z tego wynika, że ze studni musimy usunąć cały piasek zawarty w „odwróconej piramidzie" o środku w k i nachyleniu z (patrz rys. 1).

Czy to wystarczy do zapewnienia bezpieczeństwa? Otóż nie; na rysunku widać, że problem może pojawić się z dala od pozycji k lub zbocze w okolicy pozycji k może mieć nieregularny kształt.



Rys. 1: Zakreskowany teren to odwrócona piramida dla $z = 2$. Po lewej stronie widać pierwotną topografię, po prawej zaś teren po usunięciu piasku z piramidy.

Jednak po chwili zastanowienia można zaryzykować tezę, że trudności te nie miałyby miejsca, gdyby nachylenie studni przed rozpoczęciem kopania nie przekraczało z. Nasz wysiłek umysłowy z pierwszego rozdziału nie pójdzie na marne — dzięki lematowi 1 wiemy, jak optymalnie wygładzić ciąg (x_i) dla zadanego z, a potem spróbujemy sztuczki z piramidą. Najpierw jednak musimy przekonać się, czy to aby na pewno wystarczy.

Lemat 2. Wygładzenie ciągu (x_i) do nachylenia z, a następnie usunięcie piasku zawartego w piramidzie o środku w k i nachyleniu z, zadaje bezpieczny wykop do punktu k przy minimalnej liczbie ruchów łopatą.

Dowód: Przez (y_i) oznaczmy ciąg po wygładzeniu, zaś przez (u_i) — ciąg końcowy. Usunięcie piasku z piramidy oznacza, że na pozycji i otrzymamy wyraz $u_i = \min(y_i, |i-k| \cdot z)$. Upewnimy się teraz, że dla każdego i zachodzi $u_i \leqslant u_{i+1} + z$. Bez trudu dostajemy:

$$\begin{aligned} u_i &\leqslant y_i \leqslant y_{i+1} + z, \\ u_i &\leqslant |i-k| \cdot z \leqslant |(i+1)-k| \cdot z + z, \end{aligned}$$

a z połączenia tych dwóch nierówności mamy

$$u_i \leqslant \min(y_{i+1} + z,\, |(i+1)-k| \cdot z + z) = \min(y_{i+1},\, |(i+1)-k| \cdot z) + z = u_{i+1} + z.$$

Symetryczną nierówność $u_{i+1} \leqslant u_i + z$ dowodzimy w taki sam sposób. Pokazaliśmy zatem, że (u_i) zadaje bezpieczny wykop.

Niech teraz (v_i) będzie dowolnym ciągiem zadającym bezpieczny wykop przy założeniach lematu. Jest on, w szczególności, ciągiem wygładzającym (x_i), więc na mocy lematu 1 wiemy, że $v_i \leqslant y_i$ dla każdego i. W połączeniu z nierównością $v_i \leqslant |i-k| \cdot z$, otrzymujemy:

$$v_i \leqslant \min(y_i, |i-k| \cdot z) = u_i.$$

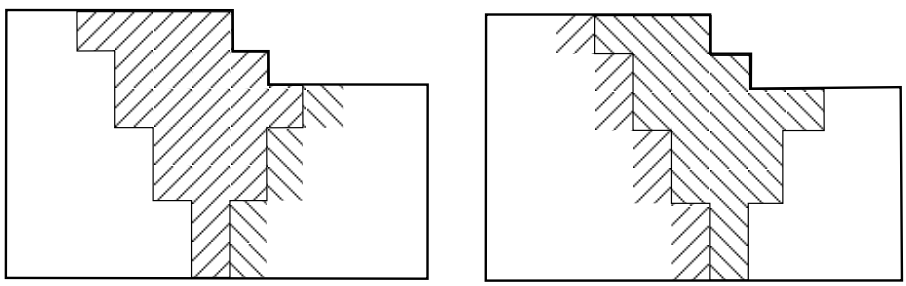
To dowodzi, że liczby ruchów łopatą nie da się poprawić. ■

Daje to następujący pomysł na lepszy algorytm: wyszukujemy binarnie z i dla ustalonego z wygładzamy teren, po czym obliczamy, ile piasku mieści się w odwróconych piramidach dla różnych k. Chcemy rozstrzygnąć, czy dla pewnego k łączna liczba ruchów łopatą nie przekracza m. Pozostaje tylko wymyślić sposób na szybkie obliczanie zawartości piramid.

Studnia pełna piramid

Na początek zauważmy, że dla $z = 0$ wystarczy sprawdzić, czy $\sum_{i=1}^{n} x_i \leqslant m$. Nieuwzględnienie tego przypadku może prowadzić do błędów w dalszej części rozważań.

Przyjmijmy teraz, że $z > 0$. Niech ciąg (y_i) oznacza wygładzony ciąg (x_i). Chcemy dla każdego k obliczyć sumę postaci $\sum_{i=1}^{n} \max(y_i - |i-k| \cdot z, 0)$. Warto pomyśleć, jak zmieni się taka suma, gdy k zwiększymy o 1. Na rysunku 2 widać dwa zbiory, w których leżą fragmenty gruntu odpowiednio wchodzące do piramidy i wychodzące z piramidy w trakcie jej przesunięcia. Nazwijmy te zbiory *prawą* i *lewą skośną*, a ich rozmiary oznaczmy przez p_i i l_i. Jasne jest, że znając liczbę pól każdej skośnej oraz zawartość piramidy dla $k = 1$, łatwo wyznaczymy odpowiedzi dla każdego k. Jako że pierwszą piramidę możemy zbadać w czasie $O(n)$, skupimy się teraz na obliczeniu ciągu p_i w czasie $O(n)$ (ciąg l_i można będzie wyznaczyć w ten sam sposób, wykonując obliczenia w odwrotnym kierunku).

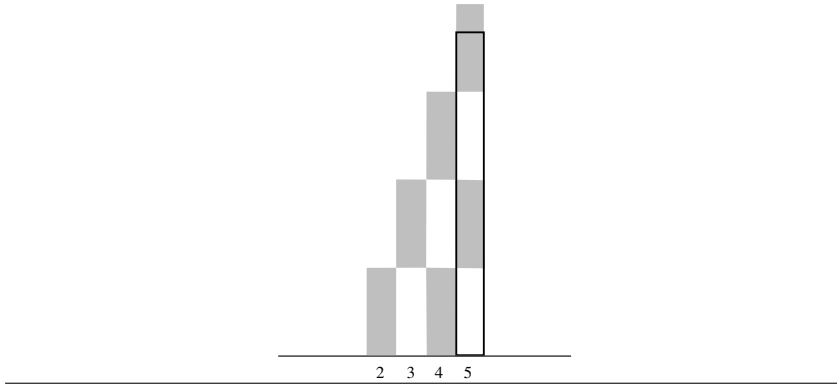


Rys. 2: Piramida o środku w k, przesuwając się o jednostkę w prawo, wchłania prawą skośną p_{k+1} i pozostawia po sobie lewą skośną l_k.

Rozważmy pewne $j \in \{1, \ldots, n\}$. Zastanówmy się, dla jakich i, i-ta prawa skośna zawiera piasek z pozycji j. Na pewno $y_j \bmod z$ najwyższych warstw trafi do skośnej o numerze $j - \left\lfloor \frac{y_j}{z} \right\rfloor$, oczywiście o ile skośna o takim numerze istnieje. Natomiast wszystkie skośne z przedziału $[\max(j + 1 - \left\lfloor \frac{y_j}{z} \right\rfloor, 1), j]$ dostaną po z warstw piasku, patrz także rys. 3.

Oczywiście, bezpośrednia aktualizacja skośnych doprowadziłaby nas z powrotem do czasu kwadratowego. Zamiast tego możemy jedynie zaznaczyć, gdzie zaczyna się i kończy przedział prawych skośnych, których rozmiary chcemy zwiększyć o z, a faktyczne sumowanie wykonać na samym końcu. Mówiąc dokładniej, jeśli w pomocniczej tablicy z licznikami $t[]$ na początku interesującego nas przedziału ustawimy 1, a tuż za jego końcem -1, to liczba przedziałów pokrywających skośną o numerze i będzie równa $t[1] + \ldots + t[i]$. Pozwala to wyznaczyć ciąg p_i następującym algorytmem (zakładamy, że tablice $t[]$ i $p[]$ są na początku wypełnione zerami).

```
1:  for j := 1 to n do begin
2:    t[max(j + 1 − y[j] div z, 1)] += 1;
3:    t[j + 1] −= 1;
4:    if j − y[j] div z > 0 then
5:      p[j − y[j] div z] += y[j] mod z;
6:  end
7:  sum := 0;
8:  for i := 1 to n do begin
9:    sum += t[i];
10:   p[i] += z · sum;
11: end
```



Rys. 3: Niech $j = 5$, $z = 3$ i $y_5 = 11$. Rysunek przedstawia, ile piasku z piątej pozycji trafi do poszczególnych prawych skośnych: 2 warstwy zasilą p_2, zaś do p_3, p_4, p_5 trafią po 3 warstwy.

Możemy w tym momencie podsumować nasze rozważania, prezentując pseudokod funkcji rozstrzygającej, czy można dokopać się do wody przy nachyleniu nieprzekraczającym z. Funkcja zwraca najmniejszy poprawny indeks k, jeśli bezpieczny wykop jest możliwy, a w przeciwnym razie zwraca -1.

```
1:  function sprawdź_nachylenie(z, m)
2:  begin
3:    y[] := wygładź(x[], z);
4:    p[] := oblicz_prawe_skośne(y[], z);
5:    l[] := oblicz_lewe_skośne(y[], z);
6:    koszt := 0;
7:    piramida := 0;
8:    for i := 1 to n do begin
9:      koszt += x[i] − y[i];
10:     piramida += max(y[i] − (i − 1) · z, 0);
11:   end
12:   if koszt + piramida ⩽ m then
13:     return 1;
```

```
14:   for k := 2 to n do begin
15:     piramida += p[k];
16:     piramida -= l[k - 1];
17:     if koszt + piramida ⩽ m then
18:       return k;
19:   end
20:   return -1;
21: end
```

Dołączając do powyższego pseudokodu wyszukiwanie binarne wartości z, uzyskujemy algorytm wykonujący $O(n \log(\max x_i))$ operacji i wykorzystujący liniową pamięć. Jego implementację można znaleźć w plikach `stu.cpp` i `stu1.pas`.

Rozwiązania niepoprawne

Autorzy testów przewidzieli takie błędy, jak wyrównywanie jedynie stoku wokół miejsca wykopu (plik `stub0.cpp`), pominięcie przypadku $z = 0$ (pliki `stub1.cpp` i `stub2.cpp`) czy też korzystanie ze zbyt małego typu do reprezentacji liczb całkowitych (plik `stub3.cpp`).

Inne niepoprawne rozwiązanie polega na próbie dokopania się do wody jedynie w miejscach o najmniejszej grubości piasku (plik `stub4.cpp`). Modyfikacja tej strategii, która wybiera np. 30 punktów o najmniejszej wysokości (plik `stub5.cpp`), działa nieco lepiej — jakkolwiek łatwo wskazać kontrprzykład na nią, trzeba przyznać, że nieźle radzi sobie z testami losowymi.

Rozwiązania wolniejsze

Oprócz wspomnianego na początku algorytmu, działającego w czasie $O(n^2 \log(\max x_i))$, można wymyślić szereg rozwiązań o złożoności czasowej $O(n \log n \log(\max x_i))$. Jednym z nich jest modyfikacja rozwiązania wzorcowego, korzystająca z tzw. drzewa licznikowego, zwanego też drzewem przedziałowym[1], do obliczenia zawartości skośnych (pliki `stus8.cpp`, `stus9.pas`, a także za pomocą drzewa potęgowego: `stus10.cpp` i `stus11.pas`). Innym przykładem jest zastosowanie wolniejszego algorytmu wygładzania terenu, w którym znajdujemy kolejno miejsca o najniższej wysokości gruntu, używając do tego kolejki priorytetowej. Wiadomo, że z najniższego miejsca nie warto usuwać piasku, więc można od razu zaktualizować sąsiednie pozycje i wyrzucić minimum z kolejki. Po wykonaniu tej operacji n razy teren zostanie wygładzony. Implementacje tego pomysłu można znaleźć w plikach `stus12.cpp`, `stus13.pas`, `stus14.cpp` i `stus15.pas`. Rozwiązania o takiej złożoności mogły zdobyć 60 punktów lub więcej, w zależności od jakości implementacji.

Rozwiązania działające co najmniej liniowo względem m lub $\max x_i$ (pliki `stus2.cpp` – `stus7.pas`) nie miały większych szans na osiągnięcie wyniku powyżej 30 punktów. Należały do nich w szczególności programy niekorzystające z wyszukiwania binarnego.

[1] Opis tej struktury danych można znaleźć np. w opracowaniu zadania *Tetris 3D* z XIII Olimpiady Informatycznej [13] czy opracowaniu zadania *Koleje* z IX Olimpiady Informatycznej [9].

Testy

Do tworzenia testów używane były funkcje generujące losowe ciągi liczb: rosnące, malejące i dowolne. W tabeli „dołek” oznacza ciąg liczb, który do pewnego miejsca jest malejący, a potem rosnący. Z kolei „nierówny dołek” powstaje z dołka przez podzielenie go na fragmenty równej długości i losowe poprzestawianie elementów w każdym fragmencie z osobna. Podobnie definiujemy „górkę” i „nierówną górkę”.

Nazwa	n	m	Opis
stu1.in	71	236	mały dołek z nierównościami na środku
stu2a.in	237	821	mały dołek z nierównościami na środku otoczony spadami
stu2b.in	842	681 835	niewielki losowy test, odpowiedź 0
stu3.in	2 042	4 423	niewielki nierówny dołek
stu4a.in	5 000	2 521	trzy niewielkie dołki, skrajne są nierówne
stu4b.in	10 000	5	małe liczby prócz jednej na środku
stu4c.in	10 000	10^{12}	duża liczba dołków, z których jeden jest szerszy i płytszy
stu5a.in	10 000	500 000	dołek otoczony losowymi wartościami
stu5b.in	10 000	$\approx 10^{14}$	dosyć duży losowy test, odpowiedź 0
stu5c.in	10 000	10^{11}	duża liczba losowych odcinków, z których jeden jest szerszy i średnio niższy
stu6a.in	486 000	2 414 746 423	54 średnie dołki
stu6b.in	75 000	811 178 223	dołek z nierównościami na środku
stu7a.in	100 000	828	test z jedną poprawną odpowiedzią
stu7b.in	140 000	411 651 544	dołek z nierównościami na środku
stu7c.in	450 000	898 889	duży losowy test
stu8a.in	150 000	2	test z jedną poprawną odpowiedzią
stu8b.in	90 000	1 059 865 233	dołek z nierównościami na środku
stu9a.in	400 000	352 247	duży dołek z nierównościami na środku
stu9b.in	550 000	$\approx 3 \cdot 10^{14}$	duży dołek z nierównościami na środku
stu10a.in	500 000	10^{18}	duża nierówna górka, odpowiedź 0
stu10b.in	700 000	$\approx 10^{12}$	duży dołek z nierównościami na środku
stu11a.in	1 000 000	15 486 352 148	150 nierównych dołków
stu11b.in	1 000 000	500 000 000	maksymalny dołek
stu11c.in	1 000 000	$\approx 5 \cdot 10^{12}$	duży dołek z nierównościami na środku
stu12a.in	1 000 000	$\approx 5 \cdot 10^{11}$	dołek z nierównościami na środku
stu12b.in	750 000	2	test z jedną poprawną odpowiedzią

Zawody II stopnia

opracowania zadań

Marek Cygan
Treść zadania, Opracowanie

Mateusz Baranowski
Program

Dostępna pamięć: 128 MB.

OI, etap II, dzień próbny, 7.02.2012

Tour de Bajtocja

W Bajtocji jest n miast. Niektóre pary miast są połączone dwukierunkowymi drogami. Drogi nie przecinają się poza końcami, w razie konieczności prowadzą tunelami lub estakadami.

Już wkrótce rozpocznie się znany wyścig kolarski **Tour de Bajtocja**. *Wiadomo, że trasa wyścigu będzie przebiegała drogami Bajtocji, będzie miała początek i koniec w tym samym mieście oraz nie będzie prowadziła żadną drogą więcej niż raz.*

Bajtazar jest słynnym bajtockim kibicem, szefem fanklubu drużyny piłkarskiej **Bajtusie**. *Bajtazar i jego klubowi przyjaciele szczerze nienawidzą wyścigów kolarskich. Chcą oni uniemożliwić sytuację, w której trasa wyścigu przebiegałaby przez miasta, w których mieszkają. Aby temu zapobiec, są gotowi zablokować część dróg. Bajtazar wie, w których miastach mieszkają poszczególni członkowie jego klubu. Chce on wyznaczyć minimalną liczbę dróg, jakie trzeba zablokować, tak aby wyścig kolarski nie mógł przebiegać przez* **żadne** *z miast, w których mieszka jakiś członek jego klubu (oczywiście, wliczając w to samego Bajtazara). Twoim zadaniem jest pomóc Bajtazarowi w wyznaczeniu takiego zbioru dróg.*

Wejście

W pierwszym wierszu standardowego wejścia znajdują się trzy liczby całkowite n, m oraz k ($1 \leqslant n \leqslant 1\,000\,000$, $0 \leqslant m \leqslant 2\,000\,000$, $1 \leqslant k \leqslant n$), pooddzielane pojedynczymi odstępami, oznaczające odpowiednio: liczbę miast, liczbę dróg oraz liczbę miast, w których mieszkają członkowie klubu. Miasta są ponumerowane od 1 do n, przy czym miasta, w których mieszkają członkowie klubu, mają numery od 1 do k. W każdym z kolejnych m wierszy znajdują się dwie liczby całkowite oddzielone pojedynczym odstępem, a_i i b_i ($1 \leqslant a_i < b_i \leqslant n$), oznaczające, że miasta a_i i b_i są połączone dwukierunkową drogą. Każda para miast Bajtocji jest połączona co najwyżej jedną drogą.

W testach wartych łącznie 40% punktów zachodzą dodatkowe warunki $n \leqslant 1\,000$ i $m \leqslant 5\,000$.

Wyjście

Twój program powinien wypisać na standardowe wyjście zbiór dróg o minimalnej liczności, których blokada uniemożliwi zorganizowanie wyścigu kolarskiego przechodzącego przez jakiekolwiek miasto, w którym mieszkają członkowie klubu.

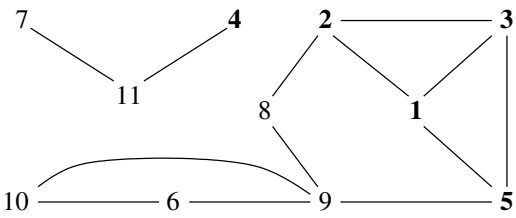
W pierwszym wierszu wyjścia należy wypisać minimalną liczbę dróg, jakie należy zablokować. W kolejnych wierszach powinien znaleźć się opis dróg, które należy zablokować, po jednej w wierszu. Każdą drogę reprezentujemy jako parę numerów miast połączonych przez tę drogę. Jako pierwsze należy wypisać miasto o mniejszym numerze. Numery miast należy oddzielić pojedynczym odstępem.

Jeśli istnieje więcej niż jedno rozwiązanie, Twój program powinien wypisać jedno, dowolne z nich.

Przykład

Dla danych wejściowych:

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



jednym z poprawnych wyników jest:

```
3
2 3
8 9
3 5
```

Rozwiązanie

W zadaniu mamy dany graf nieskierowany G, reprezentujący mapę Bajtocji, w którym pewne wierzchołki są wyróżnione (te o numerach nie większych niż k). Cykl, który zawiera co najmniej jeden wyróżniony wierzchołek, będziemy nazywać *wyróżnionym cyklem*; podobnie krawędź, która zawiera co najmniej jeden wyróżniony koniec, będziemy nazywać *wyróżnioną krawędzią*. Celem zadania jest usunięcie jak najmniejszej liczby krawędzi z grafu, aby wyeliminować wszystkie wyróżnione cykle. Dowolny taki najmniej liczny zbiór usuwanych krawędzi będziemy nazywać *rozwiązaniem*.

Na początek zastanówmy się, jakie warunki musi spełniać graf, aby nie istniał w nim żaden wyróżniony cykl. Użyjemy tu pojęcia *mostu*, czyli krawędzi, przez którą nie przechodzi żaden cykl. Równoważna definicja określa most jako krawędź, której usunięcie powoduje zwiększenie się liczby spójnych składowych w grafie. Jako że każdy cykl przechodzący przez wyróżnioną krawędź jest wyróżniony i odwrotnie, każdy wyróżniony cykl przechodzi przez co najmniej jedną wyróżnioną krawędź, otrzymujemy poniższy lemat.

Lemat 1. Graf nie zawiera wyróżnionego cyklu wtedy i tylko wtedy, gdy każda wyróżniona krawędź jest mostem.

Zauważmy, że gdyby wszystkie krawędzie grafu były wyróżnione, to wyeliminowanie wyróżnionych cykli byłoby równoważne usunięciu wszystkich cykli. Wówczas

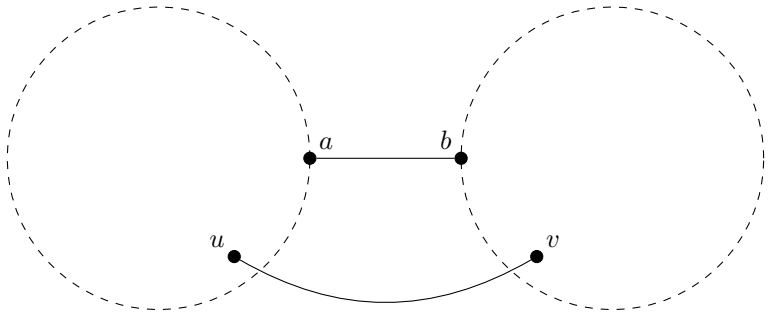
wystarczyłoby znaleźć dowolny maksymalny zbiór krawędzi, który jest lasem, i jako rozwiązanie wypisać jego dopełnienie. Przypadek, w którym wszystkie krawędzie są wyróżnione, wydaje się być przypadkiem wyjątkowo szczególnym. W dalszej części opisu pokażemy jednak, że przypadek ogólny można zredukować do omawianego przypadku szczególnego. Zacznijmy od udowodnienia poniższego, kluczowego lematu.

Lemat 2. Istnieje rozwiązanie, które zawiera jedynie wyróżnione krawędzie.

Dowód: Przez X oznaczmy rozwiązanie, które zawiera najmniejszą liczbę niewyróżnionych krawędzi. Załóżmy, że X zawiera jakąś krawędź uv, która nie jest wyróżniona (w przeciwnym przypadku lemat w oczywisty sposób zachodzi). Z metody wyboru zbioru X wynika, że w grafie G istnieje wyróżniony cykl C_0, który zawiera dokładnie jedną krawędź ze zbioru X i jest to krawędź uv; w przeciwnym przypadku zbiór $X \setminus \{uv\}$ również byłby rozwiązaniem. Niech ab będzie dowolną wyróżnioną krawędzią należącą do cyklu C_0. Wykażemy, że zbiór $X' = (X \setminus \{uv\}) \cup \{ab\}$ również jest rozwiązaniem, co będzie stanowiło sprzeczność wobec faktu, że $|X'| = |X|$ i X' zawiera mniej niewyróżnionych krawędzi niż X.

Rozważmy dowolny wyróżniony cykl C' w grafie G. Jeśli cykl C' zawiera jakąś krawędź ze zbioru $X \setminus \{uv\}$, to zawiera on co najmniej jedną krawędź ze zbioru X'. Wystarczy zatem rozważyć cykle w grafie G, których jedyną krawędzią ze zbioru X jest krawędź uv. Dlatego też aby wykazać, że graf $G \setminus X'$ [1] nie zawiera wyróżnionych cykli, wystarczy wykazać, że w grafie $G \setminus X'$ krawędź uv jest mostem, gdyż wtedy w grafie $G \setminus X'$ nie istnieje żaden cykl (nawet niewyróżniony), który przechodziłby przez krawędź uv.

Z lematu 1 wiemy, że w grafie $G \setminus X$ krawędź ab jest mostem. Jednocześnie cykl C_0 implikuje, że krawędź ab nie jest mostem w grafie $G \setminus (X \setminus \{uv\})$. Z tego wynika, że wierzchołki u oraz v znajdują się po przeciwnych stronach mostu ab w grafie $G \setminus X$, co zostało przedstawione na rysunku 1. Zatem w grafie $G \setminus (X \cup \{ab\})$ wierzchołki u oraz v znajdują w różnych spójnych składowych, stąd krawędź uv jest mostem w grafie $G \setminus X'$. To kończy dowód lematu. ■



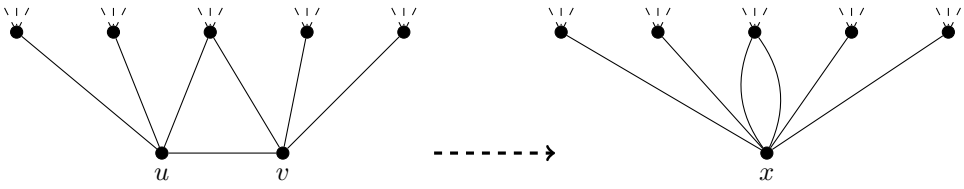
Rys. 1: Przerywane okręgi oznaczają dwie spójne składowe będące po różnych stronach mostu ab w grafie $G \setminus X$.

[1] Przez $G \setminus X'$ oznaczamy graf G po usunięciu krawędzi ze zbioru X'.

W dalszym opisie przydatna nam będzie operacja *ściągnięcia* krawędzi polegająca na zredukowaniu jej do jednego wierzchołka. Formalnie, jako operację ściągnięcia krawędzi e łączącej wierzchołki u i v określamy następujący ciąg operacji:

- do grafu dodajemy nowy wierzchołek x;
- usuwamy krawędź e;
- każdą krawędź ab mającą co najmniej jeden koniec w zbiorze $\{u, v\}$ zastępujemy krawędzią $a'b'$, gdzie $a' = x$ jeśli $a \in \{u, v\}$ oraz $a' = a$ w przeciwnym przypadku, podobnie $b' = x$ jeśli $b \in \{u, v\}$ oraz $b' = b$ w przeciwnym przypadku;
- usuwamy wierzchołki u oraz v.

Zauważmy, że operacja ściągnięcia krawędzi może spowodować, że w grafie pojawią się krawędzie wielokrotne lub pętle, nawet pomimo założenia, że wejściowy graf nie zawiera takich krawędzi, co zostało zilustrowane na rysunku 2.



Rys. 2: Sklejenie krawędzi uv.

Zauważmy, że każde rozwiązanie w grafie G' powstałym przez ściągnięcie krawędzi uv w grafie G jest też rozwiązaniem w grafie G (o ile wystąpienia wierzchołka x w tym rozwiązaniu podmienimy na odpowiednie wystąpienia wierzchołka u lub v). I odwrotnie, jeśli w grafie G istnieje rozwiązanie niezawierające krawędzi uv, to rozwiązanie to możemy zinterpretować jako rozwiązanie w grafie G'. Stąd oraz na podstawie dwóch przedstawionych powyżej lematów wynika, że poprzez ściągnięcie kolejno wszystkich niewyróżnionych krawędzi problem postawiony w zadaniu możemy zredukować do szczególnego przypadku, w którym wszystkie krawędzie są wyróżnione, a ten szczególny przypadek omówiliśmy na początku opracowania. Formalny dowód ostatniej obserwacji pozostawiamy Czytelnikowi jako ćwiczenie.

Rozwiązanie wzorcowe jest implementacją powyższego rozumowania przy pomocy przeszukiwania wszerz (algorytm BFS). Początkowo ściągamy wszystkie niewyróżnione krawędzie, a następnie z użyciem algorytmu BFS w każdej spójnej składowej wyznaczamy drzewo rozpinające. Do usunięcia przeznaczamy wszystkie wyróżnione krawędzie nienależące do znalezionego lasu rozpinającego. Złożoność czasowa i pamięciowa tego rozwiązania wynosi $O(n + m)$. Zostało ono zaimplementowane w plikach `tou.cpp` i `tou1.pas`.

W rozwiązaniu alternatywnym wykorzystujemy strukturę *Find-Union* działającą na zbiorze wierzchołków grafu. Pomimo iż asymptotycznie rozwiązanie to nie jest liniowe, jest ono bardzo efektywne w praktyce oraz proste w implementacji, gdyż zarówno ściąganie krawędzi grafu, jak i budowanie lasu rozpinającego może zostać zrealizowane za pomocą tej samej operacji *union*, patrz następujący pseudokod.

```
1: foreach ab ∈ E(G) do begin
2:   if a > k then { krawędź niewyróżniona: ściągamy }
3:     union(a, b)
4:   else { krawędź wyróżniona: do rozważenia dalej }
5:     Wyr.insert(ab);
6: end
7: foreach ab ∈ Wyr do begin
8:   if find(a) ≠ find(b) then
9:     union(a, b) { krawędź należy do drzewa rozpinającego }
10:  else
11:    wypisz(ab); { krawędź należy do rozwiązania }
12: end
```

Złożoność rozwiązania alternatywnego wynosi $O((n+m)\log^* n)$. Jego implementacje można znaleźć w plikach `tou2.cpp` oraz `tou3.pas`.

Testy

Rozwiązania zawodników były sprawdzane za pomocą 10 zestawów danych testowych, z których każdy składał się z od 2 do 4 pojedynczych testów. W poniższej tabeli przyjęto oznaczenia n, m, k zgodnie z treścią zadania, natomiast przez r oznaczono rozmiar rozwiązania, czyli liczbę krawędzi, które należy usunąć z grafu.

Nazwa	n	m	k	r	Opis
tou1a.in	10	15	5	5	graf Petersena z wyróżnionymi wierzchołkami wewnętrznymi
tou1b.in	8	12	4	5	mały test poprawnościowy
tou1c.in	1	0	1	0	test minimalny z jednym wyróżnionym wierzchołkiem
tou1d.in	6	9	2	1	mały test poprawnościowy
tou2a.in	12	26	5	14	mały test poprawnościowy
tou2b.in	40	100	10	28	mały test losowy
tou2c.in	70	300	23	133	mały test losowy
tou3a.in	300	389	30	27	losowy graf z długimi cyklami
tou3b.in	400	600	40	43	test losowy
tou3c.in	500	800	30	56	test losowy

Nazwa	n	m	k	r	Opis
tou4a.in	999	1 332	333	333	trójkąty połączone w cykl
tou4b.in	1 000	5 000	30	213	test losowy, wierzchołki o dużym stopniu
tou5a.in	15 000	19 517	6 000	4 294	trójkąty podłączone do dwóch cykli
tou5b.in	10 000	100 000	846	14 749	duża liczba spójnych składowych
tou5c.in	1 000	499 500	333	277 056	klika o 1000 wierzchołkach
tou6a.in	100 000	800 990	10 000	142 256	losowe grafy połączone mostami
tou6b.in	100 000	400 000	5 000	33 831	test losowy
tou7a.in	300 000	343 091	10 000	6 705	suma krótkich cykli
tou7b.in	300 000	900 000	37 504	138 466	losowy graf z dużą liczbą wierzchołków wyróżnionych
tou8a.in	500 000	1 799 335	1 000	799 837	graf wyróżniony podłączony do grafów niewyróżnionych
tou8b.in	500 000	666 666	1 158	166 667	wierzchołki wyróżnione mają duży stopień
tou9a.in	1 000 000	1 999 998	500 000	750 000	test wydajnościowy, cykl niewyróżniony z podłączonymi wyróżnionymi wierzchołkami
tou9b.in	1 000 000	2 000 000	507 087	685 209	test z dużą liczbą losowych spójnych składowych
tou9c.in	1 000 000	1 999 998	1	999 998	jeden centralny wierzchołek jest wyróżniony, a pozostałe tworzą cykl
tou10a.in	999 999	999 999	333 333	444 444	nieregularna siatka
tou10b.in	1 000 000	1 350 223	2 377	350 224	losowy graf maksymalny z długimi cyklami
tou10c.in	1 000 000	2 000 000	25 000	71 510	test losowy o maksymalnych rozmiarach grafu

Jakub Pachocki — Treść zadania, Opracowanie
Jakub Pachocki — Program

Dostępna pamięć: 64 MB. *OI, etap II, dzień pierwszy, 8.02.2012*

Bony

Sklep ze słodyczami, którego właścicielem jest Bajtazar, prowadzi sprzedaż pysznych cukierków karmelowych. Dla każdej liczby całkowitej dodatniej c w sklepie znajduje się dokładnie jedna paczka zawierająca c cukierków (i w chwili obecnej nie są przewidywane kolejne dostawy). Aby zachęcić klientów do kupna łakoci, Bajtazar powrzucał do m paczek bony na roczny zapas czekolady. Upewnił się przy tym, aby nie wrzucić więcej niż jednego bonu do tej samej paczki.

W przyszłym tygodniu w Bajtogrodzie rozpoczynają się obchody karnawału, który potrwa n dni; k-tego dnia karnawału odbędzie się przyjęcie, w którym będzie uczestniczyć a_k osób. Bajtazar jest przekonany, że k-tego dnia rano każdy z uczestników odbywającego się tego dnia przyjęcia kupi w jego sklepie najmniejszą dostępną paczkę cukierków, której zawartość będzie można rozdzielić równo pomiędzy wszystkie zaproszone osoby. Przykładowo, jeśli $n = 2$, $a_1 = 4$, $a_2 = 2$, to pierwszego dnia karnawału zostaną sprzedane kolejno paczki zawierające cztery, osiem, dwanaście i szesnaście cukierków, a drugiego dnia — paczki z dwoma i sześcioma cukierkami.

Bajtazar zastanawia się, którzy klienci kupią paczki z bonami. Poprosił Cię, żebyś napisał program, który pomoże mu to określić.

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita m ($1 \leqslant m \leqslant 1\,000\,000$), oznaczająca liczbę bonów. W k-tym z kolejnych m wierszy znajduje się liczba całkowita b_k ($1 \leqslant b_k \leqslant 1\,000\,000$) oznaczająca rozmiar paczki (tj. liczbę cukierków w paczce), w której Bajtazar umieścił k-ty bon. Liczby te są podane w kolejności rosnącej.

W następnym wierszu znajduje się jedna liczba całkowita n ($1 \leqslant n \leqslant 1\,000\,000$), oznaczająca liczbę dni karnawału. W k-tym z kolejnych n wierszy znajduje się liczba całkowita a_k ($1 \leqslant a_k \leqslant 1\,000\,000$), oznaczająca liczbę gości zaproszonych na przyjęcie odbywające się w k-tym dniu.

W testach wartych łącznie przynajmniej 50% punktów żadna z liczb podanych na wejściu nie przekroczy 5 000.

Wyjście

W pierwszym wierszu standardowego wyjścia Twój program powinien wypisać liczbę całkowitą z — liczbę sprzedanych paczek z bonami. W kolejnych z wierszach powinny znaleźć się numery wszystkich klientów, którzy kupili paczkę z bonem, w porządku rosnącym. Klientów numerujemy od 1 w kolejności dokonywania zakupów.

Przykład

Dla danych wejściowych:

```
1
6
8
16
3
4
2
4
```

poprawnym wynikiem jest:

```
3
2
4
6
```

Rozwiązanie

Wprowadzenie

Dla lepszego zrozumienia podanego problemu, warto na początku wyobrazić go sobie w sposób czysto matematyczny. Dane jest n liczb naturalnych $a_1, \ldots, a_n$. Tworzymy ciąg c, złożony z $a_1 + \ldots + a_n$ liczb naturalnych, przez postawienie na początku a_1 najmniejszych liczb naturalnych podzielnych przez a_1 (pierwsze przyjęcie), następnie a_2 najmniejszych niewystępujących dotąd w ciągu liczb naturalnych podzielnych przez a_2 (drugie przyjęcie), itd. Zadanie polega na efektywnym określeniu, na jakich pozycjach w ciągu c znajdują się liczby należące do zadanego zbioru $B = \{b_1, \ldots, b_m\}$.

W przykładzie z treści zadania mamy $a = (4, 2, 4)$ oraz $B = \{1, 6, 8, 16\}$. Ciąg c wyznaczony dla podanego ciągu a ma postać $c = (4, 8, 12, 16, 2, 6, 20, 24, 28, 32)$. Drugi, czwarty i szósty element tego ciągu należą do zbioru B.

Rozwiązanie wzorcowe

Oznaczmy przez M maksimum z wszystkich liczb występujących na wejściu. Rozwiązanie wzorcowe polega na symulowaniu zakupów kolejnych klientów. Należy tylko pamiętać o tym, aby nie rozważać rozmiarów paczek większych niż M (gdyż w nich i tak nie ma żadnych bonów), a w przypadku powtarzających się elementów ciągu a, przy rozpatrywaniu a_i zaczynać przeglądanie paczek od ostatniej paczki rozpatrzonej dla elementu równego a_i.

Innymi słowy, dla każdej liczby naturalnej $p \in [1, M]$ pamiętamy wartość *ostatnia*$[p]$, oznaczającą ostatnią wielokrotność liczby p zużytą w trakcie rozważania jakiegoś $a_i = p$. Jeśli *ostatnia*$[p] > M$, możemy wówczas przerwać rozpatrywanie tego a_i. Poza tym utrzymujemy tablicę wartości logicznych *paczka*$[1..M]$, wskazującą, które paczki już zostały wykupione, a także podobną tablicę *bon*$[1..M]$, informującą, w których paczkach znajdują się bony. Poniżej znajduje się pseudokod algorytmu symulującego zakupy kolejnych klientów przy użyciu podanych tablic.

```
1:  program bony
2:  begin
3:    for i := 1 to M do begin
4:      ostatnia[i] := 0;
5:      paczka[i] := false;
6:      bon[i] := false;
7:    end
8:    for i := 1 to m do
9:      bon[b[i]] := true;
10:   liczba_klientów := 0;
11:   for i := 1 to n do begin
12:     p := a[i];
13:     akt := ostatnia[p] + p;
14:     uczestnik := 1;
15:     while (akt ⩽ M) and (uczestnik ⩽ p) do begin
16:       if not paczka[akt] then begin
17:         paczka[akt] := true;
18:         if bon[akt] then
19:           wypisz(liczba_klientów + uczestnik);
20:         uczestnik := uczestnik + 1;
21:       end
22:       if uczestnik ⩽ p then akt := akt + p;
23:     end
24:     ostatnia[p] := akt;
25:     liczba_klientów := liczba_klientów + p;
26:   end
27: end
```

Powyższe rozwiązanie może na pierwszy rzut oka wydawać się niezbyt efektywne. Występująca w nim pętla **while**, zagnieżdżona w pętli **for**, może za każdym razem wykonywać wiele obrotów, w tym dla rozmiarów paczki *akt*, które zostały już wykupione (tj. gdy *paczka*[*akt*] = **true**). Zauważmy jednak, że dla każdej możliwej wartości p, jaką może przyjąć a_i, każdą z $\left\lfloor \frac{M}{p} \right\rfloor$ wielokrotności p rozpatrzymy co najwyżej raz (dzięki wykorzystaniu tablicy *ostatnia*). To daje nam następujące ograniczenie górne na łączną liczbę obrotów pętli **while**:

$$\sum_{p=1}^{M} \frac{M}{p} = M \cdot H_M = O(M \log M).$$

W powyższym wzorze H_M to M-ta liczba harmoniczna, $H_M = \frac{1}{1} + \frac{1}{2} + \ldots + \frac{1}{M}$, którą to liczbę możemy oszacować z góry przez $O(\log M)$. To oszacowanie pojawiało się już w zadaniach olimpijskich (np. w zadaniu *Korale* z XVII Olimpiady Informatycznej [17]), można o nim także poczytać w książce [25].

Całe rozwiązanie działa więc w czasie $O(M \log M)$. Jego implementacje można znaleźć w plikach `bon.cpp` i `bon1.pas`.

Inne rozwiązania

Gdyby w powyższym algorytmie dla każdego a_i przeglądać jego wielokrotności od początku, bądź też iterować po wielokrotnościach a_i przekraczających M, otrzyma się rozwiązanie o złożoności czasowej $O(M^2)$. Implementacje rozwiązania wykazującego pierwszą z wymienionych usterek można znaleźć w plikach `bons1.cpp` i `bons2.pas`. Zgodnie z informacją podaną w treści zadania, takie rozwiązania zdobywały na zawodach około 50 punktów.

Możliwym błędem w implementacji rozwiązania wzorcowego było niezauważenie, że wyniki (czyli numery klientów, którzy zakupią paczki z bonami) mogą nie mieścić się w 32-bitowym typie całkowitym (patrz pliki `bonb1.cpp` i `bonb2.pas`). Taki błąd kosztował na zawodach utratę 40 punktów.

Testy

Rozwiązania zawodników były sprawdzane za pomocą 10 zestawów danych testowych. Wszystkie testy były generowane w sposób losowy, przy czym w testach *3b*, *4c*, *5c*, *6b*, *7bcd*, *8bcd*, *9cd* oraz *10cd* liczby gości zaproszonych na poszczególne przyjęcia były w znacznej części niewielkimi liczbami całkowitymi z wybranego zakresu. W poniższej tabeli m oznacza liczbę bonów, natomiast n — liczbę przyjęć.

Nazwa	m	n
bon1a.in	5	15
bon1b.in	5	15
bon1c.in	5	15
bon2a.in	20	50
bon2b.in	1	1
bon2c.in	17	100
bon3a.in	300	700
bon3b.in	300	700
bon4a.in	2 000	2 000
bon4b.in	2 000	2 000
bon4c.in	2 000	2 000
bon5a.in	3 000	5 000
bon5b.in	5 000	5 000
bon5c.in	5 000	5 000
bon6a.in	12 345	32 000
bon6b.in	30 000	40 000

Nazwa	m	n
bon7a.in	120 345	320 000
bon7b.in	200 000	300 000
bon7c.in	200 000	300 000
bon7d.in	200 000	300 000
bon8a.in	400 000	400 000
bon8b.in	370 000	400 000
bon8c.in	370 000	400 000
bon8d.in	400 000	400 000
bon9a.in	800 000	900 000
bon9b.in	80	900 000
bon9c.in	370 000	800 000
bon9d.in	370 000	400 000
bon10a.in	1 000 000	1 000 000
bon10b.in	500 000	1 000 000
bon10c.in	1 000	1 000 000
bon10d.in	1 000	1 000 000

Jan Kanty Milczek
Treść zadania, Opracowanie

Michał Zgliczyński
Program

Dostępna pamięć: 128 MB. *OI, etap II, dzień pierwszy, 8.02.2012*

Szatnia

W Bajtocji odbywa się doroczny zlot bogatych mieszkańców. Zbierają się oni, by chwalić się swoimi zarobkami, butami Lebajtina i innymi luksusowymi rzeczami. Naturalnie, nie wszystkie rzeczy wnoszą na bankiet — płaszcze, kurtki czy parasole zostawiają w szatni, by odebrać je, wychodząc.

