MINISTERSTWO EDUKACJI NARODOWEJ I SPORTU
INSTYTUT INFORMATYKI UNIWERSYTETU WROCŁAWSKIEGO
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

X OLIMPIADA INFORMATYCZNA
2002/2003

Olimpiada Informatyczna jest organizowana przy współudziale

WARSZAWA, 2003

MINISTERSTWO EDUKACJI NARODOWEJ I SPORTU
INSTYTUT INFORMATYKI UNIWERSYTETU WROCŁAWSKIEGO
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

X OLIMPIADA INFORMATYCZNA
2002/2003

WARSZAWA, 2003

Autorzy tekstów:

Adam Borowski
prof. dr hab. Zbigniew Czech
dr hab. Krzysztof Diks
Andrzej Gąsienica-Samek
Łukasz Kaiser
mgr Łukasz Kowalik
dr Marcin Kubica
mgr Marcin Mucha
Krzysztof Onak
Paweł Parys
Rafał Rusin
mgr Piotr Sankowski
Krzysztof Sikora
mgr Marcin Stefaniak
dr Krzysztof Stencel
mgr Tomasz Waleń
Paweł Wolff

Autorzy programów na dysku CD-ROM:

Michał Adamaszek
Wojciech Dudek
dr Marcin Kubica
Krzysztof Onak
Arkadiusz Paterek
Rafał Rusin
mgr Marcin Stefaniak
Paweł Wolff

Opracowanie i redakcja:

dr hab. Krzysztof Diks
Krzysztof Onak

Skład: Krzysztof Onak

Pozycja dotowana przez Ministerstwo Edukacji Narodowej i Sportu.

Druk książki został sfinansowany przez PROKOM SOFTWARE SA

ISBN 83-917700-4-4

Spis treści

Wstęp ... *5*

Sprawozdanie z przebiegu X Olimpiady Informatycznej ... *7*

Regulamin Olimpiady Informatycznej ... *29*

Zasady organizacji zawodów ... *37*

Zawody I stopnia — opracowania zadań **43**

Ciągi bez zająknięć ... *45*

Czekolada ... *51*

Liczby Przesmyków ... *59*

Płytki drukowane ... *63*

Przemytnicy ... *71*

Zawody II stopnia — opracowania zadań **77**

Mastermind II ... *79*

Autostrady ... *81*

Trójmian ... *87*

Kafelki ... *91*

Połączenia ... *97*

Zawody III stopnia — opracowania zadań **103**

Gra w dzielniki ... *105*

Skarb ... *109*

Sumy ... *115*

Kryształ ... *119*

Małpki ... *131*

Tasowanie 139

XIV Międzynarodowa Olimpiada Informatyczna — treści zadań **143**

Kije-samobije (Two rods) 145

Kłopotliwe żabsko (The troublesome frog) 149

Przystanki autobusowe (Bus terminals) 155

Podzielona Utopia (Utopia divided) 159

Szeregowanie zadań (Batch scheduling) 163

XOR 165

IX Bałtycka Olimpiada Informatyczna — treści zadań **169**

Alokacja rejestrów (Regs) 171

Beczka (Barrel) 175

Gangi (Gangs) 177

Klejnoty (Gems) 179

Lampy (Lamps) 181

Tablica (Table) 183

X Olimpiada Informatyczna Europy Środkowej — treści zadań **185**

Kwadrat (Square) 187

Naszyjnik (The Pearl Necklace) 189

Rejestr przesuwający (Shift Register) 193

Wieże Hanoi (Towers of Hanoi) 195

Wycieczka (Trip) 197

Wyścig (The Race) 199

Literatura 201

Krzysztof Diks

Wstęp

Drogi Czytelniku!