Na nieszczęście bogaczy szajka bajtockich złodziei planuje włamać się do szatni i ukraść część pozostawionych tam przedmiotów. Póki co szef szajki przegląda zaproponowane przez gangsterów plany skoku. Każdy plan wygląda następująco: złodzieje pojawiają się w szatni w chwili m_j, zabierają przedmioty o wartości **dokładnie** *k_j i uciekają, przy czym cały skok zajmuje im czas s_j. Szef gangu chciałby przede wszystkim wiedzieć, które z planów mają szansę się udać, a które nie. Plan ma szansę się udać, jeśli w chwili m_j da się uzbierać przedmioty o łącznej wartości dokładnie k_j, w taki sposób, żeby do momentu $m_j + s_j$ włącznie nikt nie przyszedł po żaden z kradzionych przedmiotów (w takim wypadku zawiadomiłby ochronę i ucieczka nie udałaby się). W szczególności, jeśli w chwili m_j w ogóle nie da się dobrać przedmiotów o łącznej wartości k_j, plan zostaje odrzucony. Znając moment przyniesienia i zabrania każdego przedmiotu, określ, które plany mają szansę powodzenia, a które skazane są na porażkę. Zakładamy, że jeśli przedmiot zostaje przyniesiony w momencie, w którym złodzieje mają dokonać skoku, mogą go już ukraść (patrz test przykładowy).*

Wejście

W pierwszym wierszu standardowego wejścia znajduje się liczba całkowita n ($1 \leqslant n \leqslant 1\,000$), reprezentująca liczbę przedmiotów, które zostaną pozostawione w szatni. W kolejnych n wierszach znajdują się opisy przedmiotów. Każdy z nich składa się z trzech liczb całkowitych c_i, a_i i b_i ($1 \leqslant c_i \leqslant 1\,000$, $1 \leqslant a_i < b_i \leqslant 10^9$), pooddzielanych pojedynczymi odstępami, oznaczających kolejno: wartość przedmiotu, moment, w którym przedmiot zostanie przyniesiony do szatni, oraz moment, w którym zostanie z niej zabrany.

Następny wiersz zawiera liczbę p ($1 \leqslant p \leqslant 1\,000\,000$) — liczbę planów przedstawionych przez szajkę. Każdy z nich jest opisany w jednym wierszu przez trzy liczby całkowite m_j, k_j i s_j ($1 \leqslant m_j \leqslant 10^9$, $1 \leqslant k_j \leqslant 100\,000$, $0 \leqslant s_j \leqslant 10^9$), pooddzielane pojedynczymi odstępami, oznaczające kolejno: moment, w którym złodzieje weszliby do szatni, wartość, jaką chcą uzyskać, oraz czas, jaki zajmie im kradzież.

W testach wartych 16% punktów zachodzi dodatkowy warunek $p \leqslant 10$.

W innych testach, również wartych 16% punktów, wszystkie przedmioty mają równe a_i.

W jeszcze innych testach, wartych 24% punktów, wszystkie zapytania mają równe s_j.

Wyjście

Dla każdego planu szajki określ, czy plan da się zrealizować, tj. ukraść przedmioty o wartości dokładnie k_j i uciec, zanim ktoś upomni się o swoją rzecz. Jeśli plan jest wykonalny, Twój program powinien wypisać na standardowe wyjście słowo `TAK`*, a w przeciwnym wypadku słowo* `NIE`*.*

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
5	TAK
6 2 7	NIE
5 4 9	TAK
1 2 4	TAK
2 5 8	NIE
1 3 9	
5	
2 7 1	
2 7 2	
3 2 0	
5 7 2	
4 1 5	

Rozwiązanie

Wprowadzenie

Mamy zadany zbiór elementów, z których każdy jest scharakteryzowany przez trzy parametry c_i, a_i, b_i, oznaczające odpowiednio wartość elementu, moment pojawienia się oraz moment zniknięcia. Dla takiego zbioru elementów mamy określone zapytania postaci (m, k, s): czy w chwili m istnieje podzbiór elementów, które sumują się do k oraz będą istniały przez cały okres czasu $[m, m + s]$.

Rozwiązanie

Nasze zadanie jest wariacją na temat *problemu pakowania plecaka* (patrz np. [23]), możemy więc spodziewać się rozwiązania korzystającego z programowania dynamicznego. W klasycznej wersji problemu każdy przedmiot miał jednak dwie charakterystyczne dla siebie cechy — cenę c i wagę w. U nas przedmioty mają jedną cechę — wartość, a w dodatku określone są dla nich momenty pojawienia się i zniknięcia. Będziemy próbować poradzić sobie z kolejnymi trudnościami.

Brak wagi

Oczywiście w praktyce nie jest to problem, wręcz przeciwnie. Gdybyśmy mogli zupełnie pominąć problem czasu i po prostu odpowiadać na pytania, czy z zestawu n przedmiotów możemy uzyskać sumę k, wystarczyłoby przeglądać kolejne przedmioty i zapisywać odpowiedź, czy z pierwszych i przedmiotów możemy uzyskać sumę k. Zależność jest prosta — albo mogliśmy tę sumę uzyskać już za pomocą $i-1$ przedmiotów, albo uzyskamy ją poprzez dodanie i-tego przedmiotu do zbioru uzyskiwalnego $i - 1$ przedmiotami. Podejście to realizuje następujący pseudokod (zakładamy, że wartości przedmiotów są zapisane w tablicy C).

```
 1: function CzySieDa(k, n, C)
 2: begin
 3:   DaSie[0] := true;
 4:   for j := 1 to k do
 5:     DaSie[j] := false;
 6:   for i := 1 to n do
 7:     for j := k downto C[i] do
 8:       if DaSie[j − C[i]] then
 9:         DaSie[j] := true;
10:   return DaSie[k];
11: end
```

Czas dodania i czas zabrania

Te parametry wydają się symetryczne, więc zapewne warto traktować je w podobny sposób. Spróbujmy na początek skonstruować dowolne (być może za wolne) poprawne rozwiązanie. Możemy w tym celu wykorzystać naszą funkcję *CzySieDa*. Zauważmy, że w przypadku zapytania postaci (m, k, s) nie możemy użyć żadnego przedmiotu o czasie dodania a większym niż m, jak i żadnego, którego czas zabrania b jest mniejszy niż $m + s + 1$. Wszystkie inne przedmioty wykorzystać możemy — będą już w szatni i nikt nie zorientuje się na czas, że zostały zabrane. Następujący pseudokod w sposób siłowy wykorzystuje tę obserwację (tablice A i B przechowują czasy dodania i zabrania poszczególnych przedmiotów).

```
 1: function Plan(m, k, s, n, C, A, B)
 2: begin
 3:   IleOdpowiednich := 0;
 4:   for i := 1 to n do
 5:     if (A[i] ⩽ m) and (B[i] ⩾ m + s + 1) then begin
 6:       IleOdpowiednich := IleOdpowiednich + 1;
 7:       OdpowiednieC[IleOdpowiednich] := C[i];
 8:     end
 9:   return CzySieDa(k, IleOdpowiednich, OdpowiednieC);
10: end
```

Ten algorytm działa w czasie $O(n \cdot k)$ dla każdego zapytania, łącznie w czasie $O(p \cdot n \cdot k_{max})$. Treść zadania sugeruje, że ma on szansę dostać 16% punktów. Aby poprawić czas działania naszego algorytmu, musimy przestać rozpatrywać wszystkie przedmioty osobno dla każdego zapytania. Spróbujmy rozwiązać zadanie symulacyjnie — dokładać przedmioty do szatni według ich a, odpowiadać na zapytania według ich m i zabierać przedmioty według ich b. Niestety pozostawalibyśmy w momencie zapytania bez informacji, kiedy wybierane przez nas przedmioty znikną z szatni. Nie trzeba się zrażać — w testach wartych 24% punktów wszystkie plany mają równe s. Jest to równoważne sytuacji, w której dla wszystkich planów zachodzi $s = 0$ — bo wystarczy zmniejszyć o s czas zniknięcia b każdego przedmiotu. Załóżmy teraz, że każdy plan ma faktycznie $s = 0$, co sprowadzi nasze zadanie do odpowiadania, czy z przedmiotów znajdujących się w szatni w momencie m da się wybrać zbiór o łącznej

wartości k. Zauważmy, że o ile kod wewnętrznej pętli z funkcji *CzySieDa* można by bez modyfikacji przyjąć jako kod dodania nowej rzeczy do szatni, to powstałby problem przy zabieraniu przedmiotu — czy wartość **true** w $DaSie[j]$ była uzyskana tylko z wykorzystaniem tego przedmiotu, czy może bez niego, a może na oba sposoby? Aby rozwiązać ten dylemat, możemy przechowywać tam nieco inną wartość. Zamiast zapisywać, czy w ogóle da się uzyskać daną sumę, zapiszemy tam, **na ile sposobów** da się ją uzyskać. Otrzymamy następujący kod dodania przedmiotu:

```
1: procedure DodajPrzedmiot(c)
2: begin
3:   for i := k_max downto c do
4:     Sposobow[i] := Sposobow[i] + Sposobow[i − c];
5: end
```

Operacja usunięcia przedmiotu nie może oczywiście być zupełnie analogiczna; przy takim podejściu moglibyśmy otrzymać kod:

```
1: procedure UsunPrzedmiotZle(c)
2: begin
3:   for i := k_max downto c do
4:     Sposobow[i] := Sposobow[i] − Sposobow[i − c];
5: end
```

Ten kod nie działa, ponieważ zmniejsza liczbę sposobów uzyskania podzbioru o wartości i o liczbę sposobów uzyskania wartości $i - c$ z wykorzystaniem zabieranego przedmiotu. Pętla musi zostać odwrócona, a procedura zadziała poprawnie:

```
1: procedure UsunPrzedmiot(c)
2: begin
3:   for i := c to k_max do
4:     Sposobow[i] := Sposobow[i] − Sposobow[i − c];
5: end
```

Dla formalności można też zapisać kod obsługi zapytania:

```
1: function Plan(k)
2: begin
3:   return (Sposobow[k] ≠ 0);
4: end
```

To rozwiązanie powinno wywołać w czytelniku wątpliwość — czy wartości tablicy *Sposobow* nie będą rosły bardzo szybko, wymuszając użycie dużych liczb? Oczywiście tak się stanie, a implementacja dużych liczb mocno spowolniłaby rozwiązanie. Możemy poradzić sobie z tym problemem inaczej — przechowywać w tablicy *Sposobow* wartości reszt z dzielenia liczby sposobów uzyskania sumy przez dużą, ale mieszczącą się (dwukrotnie) w zakresie obliczeń, liczbę pierwszą. Występuje wtedy ryzyko, że nasz program pomyli się (jeśli przy którymś zapytaniu liczba sposobów będzie podzielna przez tę liczbę pierwszą), ale jest to zagrożenie bardzo małe (im większa liczba pierwsza, tym mniejsze). Warto więc podjąć takie ryzyko wobec przyspieszenia, jakie w ten sposób uzyskamy: rozwiązanie działa w czasie $O(n \cdot k_{max} + p)$.

Niestety, cały czas rozważaliśmy sytuację, w której wszystkie wartości s dla zapytań są równe. Gdy tak nie jest, musimy przy dodawaniu elementów przechować informację o tym, w którym momencie przedmiot zostanie zabrany. Oznacza to w praktyce, że operacje dodania i zabrania elementu nie będą symetryczne ani analogiczne. Rozbijmy je więc.

Czas dodania

W poprzednim podejściu próbowaliśmy rozpatrywać kolejne wydarzenia na osi czasu i nie ma powodu, by odejść od tego podejścia w przypadku dodawania przedmiotów do szatni. Będziemy poruszać się po osi, wykonując albo operacje dodania elementu, albo próby zrealizowania planu.

Czas zabrania

Wyobraźmy sobie inną sytuację. Goście imprezy nie przynoszą rzeczy, ale warzywa, a nasi złodzieje nie chcą kraść, ale zrobić sałatkę. Wiadomo, że data przydatności sałatki do spożycia to data przydatności najmniej trwałego składnika. Goście będą więc w momencie a przynosić składnik o wartości c i dacie przydatności w. Złodzieje w momencie m będą próbowali zrobić sałatkę o łącznej wartości k i dacie przydatności co najmniej r. Nie mamy problemu z tym zadaniem. Używając programowania dynamicznego, możemy określać przy dodawaniu każdego następnego składnika datę przydatności najtrwalszej sałatki, jaką da się uzyskać w danym momencie dla danego sumarycznego kosztu. Dodanie nowego składnika realizuje następujący kod:

```
1: procedure NowySkladnik(c, w)
2: begin
3:   for i := k_max downto c do
4:     Najtrwalsza[i] := max(Najtrwalsza[i], min(Najtrwalsza[i − c], w));
5: end
```

A zapytanie jest realizowane przez następującą funkcję:

```
1: function Salatka(k, r)
2: begin
3:   return (Najtrwalsza[k] ⩾ r);
4: end
```

Co to ma wspólnego z naszym zadaniem? Otóż wystarczy podstawić b jako w i $m + s + 1$ jako r i okazało się, że rozwiązaliśmy zadanie. Po posortowaniu wejścia, co zajmie czas $O((n + p) \cdot \log(n + p))$, pozostaje realizować fragmenty *NowySkladnik*, które zajmują czas $O(k_{max})$, i *Salatka*, które działają w czasie stałym. Łączny czas działania algorytmu to zatem $O((n + p) \cdot \log(n + p) + n \cdot k_{max} + p)$. Implementacje rozwiązania można znaleźć w plikach `sza.cpp`, `sza1.cpp`, `sza2.cpp` i `sza3.pas`.

Testy

Zadanie było sprawdzane na 12 zestawach danych testowych, patrz tabela poniżej.

Nazwa	n	p
sza1.in	20	10
sza2.in	500	10
sza3.in	777	12 372
sza4.in	567	54 336
sza5.in	198	354 420
sza6.in	187	228 608
sza7.in	200	123 200
sza8.in	98	712 068
sza9.in	666	123 504
sza10.in	777	300 312
sza11.in	950	499 700
sza12.in	1 000	1 000 000

Igor Adamski
Treść zadania, Opracowanie

Lech Duraj
Program

Dostępna pamięć: 128 MB.

OI, etap II, dzień drugi, 9.02.2012

Okropny wiersz

Bajtek musi nauczyć się na pamięć fragmentu pewnego wiersza. Wiersz, zgodnie z najlepszymi regułami sztuki współczesnej, jest długim napisem składającym się wyłącznie z małych liter alfabetu angielskiego. Brzmi oczywiście okropnie, ale nie to jest największym problemem Bajtka: przede wszystkim zapomniał on, który właściwie fragment był zadany. Wszystkie fragmenty wydają się zresztą zbyt trudne do nauczenia...

Jest jednak cień nadziei: niektóre partie wiersza wykazują pewne prawidłowości. W szczególności, czasem fragment A jest wielokrotnym powtórzeniem pewnego innego fragmentu B (innymi słowy, $A = BB\ldots B$, tzn. $A = B^k$, gdzie $k \geqslant 1$ jest liczbą całkowitą). Powiemy wtedy, że B jest **pełnym okresem** *A (w szczególności, każdy napis jest swoim pełnym okresem). Jeśli zadany fragment ma jakiś krótki pełny okres, Bajtka czeka znacznie mniej roboty. Tylko... właściwie który to był fragment?*

Zrób Bajtkowi prezent — napisz program, który wczyta pełen tekst wiersza oraz listę fragmentów, o których Bajtek podejrzewa, że mogły być tym zadanym do nauczenia, i dla każdego z nich obliczy, jaki jest jego najkrótszy pełny okres.

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita n ($1 \leqslant n \leqslant 500\,000$). W drugim wierszu znajduje się napis długości n złożony z małych liter alfabetu angielskiego — jest to tekst wiersza. Pozycje kolejnych liter numerujemy od 1 do n.

W następnym wierszu znajduje się jedna liczba całkowita q ($1 \leqslant q \leqslant 2\,000\,000$) określająca liczbę fragmentów. W kolejnych q wierszach zapisane są zapytania, po jednym w wierszu. Każdy z tych wierszy zawiera dwie liczby całkowite a_i, b_i ($1 \leqslant a_i \leqslant b_i \leqslant n$), oddzielone pojedynczym odstępem, reprezentujące zapytanie o długość najkrótszego pełnego okresu fragmentu wiersza zaczynającego się na pozycji a_i i kończącego się na pozycji b_i.

W testach wartych łącznie 42% punktów zachodzi dodatkowy warunek $n \leqslant 10\,000$. W pewnych spośród tych testów, wartych łącznie 30% punktów, zachodzi także warunek $q \leqslant 10\,000$.

Wyjście

Twój program powinien wypisać na standardowe wyjście q wierszy. W wierszu numer i powinna znaleźć się jedna liczba całkowita — odpowiedź na i-te zapytanie.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`8`	`1`
`aaabcabc`	`3`
`3`	`5`
`1 3`	
`3 8`	
`4 8`	

Rozwiązanie

Wprowadzenie

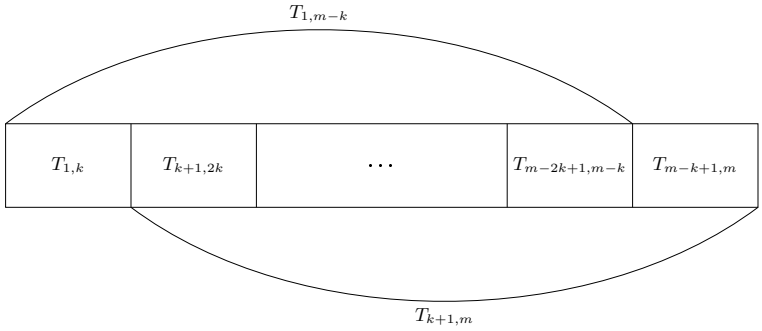
W zadaniu dane jest pewne słowo $S = s_1 s_2 \dots s_n$. Celem jest odpowiadanie na serię zapytań postaci:

$$P(i,j)\text{: } \textit{Jaki jest najkrótszy pełny okres podsłowa } S_{i,j} = s_i s_{i+1} \dots s_j\textit{?}$$

Wstępne obserwacje

Zastanówmy się na początek, w jaki sposób można szybko sprawdzić, czy dane słowo ma pełny okres o zadanej długości. Okazuje się, że zachodzi następujący fakt:

Obserwacja 1. Słowo T o długości m ma pełny okres długości k wtedy i tylko wtedy, gdy $k \mid m$ oraz $T_{1,m-k} = T_{k+1,m}$.



Rys. 1: Ilustracja równości $T_{1,m-k} = T_{k+1,m}$.

Obserwację tę można łatwo samemu uzasadnić. Jej dowód można także znaleźć np. w opracowaniu zadania *Palindromy* z XIII Olimpiady Informatycznej [13]. Na mocy obserwacji, zamiast sprawdzać, czy słowo ma pełny okres danej długości, możemy sprawdzać, czy dwa podsłowa tego słowa są sobie równe. Zajmijmy się najpierw efektywnym rozwiązaniem tego drugiego problemu.

Sprawdzanie równości dwóch podsłów

Haszowanie

Sposób ten polega na wybraniu małej liczby całkowitej p oraz dużej liczby całkowitej M i obliczeniu dla każdego sufiksu $S_{k,n}$ wartości

$$h[k] = \left(\sum_{i=k}^{n} s_i \cdot p^{i-k} \right) \bmod M.$$

Można to zrobić w czasie liniowym, korzystając z tzw. schematu Hornera:

$$h[n+1] = 0, \quad h[k] = (s_k + p \cdot h[k+1]) \bmod M \quad \text{dla} \quad k = n, n-1, \ldots, 1.$$

Teraz możemy każdemu podsłowu $S_{i,j}$ przyporządkować wartość (tzw. hasz):

$$\left(\sum_{k=i}^{j} s_i \cdot p^{k-i} \right) \bmod M = (h[i] - h[j+1] \cdot p^{j-i+1}) \bmod M.$$

Jeśli dwa podsłowa tej samej długości mają ten sam hasz, to z dużym prawdopodobieństwem podsłowa te są równe, jeśli natomiast hasze są różne, to podsłowa na pewno są różne. Po wstępnych obliczeniach w czasie $O(n)$ (spamiętanie potęg liczby p oraz obliczenie haszy sufiksów) uzyskujemy probabilistyczne rozwiązanie, które odpowiada na pytania o równość podsłów w czasie stałym. W praktyce, przy odpowiednim doborze wartości p (najczęściej mała liczba pierwsza większa od rozmiaru alfabetu) oraz M (duża liczba pierwsza), „złapanie" haszowania na błędnej odpowiedzi jest skrajnie trudne. (Za to w opisie rozwiązania zadania *Prefiksufiks* w tej książeczce można znaleźć kilka informacji o tym, w jaki sposób haszowania *nie* należy używać).

Algorytm Karpa-Millera-Rosenberga (KMR)

W tym algorytmie indeksujemy wszystkie podsłowa, których długości są potęgami dwójki, przypisując im liczby całkowite. Zachowana jest przy tym własność, że słowa są równe wtedy i tylko wtedy, gdy mają przypisane jednakowe wartości. Słowa o dowolnej długości k można porównać, znajdując najpierw takie d, dla którego $2^d \leqslant k < 2^{d+1}$, a następnie porównując identyfikatory prefiksów i sufiksów słów o długości 2^d. Tablicę indeksów buduje się w czasie $O(n \log n)$ albo $O(n \log^2 n)$, w zależności od użytego algorytmu sortowania. Wówczas odpowiedzi na pytanie o równość podsłów można udzielać w czasie stałym. Dokładniejszy opis działania oraz konstrukcji tej struktury danych można znaleźć np. w opracowaniu zadania *Powtórzenia* z VII Olimpiady Informatycznej [7] albo zadania *Korale* z XVII Olimpiady [17].

Odpowiedź na zapytanie

Umiemy już szybko odpowiadać na pytania postaci:

$P(i, j, k)$: *Czy słowo $S_{i,j}$ ma pełny okres długości k?*

Wykorzystamy to do udzielania odpowiedzi na oryginalne zapytania $P(i, j)$. Dla wygody oznaczmy $W = S_{i,j}$ i $m = |W|$, tj. $m = j - i + 1$.

Rozwiązanie wolne $O(\sqrt{m})$

Wiedząc, że długość każdego pełnego okresu musi dzielić długość całego słowa, możemy sprawdzić wszystkie dzielniki m i wybrać najmniejszy, dla którego istnieje pełny okres o tej długości. Wszystkie dzielniki liczby m możemy znaleźć w czasie $O(\sqrt{m})$, podobną metodą jak proste sprawdzanie, czy liczba jest pierwsza. Stąd otrzymujemy złożoność tego rozwiązania. Na zawodach otrzymywało około 65-75 punktów. W plikach `okrs0.cpp` i `okrs5.pas` można znaleźć implementację wykorzystującą haszowanie, a w pliku `okrs1.cpp` wykorzystany został algorytm Karpa-Millera-Rosenberga.

Rozwiązanie trochę szybsze

Powyższe rozwiązanie można usprawnić, generując i zapamiętując wcześniej wszystkie dzielniki dla każdej liczby od 1 do n. Dzięki temu odpowiedź na pojedyncze zapytanie wykonujemy w czasie zależnym dokładnie od liczby dzielników długości słowa, zamiast w pesymistycznym czasie $\Theta(\sqrt{m})$. Jest to znaczące usprawnienie, ponieważ liczba dzielników jest zazwyczaj istotnie mniejsza niż pierwiastek (zainteresowany Czytelnik więcej informacji na ten temat znajdzie np. w opracowaniu zadania *Sejf* z XVIII Olimpiady Informatycznej [18]). Implementacja rozwiązania znajduje się w pliku `okrs9.cpp`. Na zawodach za tego typu rozwiązanie można było otrzymać nawet ponad 90 punktów, gdyż bardzo ciężko ułożyć testy, które odcinałyby je od rozwiązania wzorcowego.

Rozwiązanie wzorcowe $O(\log m)$

Do rozwiązania wzorcowego potrzebujemy jeszcze dwóch prostych obserwacji:

Obserwacja 2. Jeśli słowo długości m nie ma pełnego okresu długości k, to nie ma też żadnego pełnego okresu długości l, gdzie $l \mid k$.

Dowód jest trywialny (np. przez sprzeczność).

Obserwacja 3. Jeśli słowo U jest pełnym okresem słowa T, to najkrótszy pełny okres T jest równy najkrótszemu pełnemu okresowi U.

Dowód tej obserwacji można znaleźć we wspomnianym już omówieniu zadania *Palindromy* z XIII Olimpiady [13].

Niech k będzie czynnikiem pierwszym, który dzieli m w maksymalnej potędze α. Wykorzystajmy wcześniejsze obserwacje:

- jeśli W ma pełny okres długości $\frac{m}{k}$, to z obserwacji 3 możemy obliczyć najkrótszy pełny okres słowa $W_{1,\frac{m}{k}}$; będzie on także najkrótszym pełnym okresem W;
- jeśli natomiast W nie ma pełnego okresu długości $\frac{m}{k}$, to z obserwacji 2 wiemy, że nie ma też żadnego pełnego okresu długości l, gdzie $l \mid \frac{m}{k}$. Zauważmy, że dzielniki m, które nie są dzielnikami $\frac{m}{k}$, muszą być podzielne przez k^α.

Możemy już skonstruować algorytm wzorcowy. Będziemy trzymać dwie wartości: m — długość aktualnie rozpatrywanego słowa, oraz liczbę l — iloczyn czynników, przez które musi być podzielna długość najkrótszego pełnego okresu. Zaczynamy z $m = |S_{i,j}|$ oraz $l = 1$ i wykonujemy kroki zgodnie z powyższymi punktami:

1. Znajdujemy czynnik pierwszy k, który dzieli $\frac{m}{l}$, oraz maksymalną potęgę α, w której dzieli.

2. Jeśli aktualne słowo ma pełny okres długości $\frac{m}{k}$, to zmniejszamy długość słowa do tej wartości.

3. Jeśli nie, to domnażamy do l wartość k^α.

Algorytm zatrzymuje się w momencie, gdy $m = l$. Jest to wówczas długość najkrótszego pełnego okresu wyjściowego słowa.

Podczas działania potrzebujemy często znajdować jakiś dzielnik pierwszy danej liczby. Możemy to zrealizować w czasie stałym, jeśli na samym początku użyjemy sita Eratostenesa do stablicowania po jednym dzielniku pierwszym każdej liczby od 2 do n, w czasie $O(n \log\log n)$. Opis tej metody można znaleźć np. w opracowaniu zadania *Zapytania* z XIV Olimpiady Informatycznej [14].

Złożoność czasowa rozwiązania wynika z faktu, że w każdym kroku albo wartość m jest dzielona przez co najmniej 2, albo wartość l jest przemnażana przez co najmniej 2, więc liczba kroków wykonanych do momentu, w którym wartości te będą równe, nie przekroczy $\log m$.

W plikach `okr.cpp` oraz `okr2.pas` znajduje się implementacja tego rozwiązania używająca haszowania, działająca w całkowitym czasie $O(n \log\log n + q \log n)$. W pliku `okr1.cpp` znajduje się rozwiązanie korzystające z algorytmu Karpa-Millera-Rosenberga, działające w czasie $O(n \log^2 n + q \log n)$. Oba rozwiązania uzyskiwały na zawodach komplet punktów.

Inne rozwiązania wolne

W plikach `okrs4.cpp` i `okrs8.pas` zaimplementowano rozwiązanie, które nie używa żadnej struktury danych do porównywania podsłów. Sprawdza po prostu wszystkie dzielniki długości słowa i dla każdego iteruje się po całym słowie do pierwszej niezgodności. Pesymistyczny czas działania wynosi $O(qn\sqrt{n})$. Na zawodach takie rozwiązanie otrzymywało 20-30 punktów.

Istnieją także rozwiązania zadania korzystające z tablicy prefikso-sufiksów[1]. Oba rozwiązania opierają się na pewnym wariancie obserwacji 1: długość najkrótszego pełnego okresu jest równa długości słowa minus długość najdłuższego jego prefikso-sufiksu, jeśli ta liczba dzieli długość słowa, a w przeciwnym razie najkrótszym pełnym okresem jest całe słowo. Wyznaczenie długości najdłuższego prefikso-sufiksu zajmuje czas liniowy względem długości słowa.

Pierwsze rozwiązanie, zaimplementowane w plikach `okrs3.cpp` i `okrs7.pas`, oblicza dla każdego zapytania tablicę prefikso-sufiksową dla podsłowa $S_{i,j}$ i następnie odpowiada zgodnie z powyższą obserwacją. Czas działania wynosi $O(qn)$. Takie rozwiązanie otrzymywało 30-40 punktów.

Drugie rozwiązanie wczytywało wszystkie zapytania i grupowało je po początkach podsłów. Następnie dla każdego sufiksu całego słowa obliczało tablicę prefikso-sufiksów i zapamiętywało wynik dla wszystkich zapytań, których podsłowa zaczynały

[1] O tej tablicy można przeczytać (w związku z algorytmem KMP) np. w książkach [21, 23].

się na tej samej pozycji, co ten sufiks. Dzięki temu grupowaniu złożoność rozwiązania wynosi $O(n^2)$. Takie rozwiązania otrzymywały 40-50 punktów. Implementację można znaleźć w plikach `okrs2.cpp` i `okrs6.pas`.

Testy

Przy konstrukcji testów używano następujących, charakterystycznych fragmentów:

- powtórzony i zaburzony napis losowy: krótki ciąg przypadkowych znaków, powtórzony wielokrotnie (niekoniecznie całkowitą liczbę razy), zmieniony w przypadkowym miejscu (w szczególności, powtarzany ciąg znaków stosunkowo często był jednoelementowy);
- losowa wieża: test o strukturze rekurencyjnej; w zależności od wyniku losowania wieża jest sklejeniem dwóch mniejszych wież lub powtórzoną mniejszą wieżą;
- wysoka wieża: jw., tylko wieża jest zawsze powtórzoną mniejszą wieżą, liczba powtórzeń jest między 2 a 3, aby zapewnić dużą głębokość zagnieżdżenia.
- łańcuch: słowo postaci $(\mathtt{a}^{s_1}\mathtt{b}^{t_1})^{k_1}(\mathtt{b}^{s_2}\mathtt{c}^{t_2})^{k_2}(\mathtt{c}^{s_3}\mathtt{d}^{t_3})^{k_3}\ldots$

Słowa występujące w poszczególnych testach były zazwyczaj zbudowane z kilku sklejonych fragmentów różnych typów. W znacznej części zapytań długość słowa była liczbą o dużej liczbie dzielników, co pozwalało spowolnić rozwiązania sprawdzające wszystkie dzielniki. Wszystkie testy zestawione są w poniższej tabeli.

Nazwa	n	q
okr1a.in	72	8
okr1b.in	1	1
okr1c.in	12	4
okr2.in	186	245
okr3.in	2 714	3 042
okr4a.in	8 200	9 999
okr4b.in	10 000	9 901
okr5a.in	9 900	9 992
okr5b.in	10 000	9 889
okr6a.in	9 001	90 992
okr6b.in	10 000	100 000
okr7a.in	7 691	98 227
okr7b.in	10 000	100 000

Nazwa	n	q
okr8.in	40 000	80 006
okr9.in	63 480	99 119
okr10.in	81 490	84 962
okr11.in	90 006	91 037
okr12.in	240 176	500 300
okr13.in	299 400	590 856
okr14.in	370 488	983 865
okr15a.in	500 000	968 228
okr15b.in	500 000	2 000 000
okr16a.in	466 530	975 537
okr16b.in	500 000	2 000 000

Karol Pokorski
Treść zadania, Opracowanie

Michał Zgliczyński
Program

Dostępna pamięć: 64 MB.

OI, etap II, dzień drugi, 9.02.2012

Rozkład Fibonacciego

Ciąg liczb Fibonacciego to ciąg liczb całkowitych zdefiniowany rekurencyjnie w następujący sposób:

$$Fib_0 = 0, \quad Fib_1 = 1, \quad Fib_n = Fib_{n-2} + Fib_{n-1} \text{ dla } n > 1.$$

Początkowe elementy tego ciągu to: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Bajtazar bada, w jaki sposób różne liczby można przedstawić jako sumy lub różnice liczb Fibonacciego. Aktualnie zastanawia się, jak daną dodatnią liczbę całkowitą k przedstawić jako sumę lub różnicę jak najmniejszej liczby (niekoniecznie różnych) liczb Fibonacciego. Na przykład, liczby 10, 19, 17 i 1070 można przedstawić minimalnie, odpowiednio, za pomocą 2, 2, 3 oraz 4 liczb Fibonacciego:

$$\begin{aligned} 10 &= 5 + 5 \\ 19 &= 21 - 2 \\ 17 &= 13 + 5 - 1 \\ 1070 &= 987 + 89 - 5 - 1 \end{aligned}$$

Pomóż Bajtazarowi! Napisz program, który dla danej dodatniej liczby całkowitej k wyznaczy minimalną liczbę liczb Fibonacciego potrzebnych do przedstawienia liczby k jako ich sumy lub różnicy.

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita p ($1 \leqslant p \leqslant 10$) oznaczająca liczbę zapytań. W kolejnych p wierszach znajduje się po jednej dodatniej liczbie całkowitej k ($1 \leqslant k \leqslant 4 \cdot 10^{17}$).

Wyjście

Twój program powinien wypisać na standardowe wyjście dla każdego zapytania minimalną liczbę liczb Fibonacciego potrzebnych do przedstawienia liczby k jako ich sumy lub różnicy.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
1 1070	4

Rozwiązanie

Wprowadzenie

Dana jest dodatnia liczba całkowita k. *Rozkładem* liczby k nazwiemy dowolną parę multizbiorów (P_+, P_-) spełniającą warunek:

$$\sum_{i \in P_+} Fib_i - \sum_{i \in P_-} Fib_i = k.$$

Rozmiarem rozkładu nazwiemy sumę rozmiarów multizbiorów P_+ i P_-. Rozkład nazwiemy *optymalnym*, jeśli nie istnieje żaden inny rozkład liczby k o mniejszym rozmiarze.

Pierwsze rozwiązania

Do uzyskania poprawnego, ale powolnego rozwiązania wystarczy tylko kilka dosyć intuicyjnych obserwacji. Przede wszystkim możemy bez straty ogólności przyjąć, że wszystkie elementy multizbiorów P_+ i P_- (czyli indeksy liczb Fibonacciego w rozkładzie) są nie mniejsze niż 2. Dalej:

Obserwacja 1. Jeśli w rozkładzie liczby istnieje indeks i, taki że $i \in P_+$ oraz $i \in P_-$, to rozkład ten nie jest optymalny.

Faktycznie, rozkład opisany w powyższej obserwacji można polepszyć, usuwając z obu multizbiorów po jednym wystąpieniu indeksu i.

Chociaż warunki zadania dopuszczają wykorzystanie jednej liczby Fibonacciego więcej niż raz, wydaje się naturalne, że w ten sposób nie otrzyma się rozkładu optymalnego. Przykład z treści zadania ($10 = 5 + 5$) zdaje się temu przeczyć, jednak tę samą liczbę możemy także przedstawić inaczej, tym razem już bez powtórzeń: $10 = 2 + 8$.

Obserwacja 2. Rozkład, w którym jakaś liczba Fibonacciego występuje co najmniej trzykrotnie, nie jest optymalny. Co więcej, dla każdej liczby istnieje rozkład optymalny, w którym każda liczba Fibonacciego występuje co najwyżej raz.

Dowód: Aby uzasadnić powyższe, wystarczą proste rachunki z wykorzystaniem definicji rekurencyjnej ciągu Fibonacciego:

$$\begin{aligned} 3 \cdot Fib_i &= (Fib_{i-2} + Fib_{i-1}) + Fib_i + Fib_i = Fib_{i-2} + (Fib_{i-1} + Fib_i) + Fib_i \qquad (1) \\ &= Fib_{i-2} + (Fib_{i+1} + Fib_i) = Fib_{i-2} + Fib_{i+2}; \\ 2 \cdot Fib_i &= (Fib_{i-2} + Fib_{i-1}) + Fib_i = Fib_{i-2} + (Fib_{i-1} + Fib_i) = Fib_{i-2} + Fib_{i+1}. \qquad (2) \end{aligned}$$

Za pomocą operacji (1) każde potrójne wystąpienie liczby Fibonacciego można zastąpić wystąpieniem tylko dwóch liczb Fibonacciego, co oczywiście zmniejsza rozmiar

rozkładu. Taka operacja nie może więc być wykonalna w żadnym rozkładzie optymalnym. Natomiast za pomocą operacji (2) możemy sukcesywnie eliminować wszystkie podwójne wystąpienia liczb Fibonacciego z rozkładu, nie pogarszając rozmiaru rozkładu. Należy tu jednak być ostrożniejszym, gdyż po wykonaniu operacji (2) z kolei liczby Fib_{i-2} i Fib_{i+1} mogą występować w rozkładzie wielokrotnie, więc potencjalnie moglibyśmy uzyskać nieskończoną sekwencję operacji usuwania podwójnych wystąpień. Warto jednak zauważyć, że każda taka operacja zmniejsza o jeden *sumę indeksów* liczb Fibonacciego występujących w rozkładzie. To pozwala upewnić się, że operacji drugiego typu wykonamy zawsze tylko skończenie wiele i na końcu każda liczba Fibonacciego wystąpi w rozkładzie co najwyżej raz. (Dodajmy jeszcze, że jeśli po wykonaniu którejś operacji to liczba Fib_2 występuje dwukrotnie, możemy od razu zamienić oba jej wystąpienia na Fib_3 i orzec, że rozkład nie był optymalny). ■

Z dotychczasowych rozważań wynika, że możemy skupić się na poszukiwaniu *zbiorów* (a nie multizbiorów) P_+ i P_-, w których łącznie każdej liczby Fibonacciego używamy co najwyżej raz. W dalszej części opisu będą nas interesować tylko takie rozkłady.

Dalej, nietrudno uwierzyć, że w rozkładzie optymalnym liczby nie opłaca nam się używać liczb Fibonacciego istotnie większych od tej liczby. To stwierdzenie jest sprecyzowane w poniższej obserwacji. Dowód obserwacji pomijamy — wyniknie on z dowodu poprawności rozwiązania wzorcowego.

Obserwacja 3. Jeśli $Fib_m \leqslant k < Fib_{m+1}$, to istnieje rozkład optymalny liczby k, w którym indeksy liczb Fibonacciego nie przekraczają $m+1$.

Możemy już teraz zaproponować pierwsze rozwiązanie. Wystarczy iterować po kolejnych indeksach $i = 2, 3, \ldots, m+1$ dla m określonego jak w obserwacji 3 i dla każdego z nich rozpatrzyć trzy możliwości: albo $i \in P_+$, albo $i \in P_-$, albo $i \notin P_+$ oraz $i \notin P_-$. Po każdej sekwencji wyborów sprawdzamy, czy otrzymaliśmy rozkład liczby k, a jeśli tak, czy jest to rozkład zawierający mniej liczb Fibonacciego niż najlepszy dotychczas znaleziony. Jest to, oczywiście, rozwiązanie wykładnicze, ale wykładnicze względem m; w każdym kroku mamy trzy możliwe wybory, więc złożoność czasowa tego rozwiązania to $O(3^m)$.

Aby przekonać się, jaka jest złożoność czasowa tego rozwiązania względem k, warto przypomnieć sobie znany wzór Bineta na n-tą liczbę Fibonacciego:

$$Fib_n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n.$$

Pierwsze z wyrażeń w nawiasie, równe w przybliżeniu 1,618, nazywa się zwyczajowo *złotym podziałem* i oznacza grecką literą ϕ. Z kolei drugie z wyrażeń jest równe $1-\phi \approx -0{,}618$, więc dowolne jego potęgi o naturalnym wykładniku są zawsze mniejsze co do wartości bezwzględnej niż 1. Widzimy więc, że n-ta liczba Fibonacciego jest równa, z dokładnością do stałego czynnika, ϕ^n. Mamy stąd $k \approx \phi^m$, czyli $m \approx \log_\phi k$, przy czym ta ostatnia równość zachodzi z dokładnością do stałej. Ostatecznie:

$$3^m = O\big(3^{\log_\phi k}\big) = O\big(3^{\log_3 k / \log_3 \phi}\big) = O\big((3^{\log_3 k})^{\log_\phi 3}\big) = O\big(k^{\log_\phi 3}\big).$$

Widzimy zatem, że nasze rozwiązanie jest w rzeczywistości wielomianowe względem k, przy czym wykładnik wielomianu znajduje się gdzieś pomiędzy 2 a 3 (ponieważ, jak łatwo sprawdzić z użyciem kalkulatora, $1{,}618^2 \approx 2{,}618 < 3$ i $1{,}618^3 \approx 4{,}236 > 3$). Dokładniejsze obliczenia pokazują, że $\log_\phi 3 \approx 2{,}283$.

Bezpośrednia implementacja tego rozwiązania (zawarta w plikach `rozs1.cpp` i `rozs2.cpp`) pozwalała uzyskać 12%–24% maksymalnej punktacji.

Usprawnienia

Podane rozwiązanie można znacząco usprawnić, używając pewnych klasycznych technik algorytmicznych. Techniki te podajemy raczej w charakterze dygresji, gdyż nie były one potrzebne w rozwiązaniu wzorcowym.

Jednym z pomysłów jest zastosowanie metody *meet in the middle*. W naszym przypadku polega ona na tym, aby zbiór indeksów dostępnych liczb Fibonacciego, tj. $\{2, 3, \ldots, m+1\}$, podzielić na dwa rozłączne zbiory o rozmiarze jak najbliższym $m/2$. Dla każdego z tych zbiorów możemy wygenerować $O(3^{m/2})$ możliwych kombinacji wyborów odpowiednich liczb, tworząc listę par uporządkowanych (s, r), gdzie s jest sumą, którą można uzyskać, dodając/odejmując pewne r różnych liczb Fibonacciego z danego zbioru. Możemy oczywiście dla każdego s zapamiętać za każdym razem tylko jedną parę — tę z minimalną wartością parametru r. Teraz wystarczy złożyć w całość wyniki dla obu zbiorów. W tym celu można posortować pary uzyskane dla każdego ze zbiorów i dla każdej pary (s_1, r_1) z pierwszego zbioru próbować wyszukać odpowiadającą jej parę postaci $(k - s_1, r_2)$ z drugiego zbioru. W ten sposób otrzymujemy rozkład zawierający $r_1 + r_2$ liczb Fibonacciego. Poszukiwanie pary $(k - s_1, r_2)$ można, na przykład, zrealizować wyszukiwaniem binarnym, co prowadzi do rozwiązania o złożoności czasowej $O(3^{m/2} m/2) = O(k^{(\log_\phi 3)/2} \log k)$, czyli $O(k^{1{,}142} \log k)$. Alternatywnie, odpowiadające sobie wzajemnie pary można zidentyfikować w jednym sprytnym przejściu po dwóch posortowanych listach. Takie rozwiązanie zostało zaimplementowane w pliku `rozs3.cpp`. Uzyskiwało ono 36% maksymalnej punktacji.

Inne podejście polega na dostrzeżeniu analogii między naszym problemem a problemem wydawania reszty. Liczba k jest tu kwotą do wydania, a liczby Fibonacciego (oraz liczby przeciwne do nich) są nominałami, którymi dysponujemy. Chcemy wydać k, używając jak najmniej spośród $2m$ dostępnych nominałów, których suma wartości bezwzględnych jest rzędu $O(k)$. Prosty algorytm oparty na programowaniu dynamicznym rozwiązuje ten problem w czasie $O(km)$, czyli, w naszym przypadku, $O(k \log k)$. Umiejętna implementacja tego algorytmu (patrz np. plik `rozs4.cpp`) pozwalała uzyskać 48% punktów.

Rozwiązanie wzorcowe

Z pomocą różnych sztuczek algorytmicznych udało nam się otrzymać rozwiązania działające w czasie zbliżonym do liniowego względem rozkładanej liczby k. Aby uzyskać coś istotnie szybszego, o czasie działania logarytmicznym (bądź polilogarytmicznym) względem k, potrzebujemy jeszcze kilku spostrzeżeń. Zacznijmy od następującej, bardzo prostej obserwacji, która posłuży nam do dowodu ważnej własności rozkładów optymalnych (twierdzenie 1).

Obserwacja 4. Niech (P_+, P_-) będzie rozkładem liczby k. Jeśli dla pewnego i zachodzi:

(a) $i \in P_+$ oraz $i+1 \in P_+$ lub

(b) $i \in P_-$ oraz $i+1 \in P_-$ lub

(c) $i \in P_-$ oraz $i+1 \in P_+$ lub

(d) $i \in P_-$ oraz $i+2 \in P_+$

to rozkład (P_+, P_-) nie jest rozkładem optymalnym.

Twierdzenie 1. *Jeśli $Fib_m \leqslant k < Fib_{m+1}$, to istnieje optymalny rozkład liczby k, w którym $m \in P_+$ lub $m+1 \in P_+$.*

Dowód: Dla $k \leqslant 2$ teza twierdzenia w oczywisty sposób zachodzi. Załóżmy zatem, że twierdzenie nie jest prawdziwe dla pewnej liczby $k \geqslant 3$, i rozważmy jakikolwiek rozkład optymalny (P_+, P_-) liczby k (w którym P_+ i P_- są zbiorami). Zapytajmy, ile wynosi $z = \max P_+$. Wiemy, że $z \neq m$ i $z \neq m+1$.

Załóżmy, że $z \leqslant m-1$. Na mocy obserwacji 4a, największą sumą rozkładu, jaką możemy wówczas uzyskać, jest:

$$Fib_{m-1} + Fib_{m-3} + Fib_{m-5} + \ldots$$

Powyższa suma kończy się składnikiem Fib_2 lub Fib_3, w zależności od parzystości m, i jest równa $Fib_m - Fib_1$ lub, odpowiednio, $Fib_m - Fib_2$ (czyli w obu przypadkach tyle samo). Ponieważ $k \geqslant Fib_m$, więc ten przypadek nie może zachodzić.

Załóżmy teraz, że $z \geqslant m+2$. Podobnie jak poprzednio, rozkład optymalny o najmniejszej sumie, jaki można uzyskać przy tym założeniu, to:

$$Fib_{m+2} - Fib_{m-1} - Fib_{m-3} - \ldots > Fib_{m+1}.$$

Tym razem skorzystaliśmy z obserwacji 4bcd. Ponieważ $k < Fib_{m+1}$, więc ten przypadek również nie może zachodzić.

Wykazaliśmy zatem, że żaden z przypadków $z \notin \{m, m+1\}$ nie jest możliwy. Otrzymana sprzeczność dowodzi tezy twierdzenia. ■

Powyższe twierdzenie istotnie zawęża klasę rozkładów, jakie musimy rozważać. Ale nie tylko. Jeśli założenia twierdzenia są spełnione, wówczas każda z liczb $k - Fib_m$, $Fib_{m+1} - k$ jest nie większa niż Fib_{m-1} (ponieważ $Fib_{m+1} - Fib_m = Fib_{m-1}$), więc twierdzenie natychmiast implikuje podaną wcześniej bez dowodu obserwację 3. Z tego samego spostrzeżenia wynika również algorytm o złożoności $O(2^{m/2})$, w którym w każdym kroku rozważamy dwie możliwości: albo do rozkładu wybieramy Fib_m, albo Fib_{m+1}. Tego typu rozwiązania (patrz plik `rozs5.cpp`) uzyskiwały na zawodach 87%–100% punktów.

Twierdzenie 1 wciąż dopuszcza pewną niejednoznaczność. Okazuje się, że w zależności od tego, która z liczb Fib_m, Fib_{m+1} jest bliższa k, możemy dokładnie określić, którą z liczb m, $m+1$ warto umieścić w zbiorze P_+. Prawdziwe są bowiem dwa kolejne, „siostrzane" twierdzenia.

Twierdzenie 2. *Załóżmy, że* $Fib_m \leqslant k < Fib_{m+1}$. *Jeśli* $k - Fib_m \leqslant Fib_{m+1} - k$, *to istnieje rozkład optymalny, w którym* $m \in P_+$.

Dowód: W dowodzie wystarczy ograniczyć się do przypadku, gdy $k \geqslant 3$. Oznaczmy $a = k - Fib_m$, $b = Fib_{m+1} - k$. Zauważmy, że jeśli w rozkładzie liczby k użyjemy składnika Fib_m, to pozostanie nam do rozłożenia liczba a, a jeśli użyjemy Fib_{m+1}, pozostanie nam do rozłożenia b (lub $-b$, ale to na to samo wychodzi). Należy więc wykazać, że w przypadku, gdy $a \leqslant b$, liczba składników w optymalnym rozkładzie liczby a jest nie większa niż liczba składników niezbędnych do rozłożenia liczby b. Zauważmy, że $a + b = Fib_{m-1}$. Rozważmy dwa przypadki:

- $a < Fib_{m-3}$, wtedy $b > Fib_{m-2}$. Na mocy twierdzenia 1, próbując rozłożyć liczbę b, powinniśmy użyć składnika Fib_{m-1} lub Fib_{m-2}. W pierwszym z tych przypadków pozostajemy z liczbą a do rozłożenia (wykorzystawszy dwa składniki, co jest nieopłacalne, bowiem do takiej samej sytuacji mogliśmy dojść w jednym ruchu). Jeśli zaś użyjemy składnika Fib_{m-2}, będziemy mieć dalej do rozłożenia liczbę $b - Fib_{m-2}$, jednak zauważmy, że $b - Fib_{m-2} = Fib_{m-1} - a - Fib_{m-2} = Fib_{m-3} - a$, a taki ruch mielibyśmy do wykonania także z a. W obu przypadkach nie opłaca nam się rozkładać liczby b.
- $a \geqslant Fib_{m-3}$, wtedy $Fib_{m-3} \leqslant a \leqslant b \leqslant Fib_{m-2}$. Na mocy twierdzenia 1 mamy do rozważenia dwie możliwości: następnym składnikiem rozkładu (zarówno w przypadku, gdy próbujemy dalej rozkładać a, jak i b) może być albo Fib_{m-3}, albo Fib_{m-2}. Jednakże $a - Fib_{m-3} = Fib_{m-2} - b$ oraz $Fib_{m-2} - a = b - Fib_{m-3}$. Stąd, zarówno z a, jak i z b mamy takie same możliwości dalszego rozkładu.

Ostateczna konkluzja jest następująca: rozłożenie liczby a będzie wymagało zawsze nie więcej operacji niż rozłożenie b. Można zatem bezpiecznie dodać m do zbioru P_+ i rozłożyć optymalnie to, co pozostało. ■

Twierdzenie 3. *Załóżmy, że* $Fib_m \leqslant k < Fib_{m+1}$. *Jeśli* $k - Fib_m > Fib_{m+1} - k$, *to istnieje rozkład optymalny, w którym* $m + 1 \in P_+$.

Dowód: Analogiczny do dowodu twierdzenia 2. ■

Dwa ostatnie twierdzenia podpowiadają dla każdego k zachłanny ruch, jaki należy wykonać. Jeśli przed wykonaniem tego ruchu zachodzi $Fib_m \leqslant k < Fib_{m+1}$ (i, powiedzmy, $m > 4$), to pozostała do rozłożenia liczba k' po wykonaniu ruchu nie przekracza $\max(k - Fib_m, Fib_{m+1} - k)$, czyli:

$$k' \leqslant \frac{Fib_{m+1} - Fib_m}{2} = \frac{Fib_{m-1}}{2} = \frac{Fib_{m-2} + Fib_{m-3}}{2} < \frac{2Fib_{m-2}}{2} = Fib_{m-2}.$$

To pokazuje, że po mniej więcej $m/3$ ruchach otrzymamy pełny rozkład liczby k.

W zależności od tego, jak efektywnie zaimplementujemy wykonywanie pojedynczego ruchu, otrzymamy rozwiązanie o złożoności czasowej $O(m^2) = O(\log^2 k)$ lub $O(m) = O(\log k)$. Każde z tych rozwiązań uzyskuje, oczywiście, maksymalną punktację. Stosowne implementacje można znaleźć w plikach `roz.cpp`, `roz1.pas`, `roz2.cpp` i `roz3.pas`. Poniżej znajduje się pseudokod rozwiązania działającego w czasie $O(\log k)$.

```
1:  function rozkład(k)
2:  begin
3:    x := 1;   y := 1;
4:    while (y < k) do begin
5:      z := x + y;
6:      x := y;   y := z;
7:    end
8:    r := 0;
9:    while (k > 0) do begin
10:     { Niezmiennik: x ⩽ k < y, x i y to dwie kolejne liczby Fibonacciego }
11:     if (k − x ⩽ y − k) then k := k − x else k := y − k;
12:     r := r + 1;
13:     while (x ⩾ k) do begin
14:       z := y − x;
15:       y := x;   x := z;
16:     end
17:   end
18:   return r;
19: end
```

Testy

Rozwiązania zawodników były sprawdzane za pomocą 8 testów. Poniższa tabela zawiera charakterystykę poszczególnych testów: p oznacza liczbę zapytań w danym teście, a K — maksymalną liczbę występującą w zapytaniu.

Nazwa	p	K
roz1.in	10	75
roz2.in	10	500
roz3.in	6	34 130
roz4.in	7	800 000

Nazwa	p	K
roz5.in	10	1 000 000 000
roz6.in	10	1 000 000 000 000
roz7.in	10	100 000 000 000 000
roz8.in	10	400 000 000 000 000 000

Zawody III stopnia

opracowania zadań

Jakub Wojtaszczyk — Treść zadania, Opracowanie
Alan Kutniewski — Program

Dostępna pamięć: 64 MB. *OI, etap III, dzień próbny, 27.03.2012*

Squarki

Znany bajtocki fizyk Bajtazar bada nową postać materii — **squarki**. *Są to bardzo egzotyczne cząstki, które nigdy nie występują pojedynczo, zawsze w parach. Co więcej, squarki danego rodzaju łączą się w pary tylko ze squarkami innych rodzajów.*

Po latach badań Bajtazar ustalił, że istnieje n różnych rodzajów squarków. Squarki każdego rodzaju mają inną masę, wyrażającą się dodatnią liczbą całkowitą. Bajtazar zmierzył też łączną masę każdej z $\frac{n(n-1)}{2}$ *możliwych par squarków. Zgodnie z bajtockimi prawami fizyki, masa pary squarków jest równa po prostu sumie mas squarków wchodzących w skład pary.*

Teraz Bajtazar chciałby ustalić masy pojedynczych squarków poszczególnych rodzajów. Bajtazar poprosił Cię o pomoc w napisaniu programu, który wyznaczy wszystkie rozwiązania tego problemu, tj. odtworzy wszystkie możliwe zestawy mas squarków poszczególnych rodzajów.

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita n $(3 \leqslant n \leqslant 300)$ *oznaczająca liczbę różnych rodzajów squarków. W drugim wierszu znajduje się* $\frac{n(n-1)}{2}$ *dodatnich liczb całkowitych pooddzielanych pojedynczymi odstępami, oznaczających masy wszystkich możliwych par squarków. Masa żadnej pary squarków nie przekracza* 10^8. *Dla każdych dwóch różnych rodzajów squarków, masa pary squarków tych rodzajów jest podana na wejściu dokładnie raz. Masy na wejściu są wymienione w przypadkowej kolejności.*

W testach wartych łącznie 32% punktów zachodzą dodatkowe warunki: $n \leqslant 20$ *oraz masa żadnej pary squarków nie przekracza 2 000.*

Wyjście

W pierwszym wierszu standardowego wyjścia Twój program powinien wypisać liczbę k *możliwych rozwiązań tego problemu. W następnych* k *wierszach powinny znaleźć się kolejne rozwiązania, po jednym w wierszu. Każde z rozwiązań powinno składać się z* n *różnych liczb całkowitych dodatnich podanych w kolejności rosnącej, pooddzielanych pojedynczymi odstępami, oznaczających masy squarków poszczególnych rodzajów.*

Rozwiązania można wypisać w dowolnej kolejności. Rozwiązania nie mogą się powtarzać. Możesz założyć, że dla każdych danych testowych istnieje przynajmniej jedno rozwiązanie, tzn. $k > 0$.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`4`	`1`
`3 5 4 7 6 5`	`1 2 3 4`

a dla danych:	*poprawnym wynikiem jest:*
`4`	`2`
`11 17 12 20 21 15`	`4 7 8 13`
	`3 8 9 12`

Rozwiązanie

Wprowadzenie

Przez m_A oznaczmy masę squarka A, zaś przez $m_{A,B}$ — masę pary squarków A i B, czyli (zgodnie z warunkami zadania) $m_A + m_B$.

Wpierw wyobraźmy sobie, że wiemy, która liczba z wejścia opisuje którą parę squarków. Wtedy zadanie sprowadza się po prostu do rozwiązania układu $n(n-1)/2$ równań liniowych z n niewiadomymi. Istnieją ogólne sposoby rozwiązywania takich układów (np. tzw. eliminacja Gaussa), ale w naszym przypadku można poradzić sobie jeszcze prościej. Przykładowo, aby wyznaczyć masę pewnego squarka A, można dobrać dowolne dwa inne squarki B i C i zauważyć, że

$$m_A = \frac{2m_A}{2} = \frac{m_A + m_B + m_A + m_C - m_B - m_C}{2} = \frac{m_{A,B} + m_{A,C} - m_{B,C}}{2}. \quad (1)$$

Jako że wszystkie liczby po prawej stronie są nam znane, to z tego wzoru możemy wyznaczyć niewiadomą m_A.

Tu dwie uwagi. Po pierwsze, powyższe spostrzeżenie gwarantuje, że liczba rozwiązań jest ograniczona — dla każdego przypisania liczbom na wejściu par squarków jest co najwyżej jedno rozwiązanie (czyli będzie co najwyżej $(n(n-1)/2)!$ rozwiązań). Po drugie, wybierając trzy różne squarki, wykorzystujemy fakt, że $n \geqslant 3$. Gdyby n mogło być równe dwa, to na wejściu mielibyśmy tylko jedną masę pary squarków i musielibyśmy radzić sobie z bardzo dużą liczbą rozwiązań (choć, oczywiście, dość prostych do wyznaczenia).

W ten sposób otrzymujemy pierwszy pomysł na rozwiązanie. Sprawdzamy wszystkie możliwe dopasowania liczb na wejściu do par squarków i każde z nich rozpatrujemy jak wyżej. Niestety, jak już zauważyliśmy, tych możliwych dopasowań jest trochę dużo, więc nawet dla niewielkich wartości n to rozwiązanie nie jest szczególnie realne.

Pierwsze pomysły

Po chwili zastanowienia jesteśmy w stanie istotnie ograniczyć liczbę sensownych dopasowań, jakie musimy sprawdzić. Przykładowo, najmniejsza liczba na wejściu musi być masą pary złożonej z dwóch najlżejszych squarków. Uporządkujmy zatem podane na wejściu liczby niemalejąco: $b_1, \ldots, b_{n(n-1)/2}$, zaś niech $a_1, a_2, \ldots, a_n$ oznaczają (nieznane) masy pojedynczych squarków, uporządkowane rosnąco (squarki różnych rodzajów mają różne masy). Niech dalej $m_{i,j}$ oznacza $a_i + a_j$. Wiemy teraz, że $b_1 = m_{1,2}$.

Odrobina więcej namysłu prowadzi do wniosku, że $b_2 = m_{1,3}$. Faktycznie — każda z pozostałych liczb $m_{i,j}$ spełnia $i \geqslant 1$ oraz $j \geqslant 3$, przy czym co najmniej jedna z tych nierówności jest ostra, więc $m_{i,j} > m_{1,3}$. Dalej, niestety, nie ma tak łatwo; trzecią

z kolei liczbą może być $m_{1,4}$ lub $m_{2,3}$ (oczywiście poza przypadkiem $n = 3$, w którym $b_3 = m_{2,3}$ i jesteśmy w stanie wyznaczyć wszystkie masy squarków od razu). Pojawia się w nas (no, przynajmniej w niektórych z nas) pokusa rozważenia obu możliwości, i tak dalej — z każdą nową liczbą b_i rozważać, które z nieprzypisanych jeszcze par squarków mogą jej odpowiadać. Tego typu rozwiązania, choć szybsze od poprzedniego, wciąż mają złożoność wykładniczą, i to wykładniczą względem liczby par squarków. Czyli nie najlepszą.

Rozwiązanie wzorcowe — pomysł

Lepsze efekty niż próba natychmiastowej implementacji rozwiązania da próba zmuszenia się do jeszcze odrobiny analizy zadania (to bardzo często zdarzająca się sytuacja). Zauważmy, że lżejsze od pary $m_{2,3}$ mogą być tylko pary postaci $m_{1,k}$. Mamy zatem tylko $n - 2$ możliwości przypisania masy parze squarków $m_{2,3}$: są to liczby od b_3 do b_n. A warto to zrobić dlatego, że znając wartości $m_{1,2}$, $m_{1,3}$ i $m_{2,3}$, możemy — tak jak we wzorze (1) — jednoznacznie określić a_1, a_2 i a_3.

Rozważmy, przykładowo, sytuację, gdy $m_{2,3} = b_5$. Wtedy $b_3 = m_{1,4}$ i $b_4 = m_{1,5}$. Na podstawie b_1, b_2 i b_5 wyznaczamy a_1, a_2 i a_3. Ale to nie koniec! Skoro znamy już a_1, to b_3 pozwala nam obliczyć a_4, zaś b_4 pozwala obliczyć a_5. I nagle możemy zidentyfikować znacznie więcej par — przykładowo, możemy wśród liczb na wejściu poszukać $m_{3,4}$ (którego jeszcze nie widzieliśmy, ale już wiemy, ile jest równe).

Najmniejszą nieznaną masą squarka jest teraz a_6. A jaka jest najmniejsza niewyznaczona masa pary squarków? Okazuje się, że mamy tylko jeden wybór: $m_{1,6}$. Istotnie — nie znamy jeszcze tylko mas par $m_{i,j}$ dla $j > 5$, a najmniejszą taką parą jest właśnie $m_{1,6}$. Możemy zatem znaleźć najmniejszą niewyznaczoną jeszcze masę z wejścia i na jej podstawie, znając a_1, wyznaczyć a_6. Widać, że kolejne masy squarków powinny posypać się jak kostki domina.

Rozwiązanie wzorcowe — implementacja

Spróbujmy teraz ten dość luźny pomysł przekuć na rozwiązanie. Będzie ono złożone z trzech faz. Wpierw iterujemy po możliwych wartościach k, zakładając, że $b_k = m_{2,3}$. Jak wiemy, wystarczy rozpatrzyć tylko $k = 3, 4, \ldots, n$.

Dla ustalonego k wyznaczamy wartości liczb $a_1, a_2, \ldots, a_k$ — obliczając a_1, a_2 i a_3 na podstawie trzech sum, a pozostałe masy na podstawie sum $m_{1,j}$. Umieśćmy teraz wszystkie liczby b_i w multizbiorze B (dla łatwego wyszukiwania). Następnie dla każdej pary i, j, gdzie $i < j \leqslant k$, znajdźmy w multizbiorze B liczbę $m_{i,j}$ i ją stamtąd usuńmy. Jeśli którejś z tych liczb nie ma w B, znaczy to, że nasz pierwotny wybór k był niepoprawny i możemy spokojnie przejść do sprawdzenia kolejnej wartości k.

Niech teraz l oznacza największy indeks squarka o znanej masie; początkowo $l = k$. Póki $l < n$, działamy następująco.

- Wybieramy najmniejszą liczbę b z multizbioru B.
- Wiemy, że ta liczba jest równa $m_{1,l+1}$, a stąd wyznaczamy a_{l+1}.

- Usuwamy z B wszystkie liczby $m_{i,l+1}$ dla $i \leqslant l$ (ponownie — jeśli którejś z tych liczb nie znajdziemy, oznacza to, że pierwotny wybór k był nietrafiony).
- Powiększamy l o jeden.

Gdy $l = n$, mamy rozwiązanie (być może jedno z kilku możliwych). Warto zwrócić uwagę, że z naszego algorytmu wynika, że rozwiązań będzie nie więcej niż $n - 2$.

Złożoność rozwiązania

Dla każdego z $n - 2$ możliwych wyborów parametru k musimy, w pesymistycznym przypadku, usunąć jedna po drugiej wszystkie liczby z multizbioru B. Jeśli multizbiór B zaimplementujemy w oparciu o jakieś drzewo zrównoważone (np. użyjemy jednej ze struktur danych dostępnych w bibliotece standardowej języka C++: `multiset` lub `map`), to każdą z podstawowych operacji na multizbiorze (sprawdzenie przynależności elementu, wstawienie i usunięcie elementu) będziemy mogli wykonać w czasie $O(\log n)$. To oznacza, że nasz algorytm będzie miał złożoność czasową $O(n^3 \log n)$. Tę samą złożoność czasową można także uzyskać, implementując multizbiór B jako posortowaną tablicę par postaci (element, krotność). Wtedy przy usuwaniu elementów z multizbioru wyszukujemy binarnie ich położenia, po czym zmniejszamy ich krotności. Jeszcze inaczej, multizbiór B można reprezentować za pomocą tablicy haszującej (tj. tablicy list), w której każda z wymienionych operacji w oczekiwanym przypadku działa w czasie stałym. To obniża oczekiwaną złożoność czasową algorytmu do $O(n^3)$.

Implementacje rozwiązania wzorcowego można znaleźć w plikach `squ.cpp`, `squ1.c`, `squ2.pas` i `squ3.cpp`.

Dla miłośników matematyki: ile może być rozwiązań?

Przykład w treści zadania pokazuje, że dla $n = 4$ mogą istnieć dwa różne rozwiązania. Nasze dotychczasowe rozumowanie pozwala stwierdzić, że dla $n = 3$ jest tylko jedno rozwiązanie. W miarę prosto jest też sprawdzić, że dla $n = 5$ i $n = 6$ również istnieje co najwyżej jedno rozwiązanie; a jako że im większe n, tym więcej ograniczeń nakładamy na poszukiwany układ mas squarków (dokładniej — dla n squarków będziemy mieli $n(n-1)/2$ warunków), to można podejrzewać, że dla $n > 4$ będzie zawsze co najwyżej jedno rozwiązanie. Okazuje się jednak, że nie jest to prawdą — wykażemy, że istnieją układy o więcej niż jednym rozwiązaniu, ale tylko dla n będących potęgami dwójki. Poniższe rozumowanie jest oparte na pomysłach autorstwa Wojciecha Nadary.

Pokażemy wpierw, jak skonstruować układy o przynajmniej dwóch rozwiązaniach dla n będących potęgą dwójki. Dla $n = 4$ taki układ mamy podany w treści zadania; można zresztą skonstruować układ złożony z jeszcze mniejszych liczb, np. {1, 4, 5, 6} oraz {2, 3, 4, 7}. Dla $n = 8$ przykładowym układem o równych sumach par jest teraz {1, 4, 5, 6, 12, 13, 14, 17} oraz {2, 3, 4, 7, 11, 14, 15, 16}. Dlaczego? Rozważmy dowolną parę liczb z pierwszego zbioru. Jeśli są to dwie liczby mniejsze od 10 (np. 4 i 5), to wiemy, że w drugim zbiorze możemy wybrać dwie liczby mniejsze od 10 dające tę samą sumę (w tym wypadku 2 i 7), bo sumy par dla {1, 4, 5, 6} oraz dla {2, 3, 4, 7} były takie same. Analogicznie, jeśli są to dwie liczby większe od 10 (np.

13 i 17), to w drugim zbiorze możemy wybrać dwie liczby większe od 10 o tej samej sumie (tu 14 i 16) — bo sumy par dla {11, 14, 15, 16} są takie same, jak dla {12, 13, 14, 17}; dodanie stałej do obu zbiorów nic nie zmienia. Jeśli natomiast mamy do czynienia z parą „mieszaną" (np. 4 i 12), to w drugim zbiorze możemy wybrać parę liczb, w których dziesiątka jest dodana „na odwrót" (w tym wypadku 2 i 14).

Analogicznie postępujemy dla $n = 16$ — zaczynamy od dwóch ósemek o tych samych sumach par i do każdej z nich dogrupowujemy drugą ósemkę, tym razem powiększoną o 100. Identyczne jak powyżej rozumowanie prowadzi do wniosku, że dla otrzymanych szesnastek:

$$\{1, 4, 5, 6, 12, 13, 14, 17, 102, 103, 104, 107, 111, 114, 115, 116\}$$

oraz

$$\{2, 3, 4, 7, 11, 14, 15, 16, 101, 104, 105, 106, 112, 113, 114, 117\}$$

multizbiory sum par są dokładnie takie same. W ten sam sposób możemy otrzymać układy o takich samych sumach par dla dowolnej potęgi dwójki.

Pozostało nam teraz wykazać, że potęgi dwójki to jedyne wartości n, dla których istnieją układy o więcej niż jednym rozwiązaniu. Załóżmy, że n nie jest potęgą dwójki, i rozważmy jakieś konkretne dane wejściowe (czyli układ $n(n-1)/2$ mas par squarków). Zauważmy, że na podstawie tych danych możemy wyznaczyć sumę mas wszystkich squarków: wystarczy zsumować masy wszystkich par i podzielić wynik przez $n-1$ (bo każdy squark występuje w $n-1$ różnych parach). Oznaczmy tę sumę przez S_1.

Następnie spróbujemy wyznaczyć S_2: sumę kwadratów mas squarków. Rozważmy dwa następujące wyrażenia:

$$V = (a_1 + a_2 + \ldots + a_n)^2 = S_2 + 2a_1a_2 + 2a_1a_3 + \ldots + 2a_{n-1}a_n$$

oraz

$$W = \sum_{i \neq j} (a_i + a_j)^2 = 2(n-1)S_2 + 4a_1a_2 + 4a_1a_3 + \ldots + 4a_{n-1}a_n.$$

Powyższe rozwinięcie wyrażenia W wynika stąd, że masa każdego squarka występuje w $n-1$ parach jako a_i i w $n-1$ parach jako a_j. Jako że znamy masy wszystkich par squarków oraz sumę mas squarków (S_1), to możemy wyznaczyć V i W, a zatem i $2(n-2)S_2 = W - 2V$. Stąd, skoro $n \neq 2$, możemy wyznaczyć S_2.

Ogólnie naszym celem jest wyznaczenie S_k (czyli sumy k-tych potęg mas squarków) dla każdego kolejnego k. Będziemy postępować podobnie jak dotychczas; zaczniemy od sumy k-tych potęg par:

$$\begin{aligned} W &= \sum_{i \neq j} (a_i + a_j)^k \\ &= 2(n-1)S_k + \sum_{i \neq j} \left(\binom{k}{1} a_i^{k-1} a_j + \binom{k}{2} a_i^{k-2} a_j^2 + \ldots + \binom{k}{k-1} a_i a_j^{k-1} \right). \end{aligned}$$

Teraz chcemy pozbyć się dodatkowych składników sumy. W tym celu rozważmy wyrażenia:

$$V_1 = S_{k-1}S_1 = S_k + \sum_{i \neq j} a_i^{k-1} a_j,$$

$$V_2 = S_{k-2}S_2 = S_k + \sum_{i \neq j} a_i^{k-2} a_j^2,$$

i tak aż do

$$V_{k-1} = S_1 S_{k-1} = S_k + \sum_{i \neq j} a_i a_j^{k-1}.$$

Jako że liczby S_k obliczamy po kolei, to możemy założyć, że wszystkie S_l dla $l < k$ są już nam znane. To pozwala nam wyznaczyć W oraz wszystkie V_i, więc także i wartość wyrażenia:

$$2(n-2^{k-1})S_k = (2(n-1)-(2^k-2))S_k = W - \binom{k}{1}V_1 - \binom{k}{2}V_2 - \ldots - \binom{k}{k-1}V_{k-1}.$$

Skoro n nie jest potęgą dwójki, to z tego wzoru jesteśmy w stanie wyznaczyć S_k.

Rozumowanie powyżej pokazuje, że jeśli tylko n nie jest potęgą dwójki, to możemy wyznaczyć sumy k-tych potęg mas squarków dla dowolnego naturalnego k. To jednak oznacza, że jesteśmy w stanie wyznaczyć masy pojedynczych squarków. Faktycznie, wyrażenie:

$$\sqrt[k]{S_k} = a_n \sqrt[k]{\left(\frac{a_1}{a_n}\right)^k + \ldots + \left(\frac{a_{n-1}}{a_n}\right)^k + 1}$$

zbiega, przy k dążącym do nieskończoności, do a_n (czyli masy najcięższego squarka); podobnie,

$$\sqrt[k]{S_k - a_n^k}$$

zbiega do a_{n-1}, i tak dalej.

Okazuje się, że aby wyznaczyć wartości a_i, wystarczy znać liczby S_k dla $k \leqslant n$. Opracowanie stosownej metody odtworzenia ciągu mas squarków pozostawiamy jako zadanie dla Czytelnika.

Testy

Rozwiązania zawodników były sprawdzane przy użyciu dwunastu testów, scharakteryzowanych w poniższej tabeli.

Nazwa	n
squ1.in	4
squ2.in	7
squ3.in	9
squ4.in	14

Nazwa	n
squ5.in	24
squ6.in	28
squ7.in	45
squ8.in	60

Nazwa	n
squ9.in	100
squ10.in	140
squ11.in	210
squ12.in	300

Karol Pokorski
Treść zadania, Opracowanie

Alan Kutniewski
Program

Dostępna pamięć: 128 MB.

OI, etap III, dzień pierwszy, 28.03.2012

Licytacja

Alojzy i Bajtazar grają w licytację. Do grania w tę grę potrzebny jest bardzo duży zestaw kamyków. Gracze wykonują ruchy na przemian: najpierw Alojzy, potem Bajtazar, znowu Alojzy itd. W danym momencie gry istotne są dwie wartości: aktualna stawka i aktualny rozmiar stosu. Gra zaczyna się od stawki jednego kamyka i pustego stosu. W każdym ruchu gracz wykonuje jeden z następujących ruchów:

- *podwaja stawkę,*
- *potraja stawkę,*
- *pasuje.*

W przypadku, gdy gracz spasuje, cała aktualna stawka wędruje na stos (jest to jedyny sposób powiększenia stosu), a licytacja ponownie zaczyna się od stawki jednego kamyka. Jeżeli gracz spasuje, to następny ruch należy do jego przeciwnika (Alojzy zaczyna tylko całą rozgrywkę). Gracz, który spowoduje przepełnienie stosu (następuje to, gdy na stosie znajduje się n lub więcej kamyków), przegrywa. Jeśli przed ruchem gracza łączna liczba kamyków na stosie i w stawce osiąga lub przekracza n, gracz ten nie może już podwoić ani potroić stawki. Musi spasować, tym samym dokładając stawkę na stos, co powoduje jego przegraną.

Alojzy bardzo często przegrywa. Bajtazar zaproponował mu ciekawe wyzwanie — zamiast samemu grać w licytację, lepiej napisać programy, które będą w nią grały. Niestety, Alojzy nie umie programować. Pomóż mu!

Napisz program, który będzie grał w licytację w imieniu Alojzego przeciw bibliotece napisanej przez Bajtazara.

Ocenianie

We wszystkich przypadkach testowych Twój program będzie mógł wygrać (o ile wykona odpowiednie ruchy) niezależnie od ruchów biblioteki. Twój program otrzyma punkty za dany test tylko wtedy, gdy wygra z biblioteką.

We wszystkich testach zachodzi warunek $1 \leqslant n \leqslant 30\,000$. W 50% przypadków testowych zachodzi dodatkowy warunek $n \leqslant 25$.

Opis użycia biblioteki

Aby użyć biblioteki, należy wpisać na początku programu:

- **C/C++:** `#include "cliclib.h"`
- **Pascal:** `uses pliclib;`

Biblioteka udostępnia następujące funkcje i procedury:

- `inicjuj` — *zwraca liczbę n. Powinna zostać użyta dokładnie raz, na samym początku działania programu.*
  - **C/C++:** `int inicjuj();`
  - **Pascal:** `function inicjuj: longint;`
- `alojzy` — *informuje bibliotekę o ruchu Twojego programu. Jej jedynym parametrem jest liczba całkowita x, która oznacza wykonany ruch: x = 1 oznacza spasowanie, x = 2 — podwojenie stawki, natomiast x = 3 — potrojenie stawki.*
  - **C/C++:** `void alojzy(int x);`
  - **Pascal:** `procedure alojzy(x: longint);`
- `bajtazar` — *funkcja informuje Twój program o ruchu biblioteki. Zwraca jedną liczbę x, która oznacza wykonany ruch. Analogicznie jak w przypadku funkcji* `alojzy`*, x = 1 oznacza pas, x = 2 — podwojenie, natomiast x = 3 — potrojenie stawki.*
  - **C/C++:** `int bajtazar();`
  - **Pascal:** `function bajtazar: longint;`

Po wywołaniu funkcji `inicjuj` *należy naprzemiennie wywoływać funkcje* `alojzy` *oraz* `bajtazar` *(w takiej kolejności). Złamanie protokołu komunikacji zostanie potraktowane jako błędna odpowiedź i spowoduje przyznanie 0 punktów za dany test. W tym zadaniu użycie standardowego wejścia i wyjścia jest* **zabronione**. *Jakakolwiek komunikacja powinna odbywać się tylko za pośrednictwem wyżej podanych funkcji i procedur. Biblioteka zakończy działanie programu automatycznie po zakończeniu gry.*

Rozwiązanie będzie kompilowane wraz z biblioteką przy użyciu następujących poleceń:

- **C:** `gcc -O2 -static cliclib.c lic.c -lm`
- **C++:** `g++ -O2 -static cliclib.c lic.cpp -lm`
- **Pascal:** `ppc386 -O2 -Xs -Xt lic.pas`

Eksperymenty

W katalogu **/home/zawodnik/rozw/lic/** *znajdują się przykładowe pliki bibliotek i przykładowe nieoptymalne rozwiązania ilustrujące sposób ich użycia (można je także pobrać w dziale* **Przydatne zasoby** *w SIO). Program skompilowany z przykładową biblioteką wczytuje ze standardowego wejścia liczbę n, a następnie aż do zakończenia gry wczytuje kolejne ruchy wykonywane przez Bajtazara, symulując zaimplementowaną strategię Alojzego. Na standardowe wyjście diagnostyczne (*`stderr`*) program wypisuje szczegółowy przebieg gry. W testach ocen, zarówno na komputerze zawodnika, jak i w SIO, Bajtazar odpowiada kolejno: pas, podwojenie stawki, potrojenie stawki, pas, podwojenie stawki, potrojenie stawki...*

Ostateczna ocena programów odbędzie się z wykorzystaniem innego zestawu bibliotek.

W przypadku tego zadania w SIO nie jest dostępna opcja **Test programu**.

Przykładowy przebieg programu

C/C++	Pascal	Wynik	Stos	Stawka	Wyjaśnienie
`n = inicjuj();`	`n := inicjuj;`	*15*	*0*	*1*	*Od tego momentu $n = 15$.*
`alojzy(2);`	`alojzy(2);`	—	*0*	*2*	*Twój program podwoił stawkę.*
`bajtazar();`	`bajtazar;`	*2*	*0*	*4*	*Biblioteka odpowiedziała podwojeniem stawki.*
`alojzy(3);`	`alojzy(3);`	—	*0*	*12*	*Twój program potroił stawkę.*
`bajtazar();`	`bajtazar;`	*1*	*12*	*1*	*Biblioteka spasowała. 12 kamyków wędruje na stos, a w stawce ponownie jest tylko 1 kamyk.*
`alojzy(2);`	`alojzy(2);`	—	*12*	*2*	*Twój program podwoił stawkę.*
`bajtazar();`	`bajtazar;`	*3*	*12*	*6*	*Biblioteka potroiła stawkę.*
`alojzy(1);`	`alojzy(1);`	—	*18*	*1*	*Łączna liczba kamyków na stosie i w stawce przekroczyła 15. Twój program jest zmuszony spasować.*

Powyższy przebieg programu jest poprawny, ale nieoptymalny. Twój program nie dostałby punktów za ten test. W szczególności, dla $n = 15$ istnieje możliwość wygranej Alojzego, niezależnie od ruchów Bajtazara.

Rozwiązanie

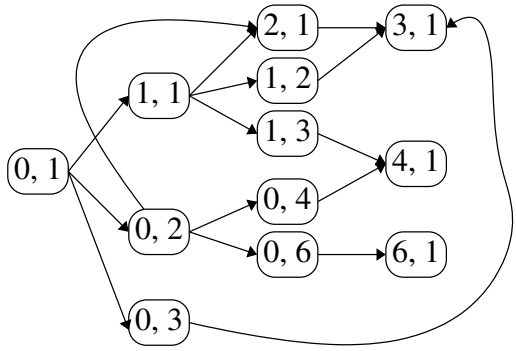
Wprowadzenie

Wyjątkowo problem sformułowany w treści tego zadania nie jest nadmiernie uwikłany w postaci historyjki. Widzimy, że mamy do czynienia z pewną grą. Jakiego typu jest to gra? W tym przypadku najprościej rozumieć grę jako zbiór pozycji i dopuszczalnych ruchów między nimi. Okazuje się, że tego typu relację najwygodniej reprezentować w postaci grafu, w którym wierzchołkami są pozycje, a możliwe ruchy są przedstawiane jako skierowane krawędzie łączące odpowiednie wierzchołki grafu.

Graf pozycji

Pozostaje jeszcze ustalić, czym są pozycje w grze z zadania. Do pełnego opisu aktualnej sytuacji w grze wystarczą nam dwie wartości: rozmiar stosu kamyków oraz stawka. Można zatem oznaczyć wierzchołki grafu pozycji za pomocą uporządkowanych par liczb całkowitych (x, y), gdzie x to aktualna liczba kamieni na stosie, natomiast y to stawka. Przy takich założeniach gra rozpoczyna się na pozycji $(0, 1)$.

Ustalenie dostępnych ruchów dla każdej pozycji jest już bardzo łatwe, należy jednak zachować pewną ostrożność przy oznaczaniu warunków końcowych gry. Ostatecznie, jeśli $x \geqslant n$, to z pozycji (x, y) nie ma żadnych dopuszczalnych ruchów, a w przeciwnym razie z pozycji (x, y) można przejść do pozycji $(x+y, 1)$, a także, o ile $x+y < n$, przejść do pozycji $(x, 2y)$ oraz $(x, 3y)$.



Rys. 1: Graf pozycji dla $n = 3$.

Podział pozycji

Warto jeszcze zauważyć, że gracze są nierozróżnialni. Przy ustalonym stanie gry zarówno Alojzy, jak i Bajtazar mogą wykonać te same ruchy. Pozwala to skupić się na określeniu, dla każdej pozycji, czy jest ona wygrywająca, czy przegrywająca, bez uwzględniania tego, który z graczy w danej chwili wykonuje ruch.

Gracz, który znajduje się na pozycji wygrywającej, może wygrać niezależnie od ruchów przeciwnika (przy założeniu optymalnej gry). Analogicznie, gracz, który znajduje się na pozycji przegrywającej, przegra niezależnie od swoich poczynań, jeśli jego przeciwnik zagra optymalnie.

Powyższa definicja jest poprawna, ale nie przybliża nas do rozwiązania zadania. Spróbujmy scharakteryzować pozycje wygrywające i przegrywające inaczej — za pomocą ruchów, które można z nich wykonać. Na dobry początek warto zauważyć, że wszystkie pozycje, z których nie można wykonać ruchu (pozycje końcowe), są pozycjami wygrywającymi. Istotnie, jeśli gracz znajduje się w pozycji końcowej, to liczba kamieni na stosie jest równa co najmniej n, a zatem jego przeciwnik (który przecież wykonał poprzedni ruch) spowodował przepełnienie stosu, co zgodnie z zasadami gry oznacza jego porażkę. Równoważnie, można by w ogóle nie rozważać pozycji,

w których stos jest przepełniony, i uznać, że gracz, który nie może wykonać ruchu, przegrywa.

Niezależnie od przyjętych założeń co do pozycji końcowych:

- jeśli z danej pozycji istnieje ruch do pozycji przegrywającej, to pozycja ta jest wygrywająca (wystarczy bowiem wykonać ruch do tej pozycji, a wówczas nasz przeciwnik znajdzie się w pozycji przegrywającej),
- jeśli z danej pozycji istnieją tylko ruchy do pozycji wygrywających, to pozycja ta jest przegrywająca (niezależnie od tego, do której pozycji gracz przejdzie, sprowadzi swojego przeciwnika do pozycji wygrywającej).

Powyższe obserwacje są podstawą analizy wielu gier kombinatorycznych. Przykładem może być tutaj zadanie *Kamyki* z XVI Olimpiady Informatycznej [16], w którym analiza pozycji wygrywających i przegrywających również stanowi trzon rozwiązania. Gorąco zachęcamy do lektury opracowania wspomnianego zadania, można bowiem dzięki niemu udoskonalić swoje umiejętności analizy takich gier[1].

Wypracowane prawa rządzące pozycjami wygrywającymi i przegrywającymi są już wystarczające do stwierdzenia, które pozycje są wygrywające, a które przegrywające, a także pozwalają wykonywać ruchy prowadzące do zwycięstwa. Wiadomo przecież, że pozycje końcowe są wygrywające, a obliczenie stanu pozostałych pozycji jest możliwe na podstawie stanów wyznaczonych wcześniej.

Należy tylko opracować kolejność rozpatrywania wierzchołków grafu pozycji, aby mieć pewność, że wszystkie pozycje niezbędne do obliczenia danego stanu zostały już przetworzone. Nie jest to jednak trudne. Otóż wystarczy zauważyć, że graf pozycji jest DAG-iem[2], a szukana kolejność przetwarzania pozycji odpowiada odwróconemu posortowaniu topologicznemu[3] grafu.

Sortowanie topologiczne można zrealizować na co najmniej dwa różne sposoby. Jeden z nich polega na stopniowym usuwaniu z grafu wierzchołków, z których nie wychodzą żadne krawędzie, i dodawaniu ich na początek uporządkowania topologicznego; w drugim zaś wykorzystujemy algorytm przeszukiwania grafu w głąb: za każdym razem, gdy algorytm w pełni przetworzy dany wierzchołek (czyli gdy zostaną odwiedzeni wszyscy jego sąsiedzi i algorytm powraca do rodzica), można dodać ów wierzchołek jako pierwszy do aktualnego uporządkowania topologicznego. Obie te metody łatwo zaimplementować w czasie $O(V + E)$, gdzie V jest liczbą wierzchołków grafu, a E — liczbą jego krawędzi.

Szacowanie liczby możliwych pozycji

Pozostaje przeanalizować, jak duży będzie nasz graf pozycji. Wierzchołków jest w nim tyle, ile par uporządkowanych (x, y), gdzie zarówno x (rozmiar stosu), jak i y (stawka)

[1] Jeszcze więcej o tego typu grach można dowiedzieć się ze strony *Wykładów z Algorytmiki Stosowanej* (`was.zaa.mimuw.edu.pl`) — Wykład 6.

[2] DAG (ang. *directed acyclic graph*) — skierowany graf acykliczny

[3] Posortowanie topologiczne skierowanego grafu acyklicznego jest uporządkowaniem jego wierzchołków, w którym każdy wierzchołek grafu poprzedza wszystkie wierzchołki, do których prowadzą wychodzące z niego krawędzie.

mogą być rzędu n. Daje to $O(n^2)$ stanów, co przy $n \leqslant 30\,000$ jest niewystarczające do uzyskania pełnej punktacji.

Czy to oznacza, że należy zmienić wypracowany pomysł na rozwiązanie? Na szczęście nie. Okazuje się, że uzyskane oszacowanie można poprawić i wówczas stanów będzie istotnie mniej niż rzędu n^2. Należy bowiem zauważyć, że stawka, na początku licytacji równa 1, może się tylko podwajać lub potrajać, a zatem zawsze będzie postaci $2^a \cdot 3^b$, gdzie a i b są liczbami całkowitymi: odpowiednio liczbą podwojeń i potrojeń stawki w aktualnej licytacji. Ponieważ oba te parametry nie przekraczają $\log n$, więc osiągalnych pozycji jest w istocie tylko $O(n \log^2 n)$. Przy przyjętych ograniczeniach na dane wejściowe takie rozwiązanie wystarczało do osiągnięcia wysokich wyników (oscylujących w granicach 80–100 punktów).

Bezpośrednia implementacja powyższego pomysłu znajduje się w pliku `lics3.cpp`.

Rozwiązania alternatywne

Powyższe rozważania okazały się wystarczające do uzyskania poprawnego i dość wydajnego rozwiązania, ale implementacja powyższego pomysłu nie jest ani najprostsza, ani najszybsza. Kluczowym problemem jest wysokie zużycie pamięci — przechowywanie grafu połączeń i posortowania topologicznego jest kosztowne i może przekroczyć nawet stosunkowo duże limity pamięciowe ustalone dla tego zadania.