Już po raz dziesiąty masz okazję wziąć do ręki „niebieską książeczkę”, a to oznacza, że Olimpiada Informatyczna ma już 10 lat. W tym szczególnym momencie pragnę powiedzieć kilka słów o osobach, które przyczyniły się szczególnie do powstania i rozwoju Olimpiady. Należą do nich profesorowie Stanisław Waligórski, Maciej Sysło i Jan Madey, dr Andrzej Walat oraz kierownik organizacyjny Olimpiady — Tadeusz Kuran. To właśnie ich śmiałej decyzji zawdzięczamy, że w szranki olimpijskie stanęło dotychczas blisko 8000 uczniów szkół średnich, którzy już wkrótce będą decydowali o przyszłości informatyki w naszym kraju. Olimpiada Informatyczna należy do najdynamiczniej rozwijających się olimpiad przedmiotowych w Polsce. Co roku młodzi Polacy, laureaci zawodów krajowych, zdobywają na zawodach międzynarodowych czołowe pozycje. Oto tylko niektóre spektakularne sukcesy z ostatnich lat:

- trzy złote medale na olimpiadach międzynarodowych Andrzeja Gąsienicy-Samka,
- drugie miejsce w świecie i złoty medal Pawła Parysa na Międzynarodowej Olimpiadzie Informatycznej w roku 2003,
- czterech Polaków w pierwszej piątce tegorocznej Bałtyckiej Olimpiady Informatycznej i pierwsze miejsce Bartka Walczaka,
- trzech Polaków w pierwszej czwórce tegorocznej Olimpiady Krajów Europy Środkowej i pierwsze miejsce Bartka Walczaka, z maksymalną możliwą do zdobycia liczbą punktów.

To tylko przykłady sukcesów, ale chyba ważniejsze jest to, że wielu naszych olimpijczyków nadal utrzymuje ścisły związek z Olimpiadą. Pomagają oni w organizacji Olimpiady, są autorami oprogramowania olimpijskiego, przygotowują zadania (są zarówno pomysłodawcami zadań, jak i autorami rozwiązań wzorcowych), dzielą się swoją wiedzą i doświadczeniami z młodszymi kolegami na obozach naukowych. Chciałbym tu szczególnie wyróżnić za lata współpracy Tomka Czajkę, Andrzeja Gąsienicę-Samka, Grzegorza Jakackiego, Marcina Muchę, Krzysztofa Onaka, Piotra Sankowskiego i Tomka Walenia. Nie mniejsze podziękowania należą się też wszystkim pozostałym jurorom, którzy przyczynili się do sukcesu naszych zawodów. Na koniec tej części chciałbym także podziękować dr Krzysztofowi Stenclowi i dr Marcinowi Kubicy za merytoryczną opiekę nad pracą młodszych kolegów jurorów.

Wiele sukcesów Olimpiady to także wynik pracy i zaangażowania wspaniałych nauczycieli. Chciałbym tu wymienić troje z nich, których osiągnięcia są naprawdę spektakularne: Andrzeja Dyrka z Krakowa, Iwonę Waszkiewicz z Bydgoszczy i Ryszarda Szubartowskiego z Gdyni.

Do sukcesów Olimpiady niewątpliwie przyczynili się też nasi sponsorzy. Dzięki nim zawodnicy mają bardzo dobre warunki do rywalizacji podczas zawodów II i III stopnia, otrzymują wartościowe nagrody, jak i mają możliwość wzięcia udziału w olimpijskich obozach naukowych. Tutaj szczególnie gorące słowa podziękowania kieruję do współorganizatora Olimpiady — firmy PROKOM SOFTWARE S.A. i jej Prezesa pana Ryszarda Krauze. Chciałbym także podziękować panu Krzysztofowi Koszewskiemu, prezesowi firmy CombiData z Sopotu, za to że od lat gości finalistów Olimpiady. Moje myśli biegną też do od lat wypróbowanych przyjaciół Olimpiady: pani Elżbiety Beuermann z Wydawnictw Naukowo-Technicznych i pana Romana Dawidsona z Ogólnopolskiej Fundacji Edukacji Komputerowej.

Olimpiada Informatyczna współpracuje też z wieloma uniwersytetami i szkołami wyższymi w kraju. W szczególności chciałbym tu wyróżnić Instytut Informatyki Uniwersytetu Wrocławskiego, Wydział Matematyki i Informatyki Uniwersytetu im. Mikołaja Kopernika w Toruniu, Politechnikę Śląską i Uniwersytet Śląski, Polsko-Japońską Wyższą Szkołę Technik Komputerowych w Warszawie, Instytut Informatyki Uniwersytetu Jagiellońskiego, Wyższą Szkołę Informatyki i Zarządzania w Rzeszowie i Akademię Górniczo-Hutniczą. Na koniec nie mogę ominąć mojej macierzystej instytucji — Instytutu Informatyki Uniwersytetu Warszawskiego, który od początku istnienia Olimpiady stanowi jej merytoryczne zaplecze.

Wkraczamy w kolejne dziesięciolecie. Jestem przekonany, że będzie one jeszcze bardziej udane niż dotychczasowe 10 lat.

Krzysztof Diks

Sprawozdanie z przebiegu X Olimpiady Informatycznej 2002/2003

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.

ORGANIZACJA ZAWODÓW

W roku szkolnym 2002/2003 odbyły się zawody X Olimpiady Informatycznej. 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.

23 września 2003 r. rozesłano plakaty zawierające zasady organizacji zawodów I stopnia oraz zestaw 5 zadań konkursowych do 3223 szkół i zespołów szkół młodzieżowych ponadpodstawowych oraz do wszystkich kuratorów i koordynatorów edukacji informatycznej. Zawody I stopnia rozpoczęły się dnia 7 października 2002 r. Ostatecznym terminem nadsyłania prac konkursowych był 4 listopada 2002 roku.

Zawody II i III stopnia były dwudniowymi sesjami stacjonarnymi, poprzedzonymi jednodniowymi sesjami próbnymi. Zawody II stopnia odbyły się w pięciu okręgach: Warszawie, Wrocławiu, Toruniu, Katowicach i Krakowie oraz w Sopocie i Rzeszowie, w dniach 11–13.02.2003 r., natomiast zawody III stopnia odbyły się w ośrodku firmy Combidata Poland S.A. w Sopocie, w dniach 8–12.04.2003 r.

Uroczystość zakończenia X Olimpiady Informatycznej odbyła się w dniu 12.04.2003 r. w Sali Posiedzeń Urzędu Miasta w Sopocie z udziałem Wiceministra Edukacji Narodowej i Sportu Adama Giersza.

SKŁAD OSOBOWY KOMITETÓW OLIMPIADY INFORMATYCZNEJ

Komitet Główny

przewodniczący:
    dr hab. Krzysztof Diks, prof. UW (Uniwersytet Warszawski)
zastępcy przewodniczącego:
    prof. dr hab. Maciej M. Sysło (Uniwersytet Wrocławski)
    prof. dr hab. Paweł Idziak (Uniwersytet Jagielloński)
sekretarz naukowy:
    dr Marcin Kubica (Uniwersytet Warszawski)

kierownik Jury:
dr Krzysztof Stencel (Uniwersytet Warszawski)
kierownik organizacyjny:
Tadeusz Kuran (OEIiZK)
członkowie:
prof. dr hab. Zbigniew Czech (Politechnika Śląska)
mgr Jerzy Dałek (Ministerstwo Edukacji Narodowej i Sportu)
dr Przemysława Kanarek (Uniwersytet Wrocławski)
dr hab. Krzysztof Loryś, prof. UWr (Uniwersytet Wrocławski)
prof. dr hab. Jan Madey (Uniwersytet Warszawski)
prof. dr hab. Wojciech Rytter (Uniwersytet Warszawski)
mgr Krzysztof J. Święcicki (Ministerstwo Edukacji Narodowej i Sportu)
dr Maciej Ślusarek (Uniwersytet Jagielloński)
dr hab. inż. Stanisław Waligórski, prof. UW (Uniwersytet Warszawski)
dr Mirosława Skowrońska (Uniwersytet Mikołaja Kopernika)
dr Andrzej Walat (OEIiZK)
mgr Tomasz Waleń (Uniwersytet Warszawski)
sekretarz Komitetu Głównego:
Monika Kozłowska-Zając (OEIiZK)

Komitet Główny mieści się w Warszawie przy ul. Nowogrodzkiej 73, w Ośrodku Edukacji Informatycznej i Zastosowań Komputerów.

Komitet Główny odbył 5 posiedzeń, ponadto 24 stycznia 2003 r. przeprowadzono seminarium przygotowujące organizację zawodów II stopnia.