Przechowywanie całego grafu w pamięci nie jest jednak konieczne. Krawędzie można generować na bieżąco, ponadto zamiast sortowania topologicznego grafu można zastosować algorytm rekurencyjny obliczania stanu pozycji, co w połączeniu ze spamiętywaniem wyników dla wcześniej obliczonych pozycji pozwoli uzyskać rozwiązanie równie efektywne, a przy tym prostsze w implementacji. Zauważmy, że pozycje możemy tu reprezentować za pomocą trójek postaci (x, a, b), gdzie x to rozmiar stosu, a stawka to $2^a \cdot 3^b$, co pozwala wygodnie tablicować wyniki dla pozycji. Programy oparte na tym pomyśle znajdują się w plikach `lic.pas`, `lic1.cpp`, `lic2.pas` oraz `lic3.cpp`.

Część zawodników zapomniała o spamiętywaniu wyników dla wcześniej obliczonych pozycji, co prowadziło do algorytmu wykładniczego. Tego typu rozwiązania zostały zaimplementowane w plikach `lics1.cpp` oraz `lics2.pas`. Na zawodach taki błąd kosztował, zgodnie z uwagą w treści zadania, około połowę możliwych do zdobycia punktów.

Niektórzy finaliści nie poradzili sobie z problemem ograniczenia liczby stanów i pozostali z algorytmem kwadratowym. Takie rozwiązania uzyskiwały na zawodach 75% możliwych do zdobycia punktów.

Rozwiązania polegające na losowaniu ruchów lub heurystycznym ich obliczaniu względem prostych miar zazwyczaj uzyskiwały bardzo mało punktów (najczęściej zero).

Testy

Spośród testów, na jakich były sprawdzane rozwiązania zawodników, połowa to testy poprawnościowe — wybrano w nich takie wartości n, aby gracz rozpoczynający miał

strategię wygrywającą, a bibliotekę ustawiono na grę optymalną (zgodną z programem wzorcowym). W poniższej tabeli takie zachowanie biblioteki zostało oznaczone jako strategia *optymalna*.

W kolejnych testach parametr n stopniowo rośnie, a także stosowane są miejscami inne strategie gry biblioteki: *prawie optymalna*, polegająca na optymalnej grze z małym prawdopodobieństwem wykonania losowego ruchu, oraz *losowa*, w której kilka pierwszych ruchów jest losowych, po czym przełączamy strategię na optymalną.

Nazwa	n	Opis
lic1.in	5	s. optymalna
lic2.in	7	s. optymalna
lic3.in	10	s. optymalna
lic4.in	13	s. optymalna
lic5.in	15	s. optymalna
lic6.in	17	s. optymalna
lic7.in	18	s. optymalna
lic8.in	20	s. optymalna
lic9.in	22	s. optymalna
lic10.in	25	s. optymalna

Nazwa	n	Opis
lic11.in	129	s. prawie optymalna
lic12.in	390	s. optymalna
lic13.in	891	s. prawie optymalna
lic14.in	1 700	s. optymalna
lic15.in	2 000	s. losowa
lic16.in	9 001	s. prawie optymalna
lic17.in	14 000	s. optymalna
lic18.in	21 060	s. losowa
lic19.in	27 400	s. optymalna
lic20.in	29 990	s. optymalna

Adam Karczmarz
Treść zadania, Opracowanie

Mateusz Baranowski
Program

Dostępna pamięć: 128 MB. *OI, etap III, dzień pierwszy, 28.03.2012*

Pensje

Bajtocka Fabryka Oprogramowania (BFO) zatrudnia n pracowników. W hierarchii pracowników każdy ma swojego bezpośredniego przełożonego, z wyjątkiem dyrektora BFO, któremu (pośrednio lub bezpośrednio) podlegają wszyscy pracownicy BFO. Pracownicy mają ustalone miesięczne pensje, przy czym każdy pracownik zarabia inną kwotę, od 1 do n bajtalarów. Każdy przełożony zarabia więcej niż każdy z jego podwładnych.

Zgodnie z bajtockim prawem, pensje pracowników na pewnych stanowiskach mogą być publicznie znane. Ponadto, jeśli pensja pewnego pracownika jest publicznie znana, to pensja jego przełożonego także jest znana.

Bajtocki Urząd Podatkowo–Antykorupcyjny (BUPA) postanowił prześwietlić BFO. Zanim BUPA wkroczy z kontrolą do BFO, chce najpierw poznać wysokość pensji wszystkich pracowników BFO, których pensje nie są publicznie znane, ale których wysokość wynika jednoznacznie z publicznie znanych wysokości pensji innych pracowników BFO.

Wejście

W pierwszym wierszu standardowego wejścia podana jest jedna liczba całkowita n ($1 \leqslant n \leqslant 1\,000\,000$) oznaczająca liczbę pracowników BFO. Pracownicy są ponumerowani od 1 do n.

Kolejne n wierszy zawiera informacje o pracownikach. Wiersz o numerze $i + 1$ opisuje pracownika numer i za pomocą dwóch liczb całkowitych p_i i z_i ($1 \leqslant p_i \leqslant n$, $0 \leqslant z_i \leqslant n$) oddzielonych pojedynczym odstępem. Liczba p_i to numer bezpośredniego przełożonego pracownika numer i. Jeżeli $p_i = i$, to i jest numerem dyrektora BFO. Jeśli $z_i > 0$, to jest to wysokość pensji pracownika numer i. Jeśli zaś $z_i = 0$, to wysokość pensji pracownika numer i nie jest publicznie znana. Dodatnie liczby z_i są parami różne.

Dane wejściowe będą tak skonstruowane, że będzie istniał co najmniej jeden sposób przypisania płac pracownikom zgodny z ich hierarchią.

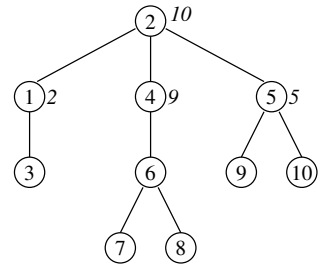
W testach wartych łącznie 54% punktów zachodzi dodatkowy warunek $n \leqslant 10\,000$.

Wyjście

Twój program powinien wypisać na standardowe wyjście n wierszy, z których każdy powinien zawierać jedną liczbę całkowitą. Jeżeli pensja pracownika numer i jest jawna lub może zostać wywnioskowana na podstawie znanych publicznie pensji innych pracowników, to i-ty wiersz wyjścia powinien zawierać pensję pracownika numer i. W przeciwnym przypadku i-ty wiersz wyjścia powinien zawierać 0.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
10	2
2 2	10
2 10	1
1 0	9
2 9	5
2 5	8
4 0	0
6 0	0
6 0	0
5 0	0
5 0	



Wyjaśnienie do przykładu: *Na rysunku liczby w kółkach to numery pracowników, a liczby zapisane kursywą to publicznie znane pensje pracowników. Pracownik numer 3 musi zarabiać 1 bajtalar, a pracownik numer 6 musi zarabiać 8 bajtalarów. Zarobki pracowników numer 7, 8, 9 i 10 nie wynikają jednoznacznie z publicznie znanych zarobków innych pracowników.*

Rozwiązanie

Wprowadzenie

Aby uprościć nasze rozważania, sformułujmy problem z zadania w języku matematyki. Struktura Bajtockiej Fabryki Oprogramowania tworzy drzewo ukorzenione, którego wierzchołki odpowiadają pracownikom BFO. Dokładniej, korzeń drzewa odpowiada dyrektorowi BFO, a ojcowie pozostałych wierzchołków odpowiadają ich bezpośrednim przełożonym. Przypisanie pensji pracownikom to nic innego jak wpisanie w każdy z wierzchołków innego klucza z zakresu od 1 do n tak, aby spełniony był *warunek kopca*. Warunek kopca jest spełniony wtedy i tylko wtedy, gdy dla każdego wierzchołka, jego klucz jest mniejszy niż klucz jego ojca.

Wyobraźmy sobie, że po przypisaniu kluczy wszystkim wierzchołkom wymazaliśmy klucze w niektórych poddrzewach. Naszym zadaniem jest wskazanie wierzchołków, dla których jesteśmy w stanie stwierdzić z całą pewnością, jaki klucz był im pierwotnie przypisany.

Analiza

Dla danego drzewa T, oznaczmy liczbę jego wierzchołków przez $|T|$. Poddrzewo o korzeniu v zawierające wszystkich potomków v oznaczać będziemy przez $T(v)$.

Na wstępie zauważmy, że dokładna struktura „górnej" części drzewa, z której nie zostały wymazane klucze, nie ma większego znaczenia dla wyniku, który chcemy uzyskać. Mianowicie, na to, jak duże mogą być klucze w wierzchołkach danego poddrzewa,

ma wpływ jedynie klucz ojca tego poddrzewa. To spostrzeżenie pozwala nam przeformułować zadanie w następujący sposób. Mamy dane t drzew $T_1, T_2, \ldots, T_t$, w których klucze wierzchołków muszą być mniejsze niż, odpowiednio, $a_1, a_2, \ldots, a_t$, a także zbiór K kluczy, którymi chcemy wypełnić te drzewa. Chcemy wskazać wierzchołki, które otrzymają ten sam klucz w każdym przyporządkowaniu kluczy zachowującym warunek kopca i podane ograniczenia.

Zbadajmy najpierw prosty przypadek, w którym mamy do wypełnienia tylko jedno drzewo, a więc gdy $t = 1$. Liczba elementów zbioru $K = \{k_1 < k_2 < \ldots < k_m\}$ musi być wtedy równa wielkości drzewa, a każdy z elementów tego zbioru musi być mniejszy niż a_1. Jasne jest, że w korzeniu drzewa należy umieścić największą liczbę z K. Jeśli korzeń drzewa ma tylko jednego syna, to synowi należy nadać klucz k_{m-1}. Jeżeli syn korzenia ma także tylko jednego syna, z całą pewnością otrzyma on klucz k_{m-2} itd. Stąd wnioskujemy, że jeśli drzewo jest ścieżką, jest tylko jedna możliwość przypisania kluczy wierzchołkom, a więc wszystkie klucze są wyznaczone jednoznacznie.

Zastanówmy się, jak to wygląda w sytuacji, gdy drzewo zawiera rozgałęzienie. Niech v będzie najbliższym korzeniowi wierzchołkiem drzewa (być może samym korzeniem), który ma co najmniej dwóch synów. Niech w będzie dowolnym potomkiem v, a s — synem v, do którego poddrzewa należy w. Załóżmy ponadto, że w ma jednoznacznie przyporządkowany klucz k. Zbiór kluczy $T(s)$ może być dowolnym z $|T(s)|$-elementowych podzbiorów zbioru $K' = \{k_1, k_2, \ldots, k_{|T(v)|-1}\}$. Ale ponieważ v ma więcej niż jednego syna, więc $|T(s)| < |T(v)| - 1$ i istnieje $|T(s)|$-elementowy podzbiór zbioru K' niezawierający k. To oznacza, że w pewnym przyporządkowaniu kluczy, k nie występuje wśród kluczy $T(s)$, więc tym bardziej nie może być on przypisany do wierzchołka w. Doszliśmy do sprzeczności z założeniem o jednoznacznym przyporządkowaniu wierzchołkowi w klucza k.

Mamy więc pierwszy rezultat.

Fakt 1. *W przypadku jednego drzewa, jednoznacznie wyznaczone są tylko klucze w wierzchołkach na ścieżce od korzenia do pierwszego rozgałęzienia.*

Przypadek wielu drzew jest bardziej skomplikowany, ponieważ zbiór nieużytych kluczy K jest wspólny dla wszystkich drzew, a zbiory kluczy odpowiadające poszczególnym poddrzewom nie muszą być jednoznacznie wyznaczone. Pokażemy, jak zredukować problem wyznaczania jednoznacznie określonych kluczy dla $t > 1$ drzew do przypadku dla $t - 1$ drzew.

Załóżmy bez straty ogólności, że drzewa są tak uporządkowane, aby zachodziło $a_1 < a_2 < \ldots < a_t$. Jako zbiór kluczy dla drzewa T_1 musimy wybrać pewien $|T_1|$-elementowy podzbiór zbioru $K_{a_1} = \{k \in K : k < a_1\}$.

Jeśli $|K_{a_1}| = |T_1|$, mamy tylko jedną możliwość — do T_1 przypisujemy wszystkie klucze z K_{a_1}, a w celu wyznaczenia jednoznacznie określonych kluczy w drzewie T_1, stosujemy wcześniej wykazany fakt. Pozostało nam obliczyć wyniki dla drzew $T_2, \ldots, T_t$ i zbioru kluczy $K' = K \setminus K_{a_1}$, a następnie dodać do wyniku jednoznacznie określone klucze z T_1.

W przypadku, gdy $|K_{a_1}| > |T_1|$, jako zbiór kluczy drzewa T_1 możemy wybrać dowolny z $|T_1|$-elementowych podzbiorów zbioru K_{a_1}. Można to łatwo uzasadnić, jeśli zauważyć, że każdy z niewybranych do T_1 elementów K_{a_1} może znaleźć się w dowolnym z pozostałych drzew, co wynika z warunku $a_2, \ldots, a_t > a_1$. Żaden

z kluczy nie jest więc jednoznacznie przypisany do T_1. Co więcej, żaden z kluczy z K_{a_1} nie może zostać jednoznacznie wyznaczony, niezależnie od tego, w którym drzewie się znajdzie. Stąd, dla ostatecznego wyniku nie ma znaczenia, które z kluczy K_{a_1} trafią do T_1, a które do pozostałych drzew.

Ta obserwacja pozwala nam przeprowadzić redukcję w następujący sposób: przypisujemy $|T_1|$ najmniejszych elementów zbioru K do pierwszego drzewa, a następnie usuwamy je z K, otrzymując K'. Rozwiązujemy problem dla pozostałych drzew i dostępnych kluczy K', otrzymując zbiór jednoznacznie przypisanych kluczy do wierzchołków drzew $T_2, \ldots, T_t$. Ostatecznie, zgodnie z obserwacją, musimy „poprawić" uzyskany wynik: ze zbioru jednoznacznie wyznaczonych par (wierzchołek, klucz) usuwamy pary z kluczem należącym do K_{a_1} (o ile takie istnieją).

Rozwiązanie wzorcowe

Z opisanej redukcji problemu do przypadku mniejszej liczby drzew wynika natychmiast algorytm rozwiązujący nasze zadanie. Niestety, bez dbałości o szczegóły techniczne i użycia odpowiednich struktur danych, może on działać w czasie kwadratowym. Aby uzyskać maksymalną liczbę punktów, potrzebujemy jego szybszej implementacji.

Algorytm będzie działał tak, że po każdym kroku będziemy mieli przeanalizowany pewien zbiór kluczy A. Dla każdego klucza ze zbioru A będziemy wiedzieli, czy jest on przypisany jednoznacznie do jakiegoś wierzchołka, czy też nie. Ponadto, pewna liczba najmniejszych kluczy ze zbioru A będzie już przyporządkowana do konkretnych drzew.

Zbiór P — nienapotkanych jeszcze kluczy, oraz zbiór K — kluczy nieprzypisanych do konkretnych drzew będziemy przechowywać w postaci uporządkowanych stosów, z których elementy będziemy wyjmować, począwszy od najmniejszego. Potrzebujemy także tablicy $niejedn[1..n]$, w której będziemy zaznaczać klucze, o których już stwierdziliśmy, że nie mogą być wyznaczone jednoznacznie. Początkowo zbiory P i K zawierają wszystkie klucze (poza wymienionymi na wejściu) i dla każdego klucza k zachodzi $niejedn[k] = \textbf{false}$.

W i-tym kroku algorytmu wykonujemy, co następuje. Ze stosu K zdejmujemy $|T_i|$ (najmniejszych) kluczy — nazwijmy ten zbiór M_i. Jeśli element ze szczytu stosu K (o ile istnieje) jest większy niż a_i, to zgodnie z Faktem 1 wyznaczamy jednoznaczne klucze w drzewie T_i z dostępnym zbiorem kluczy M_i. Ten krok można łatwo wykonać w czasie liniowym, korzystając z naszego ulubionego algorytmu przeszukiwania drzewa. Ze stosu P zdejmujemy natomiast wszystkie klucze mniejsze niż a_i.

W przeciwnym wypadku, gdy element ze szczytu stosu K jest mniejszy niż a_i, zachodzi drugi przypadek redukcji. Wiemy, że w T_i żaden z wierzchołków nie ma jednoznacznie przypisanego klucza. Wiemy także, że wszystkie dostępne na tym etapie klucze mniejsze niż a_i nie mogą być jednoznacznie przypisane do żadnego wierzchołka. Ze stosu P zdejmujemy więc wszystkie klucze mniejsze niż a_i i dla każdego z nich, ustawiamy odpowiadającą mu wartość w tablicy *niejedn* na **true**. Zachodzi tutaj niezmiennik, że wszystkie klucze k ze zbioru K, które nie należą już do zbioru P, mają przypisane $niejedn[k] = \textbf{true}$.

Po przetworzeniu wszystkich drzew, pewne wierzchołki mają przyporządkowane klucze. Jako jednoznaczne przyporządkowania wypisujemy jedynie te pary (v, k), dla których $niejedn[k] =$ **false**.

Ponieważ każdy klucz zostaje zdjęty ze stosów K i P dokładnie raz, algorytm działa w czasie liniowym. Implementację opisanego rozwiązania można znaleźć w pliku `pen.cpp`.

Testy

Testy z grup *1* i *2* zostały ułożone ręcznie. Pozostałe grupy testów zostały wygenerowane w sposób losowy.

W poniższej tabeli n oznacza rozmiar drzewa, p oznacza liczbę publicznie znanych pensji, a w — łączną liczbę pensji, które da się wywnioskować z tych danych.

Nazwa	n	p	w
pen1a.in	1	0	1
pen1b.in	5	3	2
pen1c.in	4	2	0
pen1d.in	15	4	6
pen1e.in	7	4	3
pen2a.in	9	3	2
pen2b.in	10	4	3
pen2c.in	24	4	3
pen2d.in	14	10	1
pen2e.in	15	4	2
pen3a.in	49	8	15
pen3b.in	53	14	15
pen4a.in	49	10	11
pen4b.in	65	30	29

Nazwa	n	p	w
pen5a.in	96	10	26
pen5b.in	142	32	12
pen6.in	439	100	188
pen7.in	1 167	151	680
pen8.in	4 814	500	1 574
pen9.in	6 682	2 000	879
pen10.in	85 368	17 674	36 022
pen11.in	226 992	100 000	78 452
pen12.in	408 054	100 000	44 555
pen13.in	510 646	200 000	146 154
pen14.in	535 787	9 599	41 300
pen15.in	853 829	700 000	142 287
pen16.in	988 134	17 992	609 131

Jakub Pachocki — Treść zadania, Opracowanie
Tomasz Idziaszek — Program

Dostępna pamięć: 64 MB. *OI, etap III, dzień pierwszy, 28.03.2012*

Wyrównywanie terenu

Bajtazar postanowił wybudować dom. Jako lokalizację dla niego wybrał pewną bardzo wąską dolinę. Bajtazar musi najpierw wyrównać grunt pod budowę domu. Ma on do dyspozycji dwie koparki: pierwsza z nich może zwiększyć lub zmniejszyć poziom gruntu w dowolnym spójnym fragmencie doliny o dokładnie a metrów; druga z nich może natomiast zwiększyć lub zmniejszyć poziom gruntu w dowolnym spójnym fragmencie doliny o dokładnie b metrów. Zauważ, że zarówno przed wykonaniem każdej takiej operacji jak i po jej wykonaniu grunt w rozważanym kawałku doliny nie musi być równy.

Mając daną mapę terenu, wyznacz minimalną liczbę operacji, jakie trzeba wykonać, aby wyrównać teren w całej dolinie (tj. aby grunt w całej dolinie miał poziom równy 0). W trakcie wykonywania ciągu operacji poziom gruntu w każdym fragmencie doliny może być dowolnie duży lub dowolnie mały (w szczególności ujemny).

Wejście

W pierwszym wierszu standardowego wejścia znajdują się trzy liczby całkowite n, a, b ($1 \leqslant n \leqslant 100\,000$, $1 \leqslant a, b \leqslant 10^9$), pooddzielane pojedynczymi odstępami. Liczba n oznacza długość doliny, podaną w metrach. Drugi wiersz zawiera n liczb całkowitych $h_1, h_2, \ldots, h_n$ nieprzekraczających co do wartości bezwzględnej 10^9, pooddzielanych pojedynczymi odstępami. Liczby te reprezentują poziom gruntu (wyrażony w metrach) na kolejnych kawałkach ziemi długości jednego metra.

W testach wartych 30% punktów zachodzą dodatkowe warunki $n, a, b \leqslant 200$ oraz $-200 \leqslant h_1, \ldots, h_n \leqslant 200$.

W testach wartych 60% punktów zachodzą warunki $n, a, b \leqslant 2\,000$ oraz $-2\,000 \leqslant h_1, \ldots, h_n \leqslant 2\,000$.

W testach wartych 90% punktów zachodzi warunek $a, b \leqslant 10^6$.

Wyjście

W pierwszym i jedynym wierszu standardowego wyjścia Twój program powinien wypisać jedną liczbę całkowitą — minimalną liczbę operacji potrzebnych do wyrównania gruntu lub liczbę -1, jeśli wyrównanie gruntu w dolinie za pomocą podanych koparek nie jest w ogóle możliwe.

Przykład

Dla danych wejściowych:

```
5 2 3
1 2 1 1 -1
```

poprawnym wynikiem jest:

```
5
```

Wyjaśnienie do przykładu: *Jedno z możliwych rozwiązań dla przykładowego wejścia to:*

- *zwiększ poziom gruntu na pierwszych dwóch metrach doliny o 2 metry,*
- *zmniejsz poziom gruntu na pierwszych dwóch metrach doliny o 3 metry,*
- *zwiększ poziom gruntu na czterech końcowych metrach doliny o 2 metry,*
- *zwiększ poziom gruntu na ostatnim metrze doliny o 2 metry,*
- *zmniejsz poziom gruntu na czterech końcowych metrach doliny o 3 metry.*

Rozwiązanie

Weźmy ciąg z przykładu:

$$1,\ 2,\ 1,\ 1,\ -1$$

Możemy dołożyć po jego obu stronach nieskończone ciągi zer bez zmiany wyniku:

$$\ldots,\ 0,\ 1,\ 2,\ 1,\ 1,\ -1,\ 0,\ \ldots$$

Rozważmy najpierw przypadek z jedną koparką, zmieniającą poziom ziemi o 1 metr. Widać, że wówczas musi ona być użyta na kolejnych metrach odpowiednio -1, -2, -1, -1, 1 razy. Można udowodnić, że minimalna liczba operacji potrzebnych, aby zrealizować taki schemat (tj. wykonać X_i operacji na i-tym metrze), wynosi

$$(\ldots + |X_{-1} - X_0| + |X_1 - X_2| + \ldots + |X_n - X_{n+1}| + \ldots)/2,$$

a zatem w przykładzie

$$(|0-(-1)| + |-1-(-2)| + |-2-(-1)| + |-1-(-1)| + |(-1)-1| + |1-0|)/2 = 3.$$

Rozważmy teraz przypadek z dwiema koparkami. Przede wszystkim, wszystkie poziomy gruntu muszą być podzielne przez $\mathrm{nwd}(a, b)$ (w przeciwnym przypadku odpowiedź to -1). Dalej założymy, że $\mathrm{nwd}(a, b) = 1$. Jeżeli wyznaczymy dla każdego metra, ile razy będzie na nim użyta koparka $+a$, a ile razy koparka $+b$ — oznaczmy odpowiednie ciągi przez A_i oraz B_i — to minimalny koszt realizacji takiego schematu będzie równy, jak w poprzednim przypadku, połowie sumy $|A_i - A_{i+1}| + |B_i - B_{i+1}|$.

Na początku zauważmy, że możemy wyznaczyć pewne (niekoniecznie optymalne) rozwiązanie (ciągi A_i oraz B_i), korzystając np. z rozszerzonego algorytmu Euklidesa. Następnie można zauważyć, że ponieważ $\mathrm{nwd}(a, b) = 1$, wszystkie pary ciągów rozwiązujące problem powstają z otrzymanego ciągu przez zamienianie (być może wielokrotne) pewnych par (A_i, B_i) na $(A_i + b, B_i - a)$ lub $(A_i - b, B_i + a)$. Możemy uznać taką zamianę za naszą operację bazową i dążyć do zminimalizowania sumy wartości bezwzględnych różnic kolejnych elementów ciągów.

Przyjmijmy $X_i = A_i - A_{i-1}$, $Y_i = B_i - B_{i-1}$. Oczywiście pary (X, Y) oraz (A, B) wyznaczają się wzajemnie. Jeżeli będziemy pracować ze schematem w postaci (X, Y),

nasz wynik będzie równy po prostu połowie sumy $|X_i|+|Y_i|$, a nasza operacja bazowa będzie polegała na zamianie

$$(X_i, Y_i, X_{i+1}, Y_{i+1})$$

na

$$(X_i + b, Y_i - a, X_{i+1} - b, Y_{i+1} + a)$$

lub przeciwnie. Wykonując tę operację wielokrotnie, można zamienić każdą czwórkę postaci (X_i, Y_i, X_j, Y_j) na $(X_i + b, Y_i - a, X_j - b, Y_j + a)$. W istocie więc dowolna liczba operacji bazowych na (X, Y) sprowadza się do m-krotnej zamiany par (X_i, Y_i) na $(X_i + b, Y_i - a)$ oraz m-krotnej zamiany par (X_j, Y_j) na $(X_j - b, Y_j + a)$, dla pewnego m.

Niech $F_i(k)$ będzie równe $|X_i+kb|+|Y_i-ka|$, dla dowolnego k całkowitego. Funkcja F_i jest wypukła, co więcej, zbiór liczb całkowitych można podzielić na co najwyżej trzy przedziały, na których jest to funkcja liniowa. Naszym celem (wciąż równoważnym z początkowym zadaniem!) będzie teraz znalezienie ciągu liczb całkowitych D_i o sumie równej 0, dla którego suma postaci $\sum_i F_i(D_i)$ jest minimalna.

Każde F_i przyjmuje minimum na pewnym spójnym przedziale. Nietrudno wykazać, że istnieje rozwiązanie optymalne, w którym nie ma D_i, D_j, takich że D_i jest mniejsze od wszystkich liczb w optymalnym przedziale F_i, a D_j jest większe od wszystkich liczb w optymalnym przedziale F_j. Daje to dwa przypadki do rozpatrzenia. Załóżmy zatem bez straty ogólności, że D_i nigdy nie będzie mniejsze od wszystkich liczb w optymalnym przedziale F_i. Przyjmijmy M_i jako najmniejszą liczbę, w której F_i przyjmuje minimum. Będzie zatem zachodzić $D_i \geqslant M_i$. Funkcje F_i okrojone do przedziałów $[D_i, +\infty)$ są monotoniczne i wypukłe oraz można je podzielić na co najwyżej trzy funkcje liniowe. Przyjmijmy za wyjściowe wartości $D_i = M_i$, chcemy je zwiększyć o łącznie $-\sum_i M_i$ (jeśli $-\sum_i M_i$ jest ujemna, rozwiązanie nie istnieje w tym przypadku). Możemy posortować funkcje liniowe odpowiadające F_i po współczynnikach i zachłannie przydzielić przesunięcia o łącznie $-\sum_i M_i$ najmniej nieopłacalnym przesunięciom. Rozwiązaniem zadania będzie połowa z minimum po obu przypadkach z sumy $\sum_i F_i(D_i)$.

Złożoność czasowa rozwiązania to $O(n \cdot (\log n + \log (a + b)))$ w związku z wykorzystaniem sortowania oraz algorytmu Euklidesa. Implementacje rozwiązania można znaleźć w plikach `wyr.cpp` i `wyr1.pas`.

Testy

Rozwiązania zawodników były sprawdzane na 10 zestawach testowych. Większość testów została wygenerowana w sposób losowy. Opis pozostałych typów testów:

- „brak rozwiązania”: odpowiedzią jest -1;
- „dużo operacji”: wynik jest duży (relatywnie do rozmiaru danych);
- „długie przedziały”: operacje na czwórkach (X_i, Y_i, X_j, Y_j) wymagają czwórek o dużych różnicach $j - i$;
- „ciąg bardzo zmienny”: znaczące oscylacje w ciągu (h_i).

Nazwa	n	a	b	max h_i	Opis
wyr1a.in	100	85	2	99	losowy
wyr1b.in	97	90	75	90	brak rozwiązania
wyr1c.in	17	54	81	81	losowy
wyr2a.in	200	19	118	199	losowy
wyr2b.in	193	187	78	199	dużo operacji
wyr2c.in	200	17	17	187	losowy
wyr3a.in	200	117	42	198	losowy
wyr3b.in	160	3	2	200	długie przedziały
wyr4a.in	1 000	1 680	719	1 995	losowy
wyr4b.in	1 000	1 955	1 479	1 989	losowy
wyr5a.in	2 000	163	514	2 000	losowy
wyr5b.in	914	974	919	1 000	dużo operacji
wyr6a.in	2 000	709	542	1 999	losowy
wyr6b.in	1 600	3	2	2 000	długie przedziały
wyr7a.in	10 000	975 747	488 245	999 972	losowy
wyr7b.in	10 000	751 933	773 134	999 911	losowy
wyr7c.in	10 000	2	3	300 000	ciąg bardzo zmienny
wyr8a.in	50 000	448 000	495 000	999 993 000	losowy
wyr8b.in	50 000	6 293	541 632	999 965 078	losowy
wyr8c.in	50 000	3	4	1 000 000 000	ciąg bardzo zmienny
wyr9a.in	100 000	494 853	105 495	999 987 309	losowy
wyr9b.in	100 000	613 053	992 562	999 977 022	brak rozwiązania
wyr9c.in	100 000	4	5	1 000 000 000	ciąg bardzo zmienny
wyr10a.in	100 000	160 444 645	135 419 134	999 996 983	losowy
wyr10b.in	100 000	7	3	1 125 000	długie przedziały
wyr10c.in	100 000	166 216 387	279 312 073	999 011 893	losowy
wyr10d.in	100 000	1	2	1 000 000 000	ciąg bardzo zmienny

Marek Cygan, Wojciech Rytter — Treść zadania, Opracowanie
Krzysztof Leszczyński — Program

Dostępna pamięć: 256 MB. *OI, etap III, dzień drugi, 29.03.2012*

Bezpieczeństwo minimalistyczne

Dana jest mapa sieci ulic Mafiogrodu. Sieć ta składa się ze skrzyżowań i łączących je dwukierunkowych ulic. Ulice nie przecinają się nigdzie poza skrzyżowaniami, choć mogą prowadzić tunelami lub estakadami. Między każdą parą skrzyżowań przebiega co najwyżej jedna ulica. Na każdym skrzyżowaniu v jest posterunek policji, w którym jest $p(v)$ policjantów. Aby ulica łącząca skrzyżowania u i v mogła zostać uznana za bezpieczną, w sumie z obu stron tej ulicy musi być co najmniej $b(u,v)$ policjantów. Początkowo dla każdej ulicy zachodzi $p(u) + p(v) \geqslant b(u,v)$.

Z powodu kryzysu burmistrz Bajtazar wydał zarządzenie o **bezpieczeństwie minimalistycznym** *mówiące, że:*

- *z każdego posterunku należy zwolnić pewną liczbę (być może zero) policjantów (liczbę policjantów zwolnionych z posterunku przy skrzyżowaniu v oznaczmy przez $z(v)$),*
- *po przeprowadzeniu zwolnień, łączna liczba policjantów w posterunkach na końcach każdej ulicy, łączącej skrzyżowania u i v, ma być równa dokładnie $b(u,v)$, czyli:*

$$p(u) - z(u) + p(v) - z(v) = b(u,v).$$

Z powyższych zasad nie wynika jednoznacznie, ilu policjantów ma zostać zwolnionych. Bajtazar zastanawia się, jaka jest najmniejsza i największa możliwa liczba zwolnionych policjantów (suma wartości z dla wszystkich skrzyżowań) spełniająca powyższe zasady.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie liczby całkowite n oraz m ($1 \leqslant n \leqslant 500\,000$, $0 \leqslant m \leqslant 3\,000\,000$), oddzielone pojedynczym odstępem, oznaczające liczbę skrzyżowań oraz liczbę ulic w Mafiogrodzie. Skrzyżowania są ponumerowane od 1 do n. W drugim wierszu zapisanych jest dokładnie n nieujemnych liczb całkowitych pooddzielanych pojedynczymi odstępami. Są to liczby policjantów aktualnie zatrudnionych w kolejnych posterunkach, czyli wartości $p(1), p(2), \ldots, p(n)$ ($0 \leqslant p(i) \leqslant 10^6$).

Każdy z kolejnych m wierszy zawiera opis jednej ulicy dwukierunkowej. Opis taki składa się z trzech liczb całkowitych u_i, v_i, $b(u_i,v_i)$ ($1 \leqslant u_i, v_i \leqslant n$, $u_i \neq v_i$, $0 \leqslant b(u_i,v_i) \leqslant 10^6$) oznaczających odpowiednio numery skrzyżowań będących końcami danej ulicy oraz minimalną liczbę policjantów, którzy muszą być zatrudnieni w sumie na obu końcach danej ulicy.

W testach wartych 56% punktów zachodzą dodatkowe warunki $n \leqslant 2\,000$ oraz $m \leqslant 8\,000$.

Wyjście

Jeśli spełnienie wymogów zarządzenia Bajtazara jest możliwe, Twój program powinien wypisać na standardowe wyjście dokładnie jeden wiersz zawierający dwie liczby całkowite oddzielone

pojedynczym odstępem. Mają to być najmniejsza oraz największa liczba policjantów, jakich należy zwolnić, aby spełnić wymogi zarządzenia.

Jeśli spełnienie wymogów zarządzenia nie jest możliwe, Twój program powinien wypisać jeden wiersz zawierający słowo `NIE`.

Przykład

Dla danych wejściowych:

```
3 2
5 10 5
1 2 5
2 3 3
```

poprawnym wynikiem jest:

```
12 15
```

natomiast dla danych:

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

poprawnym wynikiem jest:

```
NIE
```

Rozwiązanie

W sformułowaniu zadania w naturalny sposób występuje nieskierowany graf, którego wierzchołki odpowiadają skrzyżowaniom, natomiast krawędzie — ulicom Mafiogrodu. Zauważmy, że każdą spójną składową grafu możemy rozważać niezależnie, dlatego też bez straty ogólności załóżmy, że graf dany w zadaniu jest spójny.

Rozważmy pierwszy test przykładowy z treści zadania i przez f_1, f_2, f_3 oznaczmy liczbę policjantów zatrudnionych na poszczególnych posterunkach już po przeprowadzonej redukcji etatów. Krawędzie występujące w grafie wyznaczają równania:

$$\begin{aligned} f_1 + f_2 &= 5, \\ f_2 + f_3 &= 3. \end{aligned}$$

Jako że Bajtazar zakazał zatrudniania dodatkowych policjantów, a liczba policjantów nie może być ujemna, dodatkowo mamy ograniczenia:

$$\begin{aligned} 0 \leqslant f_1 &\leqslant p(1) = 5, \\ 0 \leqslant f_2 &\leqslant p(2) = 10, \\ 0 \leqslant f_3 &\leqslant p(3) = 5. \end{aligned}$$

Celem zadania jest wyznaczenie całkowitych wartości f_1, f_2, f_3, w taki sposób, aby spełnione były powyższe równania oraz nierówności, a liczba $f_1 + f_2 + f_3$, czyli ostateczna liczba zatrudnionych policjantów, była możliwie duża (lub mała). Oczywiście powyższe rozumowanie możemy zastosować dla grafu dowolnej wielkości. Wówczas krawędź uv wyznacza *równanie krawędziowe* $f_u + f_v = b(u, v)$, a każdy wierzchołek musi spełniać *nierówność wierzchołkową* $0 \leqslant f_v \leqslant p(v)$.

Rozwiązanie wolne

Zauważmy, że jeśli ustalimy docelową liczbę policjantów pracujących na pierwszym posterunku, to równania krawędziowe jednoznacznie wyznaczą nam liczby policjantów zatrudnionych na pozostałych posterunkach, co wynika z przyjętego założenia o spójności grafu. Żądane liczby policjantów można w praktyce wyznaczyć za pomocą dowolnego algorytmu przeszukiwania grafu, np. metody BFS czy DFS. Prowadzi nas to do rozwiązania wolnego, zaimplementowanego w pliku `bezs0.cpp`, o złożoności $O(p(1)(n+m))$, gdyż docelowa liczba policjantów na pierwszym posterunku musi być całkowita i należeć do przedziału $[0, p(1)]$.

Możemy również wykorzystać sytuację, w której początkowa liczba policjantów na poszczególnych posterunkach znacznie się różni. Zamiast wyznaczać docelową liczbę policjantów na pierwszym posterunku, możemy użyć posterunku o najmniejszej liczbie początkowo zatrudnionych policjantów. Takie rozwiązanie znajdziemy w pliku `bezs1.cpp`.

Omawiane rozwiązania pozwalały na uzyskanie 56 punktów.

Rozwiązanie wzorcowe

Zamiast rozpoczynać nasze rozwiązanie od wyznaczenia wartości f_1, potraktujmy tę wartość jako zmienną, czyli przyjmijmy $f_1 = x$. Rozpatrując dowolne drzewo rozpinające grafu ukorzenione w wierzchołku 1, za pomocą równań krawędziowych możemy jednoznacznie wyznaczyć wartości f_i jako liniowe funkcje zmiennej x. Jeśli dla wierzchołka v określiliśmy już funkcję $f_v(x) = a_v x + c_v$, a wierzchołek u jest synem wierzchołka v w drzewie, to

$$f_u(x) = -a_v x + (b(u,v) - c_v).$$

Co więcej, zauważmy, że każda z tych funkcji będzie miała postać $f_i(x) = x + c_i$ lub $f_i(x) = -x + c_i$.

Graf dany na wejściu nie musi być jednak drzewem, co oznacza, że musimy upewnić się, że równania krawędziowe dla krawędzi spoza drzewa rozpinającego również są spełnione. Rozważmy krawędź uv oraz załóżmy, że funkcje dla wierzchołków u oraz v mają postać $f_u(x) = a_u x + c_u$ oraz $f_v(x) = a_v x + c_v$. Mamy dwa następujące przypadki:

- jeśli $a_u \neq a_v$, to równanie $f_u(x) + f_v(x) = b(u,v)$ przyjmuje postać $c_u + c_v = b(u,v)$. W tej sytuacji w równaniu nie występuje zmienna x, co oznacza, że równanie to jest zawsze spełnione lub też zawsze sprzeczne (w tym drugim wypadku wiemy, że nie istnieje rozwiązanie zadania);
- jeśli natomiast $a_u = a_v$, to równanie krawędziowe przyjmuje postać

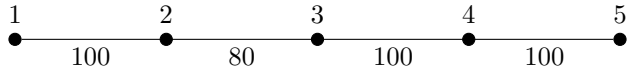
  $$x = \frac{b(u,v) - c_u - c_v}{2a_u},$$

  co pozwala nam jednoznacznie wyznaczyć wartość x i sprawdzić, czy spełnione są dla niej pozostałe równania krawędziowe, jak również nierówności wierzchołkowe. W tym przypadku zadanie ma co najwyżej jedno rozwiązanie, a w związku

z tym najmniejsza i największa liczba zwolnionych policjantów będą miały dokładnie tę samą wartość.

Pozostały nam do rozważenia nierówności wierzchołkowe. Każdy wierzchołek nakłada na zmienną x dwie nierówności, mianowicie dla wierzchołka v oraz funkcji $f_v(x) = a_v x + c_v$ mamy $0 \leqslant a_v x + c_v \leqslant p(v)$. Każdy wierzchołek wyznacza zatem pewien przedział dopuszczalnych wartości dla zmiennej x. W przypadku $a_v = 1$ mamy przedział $[-c_v, p(v) - c_v]$, natomiast w przypadku $a_v = -1$ — przedział $[c_v - p(v), c_v]$. Przecięcie wszystkich przedziałów pochodzących z nierówności wierzchołkowych pozwala nam wyznaczyć przedział $[A, B]$ reprezentujący dopuszczalne wartości dla zmiennej x. Oznacza to, że jeśli zmiennej x przypiszemy dowolną całkowitą wartość z przedziału $[A, B]$, to automatycznie spełnione będą zarówno wszystkie równania krawędziowe, jak i nierówności wierzchołkowe.

Przykładowo, w przypadku poniższego grafu (pojedyncza ścieżka; na krawędziach podano wartości $b(i, j)$) funkcje f_i oraz dopuszczalne przedziały mają następującą postać:



i	1	2	3	4	5
$p(i)$	90	50	100	40	90
$f_i(x)$	x	$100 - x$	$x - 20$	$120 - x$	$x - 20$
przedział x	$[0, 90]$	$[50, 100]$	$[20, 120]$	$[80, 120]$	$[20, 110]$

Przecięcie wszystkich przedziałów to $[80, 90]$. Jeśli do grafu dołożymy krawędź pomiędzy wierzchołkami 2 oraz 4 z wartością $b(2, 4) = 50$, to będziemy mieli tylko jedno rozwiązanie, dla $x = 85$. Gdybyśmy natomiast, zamiast tej krawędzi, dodali krawędź 2–5 z wartością $b(2, 5) = 100$, to nie mielibyśmy już żadnego rozwiązania.

Nie powiedzieliśmy jeszcze, w jaki sposób możemy wyznaczyć najmniejszą i największą liczbę policjantów, których należy zwolnić. W tym celu wystarczy zauważyć, że całkowita liczba zwolnionych policjantów wyraża się wzorem $\sum_{i=1}^{n}(p(i) - f_i(x))$. Jako że suma funkcji liniowych jest funkcją liniową, najmniejsza i największa jej wartość będzie osiągana na krańcach przedziału $[A, B]$, dzięki czemu wystarczy obliczyć liczbę zwolnionych policjantów jedynie dla wartości $x = A$ oraz $x = B$.

Przedstawione rozwiązanie ma złożoność $O(n + m)$. Jego implementację można znaleźć w pliku `bez.cpp`.

Testy

W poniższej tabeli s oznacza liczbę spójnych składowych, natomiast p to najmniejsza z liczb policjantów na poszczególnych posterunkach (tj. minimum z liczb $p(i)$).

Nazwa	**n**	**m**	**s**	**p**
bez1a.in	10	12	2	5
bez1b.in	10	9	1	10
bez1c.in	10	17	1	2
bez1d.in	1	0	1	10
bez2a.in	200	465	2	114
bez2b.in	20	19	1	1 000
bez2c.in	20	49	1	193
bez2d.in	200	411	46	160
bez3a.in	2 000	7 897	15	33
bez3b.in	20	43	1	60
bez3c.in	20	54	1	106
bez3d.in	2 000	5 019	410	14
bez4a.in	2 000	7 920	7	11 959
bez4b.in	20	19	1	1 000
bez4c.in	20	53	1	204
bez4d.in	2 000	5 000	408	30
bez5a.in	500 000	1 215 391	1 881	606
bez5b.in	20	19	1	1 000
bez5c.in	500 000	1 299 594	101 128	2
bez6a.in	100 000	249 888	198	136
bez6b.in	20	45	1	184
bez6c.in	100 000	259 677	20 224	184
bez7a.in	500 000	1 999 856	1 845	1 472
bez7b.in	20	56	1	97
bez7c.in	500 000	499 999	1	1 000 000

Tomasz Idziaszek
Treść zadania, Opracowanie

Alan Kutniewski
Program

Dostępna pamięć: 64 MB.