Komitety okręgowe

Komitet Okręgowy w Warszawie
przewodniczący:
dr Adam Malinowski (Uniwersytet Warszawski)
sekretarz:
Monika Kozłowska-Zając (OEIiZK)
członkowie:
dr Marcin Kubica (Uniwersytet Warszawski)
dr Andrzej Walat (OEIiZK)

Komitet Okręgowy mieści się w Warszawie przy ul. Nowogrodzkiej 73, w Ośrodku Edukacji Informatycznej i Zastosowań Komputerów.

Komitet Okręgowy we Wrocławiu
przewodniczący:
dr hab. Krzysztof Loryś, prof. UWr (Uniwersytet Wrocławski)
zastępca przewodniczącego:
dr Helena Krupicka (Uniwersytet Wrocławski)
sekretarz:
inż. Maria Woźniak (Uniwersytet Wrocławski)
członkowie:
mgr Jacek Jagiełło (Uniwersytet Wrocławski)

dr Tomasz Jurdziński (Uniwersytet Wrocławski)
dr Przemysława Kanarek (Uniwersytet Wrocławski)
dr Witold Karczewski (Uniwersytet Wrocławski)

Siedzibą Komitetu Okręgowego jest Instytut Informatyki Uniwersytetu Wrocławskiego we Wrocławiu, ul. Przesmyckiego 20.

Komitet Okręgowy w Toruniu

przewodniczący:
dr hab. Grzegorz Jarzembski, prof. UMK (Uniwersytet Mikołaja Kopernika)
zastępca przewodniczącego:
dr Mirosława Skowrońska (Uniwersytet Mikołaja Kopernika)
sekretarz:
dr Barbara Klunder (Uniwersytet Mikołaja Kopernika)
członkowie:
mgr Anna Kwiatkowska (IV Liceum Ogólnokształcące w Toruniu)
dr Krzysztof Skowronek (V Liceum Ogólnokształcące w Toruniu)

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

Górnośląski Komitet Okręgowy

przewodniczący:
prof. dr hab. Zbigniew Czech (Politechnika Śląska)
zastępca przewodniczącego:
mgr inż. Sebastian Deorowicz (Politechnika Śląska)
sekretarz:
mgr inż. Adam Skórczyński (Politechnika Śląska)
członkowie:
dr inż. Mariusz Boryczka (Uniwersytet Śląski)
mgr inż. Marcin Ciura (Politechnika Śląska)
mgr inż. Marcin Szołtysek (Politechnika Śląska)
mgr Jacek Widuch (Politechnika Śląska)
mgr Wojciech Wieczorek (Uniwersytet Śląski)

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

Komitet Okręgowy w Krakowie

przewodniczący:
prof. dr hab. Paweł Idziak (Uniwersytet Jagielloński)
zastępca przewodniczącego:
dr Maciej Ślusarek (Uniwersytet Jagielloński)
sekretarz:
mgr Edward Szczypka (Uniwersytet Jagielloński)
członkowie:
mgr Henryk Białek (Kuratorium Oświaty w Krakowie)
dr inż. Janusz Majewski (Akademia Górniczo-Hutnicza)

Siedzibą Komitetu Okręgowego w Krakowie jest Instytut Informatyki Uniwersytetu Jagiellońskiego, ul. Nawojki 11 w Krakowie.

Jury Olimpiady Informatycznej

W pracach Jury, które nadzorował Krzysztof Diks, a którymi kierował Krzysztof Stencel, brali udział doktoranci i studenci Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego oraz Wydziału Matematyki i Informatyki Uniwersytetu Wrocławskiego:

Michał Adamaszek
Jarosław Byrka
Tomasz Czajka
Wojciech Dudek
Andrzej Gąsienica-Samek
mgr Łukasz Kowalik
Tomasz Malesiński
mgr Marcin Mucha
Krzysztof Onak
Arkadiusz Paterek
Jakub Pawlewicz
mgr Remigiusz Różycki
Rafał Rusin
mgr Piotr Sankowski
Krzysztof Sikora
mgr Marcin Stefaniak
Paweł Wolff

ZAWODY I STOPNIA

W X Olimpiadzie Informatycznej wzięło udział 849 zawodników. Decyzją Komitetu Głównego do zawodów zostało dopuszczonych 43 uczniów gimnazjum.