OI, etap III, dzień drugi, 29.03.2012

Hurtownia

Bajtazar prowadzi hurtownię z materiałami budowlanymi. W tym sezonie hitem są panele podłogowe i większość dochodu hurtowni pochodzi z ich sprzedaży. Niestety, dość często zdarza się sytuacja, że odwiedzający hurtownię klient składa zamówienie, którego nie da się zrealizować, gdyż w magazynie jest zbyt mało paneli. Żeby nie tracić klientów, Bajtazar postanowił zminimalizować liczbę takich przypadków.

W tym celu przygotował plan pracy na najbliższe n dni. Przeanalizował umowy z producentami paneli i na ich podstawie wyznaczył ciąg $a_1, a_2, \ldots, a_n$. Liczba a_i oznacza, że rano i-tego dnia do magazynu zostanie dostarczonych a_i opakowań paneli.

Bajtazar zrobił też zestawienie ofert, które zgłosili klienci hurtowni, i na jego podstawie wyznaczył ciąg $b_1, b_2, \ldots, b_n$. Liczba b_i oznacza, że w południe i-tego dnia w hurtowni zjawi się klient, który będzie chciał zakupić b_i opakowań paneli. Jeśli Bajtazar zdecyduje się na realizację zamówienia klienta, to będzie je musiał zrealizować w całości. Jeśli w momencie wizyty klienta w magazynie jest mniej opakowań paneli niż potrzebuje klient, to Bajtazar będzie zmuszony odrzucić takie zamówienie. Jeśli natomiast paneli jest wystarczająco dużo, to Bajtazar może zdecydować, czy zrealizować zamówienie klienta, czy też nie.

Na podstawie powyższych danych Bajtazar chce stwierdzić, które zamówienia powinien zrealizować, a które odrzucić, tak by liczba odrzuconych zamówień była jak najmniejsza. Zakładamy, że na początku pierwszego dnia magazyn hurtowni jest pusty.

Wejście

W pierwszym wierszu standardowego wejścia znajduje się liczba całkowita n ($1 \leqslant n \leqslant 250\,000$). W drugim wierszu znajduje się ciąg liczb całkowitych $a_1, a_2, \ldots, a_n$ ($0 \leqslant a_i \leqslant 10^9$). W trzecim wierszu znajduje się ciąg liczb całkowitych $b_1, b_2, \ldots, b_n$ ($0 \leqslant b_i \leqslant 10^9$). Liczby w drugim i trzecim wierszu są pooddzielane pojedynczymi odstępami.

W testach wartych 50% punktów zachodzi dodatkowy warunek $n \leqslant 1\,000$.

Wyjście

W pierwszym wierszu standardowego wyjścia Twój program powinien wypisać liczbę całkowitą k oznaczającą maksymalną liczbę zamówień, które uda się zrealizować Bajtazarowi. W drugim wierszu należy wypisać rosnący ciąg k liczb pooddzielanych pojedynczymi odstępami, oznaczający numery klientów, których zamówienia trzeba w tym celu zrealizować, lub pusty wiersz, gdy nie można zrealizować żadnego zamówienia. Klientów numerujemy od 1 do n w kolejności przychodzenia do hurtowni. Jeśli istnieje więcej niż jedna poprawna odpowiedź, Twój program powinien wypisać dowolną z nich.

Przykład

Dla danych wejściowych:	*jednym z poprawnych wyników jest:*
`6`	`3`
`2 2 1 2 1 0`	`1 2 4`
`1 2 2 3 4 4`	

Rozwiązanie

Podejście pierwsze – programowanie dynamiczne

Nasze zadanie ma dość naturalne rozwiązanie oparte na metodzie programowania dynamicznego. Zauważmy, że jeśli w ciągu pierwszych i dni zrealizowaliśmy pewną liczbę zamówień, to aby wyznaczyć, które zamówienia powinny zostać zrealizowane w kolejnych dniach, wystarczy znać liczbę paneli, które zostały w magazynie po i dniach (czyli nie musimy wiedzieć, które konkretnie zamówienia z pierwszych i dni zostały zrealizowane).

Ta obserwacja prowadzi do następującego rozwiązania. Będziemy wypełniać tablicę d. Dla $0 \leqslant i \leqslant n$ i $A \geqslant 0$ mamy

$$d[i, A] = \text{maksymalna liczba zamówień z dni } \{1, \ldots, i\}, \text{ które można zrealizować, aby w magazynie zostało dokładnie } A \text{ paneli.}$$

Jeśli nie da się tak realizować zamówień, aby na końcu zostało dokładnie A paneli, to przyjmujemy, że $d[i, A] = -\infty$. Dla wygody przyjmujemy także $d[i, A] = -\infty$ dla dowolnego $A < 0$.

Jak wyznaczyć $d[i, A]$? Skoro na początku pierwszego dnia magazyn jest pusty, to $d[0, 0] = 0$ oraz $d[0, A] = -\infty$ dla $A > 0$. Załóżmy zatem, że $i \geqslant 1$. Aby wyznaczyć wartość $d[i, A]$, musimy zdecydować, czy realizować i-te zamówienie. Jeśli go nie realizujemy, to po $i - 1$ dniach musiało nam zostać $A - a_i$ paneli, w przeciwnym wypadku musiało zostać $A - a_i + b_i$ paneli. Zatem

$$d[i, A] = \max(d[i - 1, A - a_i], 1 + d[i - 1, A - a_i + b_i]).$$

Wynikiem jest maksymalna wartość spośród $d[n, A]$ dla wszystkich $A \geqslant 0$.

Problemem w tym rozwiązaniu jest to, że A może być bardzo duże. Po prostu niewłaściwie wybraliśmy, po czym indeksujemy tablicę. Spróbujmy inaczej; dla $0 \leqslant i \leqslant n$ i $0 \leqslant k \leqslant n$ mamy

$$d[i, k] = \text{maksymalna liczba paneli, które mogą zostać w magazynie po zrealizowaniu } k \text{ zamówień w dniach } \{1, \ldots, i\}.$$

Przyjmujemy, że $d[i, k] = -\infty$, jeśli w dniach $\{1, \ldots, i\}$ nie da się zrealizować k zamówień. Znowu $d[0, 0] = 0$ oraz $d[0, k] = -\infty$ dla $k > 0$. Gdy $i \geqslant 1$, musimy znów zdecydować, czy realizujemy i-te zamówienie. Jeśli nie, to w ciągu pierwszych $i - 1$ dni musimy zrealizować k zamówień, a następnie dokładamy do magazynu a_i paneli. A jeśli tak, to w ciągu pierwszych $i-1$ dni musimy zrealizować $k-1$ zamówień,

a i-tego dnia dokładamy a_i paneli i zabieramy b_i. Należy pamiętać, że drugi z tych scenariuszy jest możliwy tylko wtedy, gdy maksymalna liczba paneli, które zostaną po zrealizowaniu $k-1$ zamówień w ciągu pierwszych $i-1$ dni, powiększona o a_i jest co najmniej taka, jak b_i.

Ostatecznym wynikiem jest maksymalna wartość k taka, że $d[n,k] \neq -\infty$, tzn. taka, dla której w ciągu n dni da się zrealizować k zamówień. Poniżej przedstawiamy to rozwiązanie zapisane w postaci pseudokodu.

```
1:  d[0,0] := 0;
2:  for k := 1 to n do
3:    d[0,k] := −∞;
4:  for i := 1 to n do
5:    for k := 0 to n do begin
6:      d[i,k] := d[i−1,k] + a[i];
7:      if k > 0 and d[i−1,k−1] + a[i] ⩾ b[i] then
8:        d[i,k] := max(d[i,k], d[i−1,k−1] + a[i] − b[i]);
9:    end
10: wynik := n;
11: while d[n,wynik] = −∞ do
12:   wynik := wynik − 1;
13: return wynik;
```

Wyznaczenie na podstawie tablicy d, które zamówienia należy zrealizować, zostawiamy jako ćwiczenie dla Czytelnika. Złożoność czasowa i pamięciowa tego rozwiązania to $O(n^2)$. Takie rozwiązanie wystarczało, by zaliczyć połowę testów przygotowanych przez jurorów. Jego implementacje można znaleźć w plikach `hurs1.cpp` i `hurs2.pas`.

Podejście drugie – algorytm zachłanny

Pierwsze pytanie, które moglibyśmy sobie zadać, próbując rozwiązać nasze zadanie, jest następujące: jeśli będziemy zgadzać się na realizację każdego zamówienia, które jest możliwe do zrealizowania, to czy w ten sposób zawsze uzyskamy optymalne rozwiązanie? Dość szybko powinniśmy dojść do wniosku, że odpowiedź na to pytanie jest negatywna: dla $n = 3$ i ciągów $a = (2,0,0)$, $b = (2,1,1)$ taka strategia przydzieli obydwa panele pierwszemu klientowi, natomiast rozwiązanie optymalne przydzieli po jednym panelu drugiemu i trzeciemu klientowi. Nic w tym jednak dziwnego — zadanie nie byłoby ciekawe, gdyby takie rozwiązanie było poprawne. Zostało ono zaimplementowane w pliku `hurb1.cpp`. Nie przechodzi żadnego testu.

Nie porzucajmy jednak myśli o rozwiązaniu zachłannym. Spróbujmy nieco zmienić zadanie. Powiedzmy, że zanim jeszcze pierwszy klient przyjdzie do hurtowni, wszyscy klienci postanowili potwierdzić możliwość realizacji swojego zamówienia telefonicznie. Pytanie brzmi: czy istnieje taka kolejność telefonów, że jeśli będziemy odpowiadali klientom zachłannie (tzn. zamówienie akceptujemy, jeśli da się je zrealizować, biorąc pod uwagę wszystkie poprzednio zaakceptowane zamówienia), to uzyskane rozwiązanie będzie optymalne? Okazuje się, że tak — jeśli klienci będą dzwonić w kolejności niemalejących wielkości zamówień, to strategia zachłanna da optymalne rozwiązanie.

Dowód poprawności strategii zachłannej

Na początek zastanówmy się, jak należałoby sprawdzać, czy da się zrealizować kolejne zamówienie. Najłatwiej byłoby dla każdego klienta, którego zamówienie zdecydujemy się zrealizować, oznaczać panele, które mu sprzedamy. To prowadzi do pytania: które to będą panele? Dla ustalenia uwagi załóżmy, że dzwoni do nas klient o numerze i. Możemy mu przydzielić dowolny podzbiór b_i nieoznaczonych jeszcze paneli z dni $\{1, \ldots, i\}$. Zauważmy, że lepiej przydzielać mu panele z jak najpóźniejszych dni. Istotnie, jeśli później zadzwoni jakiś klient $j < i$, to nie będą go interesowały panele z dni $\{j+1, \ldots, i\}$. A dla klientów $j > i$ nie ma różnicy.

Niech ciąg $a'_1, \ldots, a'_i$ oznacza liczbę nieoznaczonych dotąd paneli spośród paneli dostarczonych do magazynu w poszczególnych dniach, a ciąg $c_1, \ldots, c_i$ — liczbę paneli, które oznaczymy dla klienta i (oczywiście, $0 \leqslant c_j \leqslant a'_j$ dla każdego j). Powiemy, że klientowi i przydzielamy *najświeższe* panele, jeśli $c_1 + \ldots + c_i = b_i$ oraz istnieje takie k, że

$$c_j = \begin{cases} 0 & \text{dla } j < k, \\ a'_j & \text{dla } j > k. \end{cases}$$

Niech $X \subseteq \{1, \ldots, n\}$ będzie podzbiorem klientów. Powiemy, że X *da się zrealizować*, jeśli można zrealizować zamówienia wszystkich klientów z X. Zamówienie klienta o numerze i możemy zrealizować wtedy, gdy i-tego dnia mamy w magazynie, po realizacji poprzednich zamówień, co najmniej b_i paneli. Oczywiście zadanie polega na znalezieniu najliczniejszego zbioru X, który da się zrealizować.

Niech X będzie podzbiorem klientów, a $\prec$ pewnym uporządkowaniem (porządkiem liniowym) elementów zbioru X. Powiemy, że X *da się zrealizować zachłannie względem* $\prec$, jeśli przydzielając kolejnym klientom z X według porządku $\prec$ najświeższe panele, zrealizujemy zamówienia wszystkich klientów z X.

Poniższy lemat pokazuje, że przydział według najświeższych paneli jest zawsze optymalny, niezależnie od wybranego porządku realizacji zamówień:

Lemat 1. Niech X będzie podzbiorem klientów, który da się zrealizować. Wtedy X da się zrealizować zachłannie względem dowolnego uporządkowania $\prec$.

Dowód: Załóżmy przeciwnie, że dla pewnego uporządkowania $\prec$, zbioru X nie udało się zrealizować zachłannie względem $\prec$. Zatrzymajmy się w momencie, gdy dla pewnego klienta j zabrakło paneli. Oznaczmy przez J zbiór klientów x, takich że $x \prec j$ lub $x = j$. Czyli do zbioru J należą ci klienci, którym już zrealizowaliśmy zamówienia, oraz klient j, któremu przydzieliliśmy część zamówionych przez niego paneli.

Niech i będzie największą liczbą, taką że wszystkie panele z dni $\{1, \ldots, i\}$ są oznaczone, ale część paneli z dnia $i+1$ pozostała nieoznaczona (oczywiście $i < n$, bo inaczej znaczyłoby, że X nie da się zrealizować). Wszystkie panele z dni $\{1, \ldots, i\}$ przydzielaliśmy tylko klientom ze zbioru $J_i = J \cap \{1, \ldots, i\}$, gdyż w przeciwnym przypadku wszystkie panele z dnia $i+1$ musiałyby być oznaczone. Również $j \in J_i$, bo inaczej można byłoby mu przydzielić kolejny panel z dnia $i+1$. Ponieważ ostatecznie zabrakło nam paneli, więc

$$\sum_{x \in J_i} b_x > a_1 + \ldots + a_i,$$

zatem $J_i \subseteq X$ nie da się zrealizować. Sprzeczność. ■

Twierdzenie 1. *Niech $\prec$ będzie uporządkowaniem wszystkich klientów niemalejąco względem ich zapotrzebowań (tzn. $i \prec j$ jeśli $b_i \leqslant b_j$; remisy rozstrzygamy dowolnie). Wtedy realizowanie zamówień zachłanne względem $\prec$ daje optymalne rozwiązanie.*

Dowód: Niech ALG to zbiór klientów wyznaczonych do realizacji ich zamówień przez nasz algorytm, zaś OPT to zbiór klientów wyznaczonych w pewnym rozwiązaniu optymalnym. Niech $OPT = \{i_1, i_2, \ldots, i_m\}$, przy czym $i_1 \prec i_2 \prec \ldots \prec i_m$. Niech l będzie największą liczbą, taką że $\{i_1, \ldots, i_l\} \subseteq ALG$. Zakładamy, że OPT, spośród wszystkich rozwiązań optymalnych, daje największe l (tzn. ma możliwie najdłuższy wspólny prefiks z ALG).

Na początek rozważmy przypadek, gdy OPT jest właściwym nadzbiorem ALG. Mamy wówczas $i_{l+1} \notin ALG$, choć z lematu wynika, że $ALG \cup \{i_{l+1}\} \subseteq OPT$ da się zrealizować zachłannie względem $\prec$. Sprzeczność.

Załóżmy zatem, że $ALG \setminus OPT \neq \emptyset$. Niech x będzie pierwszym klientem względem uporządkowania $\prec$, takim że $x \in ALG$ i $x \notin OPT$. Pokażemy teraz inne rozwiązanie optymalne, które będzie zawierało $\{i_1, \ldots, i_l, x\} \subseteq ALG$, a zatem znowu dojdziemy do sprzeczności, gdyż to rozwiązanie będzie miało dłuższy prefiks wspólny z ALG.

Na początek zrealizujmy zachłannie $\{i_1, \ldots, i_l\}$ względem $\prec$. Oznaczmy teraz panele, które zachłannie przydzielilibyśmy klientowi x, ale faktycznie ich nie przydzielajmy. Następnie zrealizujmy zachłannie $\{i_{l+1}, \ldots, i_m\}$ względem uporządkowania $>$, czyli w kolejności *malejących numerów klientów*. Na mocy lematu, w ten sposób przydzielimy panele wszystkim klientom z OPT.

Zauważmy teraz, że $x \prec i_{l+1}, \ldots, i_m$, gdyż w przeciwnym razie w rozwiązaniu ALG zamiast klienta x wybralibyśmy klienta i_{l+1}. Innymi słowy, zamówienia klientów $i_{l+1}, \ldots, i_m$ są co najmniej tak liczne, jak zamówienie klienta x. To pozwala nam dokonać kluczowej obserwacji: panele, które oznaczyliśmy dla klienta x, zostały przydzielone co najwyżej dwóm klientom ze zbioru $\{i_{l+1}, \ldots, i_m\}$.

Jeśli nie przydzieliliśmy tych paneli nikomu, to mamy sprzeczność — przydzielając je teraz klientowi x, dostajemy rozwiązanie liczniejsze niż OPT.

Jeśli przydzieliliśmy je jednemu klientowi $u \in \{i_{l+1}, \ldots, i_m\}$, to anulujemy realizację jego zamówienia i przydzielamy oznaczone panele klientowi x.

Jeśli przydzieliliśmy je dwóm klientom $u, v \in \{i_{l+1}, \ldots, i_m\}$, $u < v$, to anulujemy realizację zamówienia u, przydzielamy oznaczone panele klientowi x, a z nieoznaczonych paneli u rekompensujemy stratę klienta v. W ten sposób klientowi x przydzieliliśmy b' paneli klienta v, a temu drugiemu daliśmy ich w zamian $b_u - (b_x - b') \geqslant b'$.

Tak więc jedyna możliwość, która pozostała, to $ALG = OPT$. Zatem rozwiązanie wyznaczane przez nasz algorytm jest optymalne. ■

Powyższy dowód poprawności został zaproponowany przez Jakuba Pachockiego.

Implementacja algorytmu: rozwiązanie wzorcowe

W algorytmie zachłannym przeglądamy zatem klientów w kolejności niemalejących zapotrzebowań i realizujemy zamówienia zawsze, kiedy się da.

Będziemy trzymać tablicę $a[1..n]$, gdzie $a[i]$ oznacza liczbę nieoznaczonych jeszcze paneli z i-tego dnia. Zamówienie klienta i można zrealizować, gdy:

$$a[1] + a[2] + \ldots + a[i] \geqslant b_i.$$

Jeśli warunek jest spełniony, to musimy oznaczyć b_i paneli, czyli uaktualnić tablicę a. Robimy to, przeglądając kolejno elementy od $a[i]$ do $a[1]$ i za każdym razem odejmując tyle, ile możemy, dopóki w sumie nie odejmiemy b_i. Całość została zapisana w poniższym pseudokodzie (zignorujmy na razie wiersz 12).

```
 1: z := (1, ..., n);
 2: Posortuj ciąg z[i] w kolejności niemalejących zapotrzebowań b[z[i]];
 3: wynik := 0;
 4: for i := 1 to n do begin
 5:    B := b[z[i]];
 6:    A := a[1] + ... + a[z[i]];
 7:    if A ⩾ B then begin
 8:       wynik := wynik + 1;
 9:       zgłoś zlecenie z[i] do realizacji;
10:       j := z[i];
11:       while B > 0 do begin
12:          j := ostatniNiezerowy(j);
13:          if B ⩽ a[j] then begin
14:             a[j] := a[j] − B;
15:             B := 0;
16:          end else begin
17:             B := B − a[j];
18:             a[j] := 0;
19:          end
20:          j := j − 1;
21:       end
22:    end
23: end
24: return wynik;
```

Przeanalizujmy złożoność czasową naszego algorytmu. Sortowanie w wierszu 2 realizujemy w czasie $O(n \log n)$. Wyznaczenie wartości A z wiersza 6 możemy zrealizować w czasie $O(z[i])$; również pętla z wiersza 11 wykona co najwyżej $z[i]$ obrotów, gdyż warunek z wiersza 7 gwarantuje, że j będzie zawsze dodatnie. Tak więc cała pętla **for** z wiersza 4 wykonuje się w czasie $O(1 + 2 + \ldots + n) = O(n^2)$. Takie rozwiązanie zostało zaimplementowane w plikach `hurs3.cpp` i `hurs4.pas`.

No cóż, nie widać zysku w porównaniu z algorytmem dynamicznym. Spróbujmy więc zoptymalizować nasz algorytm. Potrzebujemy struktury danych, która dla tablicy $a[1..n]$ umożliwi nam obliczanie sum prefiksowych (wiersz 6) i uaktualnianie wartości elementów (wiersze 14 i 18). Stosując odpowiednie *drzewo przedziałowe* (patrz dalej), obie te operacje można zrealizować w czasie $O(\log n)$. Wciąż jednak wykonanie pętli z wiersza 11 może za każdym razem wymagać pełnych $z[i]$ obrotów (jest tak np. w sytuacji, gdy wszystkie panele zostały dostarczone pierwszego dnia). Przeanalizujmy to dokładniej. Jeśli warunek w wierszu 13 jest prawdziwy, to zerujemy B i pętla się kończy. Wobec tego cała praca jest wykonywana w wierszach 17–18. Zauważmy, że wiersz 18 zeruje element $a[j]$. Wynika z tego, że podczas całego programu ten wiersz wykona się co najwyżej n razy dla dodatniego elementu $a[j]$. Ale przypadek $a[j] = 0$

jest dla nas mało interesujący — oznacza, że wszystkie panele z j-tego dnia zostały już przydzielone. Możemy więc ograniczyć się do rozpatrywania tylko tych wartości j, dla których $a[j] > 0$. To właśnie wykonuje wiersz 12: funkcja *ostatniNiezerowy*(j) zwraca największą liczbę $x \leqslant j$, dla której $a[x] > 0$. Jak pokażemy poniżej, funkcję *ostatniNiezerowy* możemy również zrealizować na drzewie przedziałowym w czasie $O(\log n)$.

Podsumowując, pętla z wiersza 11 sumarycznie będzie miała co najwyżej $2n$ obrotów. Tak więc cały algorytm będzie miał złożoność czasową $O(n \log n)$. Został on zapisany w plikach `hur.cpp` i `hur1.pas`.

Implementacja drzewa przedziałowego

Niech 2^m będzie najmniejszą potęgą dwójki większą niż n. Trzymamy dwie tablice $sum[1..2^{m+1}-1]$ i $ost[1..2^{m+1}-1]$. Element $sum[2^l+i]$ dla $0 \leqslant i < 2^l$ oznacza sumę wartości a na przedziale $[i \cdot 2^{m-l}, (i+1) \cdot 2^{m-l})$. Natomiast element $ost[2^l+i]$ oznacza największe $j \in [i \cdot 2^{m-l}, (i+1) \cdot 2^{m-l})$, takie że $a[j] > 0$ (lub $-\infty$, gdy takie j nie istnieje). Na początku tablicę *sum* wypełniamy zerami, a tablicę *ost* wartościami $-\infty$.

Wykonanie przypisania $a[i] := v$ realizuje procedura *ustaw*(i, v). Można ją też wykorzystać do zainicjowania drzewa początkowym ciągiem a.

```
 1: procedure ustaw(i, v)
 2: begin
 3:   i := i + 2^m;
 4:   sum[i] := v;
 5:   if v > 0 then ost[i] := i; else ost[i] := −∞;
 6:   while i > 1 do begin
 7:     i := i div 2;
 8:     sum[i] := sum[2i] + sum[2i + 1];
 9:     ost[i] := max(ost[2i], ost[2i + 1]);
10:   end
11: end
```

Obliczenie sumy $a[1] + \ldots + a[i]$ realizuje poniższa funkcja *sumaPrefiksowa*(i). Ponieważ funkcja *ostatniNiezerowy*(i) jest analogiczna, w komentarzach zaznaczono, czym różnią się obie funkcje.

```
 1: function sumaPrefiksowa(i)          { ostatniNiezerowy(i) }
 2: begin
 3:   i := i + 2^m;
 4:   wynik := sum[i];                  { wynik := ost[i]; }
 5:   while i > 1 do begin
 6:     if i mod 2 = 1 then
 7:       wynik := wynik + sum[i − 1];  { wynik := max(wynik, ost[i − 1]); }
 8:     i := i div 2;
 9:   end
10:   return wynik;
11: end
```

Wszystkie trzy funkcje działają w czasie $O(m) = O(\log n)$. Dodajmy, że można zrezygnować z utrzymywania tablicy *ost*, jednak wtedy funkcja *ostatniNiezerowy* będzie musiała zostać przepisana w innej formie:

```
1:  function ostatniNiezerowy(i)
2:  begin
3:    s := sumaPrefiksowa(i);
4:    j := 1;
5:    while j < 2^m do begin
6:      j := 2j;
7:      if sum[j] < s then begin
8:        s := s − sum[j];
9:        j := j + 1;
10:     end
11:   end
12:   return j;
13: end
```

Epilog

Warto dodać, że jeśli zamiast maksymalizować liczbę zrealizowanych zamówień, będziemy chcieli maksymalizować liczbę sprzedanych paneli, to problem stanie się NP-zupełny, wobec tego prawdopodobnie nie da się go rozwiązać w czasie wielomianowym. Zainteresowani Czytelnicy mogą to łatwo udowodnić przez sprowadzenie do niego *problemu sumy podzbioru* (patrz książka [23]):

> Mając dany zbiór liczb $X = \{b_1, \ldots, b_n\}$ oraz liczbę S, sprawdzić, czy istnieje podzbiór zbioru X sumujący się do S.

Testy

Rozwiązania zawodników były sprawdzane na 12 zestawach testowych. W większości testów ciąg a składa się głównie z niewielkich wartości, a ciąg b jest losowy. Rozwiązania wykładnicze, które sprawdzają wszystkie możliwe wyniki (patrz pliki `hurs5.cpp` i `hurs6.pas`), przechodziły maksymalnie dwa pierwsze testy.

Nazwa	n
hur1a.in	10
hur1b.in	1
hur1c.in	10
hur2.in	20
hur3.in	80
hur4.in	290

Nazwa	n
hur5.in	742
hur6.in	1 000
hur7.in	9 200
hur8.in	25 400
hur9.in	52 023

Nazwa	n
hur10a.in	85 000
hur10b.in	85 000
hur11a.in	140 123
hur11b.in	140 123
hur12a.in	250 000
hur12b.in	250 000

Jacek Tomasiewicz — Treść zadania
Jakub Radoszewski — Opracowanie
Dawid Dąbrowski, **Jan Kanty Milczek** — Program

Dostępna pamięć: 64 MB. *OI, etap III, dzień drugi, 29.03.2012*

Prefiksufiks

W tym zadaniu będą nas interesować napisy złożone z małych liter alfabetu angielskiego. **Prefiksem** *danego napisu nazwiemy dowolny jego początkowy fragment.* **Sufiksem** *danego napisu nazwiemy dowolny jego końcowy fragment. W szczególności, pusty napis jest zarówno prefiksem, jak i sufiksem dowolnego napisu. Dwa napisy nazywamy* **cyklicznie równoważnymi**, *jeżeli jeden z nich można uzyskać z drugiego, przestawiając pewien jego sufiks z końca napisu na początek. Dla przykładu, napisy* `ababba` *i* `abbaab` *są równoważne cyklicznie, a napisy* `ababba` *i* `ababab` *nie są. W szczególności, każdy napis jest sam sobie cyklicznie równoważny.*

Dany jest napis t złożony z n liter. Szukamy jego prefiksu p i sufiksu s, obu tej samej długości, takich że:

- *p i s są sobie równoważne cyklicznie,*
- *długość p i s nie przekracza $\frac{n}{2}$ (czyli prefiks p i sufiks s nie zachodzą na siebie w t), oraz*
- *długość p i s jest jak największa.*

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą n ($1 \leqslant n \leqslant 1\,000\,000$), oznaczającą długość danego napisu t. Drugi wiersz wejścia zawiera napis t składający się z n małych liter alfabetu angielskiego.

W testach wartych 30% punktów zachodzi dodatkowy warunek $n \leqslant 500$.

W testach wartych 50% punktów zachodzi dodatkowy warunek $n \leqslant 5\,000$.

Wyjście

Twój program powinien wypisać w pierwszym i jedynym wierszu standardowego wyjścia jedną liczbę całkowitą, równą długości szukanego prefiksu p i sufiksu s.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`15` `ababbababababbaab`	`6`

Rozwiązanie

Problem postawiony w zadaniu jest uogólnieniem problemu znajdowania *prefikso-sufiksów* słowa t, czyli prefiksów słowa t, które są zarazem jego sufiksami. Znany jest liniowy algorytm wyznaczający wszystkie prefikso-sufiksy słowa — mowa tu o funkcji prefiksowej z algorytmu wyszukiwania wzorca w tekście metodą Knutha-Morrisa-Pratta (patrz np. książki [21, 23]). Przedstawiony w tym zadaniu problem *prefiksufiksów* jest bardziej złożony: szukany sufiks s może być dowolnym słowem równoważnym cyklicznie prefiksowi p. Nasz opis rozpocznijmy od analizy rozwiązań nieoptymalnych.

Rozwiązania wolniejsze

Załóżmy, że litery słowa t są ponumerowane od 1 do n. Najprostsze rozwiązanie zadania polega na sprawdzeniu wszystkich możliwości: dla każdej wartości parametru $m = 0, 1, \ldots, \lfloor \frac{n}{2} \rfloor$ chcemy stwierdzić, czy prefiks p i sufiks s o długości m są równoważne cyklicznie.

Jak sprawdzić, czy dwa słowa są równoważne cyklicznie? Najłatwiej zrobić to tak: m razy przenosimy pierwszą literę pierwszego słowa na jego koniec, za każdym razem sprawdzając, czy otrzymaliśmy w ten sposób drugie słowo. Ta metoda wymaga wykonania $O(m^2)$ operacji, gdyż sprawdzenie równości słów długości m zajmuje czas $O(m)$. Można jednak lepiej. Zauważmy, że p i s są równoważne cyklicznie wtedy i tylko wtedy, gdy wzorzec s występuje jako spójny fragment tekstu pp (p sklejone z p). Problem wyszukiwania wzorca w tekście można rozwiązać w czasie liniowym np. za pomocą wspomnianego już algorytmu Knutha-Morrisa-Pratta. W ten sposób otrzymujemy algorytm sprawdzania równoważności cyklicznej słów o złożoności czasowej $O(m)$. Znane są też bardziej wysublimowane algorytmy działające w tej samej złożoności czasowej, patrz opis rozwiązania zadania *Naszyjniki* z VIII Olimpiady Informatycznej [8] czy książka [21].

Stosując naiwną metodę sprawdzania równoważności cyklicznej słów, otrzymujemy rozwiązanie zadania o złożoności czasowej $O(n^3)$ (zaimplementowane w plikach `pres1.cpp` oraz `pres2.pas`), które — zgodnie z treścią zadania — uzyskiwało ok. 30 punktów. Z kolei użycie jednej z efektywnych metod o koszcie czasowym $O(m)$ daje rozwiązanie o złożoności $O(n^2)$ (patrz `pres3.cpp`, `pres4.cpp`, `pres5.pas` i `pres6.pas`), uzyskujące ok. 50 punktów.

Rozwiązanie wzorcowe

Kluczowa własność słów

Niech s^R oznacza odwrócenie słowa s, czyli słowo złożone z tych samych liter co s, tylko zapisanych od prawej do lewej. W dalszym opisie będziemy korzystać z następującej, prostej własności słów, której uzasadnienie pozostawiamy Czytelnikowi.

Fakt 1. *Dla dowolnych słów s, u, v, jeśli $s = uv$, to $s^R = v^R u^R$.*

Podzielmy dane słowo t na równej długości połówki, t_1 i t_2. Jeśli t ma nieparzystą długość, możemy usunąć środkową literę słowa, gdyż i tak nie przyda się ona w rozwiązaniu. Kluczowy pomysł polega na rozważeniu słowa będącego *przeplotem* słów t_1 i t_2^R, które otrzymujemy, biorąc po kolei: jedną literę z t_1, jedną z t_2^R, jedną z t_1, jedną z t_2^R itd. Taki przeplot oznaczymy jako $Q(t_1, t_2^R)$.

Przykład 1. Rozważmy słowo $t =$ `ababbababababbaab` z przykładu z treści zadania. W tym przypadku mamy $t_1 =$ `ababbab`, $t_2 =$ `babbaab`, $t_2^R =$ `baabbab`, więc przeplot słów t_1 i t_2^R ma postać:

$$Q(t_1, t_2^R) = \texttt{abbaaabbbbaabb}.$$

Przyjrzyjmy się teraz, jaki związek z naszym zadaniem ma tak zdefiniowany przeplot. Rozważmy najpierw wspomniany na początku opisu prostszy wariant zadania, w którym szukamy najdłuższego prefikso-sufiksu słowa, czyli najdłuższego prefiksu słowa będącego jednocześnie jego sufiksem (przy czym prefiks i sufiks nie mogą na siebie nachodzić). Przypomnijmy, że *palindrom* to słowo czytane tak samo normalnie oraz wspak, np. `anna`.

Obserwacja 1. Słowo t o długości n ma prefikso-sufiks długości m, $m \leqslant n/2$, wtedy i tylko wtedy, gdy prefiks długości $2m$ przeplotu $Q(t_1, t_2^R)$ jest palindromem.

Dowód: Zacznijmy od uzasadnienia implikacji „w prawo". Załóżmy, że słowo t ma prefiks p o długości m, który jest zarazem sufiksem słowa t. Połówki słowa t mają wtedy postać $t_1 = pt_1'$ i $t_2 = t_2'p$, gdzie t_1' i t_2' są słowami o tej samej długości (być może pustymi). Na mocy faktu 1 mamy $t_2^R = p^R(t_2')^R$. Nasz przeplot wygląda zatem tak:

$$Q(t_1, t_2^R) = Q(pt_1', p^R(t_2')^R) = Q(p, p^R)Q(t_1', (t_2')^R).$$

Wystarczy teraz zauważyć, że przeplot $Q(p, p^R)$ o długości $2m$ jest palindromem, gdyż: zaczyna się i kończy pierwszą literą słowa p, drugą i przedostatnią literą w tym przeplocie jest druga litera słowa p itd.

Uzasadnienie implikacji „w lewo" jest bardzo podobne (jeśli się dobrze przyjrzeć, właściwie już je wykonaliśmy). ■

Naszym celem jest teraz uogólnić obserwację 1 na przypadek prefiksufiksu, czyli sytuację, w której szukany prefiks p i sufiks s są cyklicznie równoważne. W takim przypadku zachodzi $p = uv$ i $s = vu$ dla pewnych słów u i v (być może pustych). Mamy zatem $t_1 = uvt_1'$, $t_2 = t_2'vu$ dla pewnych równej długości słów t_1' i t_2', czyli $t_2^R = u^Rv^R(t_2')^R$ i:

$$Q(t_1, t_2^R) = Q(u, u^R)Q(v, v^R)Q(t_1', (t_2')^R).$$

To pozwala nam już sformułować odpowiednią obserwację:

Obserwacja 2. Słowo t o długości n ma prefiksufiks długości m, $m \leqslant n/2$, wtedy i tylko wtedy, gdy prefiks długości $2m$ przeplotu $Q(t_1, t_2^R)$ jest sklejeniem dwóch palindromów parzystych.

Przykład 2. Przyjrzyjmy się przeplotowi z przykładu 1. Prefiks tego przeplotu o długości 12:

`abba` · `aabbbbaa`

jest sklejeniem dwóch palindromów, więc odpowiada prefiksowi `ababba` i sufiksowi `abbaab` słowa t, które są cyklicznie równoważne.

Rozwiązanie wzorcowe opiera się na (jakże pomysłowej) charakteryzacji prefiksufiksów zawartej w obserwacji 2. Dokończenie rozwiązania jest już tylko kwestią sprawności technicznej, pod warunkiem, że mamy do dyspozycji algorytm Manachera.

Wyszukiwanie palindromów

Algorytm Manachera, opisany np. w książce [21] lub w opracowaniu zadania *Antysymetria* z XVII Olimpiady Informatycznej [17], wyznacza w czasie liniowym dla każdej pozycji danego słowa promień palindromu parzystego o środku na tej pozycji. Dokładniej, dla danego słowa $s[1..n]$ wyznaczamy tablicę $R[1..n-1]$, w której $R[i]$ to maksymalne takie $j > 0$, że słowo $s[i-j+1..i+j]$ jest palindromem; jeśli żadne takie j nie istnieje, to przyjmujemy $R[i] = 0$. Promienie $R[i]$ zawierają w sobie strukturę wszystkich palindromów parzystych w słowie, gdyż każdy taki palindrom znajduje się w środku jakiegoś palindromu o maksymalnym promieniu.

Przykład 3. Tablica promieni palindromów parzystych dla przeplotu z przykładu 1 wygląda następująco:

i:	1	2	3	4	5	6	7	8	9	10	11	12	13	14
słowo:	a	b	b	a	a	a	b	b	b	b	a	a	b	b
$R[i]$:	0	2	0	1	1	0	1	4	1	0	3	0	1	

Na mocy obserwacji 2, wystarczy teraz pokazać, jak w danym słowie s znaleźć najdłuższy prefiks będący sklejeniem dwóch palindromów parzystych. Przede wszystkim warto zacząć od znalezienia prefiksów słowa będących palindromami parzystymi. Łatwo zauważyć, że prefiks długości $2k$ ma tę własność wtedy i tylko wtedy, gdy $R[k] = k$. Dalsza część algorytmu będzie polegała na rozpatrzeniu każdego kandydata na „drugi palindrom" w prefiksie i dobraniu do niego najlepszego „pierwszego palindromu".

Spróbujmy sformalizować podaną intuicję. Niech $P = \{2k : R[k] = k\}$. Załóżmy, że w poszukiwanym najdłuższym prefiksie drugi palindrom pochodzi z grupy palindromów odpowiadającej promieniowi $R[i]$. Oczywiście, najlepiej byłoby wybrać najdłuższy palindrom z tej grupy, czyli słowo $s[i-j+1..i+j]$ dla $j = R[i]$. Możemy tak zrobić jedynie wtedy, gdy o jeden wcześniejsza pozycja, $i-j$, należy do zbioru P. Jeśli tak nie jest, zamiast tego weźmiemy pierwszą pozycję ze zbioru P następującą po pozycji $i-j$ i przytniemy palindrom $s[i-j+1..i+j]$ tak, aby zaczynał się tuż po tej wybranej pozycji.

Jesteśmy już gotowi na to, by przedstawić nasz algorytm w postaci pseudokodu. Najpierw procedura wyznaczająca tablicę najbliższych pozycji na prawo należących do zbioru P:

```
1:  procedure wyznacz_najbliższe
2:  begin
3:    najbliższy[0] := 0;
4:    for k := 1 to n do najbliższy[k] := ∞;
5:    for k := 1 to n div 2 do
6:      if R[k] = k then  { 2k ∈ P }
7:        najbliższy[2k] := 2k;
8:    for k := n − 1 downto 1 do
9:      najbliższy[k] := min(najbliższy[k], najbliższy[k + 1]);
10: end
```

Następnie funkcja znajdująca optymalny prefiks:

```
1:  function oblicz_wynik
2:  begin
3:    wynik := 0;
4:    for i := 1 to n − 1 do begin
5:      pierwszy := najbliższy[i − R[i]];
6:      if pierwszy ⩽ i then
7:        wynik := max(wynik, 2i − pierwszy);
8:    end
9:    return wynik;
10: end
```

Rozwiązanie wzorcowe ma liniową złożoność czasową. Jego implementacje można znaleźć w plikach `pre.cpp` i `pre1.pas`.

Rozwiązanie alternatywne

Poniższe rozwiązanie ma taką samą złożoność czasową, a do tego opiera się na nieco bardziej intuicyjnych spostrzeżeniach. Ceną, jaką za to płacimy, jest wykorzystanie haszowania na słowach, która to metoda przy złym (albo pechowym) doborze parametrów może działać niepoprawnie. Haszowanie pozwala w czasie stałym odpowiadać na pytania o to, czy dane dwa podsłowa $s[a..a+k]$ i $s[b..b+k]$ słowa s są równe, i wymaga jedynie $O(n)$ wstępnych obliczeń. Więcej na temat tej metody można przeczytać w drugiej części tej sekcji.

Przypomnijmy, że jeśli prefiks p i sufiks s stanowią szukany prefiksufiks, to $p = uv$ i $s = vu$ dla pewnych słów u i v. To oznacza, że słowo t możemy zapisać jako $t = uvxvu$, przy czym x jest pewnym (być może pustym) słowem. Widzimy, że słowo u jest jednym z prefikso-sufiksów słowa t (nie dłuższym niż połowa słowa t). Dalej, słowo v jest prefikso-sufiksem słowa $t' = vxv$, czyli słowa t z usuniętym prefiksem i sufiksem u. Co więcej, słowo v jest najdłuższym prefikso-sufiksem słowa t' nie dłuższym niż połowa tego słowa.

Jak już zauważyliśmy wcześniej, wszystkie prefikso-sufiksy słowa można wyznaczyć w czasie liniowym. W tym celu możemy posłużyć się wspomnianą na wstępie funkcją prefiksową; możemy także wykorzystać haszowanie, które pozwala sprawdzać równość

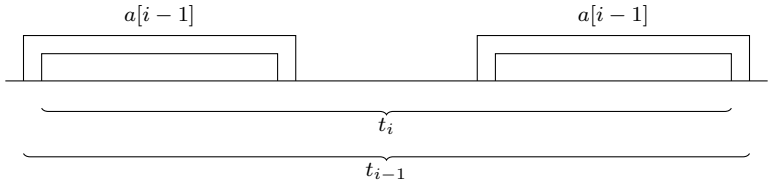
prefiksu i sufiksu słowa w czasie stałym. Aby dokończyć rozwiązanie, wystarczy teraz znaleźć najdłuższy prefikso-sufiks każdego ze słów

$$t_1 = t[1..n], \quad t_2 = t[2..n-1], \quad t_3 = t[3..n-2], \ \ldots$$

o długości nieprzekraczającej połowy długości słowa. Oznaczmy długość tak określonego prefikso-sufiksu słowa t_i jako $a[i]$. Wszystkie komórki tablicy $a[]$ możemy wyznaczyć, dla $i = \lfloor \frac{n}{2} \rfloor, \ldots, 1$, w czasie $O(n)$, korzystając z następującej obserwacji.

Obserwacja 3. $a[i-1] \leqslant a[i] + 2$.

Dowód: Prefikso-sufiks długości $a[i-1]$ słowa t_{i-1} wyznacza w słowie t_i prefikso-sufiks długości $a[i-1]-2$, patrz też rys. 1. ■



Rys. 1: $a[i] \geqslant a[i-1] - 2$.

W naszym algorytmie zastosujemy podejście (pozornie) siłowe. Jako kandydatów na $a[i]$ będziemy rozpatrywać kolejne wartości $j = a[i+1]+2, a[i+1]+1, \ldots, 0$, a sprawdzanie przerwiemy w chwili, gdy stwierdzimy istnienie prefikso-sufiksu słowa t_i o długości j. Aby przypadkiem nie wygenerować prefikso-sufiksu dłuższego niż połowa rozważanego słowa, najlepiej przed wykonaniem algorytmu wstawić w środek słowa t jakiś symbol niebędący literą. Oto pseudokod ilustrujący opisany algorytm:

```
1: j := n div 2;
2: for i := n div 2 downto 1 do begin
3:    j := j + 2;
4:    while j > 0 and t[i..i+j-1] ≠ t[n+1-(i+j-1)..n+1-i] do
5:       { sprawdzenie warunku pętli używa haszowania }
6:       j := j - 1;
7:    a[i] := j;
8: end
```

Łatwo widać, że powyższa pętla **while** wykonuje łącznie tylko $O(n)$ kroków. Faktycznie, każdy obrót tej pętli powoduje zmniejszenie zmiennej j o 1; zmienna ta ma początkowo wartość $\lfloor \frac{n}{2} \rfloor$, nie przyjmuje nigdy wartości ujemnych, a jest zwiększana tylko w instrukcji w 3. wierszu pseudokodu, łącznie o nie więcej niż n. Zakładając, że porównywanie podsłów możemy wykonywać w czasie stałym, powyższy algorytm wyznacza tablicę $a[]$ w czasie liniowym.

Długość szukanego prefiksufiksu obliczamy teraz jako maksimum z wartości postaci $i + a[i+1]$ dla i będących długościami prefikso-sufiksów słowa t, $i \geqslant 0$. Cały algorytm działa więc w czasie $O(n)$. Jego implementacje można znaleźć w plikach `preb1.cpp` i `preb2.pas`.

O wykorzystaniu haszowania

Najpopularniejszą metodą haszowania na słowach jest tak zwane haszowanie wielomianowe, wykorzystywane np. w algorytmie wyszukiwania wzorca w tekście metodą Karpa-Rabina. Wartość funkcji haszującej (tzw. *hasz*) dla słowa obliczamy, traktując poszczególne litery słowa jako współczynniki wielomianu z jedną zmienną i wyznaczając resztę z dzielenia przez M wartości tego wielomianu w wybranym punkcie p:

$$H(t) = (t[1] + t[2]p + t[3]p^2 + \ldots + t[n]p^{n-1}) \bmod M.$$

Parametr M należy dobrać tak, aby wartości haszy nie powodowały przepełnienia wbudowanych typów całkowitych.

Aby móc obliczać hasze dla podsłów słowa t, w ramach obliczeń wstępnych wyznaczamy hasze wszystkich sufiksów słowa t za pomocą schematu Hornera:

$$h[n+1] = 0, \quad h[i] = (t[i] + p \cdot h[i+1]) \bmod M \quad \text{dla} \;\; i = n, n-1, \ldots, 1.$$

Będziemy także potrzebować tablicy pierwszych n potęg liczby p modulo M. Teraz hasze dla podsłów słowa t wyznaczamy już w czasie stałym ze wzoru:

$$H(t[i..j]) = (h[i] - p^{j-i+1}h[j+1]) \bmod M.$$

Jeśli dwa podsłowa słowa t są równe, to na pewno wartości ich haszy są równe. W drugą stronę nie musi to być prawdą: ponieważ nawet bardzo długie słowa reprezentujemy za pomocą hasza będącego liczbą całkowitą z zakresu od 0 do $M-1$, możemy natrafić na *kolizję*, czyli na sytuację, gdy dwa różne słowa mają takie same hasze. Gdy hasze dwóch słów są równe, mamy zatem do wyboru dwie opcje: albo zakładamy, że mamy szczęście i rzeczywiście porównywane słowa są równe, albo dla większej pewności wykonujemy dodatkowe sprawdzenia, na przykład porównujemy litery na kilku losowo wybranych pozycjach słów (względnie na wszystkich pozycjach, jeśli jesteśmy gotowi poświęcić czas działania na rzecz stuprocentowej pewności) lub też sprawdzamy równość haszy przy kilku innych doborach parametrów p oraz M. Dobrą praktyką jest wybieranie jako p oraz M możliwie dużych liczb pierwszych. Ponadto, liczba p powinna być większa niż rozmiar alfabetu (w przeciwnym razie bardzo łatwo o kolizję już dla słów dwuliterowych).

Czasem próbuje się stosować haszowanie z pominięciem parametru M: po prostu wszystkie obliczenia wykonuje się w wybranym typie całkowitym, zazwyczaj 32- albo 64-bitowym. Parametr M jest wówczas tak naprawdę potęgą dwójki o odpowiednim wykładniku, a liczba p powinna być nieparzysta, gdyż w przeciwnym razie wpływ na wartość hasza miałoby jedynie kilkadziesiąt pierwszych liter słowa. Podana metoda pozwala pisać bardzo szybkie programy (unikamy wykonywania kosztownych operacji modulo), okazuje się jednak podatna na kolizje nawet wśród dość typowych rodzin słów.

Przykładem takiej rodziny są słowa Thuego-Morse'a, które pojawiły się np. w zadaniu *Ciągi bez zająknięć* z X Olimpiady Informatycznej [10]. Dla wygody zamiast liter `a` i `b` będziemy tu korzystać z cyfr 0 i 1. Niech $\bar{s}$ oznacza słowo powstałe poprzez negację wszystkich bitów w słowie s. Wówczas ciąg słów Thuego-Morse'a można zdefiniować rekurencyjnie:

$$s_0 \;=\; 0; \qquad s_i \;=\; s_{i-1}\bar{s}_{i-1} \;\; \text{dla} \;\; i > 0.$$

Oto kilka pierwszych wyrazów tego ciągu; zauważmy, że słowo s_i ma długość 2^i:

$$s_0 = 0,\ s_1 = 01,\ s_2 = 0110,\ s_3 = 01101001,\ s_4 = 0110100110010110,\ \ldots$$

Przyjrzyjmy się teraz, dlaczego słowa Thuego-Morse'a są podatne na występowanie kolizji przy haszowaniu z parametrem $M = 2^k$. Oznaczmy przez $W(s)$ hasz słowa s z pominięciem parametru M:

$$W(s) = s[1] + s[2]p + s[3]p^2 + \ldots + s[n]p^{n-1}.$$

Wówczas prawdziwy jest następujący fakt.

Fakt 2. *Dla dowolnego $n \geqslant 0$ oraz $2 \nmid p$ zachodzi $2^{n(n+1)/2} \mid W(\bar{s}_n) - W(s_n)$.*

Dowód: Na mocy definicji rekurencyjnej ($s_n = s_{n-1}\bar{s}_{n-1}$ i $\bar{s}_n = \bar{s}_{n-1}s_{n-1}$) dla $n \geqslant 1$ mamy:

$$\begin{aligned} W(\bar{s}_n) - W(s_n) &= W(\bar{s}_{n-1}) + p^{2^{n-1}}W(s_{n-1}) - W(s_{n-1}) - p^{2^{n-1}}W(\bar{s}_{n-1}) \\ &= W(\bar{s}_{n-1})(1 - p^{2^{n-1}}) - W(s_{n-1})(1 - p^{2^{n-1}}) \\ &= (1 - p^{2^{n-1}})(W(\bar{s}_{n-1}) - W(s_{n-1})) \end{aligned}$$

Stąd łatwo wykazać przez indukcję, że

$$W(\bar{s}_n) - W(s_n) = (1 - p^{2^{n-1}})(1 - p^{2^{n-2}}) \ldots (1 - p).$$

Aby zakończyć dowód, wystarczy udowodnić, że

$$2^i \mid 1 - p^{2^{i-1}} \quad \text{dla każdego } i \geqslant 1. \tag{1}$$

Dowód tej zależności przeprowadzimy przez indukcję. Dla $i = 1$ korzystamy z faktu, że p jest liczbą nieparzystą. Krok indukcyjny (dla $i > 1$) wynika ze wzoru skróconego mnożenia:

$$1 - p^{2^{i-1}} = (1 - p^{2^{i-2}})(1 + p^{2^{i-2}}).$$

Faktycznie, pierwszy z powyższych czynników dzieli się, na mocy założenia indukcyjnego, przez 2^{i-1}, natomiast drugi jest parzysty. To kończy dowód zależności (1). ■

Fakt 2 pozwala nam stwierdzić, że słowa s_n i $\bar{s}_n$ z pewnością spowodują wystąpienie kolizji, jeśli tylko wartość wyrażenia $n(n+1)/2$ będzie co najmniej taka, jak liczba bitów wykorzystywanego typu całkowitego. Tak więc nawet jeśli użyjemy typu 64-bitowego, już słowa s_{11} i $\bar{s}_{11}$ o długości 2048 będą miały tę samą wartość funkcji haszującej, i to niezależnie od wyboru parametru p!

Podsumowując: stosowanie metody haszowania na słowach często prowadzi do efektywnych i prostych algorytmów, należy jednak pamiętać, że otrzymane w ten sposób rozwiązania mogą czasem dawać niepoprawne wyniki, i zachować pewną ostrożność w doborze parametrów tej metody.

Rozwiązania błędne

Spostrzeżenie, że naszym celem jest znalezienie najdłuższego słowa uv spełniającego $t = uvxvu$, prowadzi do kilku możliwych rozwiązań błędnych opartych na podejściu zachłannym. Przykładowo, możemy wybrać u jako najdłuższy prefikso-sufiks słowa t nie dłuższy niż połowa słowa, a następnie wybrać v jako najdłuższy prefikso-sufiks słowa t obustronnie skróconego o u (implementacja tego podejścia: `preb4.cpp`). Takie rozwiązania nie uzyskiwały na zawodach żadnych punktów. Można też jako kandydatów na słowo u rozpatrzyć pewną liczbę najdłuższych prefikso-sufiksów słowa t (implementacje: `preb5.cpp` i `preb6.cpp`). W zależności od szczegółów implementacyjnych takie rozwiązania uzyskiwały co najwyżej 70 punktów.

Aby zobaczyć, że tego typu rozwiązania nie są poprawne, rozważmy słowo t postaci:

`aaaaaaaabaaaaaabaaaaaa`.

Wówczas szukany najdłuższy prefiksufiks pokrywa dokładnie całe słowo:

`aaaaaaaaba` · `aaaabaaaaaa`.

Pierwsze z podanych rozwiązań zachłannych w pierwszym kroku odetnie prefikso-sufiks `aaaaaa`, otrzymując obustronnie skrócone słowo:

`aabaaaaaab`.

W drugim kroku to rozwiązanie zdoła już tylko odciąć prefikso-sufiks `aab`, a zatem wynik przez nie uzyskany nie będzie optymalny. Zauważmy, że wybór w pierwszym kroku prefikso-sufiksu `aaaaa` (czyli o jeden krótszego) również nie prowadzi do znalezienia optymalnego prefiksufiksu, a odpowiedni jest dopiero wybór prefikso-sufiksu `aaaa`. Ogólnie, odpowiednio zwiększając długości podsłów złożonych z samych liter `a` (przy czym drugie podsłowo musi mieć taką samą długość jak trzecie), można uzyskać sytuację, w której rozpatrzenie nawet k najdłuższych prefikso-sufiksów nie prowadzi do optymalnego rozwiązania; tu $k > 0$ może być dowolnie dużą stałą.

Dodajmy jeszcze, że na podstawie ostatniego rozwiązania zachłannego można skonstruować rozwiązanie poprawne, ale o złożoności $O(n^2)$, w którym jako u próbujemy wybierać kolejno wszystkie prefikso-sufiksy słowa t (patrz plik `pres7.cpp`). Takie rozwiązanie uzyskiwało na zawodach ok. 70 punktów.

Testy

Rozwiązania zawodników były sprawdzane za pomocą 10 zestawów danych testowych, z których każdy zawierał od 5 do 6 testów. W teście *1f* występuje słowo jednoliterowe, a w teście *10f* — słowo zawierające milion takich samych liter. Oto charakterystyka pozostałych grup testów:

- grupa a: słowa Thuego-Morse'a;
- grupa b: słowa Fibonacciego (patrz np. zadanie *Słowa* z XVI Olimpiady Informatycznej [16]);

- grupa c: słowa o krótkim powtarzającym się okresie (rzędu $O(\sqrt{n})$);
- grupa d: słowa o dużej liczbie prefikso-sufiksów, sprawiające trudność zachłannym rozwiązaniom błędnym;
- grupa e: słowa składające się z samych liter `a` i strategicznie położonych liter `b`, sprawiające trudność rozwiązaniom o złożoności $\Theta(n^3)$.

W poniższej tabeli n oznacza długość słowa t, a m — wynik, czyli długość optymalnego prefiksufiksu.

Nazwa	n	m
pre1a.in	100	16
pre1b.in	1	0
pre1c.in	100	50
pre1d.in	100	45
pre1e.in	100	8
pre1f.in	1	0
pre2a.in	301	128
pre2b.in	2	0
pre2c.in	301	148
pre2d.in	301	149
pre2e.in	301	8
pre3a.in	500	160
pre3b.in	3	1
pre3c.in	500	242
pre3d.in	500	243
pre3e.in	500	8
pre4a.in	4 000	1 280
pre4b.in	8	3
pre4c.in	4 000	1 984
pre4d.in	4 000	1 368
pre4e.in	4 000	8
pre5a.in	5 000	2 048
pre5b.in	89	42
pre5c.in	5 000	2 480
pre5d.in	5 000	2 082
pre5e.in	5 000	8

Nazwa	n	m
pre6a.in	100 000	16 384
pre6b.in	144	68
pre6c.in	100 000	49 928
pre6d.in	100 000	10 002
pre6e.in	100 000	8
pre7a.in	200 000	32 768
pre7b.in	233	110
pre7c.in	200 000	99 872
pre7d.in	200 000	76 600
pre7e.in	200 000	8
pre8a.in	500 000	163 840
pre8b.in	1 597	754
pre8c.in	500 000	249 722
pre8d.in	500 000	232 103
pre8e.in	500 000	8
pre9a.in	1 000 000	327 680
pre9b.in	46 368	21 892
pre9c.in	1 000 000	500 000
pre9d.in	1 000 000	441 598
pre9e.in	1 000 000	8
pre10a.in	1 000 000	327 680
pre10b.in	317 811	150 050
pre10c.in	1 000 000	500 000
pre10d.in	1 000 000	458 390
pre10e.in	1 000 000	8
pre10f.in	1 000 000	500 000

XXIV Międzynarodowa Olimpiada Informatyczna,

Sirmione – Montichiari, Włochy 2012

Jakub Łącki, Jakub Pachocki, Jakub Radoszewski
Tłumaczenie

IOI 2012, dzień pierwszy, 25.09.2012

Skryba dwukierunkowy

Niektórzy twierdzą, że Leonardo da Vinci był wielkim miłośnikiem Jana Gutenberga — niemieckiego kowala, który wynalazł prasę drukarską. Zainspirowany wynalazkiem Gutenberga, Leonardo stworzył prostą maszynę drukarską: skrybę dwukierunkowego. Urządzenie to przypomina nieco prostą maszynę do pisania, która obsługuje dwa polecenia: pierwsze polecenie służy do napisania jednej litery, a drugie to polecenie **undo***, służące do cofania ostatnio wykonanych poleceń. Co istotne, polecenie undo działa wyjątkowo sprytnie, bo traktuje wcześniejsze undo również jako polecenia, a co za tym idzie, może cofać ich wykonanie.*

Zadanie

Napisz program, który będzie symulować działanie skryby. Program powinien wykonywać kolejne operacje wprowadzane przez użytkownika i odpowiadać na pytania o litery znajdujące się na wskazanych pozycjach w tekście. Początkowo tekst jest pusty. Oto lista możliwych operacji:

- `Init`() – *funkcja ta zostanie wywołana dokładnie raz, na początku wykonania programu, bez żadnych argumentów. Możesz jej użyć do zainicjalizowania struktur danych w Twoim programie. Możesz założyć, że nigdy nie będzie trzeba cofać jej wykonania.*
- `TypeLetter`(L) – *dopisz na końcu tekstu jedną literę L, przy czym L jest małą literą alfabetu angielskiego.*
- `UndoCommands`(U) – *wycofaj U ostatnich poleceń, przy czym U jest dodatnią liczbą całkowitą.*
- `GetLetter`(P) – *podaj literę znajdującą się na pozycji P w aktualnym tekście, przy czym P jest nieujemną liczbą całkowitą. Pozycje w tekście numerujemy od 0. Zapytanie nie jest poleceniem skryby, więc nie ma znaczenia dla wykonywania polecenia undo.*

Po początkowym wywołaniu `Init`(), *pozostałe funkcje mogą być wywołane zero lub więcej razy, w dowolnej kolejności. Możesz założyć, że U nie będzie większe niż liczba wcześniej otrzymanych poleceń oraz że P będzie mniejsze niż aktualna liczba liter w tekście.*

Polecenie `UndoCommands`(U) *cofa U ostatnich poleceń, począwszy od tego, które zostało wykonane jako ostatnie. Wycofanie polecenia* `TypeLetter`(L) *polega na usunięciu litery L z końca aktualnego tekstu. Z kolei wycofanie polecenia* `UndoCommands`(X) *polega na ponownym wykonaniu X ostatnich poleceń, w ich pierwotnym porządku.*

Przykład. *Poniższa tabela przedstawia przykładowy ciąg wywołań funkcji oraz aktualną postać tekstu po każdym wywołaniu.*

Wywołanie	Zwrócona wartość	Aktualny tekst
`Init()`		
`TypeLetter(a)`		`a`
`TypeLetter(b)`		`ab`
`GetLetter(`*1*`)`	`b`	`ab`
`TypeLetter(d)`		`abd`
`UndoCommands(`*2*`)`		`a`
`UndoCommands(`*1*`)`		`abd`
`GetLetter(`*2*`)`	`d`	`abd`
`TypeLetter(e)`		`abde`
`UndoCommands(`*1*`)`		`abd`
`UndoCommands(`*5*`)`		`ab`
`TypeLetter(c)`		`abc`
`GetLetter(`*2*`)`	`c`	`abc`
`UndoCommands(`*2*`)`		`abd`
`GetLetter(`*2*`)`	`d`	`abd`

Podzadanie 1 [5 punktów]

Łącznie będzie co najwyżej 100 poleceń i zapytań. Polecenie `UndoCommands` *nie będzie wykonywane.*

Podzadanie 2 [7 punktów]

Łącznie będzie co najwyżej 100 poleceń i zapytań. Polecenie `UndoCommands` *nigdy nie będzie wycofywane.*

Podzadanie 3 [22 punkty]

Łącznie będzie co najwyżej 5 000 poleceń i zapytań.

Podzadanie 4 [26 punktów]

Łącznie będzie co najwyżej 1 000 000 poleceń i zapytań. Wszystkie wywołania `GetLetter` *będą miały miejsce po wszystkich wywołaniach* `TypeLetter` *i* `UndoCommands`*.*

Podzadanie 5 [40 punktów]

Łącznie będzie co najwyżej 1 000 000 poleceń i zapytań.

Szczegóły implementacji

Należy zgłosić dokładnie jeden plik o nazwie `scrivener.c`*,* `scrivener.cpp` *lub* `scrivener.pas`*. Powinien on zawierać implementacje funkcji opisanych powyżej.*

Programy w C/C++

```
void Init();
void TypeLetter(char L);
void UndoCommands(int U);
char GetLetter(int P);
```

Programy w Pascalu

```
procedure Init;
procedure TypeLetter(L : Char);
procedure UndoCommands(U : LongInt);
function GetLetter(P : LongInt) : Char;
```

Funkcje powinny działać dokładnie tak, jak opisano powyżej. Twój program nie powinien korzystać ze standardowego wejścia, standardowego wyjścia lub jakichkolwiek plików.

Przykładowy moduł oceniający

Przykładowy moduł oceniający wczytuje dane w następującym formacie:

- *wiersz 1: łączna liczba poleceń i zapytań opisanych w dalszej części wejścia;*
- *w każdym z kolejnych wierszy:*
  - `T` x – *operacja* `TypeLetter`(x)*;*
  - `U` y – *operacja* `UndoCommand`(y)*;*
  - `P` z – *operacja* `GetLetter`(z)*.*

Przykładowy moduł oceniający wypisuje litery zwracane przez `GetLetter`*, każdą w osobnym wierszu.*

Jakub Łącki, Jakub Pachocki, Jakub Radoszewski
Tłumaczenie

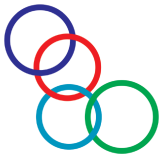
IOI 2012, dzień pierwszy, 25.09.2012

Ogniwa

Około roku 1485 Leonardo da Vinci opisał dosyć skomplikowaną wersję tego, co obecnie nazywamy spadochronem. Spadochron Leonarda był wykonany z materiału rozłożonego na konstrukcji w kształcie piramidy.

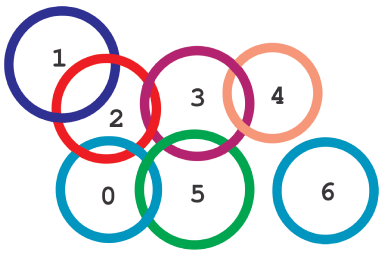
Połączone ogniwa

Ponad 500 lat później spadochroniarz Adrian Nicholas zrealizował projekt Leonarda. Stworzył nowoczesną, lekką konstrukcję, która pozwalała przymocować spadochron Leonarda do ludzkiego ciała. Częścią konstrukcji są połączone ze sobą ogniwa, do których mocuje się materiał. Każde ogniwo to mały pierścień wykonany ze sprężystego, wytrzymałego materiału. Ogniwa można łatwo łączyć, gdyż każde z nich można otworzyć, a następnie ponownie zamknąć. **Łańcuchem** *nazywamy taki układ połączonych ogniw, w którym każde ogniwo jest połączone z co najwyżej dwoma sąsiednimi ogniwami. Łańcuch posiada początek i koniec, czyli ogniwa mające co najwyżej jednego sąsiada. W szczególności, pojedyncze ogniwo również tworzy łańcuch.*



Możliwe są także różne inne konfiguracje ogniw, a każde ogniwo może być połączone z dowolnie dużą liczbą innych ogniw. Ogniwo nazywamy **krytycznym**, *jeśli po jego otworzeniu i usunięciu wszystkie pozostałe ogniwa tworzą zbiór łańcuchów (w szczególności, jeśli jest tylko jedno ogniwo, to jest ono krytyczne).*

Przykład. *Spójrzmy na poniższy obrazek z siedmioma ogniwami ponumerowanymi od 0 do 6. Dwa ogniwa są krytyczne. Jedno z nich to ogniwo 2 — po jego usunięciu pozostałe ogniwa tworzą trzy łańcuchy: $[1]$, $[0, 5, 3, 4]$ oraz $[6]$. Drugie krytyczne ogniwo to 3 — po jego usunięciu pozostałe ogniwa tworzą trzy łańcuchy: $[1, 2, 0, 5]$, $[4]$ i $[6]$. Jeśli usuniemy jakiekolwiek inne ogniwo, nie otrzymamy zbioru rozłącznych łańcuchów. Załóżmy, na przykład, że usuwamy ogniwo 5. W wyniku wprawdzie otrzymamy łańcuch $[6]$, jednak ogniwa 0, 1, 2, 3 i 4 nie będą tworzyły łańcucha.*



Zadanie

Twoim zadaniem jest napisanie programu, który będzie zliczał krytyczne ogniwa w podanej konfiguracji ogniw. Początkowa konfiguracja składa się z pewnej liczby niepołączonych ogniw. Następnie ogniwa są ze sobą łączone. W każdym momencie Twój program może zostać poproszony o wyznaczenie liczby krytycznych ogniw w aktualnej konfiguracji. Powinieneś zaimplementować trzy funkcje:

- `Init`(N) – *funkcja ta zostanie wywołana dokładnie raz, na samym początku działania programu. Parametr N oznacza liczbę niepołączonych ogniw w początkowej konfiguracji. Ogniwa są ponumerowane od 0 do $N-1$.*
- `Link`(A, B) – *ogniwa o numerach A i B zostają ze sobą połączone. Możesz założyć, że $A \neq B$ i ogniwa te nie zostały ze sobą wcześniej (bezpośrednio) połączone. Poza tym, ogniwa można łączyć zupełnie dowolnie, kompletnie ignorując prawa fizyki i zdrowy rozsądek. Oczywiście wywołania* `Link`(A, B) *i* `Link`(B, A) *są równoważne.*
- `CountCritical`$()$ – *zwraca liczbę krytycznych ogniw w aktualnej konfiguracji.*

Przykład. *Rozważmy przykład, w którym występuje $N = 7$ ogniw. Początkowo są one niepołączone. Pokażemy możliwy ciąg wywołań, po którym otrzymamy konfigurację z rysunku.*

Wywołanie	Zwrócona wartość
`Init`(7)	
`CountCritical`$()$	7
`Link`$(1, 2)$	
`CountCritical`$()$	7
`Link`$(0, 5)$	
`CountCritical`$()$	7
`Link`$(2, 0)$	
`CountCritical`$()$	7
`Link`$(3, 2)$	
`CountCritical`$()$	4
`Link`$(3, 5)$	
`CountCritical`$()$	3
`Link`$(4, 3)$	
`CountCritical`$()$	2

Podzadanie 1 [20 punktów]

$N \leqslant 5\,000$
Funkcja `CountCritical` *zostanie wywołana tylko raz, po wszystkich innych wywołaniach; funkcja* `Link` *zostanie wywołana co najwyżej 5 000 razy.*

Podzadanie 2 [17 punktów]

$N \leqslant 1\,000\,000$
Funkcja `CountCritical` *zostanie wywołana tylko raz, po wszystkich innych wywołaniach; funkcja* `Link` *zostanie wywołana co najwyżej 1 000 000 razy.*

Podzadanie 3 [18 punktów]

$N \leqslant 20\,000$
Funkcja `CountCritical` *zostanie wywołana co najwyżej 100 razy; funkcja* `Link` *zostanie wywołana co najwyżej 10 000 razy.*

Podzadanie 4 [14 punktów]

$N \leqslant 100\,000$
Funkcje `CountCritical` *i* `Link` *zostaną wywołane (łącznie) co najwyżej 100 000 razy.*

Podzadanie 5 [31 punktów]

$N \leqslant 1\,000\,000$
Funkcje `CountCritical` *i* `Link` *zostaną wywołane (łącznie) co najwyżej 1 000 000 razy.*

Szczegóły implementacji

Należy zgłosić dokładnie jeden plik o nazwie `rings.c`, `rings.cpp` *lub* `rings.pas`. *Powinien on zawierać implementacje opisanych powyżej funkcji.*

Programy w C/C++

```
void Init(int N);
void Link(int A, int B);
int CountCritical();
```

Programy w Pascalu

```
procedure Init(N : LongInt);
procedure Link(A, B : LongInt);
function CountCritical() : LongInt;
```

Funkcje powinny działać dokładnie tak, jak opisano powyżej. Twój program nie powinien korzystać ze standardowego wejścia, standardowego wyjścia lub jakichkolwiek plików.

Przykładowy moduł oceniający

Przykładowy moduł oceniający wczytuje dane w następującym formacie:

- *wiersz 1:* N, L;
- *wiersze* $2, \ldots, L + 1$:
  - -1 *oznacza wywołanie* `CountCritical`;
  - *dwie liczby* A, B *oznaczają wywołanie* `Link` *z parametrami* A *i* B.

Przykładowy moduł oceniający wypisuje wszystkie wyniki zwrócone przez `CountCritical`.

Jakub Łącki, Jakub Pachocki, Jakub Radoszewski
Tłumaczenie

IOI 2012, dzień pierwszy, 25.09.2012

Odometr na kamyki

Jednym z wynalazków Leonarda da Vinci jest odometr (drogomierz). Stanowi on rodzaj pojazdu służący do mierzenia odległości za pomocą kamyków, które wypadają w miarę kręcenia się kółek. Liczba upuszczonych kamyków jest równa liczbie obrotów kółek, co pozwala wyznaczyć odległość przebytą przez odometr. Jako informatycy będący fanami Leonarda, postanowiliśmy dodać do jego odometru oprogramowanie sterujące oraz nieco poszerzyć jego możliwości. Twoim zadaniem jest oprogramowanie odometru zgodnie z poniższą specyfikacją.

Opis planszy

Nasz odometr będzie poruszał się po planszy złożonej z 256×256 pól. Na każdym z pól może znajdować się co najwyżej 15 kamyków. Położenie pola określamy za pomocą pary współrzędnych (wiersz, kolumna), przy czym każda współrzędna jest z zakresu $0, \ldots, 255$. Pola sąsiadujące z polem (i, j) to (jeśli istnieją): $(i-1, j)$, $(i+1, j)$, $(i, j-1)$ oraz $(i, j+1)$. Pola znajdujące się w pierwszym i ostatnim wierszu planszy oraz w pierwszej i ostatniej kolumnie planszy nazywamy polami **brzegowymi**. *Na początku odometr znajduje się na polu $(0, 0)$ (czyli w północno-zachodnim narożniku planszy) i jest zwrócony na północ.*

Podstawowe komendy

Odometr wykonuje **program** *składający się z następujących komend:*

- `left` – *skręć o 90 stopni w lewo (tj. przeciwnie do kierunku ruchu wskazówek zegara) i pozostań na obecnie zajmowanym polu. Przykładowo, jeśli przed wykonaniem tej komendy odometr był zwrócony na południe, to po jej wykonaniu będzie zwrócony na wschód.*
- `right` – *skręć o 90 stopni w prawo (tj. zgodnie z kierunkiem ruchu wskazówek zegara) i pozostań na obecnie zajmowanym polu. Przykładowo, jeśli przed wykonaniem tej komendy odometr był zwrócony na zachód, to po jej wykonaniu będzie zwrócony na północ.*
- `move` – *wykonaj ruch o jedno pole naprzód (czyli w kierunku, w którym jest obecnie zwrócony odometr). Jeśli docelowe pole nie istnieje (tzn. odometr dotarł już do brzegu planszy w rozważanym kierunku), to ta komenda nie ma żadnego efektu.*
- `get` – *zabierz jeden kamyk z obecnie zajmowanego pola. Jeśli na polu nie ma żadnego kamyka, ta komenda nie ma żadnego efektu.*
- `put` – *upuść jeden kamyk na obecnie zajmowane pole. Jeśli pole zawiera już 15 kamyków, ta komenda nie ma żadnego efektu. Odometr ma zawsze w zapasie odpowiednią liczbę kamyków.*
- `halt` – *zakończ działanie odometru.*

182 *Odometr na kamyki*

Odometr wykonuje komendy w kolejności, w której występują one w programie. Każdy wiersz programu może zawierać co najwyżej jedną komendę. Puste wiersze programu są ignorowane. Symbol `#` *oznacza komentarz; wszystko, co znajduje w wierszu za znakiem* `#`*, jest ignorowane. Jeśli odometr dojdzie do końca programu, kończy swoje działanie.*

Przykład 1. *Rozważmy poniższy, bardzo prosty program. Na końcu działania programu odometr znajduje się na polu* $(0, 2)$ *i jest zwrócony na wschód. Zauważ, że w poniższym programie pierwsza komenda* `move` *zostanie zignorowana, ponieważ na początku odometr znajduje się w północno-zachodnim narożniku planszy i jest zwrócony na północ.*

```
move  # nie ma żadnego efektu
right
# teraz odometr jest zwrócony na wschód
move
move
```

Etykiety

Aby modyfikować działanie programu zależnie od chwilowego stanu wykonania, będziemy używać etykiet. Etykiety to napisy złożone z co najwyżej 128 symboli `a`, ..., `z`, `A`, ..., `Z`, `0`, ..., `9` *(w etykietach rozróżniamy wielkie i małe litery). Poniżej znajduje się lista komend związanych z etykietami. W poniższym opisie* L *oznacza jakąkolwiek poprawną etykietę.*

- L`:` *(tj.* L *plus dwukropek* `':'`*) – deklaruje w danym miejscu programu etykietę* L*. Wszystkie zadeklarowane etykiety muszą być różne. Deklaracja etykiety nie zmienia w żaden sposób ruchu odometru.*
- `jump` L *– kontynuuj działanie programu od miejsca, w którym zadeklarowano etykietę* L*.*
- `border` L *– przeskocz do miejsca w programie, w którym zadeklarowano etykietę* L*, pod warunkiem, że w chwili wykonywania tej komendy odometr znajduje się na brzegu i jest zwrócony ku krawędzi planszy (tzn. w tym położeniu komenda* `move` *nie miałaby żadnego efektu); w przeciwnym razie komenda* `border` L *nie ma żadnego efektu, a program kontynuuje działanie od następnej komendy.*
- `pebble` L *– przeskocz do miejsca w programie, w którym zadeklarowano etykietę* L*, pod warunkiem, że w chwili wykonywania tej komendy na polu zajmowanym przez odometr znajduje się co najmniej jeden kamyk; w przeciwnym razie komenda nie ma żadnego efektu, a program kontynuuje działanie od następnej komendy.*

Przykład 2. *Poniższy program znajduje pierwszy (skrajnie lewy) kamyk w wierszu 0 i w tym miejscu zatrzymuje odometr. Jeśli wiersz 0 nie zawiera żadnych kamyków, odometr zatrzymuje się na końcu tego wiersza. W programie używamy dwóch etykiet,* `leonardo` *oraz* `davinci`*.*

```
right
leonardo:
pebble davinci  # znaleziono kamyk
border davinci  # koniec wiersza
```

```
move
jump leonardo
davinci:
halt
```

Na początku odometr wykonuje obrót w prawo. W programie znajduje się pętla, której początek stanowi deklaracja etykiety `leonardo`*, a koniec stanowi komenda* `jump leonardo`*. Wewnątrz pętli odometr sprawdza, czy znajduje się na polu zawierającym kamyk albo na końcu wiersza, a jeśli nie, wykonuje komendę* `move`*, wskutek której przemieszcza się z bieżącego pola* $(0, j)$ *na sąsiednie pole* $(0, j + 1)$ *(wiadomo teraz, że to pole istnieje). Zauważ, że komenda* `halt` *nie jest tu konieczna, gdyż bez niej program i tak by się zakończył.*

Zadanie

Napisz program sterujący odometrem (zgodnie z językiem programowania wyspecyfikowanym powyżej), który spowoduje, że odometr zrealizuje cele wyznaczone mu w poszczególnych podzadaniach. Każde podzadanie zawiera pewne ograniczenia, które Twój program musi spełniać. Ograniczenia te dotyczą dwóch następujących pojęć.

- **Długość programu** – *jest to liczba komend zawartych w programie. Deklaracje etykiet, komentarze i puste wiersze w programie nie są brane pod uwagę przy obliczaniu długości programu.*
- **Czas wykonania** – *jest to liczba kroków wykonanych przez program. Krokiem nazywamy dowolne wykonanie komendy, niezależnie od tego, czy ma ono jakikolwiek efekt. Deklaracje etykiet, komentarze i puste wiersze w programie nie są liczone jako kroki.*

W przykładzie 1 długość programu to 4 i czas wykonania to 4. W przykładzie 2 długość programu to 6, a czas wykonania dla planszy zawierającej tylko jeden kamyk, na polu $(0, 10)$*, to 43. Poszczególne kroki to: komenda* `right`*, następnie 10 obrotów pętli, z których każdy składa się z czterech kroków (*`pebble davinci`*;* `border davinci`*;* `move`*;* `jump leonardo`*), a na końcu komendy* `pebble davinci` *oraz* `halt`*.*

Podzadanie 1. [9 punktów]

Na początku na polu $(0, 0)$ *znajduje się* x *kamyków, a na polu* $(0, 1)$ *znajduje się* y *kamyków. Wszystkie pozostałe pola są puste. Napisz program, po wykonaniu którego odometr zakończy działanie na polu* $(0, 0)$*, jeśli* $x \leqslant y$*, a w przeciwnym przypadku na polu* $(0, 1)$*. (Nie interesuje nas kierunek, w którym zwrócony jest odometr na końcu programu; nie interesuje nas również to, ile kamyków znajduje się na końcu na planszy ani jak są one rozmieszczone). Pamiętaj, że na każdym polu planszy może znajdować się co najwyżej 15 kamyków.*

Ograniczenia: *długość programu* $\leqslant 100$*, czas wykonania* $\leqslant 1\,000$*.*

Podzadanie 2. [12 punktów]

Twoje zadanie jest takie samo jak poprzednio, tyle że na końcu programu na polu $(0, 0)$ *musi znajdować się* x *kamyków, a na polu* $(0, 1)$ *musi znajdować się* y *kamyków.*

Ograniczenia: *długość programu* $\leqslant 200$*, czas wykonania* $\leqslant 2\,000$*.*

Podzadanie 3. [19 punktów]

Gdzieś w wierszu 0 znajdują się dwa kamyki: jeden na polu $(0, x)$, a drugi na polu $(0, y)$. Liczby x i y są różne, a ich suma jest parzysta. Napisz program, który umieszcza odometr na polu o współrzędnych $(0, (x + y)/2)$, tj. na polu znajdującym się dokładnie w połowie drogi między rozważanymi polami zawierającymi kamyki. Końcowa zawartość pól planszy jest tu nieistotna.

Ograniczenia: *długość programu $\leqslant$ 100, czas wykonania $\leqslant$ 200 000.*

Podzadanie 4. [co najwyżej 32 punkty]

Na planszy znajduje się co najwyżej 15 kamyków, przy czym każde pole zawiera co najwyżej jeden kamyk. Napisz program, który zbierze wszystkie kamyki w północno-zachodnim narożniku planszy. Dokładniej, jeśli na początku plansza zawierała x kamyków, to na końcu na polu $(0, 0)$ musi znajdować się dokładnie x kamyków, a wszystkie pozostałe pola muszą być puste.

Liczba punktów, jaką zdobędziesz za to podzadanie, będzie zależeć od czasu wykonania Twojego programu. Niech L oznacza maksymalny czas wykonania ze wszystkich przypadków testowych w tym podzadaniu. Otrzymasz:

- *32 punkty, jeśli $L \leqslant 200\,000$;*
- *$32 - 32 \cdot \log_{10}(L/200\,000)$ punktów, jeśli $200\,000 < L < 2\,000\,000$;*
- *0 punktów, jeśli $L \geqslant 2\,000\,000$.*

Ograniczenia: *długość programu $\leqslant$ 200.*

Podzadanie 5. [co najwyżej 28 punktów]

Każde pole planszy może zawierać dowolną liczbę kamyków (rzecz jasna, między 0 a 15). Napisz program, który wyznaczy minimum, tj. zakończy działanie odometru na polu (i, j), takim że każde z pozostałych pól zawiera co najmniej tyle kamyków, co pole (i, j). Po zakończeniu działania programu każde pole musi zawierać dokładnie tyle samo kamyków, ile zawierało przed wykonaniem programu.

Liczba punktów, jaką zdobędziesz za to podzadanie, będzie zależeć od długości P Twojego programu. Otrzymasz:

- *28 punktów, jeśli $P \leqslant 444$;*
- *$28 - 28 \cdot \log_{10}(P/444)$ punktów, jeśli $444 < P < 4\,440$;*
- *0 punktów, jeśli $P \geqslant 4\,440$.*

Ograniczenia: *czas wykonania $\leqslant$ 44 400 000.*

Szczegóły implementacji

Jako rozwiązanie każdego podzadania powinieneś wysłać dokładnie jeden plik, zawierający program zapisany zgodnie ze składnią języka sterowania odometru. Rozmiary poszczególnych plików nie mogą przekraczać 5 MiB. Dla każdego podzadania, Twój program zostanie sprawdzony za pomocą kilku przypadków testowych i otrzymasz pewną informację o tym, ile zasobów zużył. Jeśli składnia Twojego programu będzie niepoprawna, otrzymasz informację o tym, jaki błąd składni w nim występuje.

Symulator

Masz do dyspozycji symulator odometru, który potrafi uruchomić program sterujący odometrem na konkretnych planszach. Symulator wykonuje programy o składni takiej, jak opisana powyżej.

Plik z opisem planszy powinien mieć następujący format: każdy wiersz zawiera trzy liczby całkowite R, C oraz P, oznaczające, że pole (R, C) zawiera P kamyków. Pola niewystępujące w opisie planszy nie zawierają żadnych kamyków. Dla przykładu, rozważmy plik o następującej zawartości:

```
0 10 3
4 5 12
```

Plansza opisana przez ten plik zawiera 15 kamyków: 3 kamyki na polu $(0, 10)$ i 12 kamyków na polu $(4, 5)$.

Symulator uruchamia się przez wywołanie programu `simulator.py` *znajdującego się w katalogu tego zadania. Jako argument należy podać nazwę pliku z programem sterującym odometrem. Symulator przyjmuje następujące parametry wiersza poleceń:*

- `-h` – *podaje krótki opis dostępnych parametrów wiersza poleceń;*
- `-g GRID_FILE` – *ładuje opis planszy z pliku* `GRID_FILE` *(domyślnie plansza jest pusta);*
- `-s GRID_SIDE` – *ustawia rozmiar planszy na* `GRID_SIDE x GRID_SIDE` *(domyślna wartość to 256, taka sama jak występująca wszędzie w treści zadania); używanie mniejszych plansz może pomóc Ci w usuwaniu usterek z programu;*
- `-m STEPS` – *ustawia górny limit na czas wykonania programu na* `STEPS` *kroków;*
- `-c` – *uruchamia tryb kompilacji. W trybie kompilacji symulator generuje i kompiluje program w języku C, który będzie przeprowadzał symulację. Wprawdzie przygotowanie programu zajmuje pewien czas, jednak gotowy program wykonuje symulację istotnie szybciej niż* `simulator.py`*. Polecamy Ci używać tej opcji, gdy Twój program wykonuje 10 000 000 lub więcej kroków.*

Jakub Łącki, Jakub Pachocki, Jakub Radoszewski
Tłumaczenie

IOI 2012, dzień drugi, 27.09.2012

Turniej rycerski

Przed swym ślubem z Beatrice d'Este w 1491 roku, Ludovico Sforza poprosił Leonarda da Vinci o przygotowanie atrakcji weselnych. Jedną z nich miał być wielki turniej rycerski, trwający całe trzy dni. Niestety, najbardziej popularny rycerz się spóźnia...

Turniej

W turnieju bierze udział N rycerzy. Są oni ustawieni w szereg; ich pozycje numerujemy od 0 do $N-1$, zgodnie z porządkiem w szeregu. Turniej składa się z pewnej liczby **rund**. *Mistrz turnieju rozpoczyna rundę, ogłaszając dwie pozycje, S i E (przy czym $0 \leqslant S < E \leqslant N-1$). W tym momencie wszyscy rycerze znajdujący się w szeregu na pozycjach od S do E (włącznie) walczą na kopie; zwycięzca pozostaje w turnieju i wraca na swoje miejsce w szeregu, podczas gdy wszyscy pozostali odpadają z turnieju i opuszczają pole gry. Następnie rycerze pozostający w turnieju przesuwają się ku początkowi szeregu (nie zmieniając względnego porządku), tj. przechodzą na pozycje od 0 do $N-(E-S)-1$. Potem mistrz turnieju rozpoczyna kolejną rundę i powtarza tę procedurę, aż w szeregu pozostanie tylko jeden rycerz.*

Leonardo wie, że rycerze różnią się umiejętnościami, i potrafi przydzielić im parami różne rangi z zakresu od 0 (najsłabszy) do $N-1$ (najsilniejszy). Zna on również dokładne polecenia, jakie wyda mistrz turnieju w każdej z C rund — w końcu to Leonardo... Jest on przy tym pewny, że w każdej rundzie wygra rycerz o największej randze.

Spóźniony rycerz

$N-1$ spośród N rycerzy stoi już w szeregu, brakuje tylko najpopularniejszego zawodnika. Rycerz ten ma rangę R i przybędzie odrobinę spóźniony. Aby ucieszyć publikę, Leonardo chciałby wykorzystać jego popularność i ustawić go na pozycji w szeregu, która zmaksymalizuje liczbę wygranych przez niego rund. Zauważ, że nie liczymy rund, w których spóźniony rycerz nie walczy; interesują nas tylko rundy, w których bierze on udział i wygrywa.

Przykład. *Rozważmy sytuację, w której $N=5$ i $N-1$ rycerzy ustawionych w szeregu ma kolejno rangi $[1,0,2,4]$. Spóźniony rycerz ma zatem rangę $R=3$. Przez $C=3$ rundy mistrz turnieju zamierza wywołać kolejno następujące pary pozycji (S,E): $(1,3)$, $(0,1)$, $(0,1)$.*

Jeżeli Leonardo umieści spóźnionego rycerza na pierwszej pozycji, rangi rycerzy w szeregu będą wynosiły kolejno $[3,1,0,2,4]$. W pierwszej rundzie walczą rycerze z pozycji $1,2,3$, o rangach $1,0,2$. W turnieju pozostaje więc rycerz o randze 2, a nowy szereg to $[3,2,4]$. W kolejnej rundzie walczą rycerze o rangach 3 i 2 (z pozycji 0 i 1) i zwycięża rycerz o randze $R=3$. Przed trzecią rundą szereg to $[3,4]$. W ostatniej walce (pozycje 0 i 1) wygrywa rycerz o randze 4. Spóźniony rycerz wygrał zatem tylko jedną rundę (drugą).