- Gimnazjum nr 24, Gdynia: 17 uczniów
- Gimnazjum nr 50, Bydgoszcz: 4
- Gimnazjum nr 13, Wrocław: 2
- Gimnazjum nr 16, Szczecin: 2
- Publiczne Gimnazjum nr 52 Ojców Pijarów, Kraków: 2
- Gimnazjum, Zambrów: 1
- Gimnazjum i Liceum Akademickie, Toruń: 1
- Gimnazjum im. św. Jadwigi Królowej, Kielce: 1
- Gimnazjum Niepubliczne nr 12, Warszawa: 1
- Gimnazjum nr 1, Gdynia: 1
- Gimnazjum nr 1 w Krzepicach, Krzepice: 1

- Gimnazjum nr 2, Żywiec: 1
- Gimnazjum nr 8, Elbląg: 1
- Gimnazjum nr 9, Bielsko-Biała: 1
- Gimnazjum nr 13, Szczecin: 1
- Gimnazjum nr 13, Warszawa: 1
- Gimnazjum nr 34 w Łodzi, Łódź: 1
- Gimnazjum nr 50, Poznań: 1
- Kolegium Szkół Prywatnych, Kielce: 1
- Publiczne Gimnazjum nr 1 im. Wł. Jagiełły, Choszczno: 1
- Publiczne Gimnazjum nr 5, Kluczbork: 1

Kolejność województw pod względem liczby uczestników była następująca:

mazowieckie	131
pomorskie	109
małopolskie	104
śląskie	94
kujawsko-pomorskie	81
dolnośląskie	54
zachodniopomorskie	51
łódzkie	38
wielkopolskie	37
lubelskie	29
podkarpackie	29
podlaskie	27
lubuskie	20
świętokrzyskie	20
opolskie	16
warmińsko-mazurskie	9

W zawodach I stopnia najliczniej reprezentowane były szkoły:

III LO im. Marynarki Wojennej RP, Gdynia	47 uczniów
V LO im. A. Witkowskiego, Kraków	46
XIV LO im. St. Staszica, Warszawa	31
VI LO im. J. i J. Śniadeckich, Bydgoszcz	21
Gimnazjum nr 24, Gdynia	17
VIII LO im. A. Mickiewicza, Poznań	15

VI LO im. W. Sierpińskiego, Gdynia	12
XIII LO, Szczecin	11
I LO im. Adama Mickiewicza, Białystok	10
IV LO im. T. Kościuszki, Toruń	9
XXVII LO im. T. Czackiego, Warszawa	9
V LO im. A. Struga, Gliwice	7
V LO im. Księcia J. Poniatowskiego, Warszawa	7
VIII LO im. Marii Skłodowskiej, Katowice	7
Z. S. O. nr 2 im. Hugona Kołłątaja, Wałbrzych	7
I LO im. A. Mickiewicza, Góra	6
II LO im. M. Konopnickiej, Opole	6
V Liceum Ogólnokształcące, Bielsko-Biała	6
X LO im. Królowej Jadwigi, Warszawa	6
Katolickie LO Ojców Pijarów, Kraków	6
I LO im. M. Kopernika, Łódź	5
IV LO im. M. Kopernika, Rzeszów	5
VI LO im. J. Kochanowskiego, Radom	5
VIII LO im. Władysława IV, Warszawa	5
XII LO, Łódź	5
XIV LO im. Polonii Belgijskiej, Wrocław	5
Gimnazjum nr 50, Bydgoszcz	4
I LO im. A. Osuchowskiego, Cieszyn	4
I LO im. St. Dubois, Koszalin	4
II LO im. R. Traugutta, Częstochowa	4
IV LO im. Kazimierza Wielkiego, Bydgoszcz	4
LO im. Bogusława X, Białogard	4
I LO im. C. K. Norwida, Bydgoszcz	3
I LO im. Jana Kasprowicza, Inowrocław	3
I LO im. M. Kopernika, Gdańsk	3
I LO im. S. Czarnieckiego, Chełm	3
II LO, Gorzów Wielkopolski	3
II LO im. Króla Jana III Sobieskiego, Grudziądz	3
VI LO im. J. Słowackiego, Kielce	3
VIII Liceum Ogólnokształcące, Bydgoszcz	3

Najliczniej reprezentowane były miasta:

Warszawa	104
Gdynia	79
Kraków	74
Bydgoszcz	39
Szczecin	26
Wrocław	22
Poznań	20
Łódź	18
Lublin	17
Toruń	17

Białystok	14
Częstochowa	13
Katowice	12
Rzeszów	11
Bielsko-Biała	10
Kielce	10
Opole	10
Wałbrzych	10
Gdańsk	9
Gliwice	9
Radom	8
Koszalin	7
Wodzisław Śląski	7
Zielona Góra	7
Dąbrowa Górnicza	6
Góra	6
Tarnów	6
Tychy	6
Inowrocław	5
Nowy Sącz	5
Ostrowiec Świętokrzyski	5

Zawodnicy uczęszczali do następujących klas:

do klasy I	gimnazjum	8 zawodników
do klasy II	gimnazjum	9
do klasy III	gimnazjum	23
do klasy I	szkoły średniej	138
do klasy II	szkoły średniej	9
do klasy III	szkoły średniej	300
do klasy IV	szkoły średniej	330
do klasy V	szkoły średniej	18

14 zawodników nie podało informacji o tym, do której klasy uczęszczają.

W zawodach I stopnia zawodnicy mieli do rozwiązania pięć zadań: „Ciągi bez zająknięć", „Czekolada", „Liczby Przesmyków", „Płytki drukowane" i „Przemytnicy".

W wyniku zastosowania procedury sprawdzającej wykryto niesamodzielne rozwiązania, które, zgodnie z Regulaminem, nie zostały wzięte pod uwagę.

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

- Ciągi bez zająknięć

	liczba zawodników	czyli
100 pkt.	151	17,8%
75–99 pkt.	49	5,8%
50–74 pkt.	68	8%
1–49 pkt.	269	31,7%
0 pkt.	312	36,7%

- Czekolada

	liczba zawodników	czyli
100 pkt.	587	69,1%
75–99 pkt.	26	3,1%
50–74 pkt.	82	9,7%
1–49 pkt.	78	9,2%
0 pkt.	76	8,9%

- Liczby Przesmyków

	liczba zawodników	czyli
100 pkt.	109	12,8%
75–99 pkt.	35	4,1%
50–74 pkt.	25	3,0%
1–49 pkt.	114	13,4%
0 pkt.	566	66,7%

- Płytki drukowane

	liczba zawodników	czyli
100 pkt.	67	7,9%
75–99 pkt.	2	0,2%
50–74 pkt.	13	1,5%
1–49 pkt.	326	38,5%
0 pkt.	441	51,9%

- Przemytnicy

	liczba zawodników	czyli
100 pkt.	138	16,3%
75–99 pkt.	105	12,4%
50–74 pkt.	124	14,6%
1–49 pkt.	160	18,8%
0 pkt.	322	37,9%

W sumie za wszystkie 5 zadań konkursowych:

SUMA	liczba zawodników	czyli
500 pkt.	23	2,7%
375–499 pkt.	92	10,9%
250–374 pkt.	148	17,4%
1–249 pkt.	546	64,3%
0 pkt.	40	4,7%

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

ZAWODY II STOPNIA

Do zawodów II stopnia zakwalifikowano 307 zawodników, którzy osiągnęli w zawodach I stopnia wynik nie mniejszy niż 202 pkt.

Jeden zawodnik nie stawił się na zawody. W zawodach II stopnia uczestniczyło więc 306 zawodników.