Jeżeli zamiast tego Leonardo umieści spóźnionego rycerza pomiędzy tymi o rangach 1 i 0, szereg będzie wyglądał następująco: $[1,3,0,2,4]$. Tym razem w pierwszej rundzie biorą udział rycerze o rangach $3,0,2$ i rycerz o randze $R=3$ wygrywa. Na początku kolejnej rundy szereg to $[1,3,4]$ i w walce (1 przeciwko 3) ponownie wygrywa rycerz o randze 3. Ostatni szereg to

[3, 4] i trzecią rundę wygrywa rycerz o randze 4. Spóźniony rycerz wygrywa więc dwie rundy. Okazuje się, że jest to najlepsze możliwe rozwiązanie, gdyż w tej sytuacji nie da się umieścić spóźnionego rycerza tak, aby wygrał więcej niż dwa razy.

Zadanie

Napisz program, który wybierze najlepszą pozycję dla spóźnionego rycerza, tak aby zmaksymalizować liczbę wygranych przez niego rund.

Zaimplementuj funkcję `GetBestPosition`(N, C, R, K, S, E)*, przy czym:*

- N *to liczba rycerzy;*
- C *to liczba rund zaplanowanych przez mistrza turnieju* ($1 \leqslant C \leqslant N - 1$)*;*
- R *to ranga spóźnionego rycerza; rangi wszystkich rycerzy (tych stojących w szeregu oraz tego spóźnionego) są parami różne i należą do zbioru* $\{0, \ldots, N - 1\}$*. Ranga* R *spóźnionego rycerza jest dana wprost, choć można ją wywnioskować na podstawie pozostałych danych;*
- K *to tablica* $N - 1$ *liczb całkowitych reprezentujących rangi* $N - 1$ *rycerzy, którzy stoją już w szeregu (w kolejności, w jakiej rycerze stoją w szeregu);*
- S *i* E *to dwie tablice rozmiaru* C*; dla każdego* i *od 0 do* $C - 1$ *włącznie, w* $(i + 1)$*-szej rundzie rozpoczętej przez mistrza turnieju wezmą udział wszyscy rycerze z pozycji od* $S[i]$ *do* $E[i]$ *włącznie. Możesz założyć, że dla każdego* i *zachodzi* $S[i] < E[i]$*.*

Możesz założyć, że $E[i]$ *będzie zawsze mniejsze niż liczba rycerzy pozostałych do rundy* $(i + 1)$*-szej, a po wszystkich* C *rundach pozostanie dokładnie jeden rycerz.*

`GetBestPosition`(N, C, R, K, S, E) *powinno zwrócić najlepszą pozycję* P*, na której Leonardo powinien umieścić spóźnionego rycerza* ($0 \leqslant P \leqslant N - 1$)*. Jeżeli istnieje wiele równie dobrych pozycji,* **zwróć najmniejszą z nich***. (Pozycja* P *to pozycja spóźnionego rycerza w szeregu na początku turnieju, indeksowana od 0. Innymi słowy,* P *jest równe liczbie rycerzy stojących przed spóźnionym rycerzem w optymalnym rozwiązaniu. W szczególności,* $P = 0$ *oznacza, że spóźniony rycerz stoi na początku szeregu, a* $P = N - 1$ *oznacza, że stoi na jego końcu.)*

Podzadanie 1 [17 punktów]

$N \leqslant 500$

Podzadanie 2 [32 punkty]

$N \leqslant 5\,000$

Podzadanie 3 [51 punktów]

$N \leqslant 100\,000$

Szczegóły implementacji

Należy zgłosić dokładnie jeden plik o nazwie `tournament.c`, `tournament.cpp` *lub* `tournament.pas`. *Powinien on zawierać implementację opisanej powyżej funkcji.*

Programy w C/C++

```
int GetBestPosition(int N, int C, int R, int *K, int *S, int *E);
```

Programy w Pascalu

```
function GetBestPosition(N, C, R : LongInt;
    var K, S, E : array of LongInt) : LongInt;
```

Funkcja powinna działać dokładnie tak, jak opisano powyżej. Twój program nie powinien korzystać ze standardowego wejścia, standardowego wyjścia lub jakichkolwiek plików.

Przykładowy moduł oceniający

Przykładowy moduł oceniający wczytuje dane w następującym formacie:

- *wiersz 1:* N, C, R;
- *wiersze 2,* $\ldots$, N: $K[i]$;
- *wiersze* $N+1$, $\ldots$, $N+C$: $S[i]$, $E[i]$.

Jakub Łącki, Jakub Pachocki, Jakub Radoszewski
Tłumaczenie

IOI 2012, dzień drugi, 27.09.2012

Idealne miasto

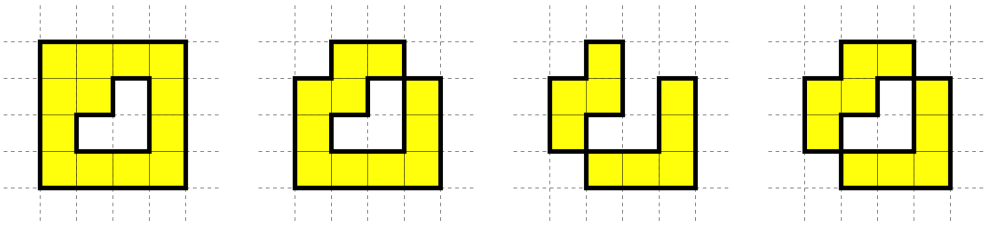
Podobnie jak wielu współczesnych mu naukowców i artystów, Leonardo da Vinci interesował się zagadnieniami urbanistyki i planowania przestrzennego. Postawił sobie za cel zaprojektowanie miasta idealnego: wygodnego i przestronnego, w efektywny sposób gospodarującego zasobami naturalnymi. Ta koncepcja zdecydowanie przewyższała średniowieczne standardy wąskich, wręcz klaustrofobicznych miast.

Idealne miasto

W tym zadaniu miasto będziemy wyobrażać sobie jako N kwartałów zabudowy, dla prostoty nazywanych blokami, rozmieszczonych na nieskończonej kratce. Położenie każdego pola kratki określamy za pomocą pary współrzędnych (wiersz, kolumna). Pola sąsiadujące z polem (i, j) to: $(i-1, j)$, $(i+1, j)$, $(i, j-1)$ oraz $(i, j+1)$. Każdy blok zajmuje dokładnie jedno pole kratki. Każde pole (i, j) zawierające blok miasta spełnia warunek $1 \leqslant i, j \leqslant 2^{31} - 2$. Jako współrzędne bloku przyjmujemy współrzędne pola kratki, które ten blok zajmuje. Dwa bloki nazywamy sąsiednimi, jeśli zajmują one sąsiednie pola. Idealne miasto składa się z pewnej liczby połączonych bloków, przy czym jego brzeg nie zawiera „dziur". Dokładniej, muszą być spełnione dwa następujące warunki:

- *Pomiędzy każdymi dwoma* **pustymi** *polami można przejść, poruszając się jedynie po parami sąsiednich* **pustych** *polach.*
- *Pomiędzy każdymi dwoma* **niepustymi** *polami można przejść, poruszając się jedynie po parami sąsiednich* **niepustych** *polach.*

Przykład 1. *Żaden z poniższych układów bloków nie przedstawia idealnego miasta. Dwa pierwsze z lewej nie spełniają pierwszego z powyższych warunków, trzeci nie spełnia drugiego z powyższych warunków, natomiast czwarty nie spełnia żadnego z powyższych warunków.*

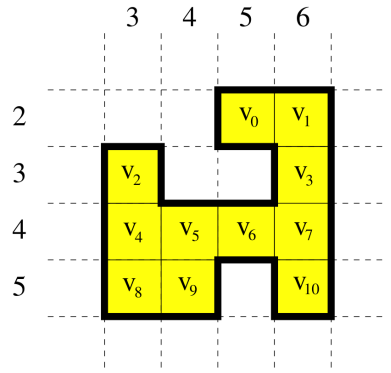


Odległość pól

Po mieście można przemieszczać się za pomocą **ruchów** *polegających na przejściu z pewnego bloku do bloku, który z nim sąsiaduje. Ruchy nie mogą prowadzić przez puste pola kratki. Niech $v_0, v_1, \ldots, v_{N-1}$ oznaczają wszystkie bloki miasta. Odległością między blokami v_i oraz*

v_j, oznaczaną przez $d(v_i, v_j)$, nazywamy najmniejszą liczbę ruchów potrzebnych do przemieszczenia się z jednego z tych bloków do drugiego.

Przykład 2. *Poniższy układ przedstawia idealne miasto złożone z $N = 11$ bloków: $v_0 = (2,5)$, $v_1 = (2,6)$, $v_2 = (3,3)$, $v_3 = (3,6)$, $v_4 = (4,3)$, $v_5 = (4,4)$, $v_6 = (4,5)$, $v_7 = (4,6)$, $v_8 = (5,3)$, $v_9 = (5,4)$ i $v_{10} = (5,6)$. W tym przykładzie $d(v_1, v_3) = 1$, $d(v_1, v_8) = 6$, $d(v_6, v_{10}) = 2$, a $d(v_9, v_{10}) = 4$.*



Zadanie

Napisz program, który mając dany plan rozmieszczenia bloków w pewnym idealnym mieście, obliczy sumę odległości między wszystkimi parami bloków v_i, v_j dla $i < j$, tzn. sumę

$$\sum d(v_i, v_j), \quad \text{gdzie } 0 \leqslant i < j \leqslant N-1.$$

Zaimplementuj funkcję `DistanceSum`*(N, X, Y), która przyjmuje opis miasta w postaci liczby N oraz tablic X i Y i oblicza wartość powyższej sumy. Tablice X, Y mają rozmiar N i określają współrzędne bloków; i-ty blok znajduje się na polu o współrzędnych $(X[i], Y[i])$, dla $0 \leqslant i \leqslant N-1$, przy czym $1 \leqslant X[i], Y[i] \leqslant 2^{31} - 2$. Ponieważ wynik funkcji mógłby przekroczyć rozmiar 32-bitowych typów całkowitych, należy obliczyć resztę z dzielenia wyniku przez 1 000 000 000 (miliard).*

W przykładzie 2 mamy $11 \cdot 10/2 = 55$ par bloków. Suma odległości wszystkich par bloków to 174.

Podzadanie 1 [11 punktów]

$N \leqslant 200$

Podzadanie 2 [21 punktów]

$N \leqslant 2\,000$

Podzadanie 3 [23 punktów]

$N \leqslant 100\,000$

W tym podzadaniu zachodzą dwa dodatkowe warunki: dla dowolnych dwóch niepustych pól i oraz j, takich że $X[i] = X[j]$, każde pole znajdujące się pomiędzy nimi jest także niepuste; dla dowolnych dwóch niepustych pól i oraz j, takich że $Y[i] = Y[j]$, każde pole znajdujące się pomiędzy nimi jest także niepuste.

Podzadanie 4 [45 punktów]

$N \leqslant 100\,000$

Szczegóły implementacji

Należy zgłosić dokładnie jeden plik o nazwie `city.c`, `city.cpp` *lub* `city.pas`. *Powinien on zawierać implementację opisanej powyżej funkcji.*

Programy w C/C++

```
int DistanceSum(int N, int *X, int *Y);
```

Programy w Pascalu

```
function DistanceSum(N : LongInt; var X, Y : array of LongInt) : LongInt;
```

Funkcja powinna działać dokładnie tak, jak opisano powyżej. Twój program nie powinien korzystać ze standardowego wejścia, standardowego wyjścia lub jakichkolwiek plików.

Przykładowy moduł oceniający

Przykładowy moduł oceniający wczytuje dane w następującym formacie:

- *wiersz 1:* N*;*
- *wiersze* $2, \ldots, N + 1$*:* $X[i], Y[i]$*.*

Jakub Łącki, Jakub Pachocki, Jakub Radoszewski
Tłumaczenie

IOI 2012, dzień drugi, 27.09.2012

Ostatnia Wieczerza

Gdy Leonardo da Vinci malował swój najsłynniejszy fresk, Ostatnią Wieczerzę, pracował całymi dniami. Każdego dnia, wczesnym rankiem zastanawiał się, których kolorów farb będzie w ciągu tego dnia potrzebować. Zazwyczaj używał on wielu barw, jednak pod ręką (tj. na rusztowaniu) mógł mieć jedynie ograniczoną liczbę pojemników z farbą. Wnoszeniem pojemników z farbą na rusztowanie i znoszeniem ich na półkę stojącą na podłodze zajmował się asystent Leonarda.

Twoim zadaniem będzie napisanie dwóch osobnych programów, które pomogą asystentowi. Pierwszy program otrzyma opis zamiarów Leonarda (ciąg kolorów farb, których będzie kolejno potrzebować w ciągu dnia) i obliczy **krótki** *ciąg bitów, który nazywać będziemy* **podpowiedzią**. *Podczas obsługiwania próśb Leonarda o pojemniki z farbą, asystent nie będzie znał jego przyszłych żądań, a jedynie podpowiedź obliczoną przez Twój program. Drugi program otrzyma podpowiedź, a następnie otrzymywać będzie żądania Leonarda* **online** *(tj. po jednym na raz). Program będzie musiał zrozumieć podpowiedź i użyć jej do optymalnego przenoszenia pojemników z farbą. Jeśli chcesz poznać szczegóły, czytaj dalej.*

Przenoszenie pojemników między półką a rusztowaniem

W tym zadaniu przyjmiemy, że Leonardo dysponuje N pojemnikami z farbą ponumerowanymi od 0 do $N-1$ (każdy pojemnik zawiera farbę innego koloru) i każdego dnia dokładnie N razy prosi asystenta o pewien pojemnik. Niech C oznacza ciąg opisujący żądania Leonarda. Ciąg ten ma długość N, a jego wyrazy to liczby z zakresu od 0 do $N-1$. Może się zdarzyć, że pewne liczby będą występować w ciągu C wiele razy; może się również zdarzyć, że pewne liczby w ogóle w nim nie wystąpią.

Rusztowanie jest przez cały czas w pełni zastawione pojemnikami z farbą: w każdej chwili znajduje się na nim K (spośród N) pojemników, przy czym $K < N$. Początkowo rusztowanie zawiera pojemniki o numerach od 0 do $K-1$.

Asystent obsługuje żądania Leonarda po kolei. Jeśli Leonardo prosi o pojemnik z farbą, który aktualnie znajduje się na rusztowaniu, asystent nie musi nic robić. W przeciwnym razie powinien on wziąć pojemnik z farbą żądanego koloru z półki i przenieść go na rusztowanie. Oczywiście na rusztowaniu nie ma miejsca na nowy pojemnik, więc asystent musi wybrać jeden z pojemników z rusztowania i znieść go na półkę.

Optymalna strategia przenoszenia pojemników

Asystent chciałby się jak najmniej napracować. Jest on świadom, że liczba żądań, które spowodują, że będzie on musiał nosić pojemniki, zależy od jego wyborów. Dzieje się tak, gdyż za każdym razem, gdy asystent musi zdecydować, który pojemnik powinien znieść z rusztowania na półkę, jego decyzja wpływa na przebieg przyszłych wydarzeń. Leonardo wie, jak powinien postępować asystent (jeśli zna ciąg C), by mieć jak najmniej pracy. Otóż przy podejmowaniu decyzji asystent powinien zwrócić uwagę na pojemniki aktualnie stojące na rusztowaniu oraz

żądania, które w przyszłości wystosuje Leonardo, a następnie wybrać pojemnik w następujący sposób:

- *Jeśli na rusztowaniu znajduje się pojemnik, który nie będzie potrzebny w przyszłości, asystent powinien znieść go na półkę.*
- *W przeciwnym razie asystent powinien znieść pojemnik, o który Leonardo poprosi najpóźniej. Innymi słowy, dla każdego pojemnika na półce znajdujemy jego następne wystąpienie w ciągu żądań Leonarda. Na półkę należy odnieść pojemnik, którego następne wystąpienie jest najpóźniejsze.*

Można wykazać, że postępując zgodnie ze strategią Leonarda, asystent wykona minimalną możliwą liczbę przeniesień.

Przykład 1. *Załóżmy, że $N = 4$. Mamy wówczas cztery kolory (ponumerowane od 0 do 3) oraz cztery żądania. Przyjmijmy, że ciąg żądań to $C = (2, 0, 3, 0)$. Ponadto załóżmy, że $K = 2$. Oznacza to, że na rusztowaniu mieszczą się dwa pojemniki z farbą. Początkowo rusztowanie zawiera pojemniki o numerach 0 i 1 (mówimy, że zawartość rusztowania to $[0, 1]$). Asystent może wówczas obsłużyć żądania Leonarda na przykład tak:*

- *Pierwszego pojemnika, o który prosi Leonardo (numer 2), nie ma na rusztowaniu. Asystent wnosi więc pojemnik na rusztowanie i postanawia znieść pojemnik 1 na półkę. Zawartość rusztowania to $[0, 2]$.*
- *Następny pojemnik (numer 0) znajduje się na rusztowaniu, więc asystent nic nie musi robić.*
- *Przy trzecim żądaniu (pojemnik 3) asystent zdejmuje pojemnik 0 na półkę; nowa zawartość rusztowania to $[3, 2]$.*
- *Ostatni pojemnik, o który prosi Leonardo (numer 0), musi zostać przyniesiony z półki. Asystent postanawia zdjąć pojemnik 2 i w efekcie zawartość rusztowania to $[3, 0]$.*

Zwróć uwagę, że w powyższym przykładzie asystent nie zastosował optymalnej strategii podanej przez Leonarda. Przy optymalnej strategii asystent powinien bowiem w trzecim kroku usunąć pojemnik numer 2 z rusztowania, co pozwoliłoby mu nic nie robić przy ostatnim żądaniu.

Strategia asystenta przy ograniczonej pamięci

Wczesnym rankiem asystent poprosił Leonarda o zapisanie ciągu C na kartce, żeby móc zawczasu znaleźć optymalną strategię działania. Jednak Leonardo zazdrośnie strzeże sekretów swojej pracy i nie zgodził się przekazać asystentowi ciągu C na piśmie. Pozwolił mu jedynie przeczytać ciąg C; asystent może więc podjąć próbę jego zapamiętania.

Niestety pamięć asystenta potrafi zmieścić tylko M bitów informacji. Może to oznaczać, że asystent nie będzie mógł zapamiętać całego ciągu C. Asystent musi zatem wymyślić sprytny sposób na zapamiętanie pewnych informacji w M bitach. Zapamiętany przez niego ciąg bitów będziemy nazywać **ciągiem podpowiedzi** *i oznaczymy przez A.*

Przykład 2. *Po przeczytaniu ciągu C z notatek Leonarda, asystent może, na przykład, wyznaczyć zawartość rusztowania po każdym żądaniu. Jeśli postanowiłby użyć (nieoptymalnej) strategii z przykładu 1, musiałby zapamiętać następujące zawartości rusztowania: $[0, 2]$, $[0, 2]$,*

[3, 2], [3, 0]. Zauważ, że jasne jest, iż początkowa zawartość rusztowania to [0, 1], i nie trzeba tej informacji zapamiętywać.

Przyjmijmy teraz, że $M = 16$, czyli asystent potrafi zapamiętać 16 bitów informacji. Skoro $N = 4$, każdy kolor da się zapisać przy użyciu dwóch bitów. Zatem 16 bitów wystarcza do zapisania zawartości rusztowania. Asystent obliczy więc następujący ciąg podpowiedzi: $A = (0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0)$.

W ciągu dnia asystent może odkodować ciąg podpowiedzi i użyć go do podejmowania decyzji.

(Oczywiście, skoro $M = 16$, asystent mógłby po prostu zapamiętać cały ciąg C przy użyciu zaledwie 8 bitów, jednak celem tego przykładu jest pokazanie, że asystent ma także inne możliwości.)

Zadanie

Twoim zadaniem jest napisanie **dwóch programów** *w tym samym języku programowania. Programy te zostaną uruchomione osobno (jeden po drugim) i nie będą mogły się ze sobą komunikować w trakcie wykonania.*

Pierwszy program powinien symulować poranne zachowanie asystenta. Program ten otrzyma ciąg C i powinien obliczyć ciąg podpowiedzi A.

Drugi program powinien symulować zachowanie asystenta w ciągu dnia. Program ten otrzyma ciąg podpowiedzi A (obliczony przez pierwszy program) i powinien obsłużyć ciąg C, opisujący żądania Leonarda. Ciąg C będzie odsłaniany drugiemu programowi wyraz po wyrazie i każde żądanie należy obsłużyć przed poznaniem kolejnego żądania.

Mówiąc dokładniej, w pierwszym programie należy zaimplementować jedną funkcję `ComputeAdvice`(C, N, K, M)*, która jako argument otrzymuje ciąg C (składający się z N liczb całkowitych z przedziału $[0, N-1]$), liczbę K oznaczającą liczbę pojemników, które mieszczą się na rusztowaniu, oraz liczbę M oznaczającą, ile bitów potrafi zapamiętać asystent. Program powinien obliczyć ciąg podpowiedzi A, który składa się z co najwyżej M bitów, a następnie podać zawartość tego ciągu przez wywołanie poniższej funkcji dla kolejnych bitów:*

- `WriteAdvice`(B) *– dopisz bit B na końcu aktualnego ciągu podpowiedzi A. Możesz wywołać tę funkcję co najwyżej M razy.*

W drugim programie należy zaimplementować funkcję `Assist`(A, N, K, R)*. Otrzymuje ona ciąg podpowiedzi A, liczby całkowite N i K opisane powyżej oraz liczbę R, która oznacza długość ciągu A w bitach ($R \leqslant M$). Funkcja* `Assist` *powinna zrealizować strategię asystenta przy użyciu następujących funkcji:*

- `GetRequest`$()$ *– zwraca numer następnego pojemnika, o który prosi Leonardo. (Nie dostarcza ona jednak informacji o przyszłych żądaniach Leonarda).*

- `PutBack`(T) *– odłóż pojemnik T z rusztowania na półkę. T musi być numerem jednego z pojemników, które aktualnie znajdują się na rusztowaniu.*

Funkcja `Assist` *powinna wywołać* `GetRequest` *dokładnie N razy, za każdym razem otrzymując kolejne żądanie Leonardo. Po każdym wywołaniu* `GetRequest`*, jeśli żądany pojemnik* **nie** *znajduje się na rusztowaniu,* **należy koniecznie** *wywołać* `PutBack`(T) *dla pewnego T. W przeciwnym razie* **nie należy** *wywoływać* `PutBack`*. Niestosowanie się do tego wymagania*

powoduje natychmiastowe zakończenie programu. Pamiętaj, że na początku na rusztowaniu znajdują się pojemniki o numerach od 0 do $K-1$.

Dany test uznaje się za zaliczony, jeśli obydwie napisane przez Ciebie funkcje stosują się do podanych ograniczeń, a liczba wywołań `PutBack` *jest* **taka sama**, *jak w optymalnej strategii Leonarda. Jeśli istnieje wiele strategii o takiej samej liczbie wywołań* `PutBack`, *jak w optymalnej strategii Leonarda, Twój program może użyć dowolnej z nich. (W szczególności, nie trzeba używać strategii Leonarda, jeśli istnieje inna, równie dobra strategia).*

Przykład 3. *Wróćmy do przykładu 2 i załóżmy, że funkcja* `ComputeAdvice` *wyznaczyła ciąg $A = (0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0)$. Aby zwrócić ten ciąg, należy wywołać:* `WriteAdvice`(0), `WriteAdvice`(0), `WriteAdvice`(1), `WriteAdvice`(0), `WriteAdvice`(0), `WriteAdvice`(0), `WriteAdvice`(1), `WriteAdvice`(0), `WriteAdvice`(1), `WriteAdvice`(1), `WriteAdvice`(1), `WriteAdvice`(0), `WriteAdvice`(1), `WriteAdvice`(1), `WriteAdvice`(0), `WriteAdvice`(0).

Następnie wywołana zostaje funkcja `Assist`. *Jej parametry to powyższy ciąg A oraz $N = 4$, $K = 2$ i $R = 16$. Funkcja* `Assist` *musi $N = 4$ razy wywołać funkcję* `GetRequest`. *Ponadto po niektórych wywołaniach* `GetRequest`, *funkcja* `Assist` *powinna wywołać również funkcję* `PutBack`(T) *dla odpowiednio dobranej wartości T.*

Poniższa tabela pokazuje ciąg wywołań odpowiadający (nieoptymalnym) wyborom z przykładu 1. Kreska oznacza, że `PutBack` *nie jest wywoływana.*

`GetRequest()`	`PutBack`
2	`PutBack`(1)
0	–
3	`PutBack`(0)
0	`PutBack`(2)

Podzadanie 1 [8 punktów]

$N \leqslant 5\,000$
Można zapamiętać co najwyżej $M = 65\,000$ bitów.

Podzadanie 2 [9 punktów]

$N \leqslant 100\,000$
Można zapamiętać co najwyżej $M = 2\,000\,000$ bitów.

Podzadanie 3 [9 punktów]

$N \leqslant 100\,000$
$K \leqslant 25\,000$
Można zapamiętać co najwyżej $M = 1\,500\,000$ bitów.

Podzadanie 4 [35 punktów]

$N \leqslant 5\,000$
Można zapamiętać co najwyżej $M = 10\,000$ bitów.

Podzadanie 5 [co najwyżej 39 punktów]

$N \leqslant 100\,000$
$K \leqslant 25\,000$
Można zapamiętać co najwyżej $M = 1\,800\,000$ *bitów.*

Punktacja za to podzadanie zależy od długości ciągu podpowiedzi, który wyznaczy Twój program. Niech R_{max} *oznacza maksymalną (po wszystkich testach) długość ciągu podpowiedzi obliczoną przez* `ComputeAdvice`*. Otrzymasz:*

- *39 punktów, jeśli* $R_{max} \leqslant 200\,000$*;*
- $39 \cdot (1\,800\,000 - R_{max})/1\,600\,000$ *punktów, jeśli* $200\,000 < R_{max} < 1\,800\,000$*;*
- *0 punktów, jeśli* $R_{max} \geqslant 1\,800\,000$*.*

Szczegóły implementacji

Powinieneś zgłosić dokładnie dwa pliki z programami napisanymi **w tym samym języku programowania**.

Pierwszy plik powinien nazywać się `advisor.c`*,* `advisor.cpp` *lub* `advisor.pas`*. Musi on zawierać implementację funkcji* `ComputeAdvice` *zgodną z powyższym opisem. W implementacji można wywoływać funkcję* `WriteAdvice`*. Drugi plik powinien nazywać się* `assistant.c`*,* `assistant.cpp` *lub* `assistant.pas`*. Musi on zawierać implementację funkcji* `Assist` *zgodną z powyższym opisem. W implementacji można wywoływać funkcje* `GetRequest` *i* `PutBack`*.*

Programy w C/C++

```
void ComputeAdvice(int *C, int N, int K, int M);
void WriteAdvice(unsigned char a);

void Assist(unsigned char *A, int N, int K, int R);
void PutBack(int T);
int GetRequest();
```

Programy w Pascalu

```
procedure ComputeAdvice(var C : array of LongInt; N, K, M : LongInt);
procedure WriteAdvice(a : Byte);

procedure Assist(var A : array of Byte; N, K, R : LongInt);
procedure PutBack(T : LongInt);
function GetRequest : LongInt;
```

Funkcje powinny działać dokładnie tak, jak opisano powyżej. Jeśli w obydwóch swoich programach zechcesz zaimplementować jakieś dwie funkcje, które będą miały taką samą nazwę, ich deklaracje powinieneś poprzedzić słowem kluczowym `static`*. Twoje programy nie powinny korzystać ze standardowego wejścia, standardowego wyjścia lub jakichkolwiek plików.*

Twoje rozwiązanie powinno także spełniać poniższe wymagania (w szczególności, spełniają je szablony rozwiązań na Twoim komputerze).

Programy w C/C++

Na początku Twojego rozwiązania powinieneś dołączyć plik `advisor.h` *(w programie obliczającym ciąg podpowiedzi) lub* `assistant.h` *(w programie symulującym przenoszenie pojemników) przy pomocy poniższych dyrektyw:*

```
#include "advisor.h"   lub   #include "assistant.h"
```

Pliki `advisor.h` *i* `assistant.h` *znajdują się w katalogu na dysku Twojego komputera. Można je również znaleźć na stronie systemu sprawdzającego. Dostępny jest również (w ten sam sposób) kod i skrypty kompilujące i testujące rozwiązania. Po skopiowaniu Twojego rozwiązania do katalogu z tymi skryptami wystarczy uruchomić* `compile_c.sh` *lub* `compile_cpp.sh` *(w zależności od języka programowania).*

Programy w Pascalu

Powinieneś użyć modułu `advisorlib` *(w programie obliczającym ciąg podpowiedzi) lub* `assistantlib` *(w programie symulującym przenoszenie pojemników). W tym celu, plik z rozwiązaniem powinien zawierać:*

```
uses advisorlib;   lub   uses assistantlib;
```

Pliki `advisorlib.pas` *i* `assistantlib.pas` *znajdują się w katalogu na dysku Twojego komputera. Można je również znaleźć na stronie systemu sprawdzającego. Dostępny jest również (w ten sam sposób) kod i skrypty kompilujące i testujące rozwiązania. Po skopiowaniu Twojego rozwiązania do katalogu z tymi skryptami wystarczy uruchomić* `compile_pas.sh`*.*

Przykładowy moduł oceniający

Przykładowy moduł oceniający wczytuje dane w poniższym formacie:

- *wiersz 1:* N, K, M*;*
- *wiersze 2,* $\ldots$*,* $N + 1$*:* $C[i]$*.*

Przykładowy moduł oceniający na początku wywoła funkcję `ComputeAdvice`*, co spowoduje utworzenie pliku* `advice.txt`*, który zawierać będzie poszczególne bity ciągu podpowiedzi, pooddzielane pojedynczymi odstępami. Ciąg będzie zakończony liczbą 2.*

Obliczony ciąg zostanie następnie przekazany Twojej implementacji funkcji `Assist`*. Na wyjście zostanie wypisana pewna liczba wierszy. Każdy z tych wierszy zawierać będzie albo* „`R [liczba]`”*, albo* „`P [liczba]`”*. Wiersze pierwszego typu oznaczają wywołania* `GetRequest()` *i otrzymane odpowiedzi. Wiersze drugiego typu odpowiadają wywołaniom* `PutBack()` *i zawierają numery pojemników do odłożenia na półkę. Na końcu zostanie wypisany wiersz o treści* „`E`”*.*

XVIII Bałtycka Olimpiada Informatyczna,

Ventspils, Łotwa 2012

Dawid Dąbrowski, Jakub Radoszewski
Tłumaczenie

BOI 2012, dzień pierwszy, 4.05.2012

Nawiasy

Zdefiniujmy **poprawne wyrażenie nawiasowe** *jak następuje:*

- `()` *oraz* `[]` *są poprawnymi wyrażeniami nawiasowymi;*
- *jeśli* `A` *jest poprawnym wyrażeniem nawiasowym, to* `(A)` *oraz* `[A]` *również są poprawnymi wyrażeniami nawiasowymi;*
- *jeśli* `A` *oraz* `B` *są poprawnymi wyrażeniami nawiasowymi, to ich sklejenie* `AB` *również jest poprawnym wyrażeniem nawiasowym.*

Jeśli weźmiemy dowolne poprawne wyrażenie nawiasowe, w którym występuje przynajmniej jedna para nawiasów kwadratowych — czyli `[` *i odpowiadający* `]` *— i zamienimy w nim każdy nawias kwadratowy (zarówno otwierający, jak i zamykający) na* **otwierający** *nawias okrągły, otrzymamy wyrażenie, które nazwiemy* **zaburzonym wyrażeniem nawiasowym**.

Na przykład, `((` *oraz* `(((((()))` *są zaburzonymi wyrażeniami nawiasowymi. Pierwsze powstaje z poprawnego wyrażenia nawiasowego* `[]`*. Drugie można uzyskać z czterech poprawnych wyrażeń:* `[](((())))`*,* `([](()))`*,* `(([]()))` *i* `(([]))`*.*

Twoim zadaniem jest, dla danego zaburzonego wyrażenia nawiasowego, obliczyć, z ilu poprawnych wyrażeń nawiasowych mogło ono powstać.

Wejście

Pierwszy wiersz pliku `brackets.in` *zawiera jedną parzystą liczbę całkowitą* N *(*$2 \leqslant N \leqslant 30\,000$*) — długość danego zaburzonego wyrażenia nawiasowego. Drugi wiersz zawiera* N *znaków* `(` *i* `)` *opisujących dane wyrażenie.*

Wyjście

Jedyny wiersz pliku `brackets.out` *powinien zawierać jedną liczbę całkowitą — resztę z dzielenia szukanej liczby wyrażeń przez 1 000 000 009.*

Przykład

Dla pliku wejściowego `brackets.in`:

```
4
(((()
```

poprawnym wynikiem jest plik wyjściowy `brackets.out`:

```
2
```

Odpowiadające poprawne wyrażenia nawiasowe: `[]()`*,* `([])`*.*

Dla pliku wejściowego `brackets.in`:
```
8
((((((((
```
poprawnym wynikiem jest plik wyjściowy `brackets.out`:
```
14
```

Odpowiadające poprawne wyrażenia nawiasowe: `[][][][]`, `[[]][][]`, `[[]][[]]`, `[][][[]]`, `[[]]]][]`, `[[][]][]`, `[][[][]]`, `[[[[]]]]`, `[[][[]]]`, `[[[]][]]`, `[[][][]]`, `[[[][]]]`, `[][[]][]`.

Ocenianie

W testach wartych 20 punktów zachodzi warunek $N \leqslant 50$.
W testach wartych 45 punktów zachodzi warunek $N \leqslant 1\,000$.

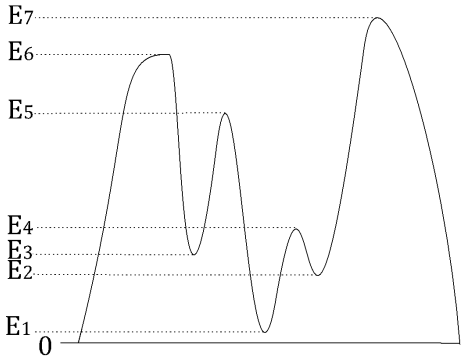
Dawid Dąbrowski, Jakub Radoszewski
Tłumaczenie

BOI 2012, dzień pierwszy, 4.05.2012

Szczyty

Alpinista wspiął się na jeden ze szczytów pewnej górzystej wyspy. Teraz chciałby dotrzeć na jakikolwiek wyższy szczyt.

Każdy punkt wyspy charakteryzuje się pewną dodatnią wysokością nad poziomem morza (morze ma wysokość 0). Jeśli alpinista znajduje się właśnie na szczycie o wysokości E_i, to jego kolejnym celem jest dotarcie na jakiegokolwiek szczyt o wysokości $E_j > E_i$. Oczywiście, ze szczytu nie ma żadnej bezpośredniej ścieżki w górę — aby zrealizować swój cel, alpinista musi zejść do jakiegoś punktu położonego niżej i dopiero wówczas będzie mógł znowu pójść w górę. Schodzenie w dół nie jest tak ekscytujące jak wspinaczka, więc alpinista chciałby wyznaczyć maksymalną wysokość najniższego punktu na ścieżce z bieżącego szczytu na jakikolwiek wyższy szczyt.



Przykładowo, jeśli profil wysokościowy wyspy jest taki, jak na rysunku powyżej, i alpinista znajduje się na szczycie o wysokości E_4, to ma on do wyboru trzy wyższe szczyty (E_5, E_6 i E_7), a ścieżka o najniższym punkcie położonym najwyżej prowadzi na szczyt o wysokości E_7 — aby się tam dostać, trzeba zejść w dół do poziomu E_2 (w pozostałych przypadkach trzeba zejść aż do poziomu E_1). Gdyby zacząć na szczycie E_5, wynikiem (najniższym punktem trasy) byłoby E_3 (ścieżka do E_6), natomiast dla E_6 wynikiem byłoby E_1.

Mapa wyspy jest dwuwymiarową prostokątną tablicą zawierającą $N \times M$ pól i opisującą wysokości poszczególnych części wyspy — liczba wpisana w dane pole oznacza wysokość nad poziomem morza odpowiadającego mu regionu wyspy. Dwa pola uznajemy za sąsiednie, gdy mają co najmniej jeden punkt wspólny. Tak więc każde pole (poza tymi na obrzeżu wyspy) sąsiaduje z ośmioma innymi polami. **Ścieżką** *nazywamy ciąg kolejno sąsiadujących ze sobą pól.* **Obszarem płaskim** *nazywamy maksymalny zbiór pól o tej samej wysokości, w którym każde dwa pola są połączone ścieżką przechodzącą wyłącznie przez pola tego obszaru.* **Szczytem** *nazywamy obszar płaski, którego żadne z pól nie sąsiaduje z polami o większej wysokości.*

Napisz program, który znajdzie wszystkie szczyty na wyspie i dla każdego z nich obliczy wysokość najwyższego spośród najniższych punktów na ścieżce prowadzącej na jakiś wyższy szczyt. Jeśli zaś chodzi o najwyższy szczyt wyspy (od którego, rzecz jasna, nie ma na wyspie

wyższych szczytów), to zakładamy, że po znalezieniu się na nim alpinista opuści wyspę w poszukiwaniu innych, jeszcze wyższych szczytów, więc szukanym wynikiem dla niego będzie 0 (poziom morza).

Wejście

W pierwszym wierszu pliku `peaks.in` *znajdują się dwie dodatnie liczby całkowite N oraz M ($1 \leqslant N, M \leqslant 2\,000$, $N \cdot M \leqslant 10^5$), oznaczające wysokość i szerokość mapy wyspy. W kolejnych N wierszach podany jest opis mapy. W i-tym z tych wierszy znajduje się M liczb całkowitych E_{ij} ($1 \leqslant E_{ij} \leqslant 10^6$ dla $1 \leqslant j \leqslant M$) pooddzielanych pojedynczymi odstępami.*

Wyjście

W pierwszym wierszu pliku `peaks.out` *należy wypisać jedną liczbę całkowitą P, oznaczającą liczbę szczytów znalezionych na wyspie. W kolejnych P wierszach powinny znajdować się po dwie liczby całkowite: wysokość szczytu i wysokość najwyższego spośród najniższych punktów na ścieżce prowadzącej na jakiś wyższy szczyt. Informacje o szczytach należy wypisać w porządku nierosnących wysokości; jeśli jest kilka szczytów o tej samej wysokości, należy je wypisać w porządku nierosnących wysokości szukanych najniższych punktów.*

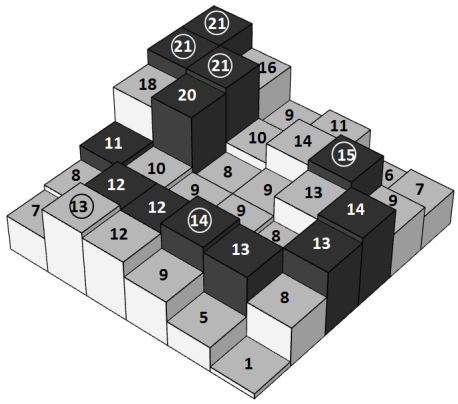
Przykład

Dla pliku wejściowego `peaks.in`:

```
6 6
21 16 9 11 6 7
21 21 10 14 15 9
18 20 8 9 13 14
11 10 9 9 8 13
8 12 12 14 13 8
7 13 12 9 5 1
```

poprawnym wynikiem jest plik wyjściowy `peaks.out`:

```
4
21 0
15 11
14 13
13 12
```



Wszystkie szczyty są oznaczone kółkami. Zaznaczono optymalną ścieżkę ze szczytu o wysokości 15.

Dla pliku wejściowego `peaks.in`:

```
5 3
16 14 16
14 14 15
12 17 16
12 13 10
16 11 16
```

poprawnym wynikiem jest plik wyjściowy `peaks.out`:

```
5
17 0
16 15
16 14
16 13
16 13
```

Ocenianie

W testach wartych 15 punktów zachodzi warunek $N \leqslant 2$ *lub* $M \leqslant 2$.
W testach wartych 50 punktów zachodzi warunek $P \leqslant 500$.
W testach wartych 80 punktów zachodzi warunek $P \leqslant 5\,000$.

Dawid Dąbrowski, Jakub Radoszewski
Tłumaczenie

BOI 2012, dzień pierwszy, 4.05.2012

Telefonia komórkowa

Znany operator telefonii komórkowej Totalphone rozstawił pewną liczbę nadajników, dzięki którym zamierza zapewnić zasięg komórkowy na nowo wybudowanej autostradzie. Niestety, programiści pracujący w firmie Totalphone to partacze; z tego względu nadajniki nie są konfigurowalne, lecz wszystkie muszą mieć ustawiony dokładnie taki sam zasięg. Firma chciałaby zminimalizować zużycie mocy w sieci. Pomóż jej wyznaczyć maksimum z odległości między punktami autostrady a najbliższymi im nadajnikami sieci.

Wejście

W pierwszym wierszu pliku `mobile.in` *znajdują się dwie liczby całkowite* N *(*$1 \leqslant N \leqslant 10^6$*) oraz* L *(*$1 \leqslant L \leqslant 10^9$*) oznaczające liczbę nadajników oraz długość autostrady. Dalej następuje* N *wierszy, z których każdy zawiera parę liczb całkowitych* x_i, y_i *(*$-10^9 \leqslant x_i, y_i \leqslant 10^9$*) opisującą współrzędne jednego z nadajników. Wszystkie punkty będą różne. Punkty będą uporządkowane niemalejąco według współrzędnych* x_i*. W przypadku równych współrzędnych* x_i *punkty będą uporządkowane rosnąco według współrzędnych* y_i*.*

Autostrada jest odcinkiem o końcach w punktach $(0, 0)$ *i* $(L, 0)$*.*

Wyjście

W pierwszym i zarazem jedynym wierszu pliku `mobile.out` *powinna znaleźć się jedna liczba — maksymalna z odległości od punktu autostrady do najbliższego mu nadajnika sieci. Twój wynik zostanie uznany za poprawny, jeśli będzie różnił się od idealnie dokładnego wyniku o co najwyżej* 10^{-3}*.*

Przykład

Dla pliku wejściowego `mobile.in`:

```
2 10
0 0
11 1
```

poprawnym wynikiem jest plik wyjściowy `mobile.out`:

```
5.545455
```

Ocenianie

W testach wartych 25 punktów zachodzi warunek $N \leqslant 5\,000$*.*
W testach wartych 50 punktów zachodzi warunek $N \leqslant 100\,000$*.*

Uwaga

W obliczeniach użyj zmiennych rzeczywistych podwójnej precyzji; w przeciwnym przypadku Twój wynik może nie zmieścić się w zadanej dokładności.

Dawid Dąbrowski, Jakub Radoszewski
Tłumaczenie

BOI 2012, dzień drugi, 5.05.2012

Fajerwerki w RightAngleles

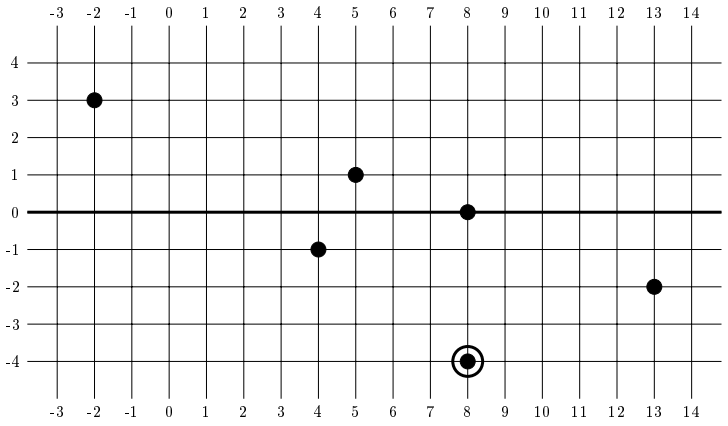
W mieście RightAngleles ulice tworzą regularną siatkę — każde dwie ulice są albo prostopadłe, albo równoległe. Odległości między dwiema sąsiednimi równoległymi ulicami są równe i wynoszą jedną **jednostkę**. *Ulice biegnące z zachodu na wschód nazywamy* **poziomymi** *i numerujemy kolejnymi liczbami całkowitymi z południa na północ. Ulice biegnące z południa na północ nazywamy* **pionowymi** *i numerujemy kolejnymi liczbami całkowitymi z zachodu na wschód.*

Każdy mieszkaniec RightAngleles mieszka na przecięciu pewnej ulicy pionowej z pewną ulicą poziomą. W tym samym miejscu mogą znajdować się domy wielu mieszkańców.