Zawody II stopnia odbyły się w dniach 11–13 lutego 2003 r. w pięciu stałych okręgach oraz w Sopocie i Rzeszowie:

- w Toruniu — 31 zawodników z następujących województw:
  - kujawsko-pomorskie (28)
  - mazowieckie (2)
  - warmińsko-mazurskie (1)
- we Wrocławiu — 54 zawodników z następujących województw:
  - dolnośląskie (19)
  - lubuskie (1)
  - opolskie (5)
  - śląskie (5)
  - wielkopolskie (13)
  - zachodniopomorskie (11)
- w Warszawie — 79 zawodników z następujących województw:
  - łódzkie (7)
  - mazowieckie (57)
  - podlaskie (11)
  - świętokrzyskie (4)
- w Krakowie — 46 zawodników z następujących województw:
  - małopolskie (46)
- w Gliwicach — 29 zawodników z następujących województw:
  - małopolskie (3)
  - śląskie (26)
- w Sopocie — 45 zawodników z następujących województw:
  - pomorskie (43)

- zachodniopomorskie (2)

- w Rzeszowie — 22 zawodników z następujących województw:
  - lubelskie (5)
  - małopolskie (3)
  - podkarpackie (14)

W zawodach II stopnia najliczniej reprezentowane były szkoły:

V LO im. A. Witkowskiego, Kraków	35 zawodników
III LO im. Marynarki Wojennej RP, Gdynia	23
XIV LO im. St. Staszica, Warszawa	22
VI LO im. J. i J. Śniadeckich, Bydgoszcz	11
VI Liceum Ogólnokształcące, Gdynia	10
I LO im. A. Mickiewicza, Białystok	8
VIII LO im. A. Mickiewicza, Poznań	8
XIII LO, Szczecin	7
IV LO im. T. Kościuszki, Toruń	6
VIII LO im. M. Skłodowskiej, Katowice	6
Z. S. O. nr 2 im. H. Kołłątaja, Wałbrzych	6
IV LO im. M. Kopernika, Rzeszów	5
V LO im. Księcia J. Poniatowskiego, Warszawa	5
XXVII LO im. T. Czackiego, Warszawa	5
Gimnazjum nr 24, Gdynia	4
I LO im. M. Kopernika, Łódź	4
V LO, Bielsko-Biała	4
XIV LO im. Polonii Belgijskiej, Wrocław	4
I LO im. J. Kasprowicza, Inowrocław	3
I Społeczne LO, Warszawa	3
II LO im. R. Traugutta, Częstochowa	3
VI LO im. J. Kochanowskiego, Radom	3
L. LO im. Ruy Barbosa, Warszawa	3
LO Przymierza Rodzin im. Jana Pawła II, Warszawa	3

Najliczniej reprezentowane były miasta:

Warszawa	53
Kraków	46
Gdynia	37
Bydgoszcz	14
Poznań	10
Wrocław	10
Białystok	8
Katowice	8
Szczecin	8
Toruń	8

Rzeszów	7
Bielsko-Biała	6
Łódź	6
Wałbrzych	6
Częstochowa	5
Inowrocław	4
Opole	4
Gdańsk	3
Gliwice	3
Kielce	3
Radom	3
Wodzisław Śląski	3

W dniu 11 lutego odbyła się sesja próbna, na której zawodnicy rozwiązywali, nie liczące się do ogólnej klasyfikacji, zadanie „Mastermind II". W dniach konkursowych zawodnicy rozwiązywali zadania: „Autostrady", „Trójmian", „Kafelki" oraz „Połączenia", każde oceniane maksymalnie po 100 punktów.

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

- Mastermind II

	liczba zawodników	czyli
100 pkt.	77	25,2%
75–99 pkt.	9	2,9%
50–74 pkt.	25	8,2%
1–49 pkt.	71	23,2%
0 pkt.	124	40,5%

- Autostrady

	liczba zawodników	czyli
100 pkt.	1	0,3%
75–99 pkt.	3	1%
50–74 pkt.	43	14,1%
1–49 pkt.	142	46,4%
0 pkt.	117	38,2%

- Trójmian

	liczba zawodników	czyli
100 pkt.	2	0,7%
75–99 pkt.	0	0%
50–74 pkt.	1	0,3%
1–49 pkt.	119	38,9%
0 pkt.	184	60,1%

- Kafelki

	liczba zawodników	czyli
100 pkt.	7	2,3%
75–99 pkt.	8	2,6%
50–74 pkt.	38	12,5%
1–49 pkt.	132	43,1%
0 pkt.	121	39,5%

- Połączenia

	liczba zawodników	czyli
100 pkt.	9	2,9%
75–99 pkt.	10	3,3%
50–74 pkt.	15	4,9%
1–49 pkt.	158	51,6%
0 pkt.	114	37,3%

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

SUMA	liczba zawodników	czyli
400 pkt.	0	0%
300–399 pkt.	0	0%
200–299 pkt.	11	3,6%
1–199 pkt.	274	89,5%
0 pkt.	21	6,9%

Wszystkim zawodnikom przesłano informację o uzyskanych wynikach.