Burmistrz RightAngleles chciałby podnieść swoją popularność poprzez zorganizowanie pokazu fajerwerków na przecięciu głównej ulicy poziomej (o numerze 0) z pewną ulicą pionową. Wiemy, którzy mieszkańcy byliby zainteresowani przyjściem na pokaz. Fajerwerki będą widoczne na obu ulicach, na których przecięciu będzie odbywał się pokaz (czyli na poziomej ulicy 0 oraz na pewnej ulicy pionowej). Z powodów bezpieczeństwa, obserwatorzy muszą zająć miejsca w odległości co najmniej S jednostek od miejsca pokazu. Tak więc, jeśli pokaz będzie odbywał się na przecięciu głównej ulicy poziomej z pewną ulicą pionową V, to obserwatorzy muszą przyjść na główną ulicę poziomą lub na ulicę pionową V, ale w odległości co najmniej S jednostek od miejsca pokazu. Na przykład, jeśli $S = 2$, to mieszkańcy chcący obejrzeć pokaz mogą przyjść na dowolne skrzyżowanie położone na głównej ulicy poziomej oprócz przecięć z ulicami pionowymi $V - 1$, V oraz $V + 1$ lub na dowolne skrzyżowanie położone na ulicy pionowej V oprócz jej przecięć z ulicami poziomymi -1, 0 oraz 1.

Pozytywne wrażenie na mieszkańcach wywarte przez pokaz jest ściśle związane z odległością, jaką muszą przebyć, by na niego dotrzeć. Trzeba więc tak wybrać miejsce pokazu, by łączna odległość, jaką muszą pokonać mieszkańcy, aby na niego dotrzeć, była jak najmniejsza.

Na przykład, jeśli $S = 2$ i w mieście jest siedmiu mieszkańców zainteresowanych pokazem, których miejsca zamieszkania są pokazane na rysunku (w punkcie $(-4, 8)$ są domy dwóch mieszkańców), to najlepszym miejscem na zorganizowanie pokazu jest przecięcie 8. ulicy pionowej z główną ulicą poziomą — suma odległości, jaką muszą wówczas przebyć mieszkańcy, to 9 jednostek.



Napisz program, który obliczy minimalną sumę dystansów (w jednostkach), jakie muszą przebyć mieszkańcy, by zobaczyć pokaz, przy założeniu, że miejsce jego organizacji zostało wybrane optymalnie.

Wejście

Dane wejściowe znajdują się w pliku `fire.in`. *W pierwszym wierszu są podane dwie dodatnie liczby całkowite oddzielone pojedynczym odstępem: liczba mieszkańców zainteresowanych pokazem N ($N \leqslant 10^5$) oraz bezpieczna odległość S ($S \leqslant 10^6$), wyrażona w jednostkach. W kolejnych N wierszach znajdują się opisy domów mieszkańców. Każdy z nich składa się z dwóch liczb całkowitych H_i oraz V_i oddzielonych pojedynczym odstępem. H_i ($-10^9 \leqslant H_i \leqslant 10^9$) to numer ulicy poziomej, a V_i ($-10^9 \leqslant V_i \leqslant 10^9$) — numer ulicy pionowej, na przecięciu których znajduje się dom danego mieszkańca.*

Wyjście

W pierwszym i jedynym wierszu pliku `fire.out` *powinna znaleźć się jedna liczba całkowita — minimalna sumaryczna odległość (w jednostkach), jaką muszą pokonać mieszkańcy, by móc obejrzeć pokaz.*

Przykład

Dla pliku wejściowego `fire.in`:

```
7 2
3 -2
0 8
-4 8
-1 4
-2 13
-4 8
```

poprawnym wynikiem jest plik wyjściowy `fire.out`:

```
9
```

Uwaga: *Przykład odpowiada rysunkowi z treści zadania.*

Ocenianie

W testach wartych 20 punktów zachodzi warunek $0 \leqslant V_i \leqslant 5\,000$.
W testach wartych 40 punktów zachodzi warunek $N \leqslant 5\,000$.

Dawid Dąbrowski, Jakub Radoszewski
Tłumaczenie

BOI 2012, dzień drugi, 5.05.2012

Melodia

Linus lubi grać na pewnym instrumencie muzycznym. Instrument ten ma S otworów. Za jego pomocą można zagrać N różnych nut (ponumerowanych od 1 do N), zakrywając każdy z otworów na jeden z 10 sposobów (ponumerowanych od 0 do 9). Każdą nutę można zagrać na dokładnie jeden sposób, który można przedstawić za pomocą sekwencji cyfr opisującej sposób zakrycia poszczególnych otworów. Jeśli otwory zostaną zakryte w niepoprawny sposób (nieodpowiadający żadnej nucie), instrument wydaje bardzo nieprzyjemne odgłosy. Linus woli więc czasem w poprawny sposób zagrać złą nutę niż zakryć otwory w sposób nieodpowiadający żadnej nucie.

Linus jest członkiem zespołu muzycznego i musi umieć grać różne skomplikowane utwory. Ma on zapisaną melodię (czyli ciąg liczb reprezentujący kolejne nuty), którą chciałby zagrać wraz z zespołem. Niestety, Linus nie jest wirtuozem swojego instrumentu. Potrafi kolejno zagrać dwie dane nuty, tylko jeśli różnią się one sposobem zakrycia co najwyżej G otworów. Linus zdecydował się zatem zmienić niektóre nuty w melodii tak, by był w stanie zagrać ją w całości. Zmiana jednej nuty liczy się jako jeden błąd.

Zadanie

Dla danej melodii, wyznacz, jaką (zmodyfikowaną) melodię powinien zagrać Linus, by popełnić jak najmniej błędów.

Wejście

Pierwszy wiersz pliku `melody.in` *zawiera trzy liczby całkowite: liczbę nut możliwych do zagrania N ($1 \leqslant N \leqslant 100$), liczbę otworów w instrumencie S oraz liczbę G oznaczającą sprawność Linusa ($0 \leqslant G < S \leqslant 100$). Kolejne N wierszy zawiera opisy nut. Każdy z nich zawiera S cyfr bez jakichkolwiek odstępów. j-ta cyfra w $(i+1)$-szym wierszu opisuje sposób zakrycia j-tego otworu podczas grania i-tej nuty (różne sposoby zakrycia otworów są oznaczane cyframi od 0 do 9). Opisy wszystkich nut będą różne.*

$(N+2)$-gi wiersz wejścia zawiera długość melodii L ($1 \leqslant L \leqslant 10^5$). Ostatni wiersz zawiera opis melodii: L liczb całkowitych pooddzielanych pojedynczymi odstępami, oznaczających numery kolejnych nut w melodii.

Wyjście

W pierwszym wierszu pliku `melody.out` *powinna zostać zapisana jedna nieujemna liczba całkowita — najmniejsza możliwa liczba błędów, jaką Linus może popełnić. Drugi wiersz powinien zawierać opis melodii, którą powinien zagrać: L liczb całkowitych pooddzielanych pojedynczymi odstępami — numery nut w melodii. Jeśli istnieje wiele poprawnych odpowiedzi, Twój program może wypisać dowolną z nich.*

Przykład

Dla pliku wejściowego `melody.in`:

```
5 4 2
1111
2101
2000
0100
0000
7
1 5 4 5 3 2 1
```

poprawnym wynikiem jest plik wyjściowy `melody.out`:

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

Komentarz: *Linus nie może zagrać nuty numer 5 bezpośrednio po nucie numer 1.*

Ocenianie

W testach wartych 40 punktów zachodzi warunek $L \leqslant 100$.
W testach wartych 65 punktów zachodzi warunek $L \leqslant 5\,000$.

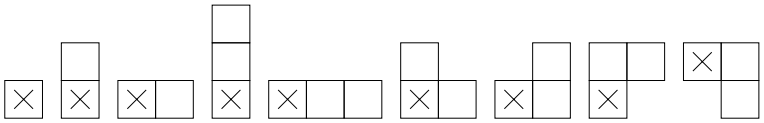
Dawid Dąbrowski, Jakub Radoszewski
Tłumaczenie

BOI 2012, dzień drugi, 5.05.2012

Tiny

Starsi ludzie pamiętają jeszcze grę komputerową o nazwie Tetris, stworzoną przez Aleksieja Pażytnowa. W tej grze klocki złożone z czterech małych kwadratów (tetromino) spadają na prostokątną planszę, a celem gracza jest w taki sposób obracać kolejne klocki i przemieszczać je w poziomie w ramach planszy, aby zapełnić jak największą liczbę wierszy planszy. Zapełniony wiersz znika, tworząc tym samym miejsce dla kolejnych spadających klocków.

W tym zadaniu zajmiemy się uproszczoną wersją gry, którą nazwiemy Tiny Tetris (albo po prostu Tiny). W grze Tiny jest tylko dziewięć różnych klocków (będziemy je nazywać t-klockami), z których każdy składa się z jednego, dwóch lub trzech kwadratów:



Numery na rysunku oznaczają typy t-klocków. Za pomocą tych liczb będziemy w dalszym tekście odwoływać się do konkretnych t-klocków.

Cel gry jest ten sam — za pomocą spadających t-klocków będziemy zapełniać prostokątną planszę o wysokości i szerokości 9 kwadratów. W przeciwieństwie do klocków z gry Tetris, t-klocków nie można w locie obracać ani przemieszczać na lewo czy prawo. Gracz może jedynie wybrać numer kolumny planszy (liczba całkowita między 1 a 9), wzdłuż której będzie spadać skrajnie lewy kwadrat t-klocka (oznaczony na rysunku jako $\times$).

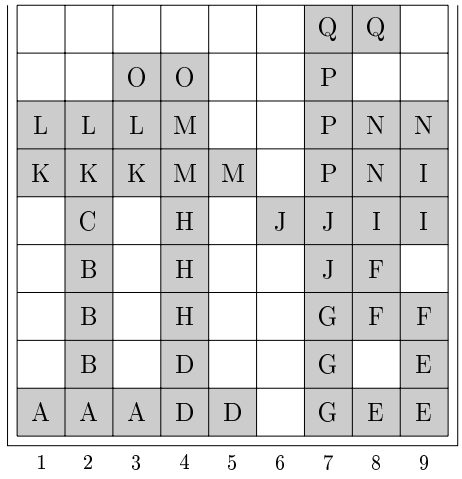
W grze mamy daną sekwencję złożoną z N t-klocków. Należy jak najwięcej z nich upuścić na planszę, wykonując jedynie legalne ruchy oraz nie dopuszczając, by jakiś klocek wystawał poza górny brzeg planszy. Wynik gry jest równy liczbie poprawnie upuszczonych t-klocków.

Na początku gry licznik jest ustawiony na 0. Przebieg gry jest następujący:

1. *Gracz wybiera kolumnę, w której zostanie upuszczony dany t-klocek.*
2. *Jeśli kolumna została wybrana poprawnie (dla przykładu, kolumna numer 8 nie będzie nigdy poprawną kolumną dla t-klocka typu 5), t-klocek spada w dół aż do napotkania przeszkody. W przeciwnym razie gra się kończy.*
3. *Jeśli t-klocek zmieścił się całkowicie na planszy (czyli wszystkie jego kwadraty zmieściły się w prostokącie 9×9), licznik jest zwiększany o 1. W przeciwnym razie gra się kończy.*
4. *Następnie sprawdza się, czy zostały zapełnione jakieś wiersze planszy. Jeżeli są takie wiersze, to znikają one z planszy, a wiersze znajdujące się powyżej nich zostają przesunięte w dół, bez zmiany ich konfiguracji.*
5. *Jeśli w sekwencji zostały jeszcze jakieś t-klocki, przechodzimy do punktu 1. W przeciwnym razie gra się kończy.*

Wynik gry jest równy wartości licznika w chwili zakończenia gry.

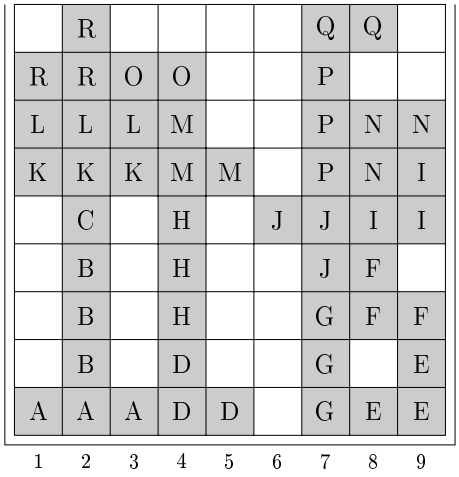
Przeanalizujmy pewien konkretny przebieg gry:



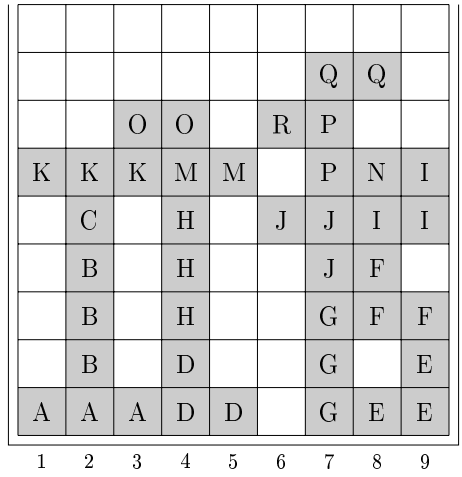
Dana jest następująca sekwencja 20 t-klocków: 5, 4, 1, 6, 7, 6, 4, 4, 7, 9, 5, 5, 6, 8, 3, 4, 3, 7, 4, 2. Załóżmy, że pierwsze 17 t-klocków zostało już upuszczonych na planszę w następujących kolumnach: 1, 2, 2, 4, 8, 8, 7, 4, 8, 6, 1, 1, 4, 8, 3, 7, 7. Do tego momentu żaden wiersz nie został zapełniony, licznik ma wartość 17 i teraz powinniśmy upuścić t-klocek typu 7 (kolejne litery na rysunku odpowiadają kolejnym t-klockom w sekwencji).

Są tylko dwie kolumny, w których możemy teraz upuścić rozważany t-klocek typu 7:

a) kolumna 1:



b) kolumna 5 (w tym przypadku zapełniamy jeden wiersz, który natychmiast znika):



Zadanie

Masz dane pięć plików opisujących sekwencje t-klocków w poszczególnych grach: `tiny.i1`, `tiny.i2`, `tiny.i3`, `tiny.i4` *oraz* `tiny.i5`. *Każdy z tych plików ma następujący format:*

W pierwszym wierszu znajduje się jedna liczba całkowita N. W kolejnych N wierszach znajdują się opisy t-klocków. Każdy z tych wierszy zawiera jedną liczbę całkowitą od 1 do 9 — typ kolejnego t-klocka. T-klocki są podane w kolejności spadania na planszę.

Dla każdego z podanych plików wejściowych powinieneś zgłosić odpowiedni plik wyjściowy (`tiny.o1`, `tiny.o2`, `tiny.o3`, `tiny.o4` *oraz* `tiny.o5`*) zawierający co najwyżej N wierszy — numery kolumn, w których należy upuszczać poszczególne t-klocki. i-ty wiersz pliku wyjściowego powinien zawierać numer kolumny, w której należy upuścić i-ty t-klocek z wejścia.*

Możesz założyć, że dla każdego pliku wejściowego istnieje sekwencja numerów kolumn, w których należy upuszczać kolejne t-klocki, tak aby wszystkie t-klocki znalazły się na planszy (co daje wynik gry równy N).

Ocenianie

Każdy z pięciu testów jest wart 20 punktów. Liczba punktów, jaka zostanie Ci przyznana za dany plik wyjściowy (dany test), zostanie obliczona zgodnie ze wzorem:

$$20 \cdot twój\_wynik/maksimum\_z\_wyników\_wszystkich\_zawodników$$

i zaokrąglona do dwóch cyfr po przecinku.

Podczas zawodów dla każdego nadesłanego przez Ciebie pliku wyjściowego otrzymasz jako informację zwrotną wynik oraz liczbę punktów, jaką zdobyłbyś, gdyby pewien zawodnik uzyskał maksymalny wynik za ten test. Po zakończeniu konkursu wszystkie pliki wyjściowe zostaną ocenione ponownie, biorąc pod uwagę faktyczne maksymalne wyniki uzyskane przez zawodników, w związku z czym może zostać Ci przyznana większa liczba punktów.

XIX Olimpiada Informatyczna Krajów Europy Środkowej

Tata, Węgry 2012

Jakub Pawlewicz, Jakub Radoszewski
Tłumaczenie

CEOI 2012, dzień pierwszy, 9.07.2012

Szeregowanie zleceń

Centrum Efektywnych Obliczeń Informatycznych (CEOI) ma w najbliższym czasie otrzymać M zleceń. Wszystkie zlecenia muszą zostać przetworzone w ciągu najbliższych N dni. Przetworzenie jednego zlecenia wymaga wykorzystania jednego komputera i zajmuje dokładnie jeden dzień. CEOI ma dostęp do pewnej liczby komputerów, z których każdy może przetwarzać co najwyżej jedno zlecenie dziennie. Tak więc centrum może zrealizować dziennie co najwyżej tyle zleceń, ile ma do dyspozycji komputerów. CEOI zamierza obsługiwać zlecenia z co najwyżej D-dniowym opóźnieniem, tzn. jeśli klient zgłosi zlecenie w dniu S, musi ono zostać przetworzone najpóźniej w dniu $S + D$.

Zadanie

Napisz program, który wyznaczy minimalną liczbę komputerów, które należy wykorzystać, aby zrealizować wszystkie zlecenia z co najwyżej D-dniowym opóźnieniem.

Wejście

Pierwszy wiersz wejścia zawiera trzy liczby całkowite: N ($1 \leqslant N \leqslant 100\,000$) — liczbę dni, przez które centrum będzie przetwarzać zlecenia, D ($0 \leqslant D < N$) — maksymalną liczbę dni dopuszczalnego opóźnienia, oraz M ($1 \leqslant M \leqslant 1\,000\,000$) — liczbę zleceń do przetworzenia. Dni numerujemy od 1 do N, a zlecenia od 1 do M. Drugi wiersz zawiera M liczb całkowitych pooddzielanych pojedynczymi odstępami; i-ta z tych liczb oznacza numer dnia, w którym zostanie zgłoszone i-te zlecenie. Wszystkie zlecenia zostaną zgłoszone w ciągu pierwszych $N - D$ dni.

Wyjście

Pierwszy wiersz wyjścia powinien zawierać jedną liczbę całkowitą K — minimalną liczbę komputerów potrzebnych do przetworzenia wszystkich zleceń z co najwyżej D-dniowym opóźnieniem. Kolejne N wierszy powinno opisywać poprawny plan przetwarzania zleceń. Wiersz $(i+1)$-szy powinien zawierać numery co najwyżej K zleceń, które zostaną przetworzone i-tego dnia. Liczby w każdym wierszu należy rozdzielić pojedynczymi odstępami, a na końcu wiersza należy wypisać 0. Jeśli jest więcej niż jedno poprawne rozwiązanie, Twój program powinien wypisać dowolne z nich.

Przykład

Dla danych wejściowych:

```
8 2 12
1 2 4 2 1 3 5 6 2 3 6 4
```

jednym z poprawnych wyników jest:

```
2
5 1 0
9 4 0
2 10 0
6 12 0
3 7 0
11 8 0
0
0
```

Ocenianie

W 50% testów M nie przekroczy 100 000.
Jeśli poprawny będzie tylko pierwszy wiersz wyjścia, Twój program otrzyma 40% punktów.

Jakub Pawlewicz, Jakub Radoszewski
Tłumaczenie

CEOI 2012, dzień pierwszy, 9.07.2012

Obwód drukowany

Jako rzecze Wikipedia, obwód drukowany (ang. Printed Circuit Board, PCB) jest to „płytka z materiału izolacyjnego z połączeniami elektrycznymi (tzw. ścieżkami) i punktami lutowniczymi (tzw. padami), przeznaczona do montażu podzespołów elektronicznych". Twoja firma zamierza wyprodukować nowe urządzenie elektroniczne na bazie płytki PCB.

Projekt płytki jest już częściowo gotowy. Układ płytki ma kształt wielokąta, którego wierzchołki stanowi N padów ponumerowanych od 1 do N. Dla każdego $u = 1, 2, \ldots, N-1$, pady o numerach u oraz $u+1$ są połączone ścieżką w kształcie odcinka; ponadto pad numer N jest połączony z padem numer 1 ścieżką w kształcie odcinka. Ścieżki nie przecinają się, tzn. jeśli jakieś dwie ścieżki mają punkt wspólny, to są to dwie kolejne ścieżki na obwodzie wielokąta, a ów punkt wspólny to pad będący wspólnym końcem tych ścieżek. Położenie każdego padu jest opisane przez współrzędne x i y. Dodatkowo, w lewym dolnym rogu płytki znajduje się pad o współrzędnych $(0, 0)$, reprezentujący wyjście płytki.

Zadanie

Napisz program, który wyznaczy wszystkie pady, które można połączyć z wyjściem płytki ścieżką w kształcie odcinka tak, aby jedynym punktem wspólnym tej ścieżki z wielokątem był dany pad.

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę całkowitą N ($1 \leqslant N \leqslant 100\,000$), oznaczającą liczbę padów płytki. Każdy z kolejnych N wierszy zawiera dwie liczby całkowite x i y ($0 < x, y \leqslant 1\,000\,000$), oddzielone pojedynczym odstępem — współrzędne pada będącego jednym z wierzchołków wielokąta. Wiersz o numerze $i+1$ zawiera współrzędne pada o numerze i.

Wyjście

Pierwszy wiersz wyjścia powinien zawierać jedną liczbę całkowitą M — liczbę padów, które mogą zostać połączone ścieżką z wyjściem płytki, tak aby ścieżka ta przecinała wielokąt tylko w danym padzie. Drugi wiersz powinien zawierać numery tych M padów, w kolejności rosnącej.

Przykład

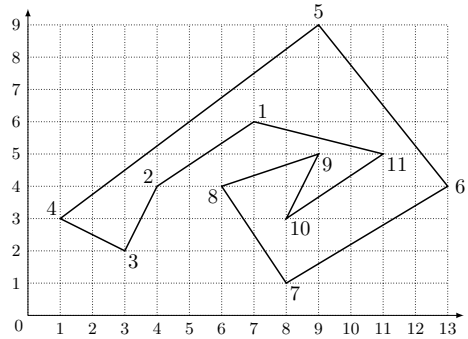
Dla danych wejściowych:

```
11
7 6
4 4
3 2
1 3
9 9
```

```
13 4
8 1
6 4
9 5
8 3
11 5
```

poprawnym wynikiem jest:

```
3
3 4 7
```



Ocenianie

W 10% testów N nie przekroczy 1 000.
Jeśli poprawny będzie tylko pierwszy wiersz wyjścia, Twój program otrzyma 40% punktów.

Jakub Pawlewicz, Jakub Radoszewski
Tłumaczenie

CEOI 2012, dzień pierwszy, 9.07.2012

Regaty żeglarskie

Na jeziorze o kształcie koła co roku organizowane są zawody żeglarskie. Wzdłuż brzegu jeziora położonych jest N portów ponumerowanych lewoskrętnie od 1 do N. Zawody są podzielone na kilka etapów. Każdy etap polega na przepłynięciu w linii prostej między dwoma portami. Trasa zawodów może przebiegać przez każdy port co najwyżej raz.

Organizatorzy chcą wyznaczyć trasę złożoną z możliwie największej liczby etapów. Muszą jednak wziąć pod uwagę to, że między niektórymi parami portów nie można przepłynąć w linii prostej bez napotykania na przeszkody. Na szczęście, dla każdego portu A organizatorzy zaopatrzyli się już w listę portów docelowych, do których można dopłynąć bezpośrednio z portu A.

Zazwyczaj, aby uniknąć możliwości kolizji jachtów, trasę zawodów dobiera się tak, aby żadne dwa etapy nie przecinały się. Jednak w tym roku, dzięki zastosowaniu nowoczesnego systemu sygnalizacji na trasie, można pozwolić sobie na jedno przecięcie etapów — jednak tylko wtedy, gdy znajduje się ono już na pierwszym etapie. Zatem jeśli trasa zaczyna się w porcie S i następnym portem na trasie jest T, to co najwyżej jeden etap trasy może przeciąć etap S-T. Organizatorzy wciąż zostawiają sobie możliwość wyboru, czy pozwolą na takie przecięcie, czy też pozostaną przy klasycznej trasie bez przecinających się etapów.

Zadanie

Napisz program, który wyznaczy trasę zawodów danego rodzaju złożoną z największej możliwej liczby etapów.

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby całkowite, liczbę portów N ($1 \leqslant N \leqslant 500$) i rodzaj trasy k. Jeśli $k = 0$, to wymaga się trasy klasycznej (bez samoprzecięć), a jeśli $k = 1$, to trasa może zawierać co najwyżej jedno samoprzecięcie w podany wyżej sposób. Kolejne N wierszy zawiera listy portów docelowych. Wiersz o numerze $i + 1$ zawiera listę liczb całkowitych pooddzielanych pojedynczymi odstępami, zakończoną zerem — są to porty docelowe i-tego portu.

Wyjście

Pierwszy wiersz wyjścia powinien zawierać jedną liczbę całkowitą — maksymalną możliwą liczbę etapów trasy zawodów danego rodzaju. Drugi wiersz powinien zawierać numer portu startowego takiej trasy zawodów. Jeżeli istnieje wiele rozwiązań, należy wypisać dowolne z nich.

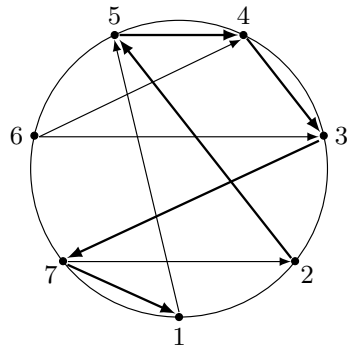
Przykład

Dla danych wejściowych:

```
7 1
5 0
5 0
7 0
3 0
4 0
4 3 0
2 1 0
```

poprawnym wynikiem jest:

```
5
2
```



Ocenianie

W 40% testów zachodzi warunek $k = 0$. W 50% testów N nie przekracza 100.
Jeśli poprawny będzie tylko pierwszy wiersz wyjścia, i tak nie otrzymasz za to żadnych punktów.

Jakub Pawlewicz, Jakub Radoszewski
Tłumaczenie

CEOI 2012, dzień drugi, 11.07.2012

Projektowanie autostrad

Rząd pewnego kraju postanowił zbudować sieć autostrad łączącą N najważniejszych miast kraju. Każde z miast wybrało już lokalizację swojego węzła autostradowego. Autostrady mają być budowane w linii prostej. Szczęśliwie okazało się, że do pokrycia wszystkich miast wystarczą dwie autostrady, tzn. istnieją takie dwie proste, że każdy z węzłów znajduje się na jednej z nich. Ponadto każda z prostych zawiera co najmniej trzy węzły i żaden węzeł nie leży na obu prostych. Łatwo zauważyć, że przy tych założeniach proste są wyznaczone jednoznacznie.

Twoja firma została zatrudniona do zbudowania owych dwóch autostrad. Współrzędne węzłów są ewidencjonowane w biurze rejestru gruntów. Do otrzymywania wypisów z rejestru służy formularz, który pozwala spytać, czy dane trzy węzły są współliniowe. Trzeba za to uiścić niemałą opłatę skarbową (taka już jest ta biurokracja), więc docelowo chcesz wykonać jak najmniej takich wypisów. Ponieważ do opisania prostej wystarczą dwa węzły, Twoim zadaniem jest podać takie dwa węzły dla każdej z szukanych prostych.

Zadanie

Napisz program, który wyznaczy po dwa węzły dla każdej autostrady, stosując jak najmniej wypisów.

Biblioteka

Do pobierania wypisów służy biblioteka `office` *implementująca trzy operacje:*

- `GetN`*, wywoływana tylko raz, na początku, bez żadnych argumentów; zwraca N — liczbę miast. Węzły odpowiadające miastom numerujemy od 1 do N.*
- `isOnLine`*, wywoływana z trzema argumentami — identyfikatorami węzłów.* `isOnLine`(x,y,z) *zwraca 1, jeśli węzły x, y i z są współliniowe, a 0 w przeciwnym przypadku.*
- `Answer`*, wywoływana tylko raz, na końcu; wysyła odpowiedź i kończy działanie programu.* `Answer` *przyjmuje jako argumenty cztery liczby całkowite: a1, b1, a2, b2, przy czym a1 i b1 wyznaczają prostą reprezentującą pierwszą autostradę, a a2 i b2 — drugą autostradę.*

Rozwiązania w Pascalu: *w kodzie źródłowym należy dodać deklarację*

```
uses office;
```

Rozwiązania w C/C++: *należy użyć*

```
#include "office.h"
```

Przykładowa komenda kompilacji w C++:

```
g++ -O2 -static highway.cpp office.cpp -lm -o highway
```

Testowanie

Z systemu zawodów możesz pobrać plik `sample.zip` *zawierający implementację przykładowej biblioteki* `office`*. Biblioteka ta czyta dane ze standardowego wejścia. Pierwszy wiersz wejścia powinien zawierać jedną liczbę całkowitą — liczbę miast. Drugi wiersz powinien zawierać listę liczb całkowitych pooddzielanych pojedynczymi odstępami — identyfikatory węzłów należących do pierwszej autostrady. Wszystkie pozostałe węzły, które nie znalazły się na tej liście, leżą na drugiej autostradzie.*

Ustalenia

- *Liczba miast N spełnia warunek $40 \leqslant N \leqslant 100\,000$.*
- *Deklaracje funkcji w Pascalu:*
```
function GetN: longint;
function isOnLine(x, y, z: longint): integer;
procedure Answer(a1, b1, a2, b2: longint);
```
- *Deklaracje funkcji w C/C++:*
```
int GetN(void);
int isOnLine(int x, int y, int z);
void Answer(int a1, int b1, int a2, int b2);
```
- *Zabronione jest czytanie z plików i pisanie do plików, włączając w to standardowe wejście i wyjście.*
- *Limit pamięciowy: 16 MB (przy czym biblioteka zużywa nie więcej niż 1 MB pamięci).*

Ocenianie

Do sprawdzania rozwiązań będzie wykorzystywana biblioteka, która nie bazuje na z góry ustalonym rozwiązaniu, ale zawsze udziela niesprzecznych odpowiedzi. Twój program zostanie zaakceptowany tylko wtedy, gdy udzieli odpowiedzi wynikającej z uprzednio zebranych wypisów. (Czyli nie ma sensu zgadywać wyniku).

Do oceny rozwiązań zostanie wykorzystanych 25 testów. Jeśli Twoja odpowiedź w danym teście będzie poprawna i liczba wypisów wykonanych przez Twój program wyniesie K, uzyskasz:

- *4 punkty, jeśli $K \leqslant N/2 + 2$, a w przeciwnym razie*
- *2 punkty, jeśli $K \leqslant 2N/3$, a w przeciwnym razie*
- *1 punkt, jeśli $K \leqslant N - 3$, a w przeciwnym razie*
- *0 punktów.*

Możesz założyć, że system oceniający zmusi Twój program do wykonania co najmniej $N/3$ wypisów, zanim będziesz mógł podać poprawną odpowiedź.

Jakub Pawlewicz, Jakub Radoszewski
Tłumaczenie

CEOI 2012, dzień drugi, 11.07.2012

Sieć

Inżynierowie z pewnej firmy zaprojektowali sieć komunikacyjną, która składa się z węzłów i bezpośrednich jednokierunkowych połączeń między niektórymi parami węzłów. Mówimy, że węzeł q jest osiągalny z węzła p ścieżką składającą się z (różnych) węzłów $p_1, p_2, \ldots, p_k$, jeżeli $p = p_1$, $q = p_k$ oraz dla każdego $i = 1, \ldots, k-1$ istnieje połączenie z p_i do p_{i+1}. W sieci wyróżniamy węzeł główny r, który ma tę własność, że każdy inny węzeł p jest z niego osiągalny. Ponadto, dla każdej pary węzłów p i q istnieje **co najwyżej jedna** *ścieżka, za pomocą której q jest osiągalny z p.*

Wkrótce planowana jest modernizacja sieci, ale nie zdecydowano jeszcze, jak ma ona przebiegać. Można rozważyć przypisanie roli węzła głównego jakiemuś innemu węzłowi. W tym celu warto by wiedzieć, dla każdego węzła sieci, ile węzłów jest z niego osiągalnych. Innym pomysłem jest zdecentralizowanie sieci. Wtedy chcielibyśmy wiedzieć, ile nowych połączeń należałoby dodać, tak aby dla każdej pary węzłów p i q była dokładnie jedna ścieżka, za pomocą której q byłby osiągalny z p.

Zadanie

Napisz program, który wyznaczy liczbę węzłów osiągalnych z poszczególnych węzłów sieci (podzadanie A) oraz wyznaczy najmniejszą liczbę nowych połączeń, które należy dodać, aby każdy węzeł był osiągalny z każdego innego węzła dokładnie jedną ścieżką (podzadanie B). Należy również wypisać listę tych nowych połączeń.

Wejście

W pierwszym wierszu wejścia znajdują się trzy liczby całkowite: liczba węzłów N ($1 \leqslant N \leqslant 100\,000$), liczba połączeń M ($1 \leqslant M \leqslant 500\,000$) i numer węzła głównego r ($1 \leqslant r \leqslant N$). Węzły są ponumerowane od 1 do N. W następnych M wierszach znajduje się opis połączeń. Każdy z tych wierszy zawiera dwie liczby całkowite p i q oddzielone pojedynczym odstępem, oznaczające połączenie prowadzące z p do q.

Wyjście

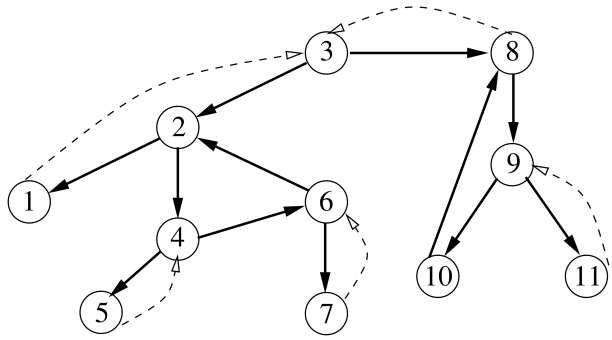
Pierwszy wiersz wyjścia powinien zawierać odpowiedź dla podzadania A: N liczb całkowitych pooddzielanych pojedynczymi odstępami; i-ta z tych liczb powinna oznaczać liczbę węzłów osiągalnych z węzła i (włączając i).

Pozostałe wiersze powinny zawierać odpowiedź dla podzadania B. Drugi wiersz wyjścia powinien zawierać jedną liczbę całkowitą K — najmniejszą liczbę połączeń, które trzeba dodać, aby uzyskać sieć, w której każdy węzeł jest osiągalny z każdego innego węzła dokładnie jedną ścieżką. Następne K wierszy powinno zawierać opis nowych połączeń: każdy z nich powinien zawierać dwie liczby całkowite u i v oddzielone pojedynczym odstępem, reprezentujące połączenie prowadzące z u do v. Jeżeli istnieje wiele rozwiązań, należy wypisać dowolne z nich.

Przykład

Dla danych wejściowych:

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



jednym z poprawnych wyników jest:

```
1 6 11 6 1 6 1 4 4 4 1
5
1 3
5 4
7 6
11 9
8 3
```

Ocenianie

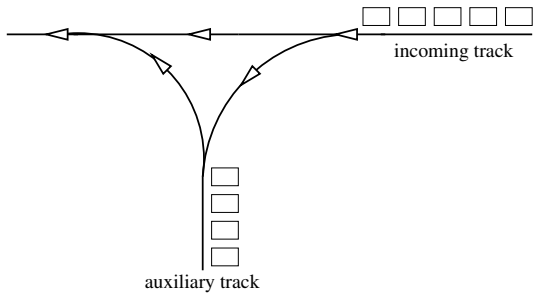
W 50% testów N jest nie większe niż 10 000.
Podzadanie A jest warte 40% punktów, podzadanie B jest warte pozostałe 60% punktów.
W przypadku rozwiązania tylko podzadania B, w pierwszym wierszu wyjścia należy wypisać N jakichkolwiek liczb całkowitych (z zakresu od 1 do N).

Jakub Pawlewicz, Jakub Radoszewski
Tłumaczenie

CEOI 2012, dzień drugi, 11.07.2012

Utylizacja odpadów

Pewna firma zajmuje się utylizacją odpadów przemysłowych. Odpady te przewozi się w specjalnych wagonach. Na torze wejściowym (ang. incoming track*) na przetworzenie oczekuje obecnie N wagonów z odpadami. W każdym wagonie znajduje się dokładnie jeden rodzaj odpadów. Są różne tryby utylizacji odpadów. W danym trybie możliwa jest utylizacja tylko niektórych rodzajów odpadów. Niestety, zmiana trybu jest bardzo kosztowna, tak więc firma zdecydowała się używać tylko jednego trybu dziennie. Aby przyspieszyć proces utylizacji, firma wykorzystuje dodatkowy boczny tor (ang.* auxiliary track*) — patrz rysunek. Dzięki temu, jeśli kolejny wagon z toru wejściowego zawiera odpady, których nie da się przetworzyć w danym trybie, wagon ten można przenieść na boczny tor, spychając wagony, które już się tam znajdują, w głąb toru. Jako kolejny wagon do przetworzenia wybrany zostaje zawsze pierwszy wagon z toru wejściowego albo z toru bocznego. Zauważ, że wagonu nie można nigdy przestawić z bocznego toru z powrotem na tor wejściowy. Firma zamierza w ciągu trzech najbliższych dni przetworzyć jak największą liczbę wagonów. Na końcu trzeciego dnia boczny tor musi być opróżniony.*



Zadanie

Napisz program, który wyznaczy tryby pracy na trzy najbliższe dni, które pozwolą przetworzyć możliwie największą liczbę wagonów i zakończyć przetwarzanie z pustym bocznym torem. Jeśli wszystkie wagony da się przetworzyć w ciągu mniej niż trzech dni, Twój program powinien zwrócić rozwiązanie wymagające najmniejszej możliwej liczby dni.

Wejście

Pierwszy wiersz wejścia zawiera trzy liczby całkowite: liczbę wagonów N ($1 \leqslant N \leqslant 20\,000$), liczbę rodzajów odpadów K ($1 \leqslant K \leqslant 1\,000$) i liczbę możliwych trybów pracy S ($1 \leqslant S \leqslant 1\,000$). Wagony numerujemy od 1 do N, rodzaje odpadów numerujemy od 1 do K, a tryby numerujemy od 1 do S. Kolejne S wierszy zawiera opisy trybów; i-ty z tych wierszy zawiera ciąg liczb całkowitych pooddzielanych pojedynczymi odstępami, zakończony zerem, opisujący rodzaje odpadów, które można przetworzyć w i-tym trybie. Ostatni wiersz wejścia zawiera ciąg N liczb całkowitych opisujący wagony: i-ta z tych liczb oznacza rodzaj

odpadów znajdujących się w wagonie numer i. Dla każdego rodzaju odpadów jest od jednego do dziesięciu trybów, w których można utylizować ten rodzaj odpadów.

Wyjście

Pierwszy wiersz wyjścia powinien zawierać jedną liczbę całkowitą — maksymalną liczbę wagonów, które można przetworzyć. Drugi wiersz powinien zawierać trzy liczby całkowite rozdzielone pojedynczymi odstępami — numery trybów pracy odpowiednio na pierwszy, drugi i trzeci dzień. Jeśli na przetworzenie wszystkich wagonów wystarczą dwa dni, trzecia z tych liczb powinna być zerem. Podobnie, jeśli wszystkie wagony można przetworzyć w ciągu jednego dnia, także druga z tych liczb powinna być zerem. Jeśli istnieje więcej niż jedno poprawne rozwiązanie, należy wypisać dowolne z nich.

Przykład

Dla danych wejściowych:

```
13 5 4
1 0
4 5 0
5 3 0
2 5 0
4 5 2 5 5 4 1 1 5 4 5 3 3
```

jednym z poprawnych wyników jest:

```
11
2 1 4
```

Ocenianie

W 50% testów N nie przekroczy 10 000 i S nie przekroczy 300.
Jeśli poprawny będzie tylko pierwszy wiersz wyjścia, Twój program otrzyma 40% punktów.

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] *XVII Olimpiada Informatyczna 2009/2010*. Warszawa, 2010.

[18] *XVIII Olimpiada Informatyczna 2010/2011*. Warszawa, 2011.

[19] A. V. Aho, J. E. Hopcroft, J. D. Ullman. *Projektowanie i analiza algorytmów komputerowych*. PWN, Warszawa, 1983.

[20] L. Banachowski, A. Kreczmar. *Elementy analizy algorytmów*. WNT, Warszawa, 1982.

[21] L. Banachowski, K. Diks, W. Rytter. *Algorytmy i struktury danych.* WNT, Warszawa, 1996.

[22] J. Bentley. *Perełki oprogramowania.* WNT, Warszawa, 1992.

[23] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Wprowadzenie do algorytmów.* WNT, Warszawa, 2004.

[24] M. de Berg, M. van Kreveld, M. Overmars. *Geometria obliczeniowa. Algorytmy i zastosowania.* WNT, Warszawa, 2007.

[25] R. L. Graham, D. E. Knuth, O. Patashnik. *Matematyka konkretna.* PWN, Warszawa, 1996.

[26] D. Harel. *Algorytmika. Rzecz o istocie informatyki.* WNT, Warszawa, 1992.

[27] D. E. Knuth. *Sztuka programowania.* WNT, Warszawa, 2002.

[28] W. Lipski. *Kombinatoryka dla programistów.* WNT, Warszawa, 2004.

[29] E. M. Reingold, J. Nievergelt, N. Deo. *Algorytmy kombinatoryczne.* WNT, Warszawa, 1985.

[30] K. A. Ross, C. R. B. Wright. *Matematyka dyskretna.* PWN, Warszawa, 1996.

[31] R. Sedgewick. *Algorytmy w C++. Grafy.* RM, 2003.

[32] S. S. Skiena, M. A. Revilla. *Wyzwania programistyczne.* WSiP, Warszawa, 2004.

[33] P. Stańczyk. *Algorytmika praktyczna. Nie tylko dla mistrzów.* PWN, Warszawa, 2009.

[34] M. M. Sysło. *Algorytmy.* WSiP, Warszawa, 1998.

[35] M. M. Sysło. *Piramidy, szyszki i inne konstrukcje algorytmiczne.* WSiP, Warszawa, 1998.

[36] R. J. Wilson. *Wprowadzenie do teorii grafów.* PWN, Warszawa, 1998.

[37] N. Wirth. *Algorytmy + struktury danych = programy.* WNT, Warszawa, 1999.

[38] *W poszukiwaniu wyzwań. Wybór zadań z konkursów programistycznych Uniwersytetu Warszawskiego.* Warszawa, 2012.

Niniejsza publikacja stanowi wyczerpujące źródło informacji o zawodach XIX Olimpiady Informatycznej, przeprowadzonych w roku szkolnym 2011/2012. 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 XXIV Międzynarodowej Olimpiady Informatycznej, XVIII Bałtyckiej Olimpiady Informatycznej oraz XIX Olimpiady Informatycznej Krajów Europy Środkowej.

XIX 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

ASSECO
POLAND

ISBN 978-83-930856-8-2