W zawodach II stopnia wyróżniono 32 zawodników, którzy uzyskali wynik powyżej 100 pkt. Zawodnikom tym Komitet Główny wystawił stosowne zaświadczenia.

ZAWODY III STOPNIA

Zawody III stopnia odbyły się w ośrodku firmy Combidata Poland S.A. w Sopocie w dniach od 8 do 12 kwietnia 2003 r.

W zawodach III stopnia wzięło udział 44 najlepszych uczestników zawodów II stopnia, którzy uzyskali wynik nie mniejszy niż 128 pkt. Zawodnicy pochodzili z następujących województw:

pomorskie	11
mazowieckie	8
małopolskie	6
śląskie	4
wielkopolskie	3
dolnośląskie	2
kujawsko-pomorskie	2
podkarpackie	2
podlaskie	2

zachodniopomorskie	2
łódzkie	1
świętokrzyskie	1

Niżej wymienione szkoły miały w finale więcej niż jednego zawodnika:

III LO im. Marynarki Wojennej RP, Gdynia	8 zawodników
V LO im. A. Witkowskiego, Kraków	5
XIV LO im. St. Staszica, Warszawa	5
VIII LO im. A. Mickiewicza, Poznań	3
V LO im. Księcia J. Poniatowskiego, Warszawa	2
VI Liceum Ogólnokształcące, Gdynia	2
VIII LO im. M. Skłodowskiej, Katowice	2
XIII LO, Szczecin	2

8 kwietnia odbyła się sesja próbna, na której zawodnicy rozwiązywali, nie liczące się do ogólnej klasyfikacji, zadanie „Gra w dzielniki". W dniach konkursowych zawodnicy rozwiązywali zadania: „Skarb", „Sumy", „Kryształ", „Małpki" i „Tasowanie", 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:

- Gra w dzielniki

	liczba zawodników	czyli
100 pkt.	23	52,2%
75–99 pkt.	9	20,5%
50–74 pkt.	4	9,1%
1–49 pkt.	7	15,9%
0 pkt.	1	2,3%

- Skarb

	liczba zawodników	czyli
100 pkt.	1	2,3%
75–99 pkt.	3	6,8%
50–74 pkt.	18	40,9%
1–49 pkt.	16	36,4%
0 pkt.	6	13,6%

- Sumy

	liczba zawodników	czyli
100 pkt.	3	6,8%
75–99 pkt.	1	2,3%
50–74 pkt.	17	38,6%
1–49 pkt.	19	43,2%
0 pkt.	4	9,1%

- Kryształ

	liczba zawodników	czyli
100 pkt.	0	0%
75–99 pkt.	0	0%
50–74 pkt.	0	0%
1–49 pkt.	24	54,6%
0 pkt.	20	45,4%

- Małpki

	liczba zawodników	czyli
100 pkt.	8	18,2%
75–99 pkt.	0	0%
50–74 pkt.	1	2,3%
1–49 pkt.	14	31,8%
0 pkt.	21	47,7%

- Tasowanie

	liczba zawodników	czyli
100 pkt.	2	4,6%
75–99 pkt.	0	0%
50–74 pkt.	0	0%
1–49 pkt.	14	31,8%
0 pkt.	28	63,6%

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

SUMA	liczba zawodników	czyli
500 pkt.	0	0%
375–499 pkt.	1	2,3%
250–374 pkt.	3	6,8%
125–249 pkt.	15	34,1%
1–124 pkt.	24	54,5%
0 pkt.	1	2,3%

W dniu 12 kwietnia 2003 roku, w Sali Posiedzeń Urzędu Miasta w Sopocie, ogłoszono wyniki finału X Olimpiady Informatycznej 2002/2003 i rozdano nagrody ufundowane przez: PROKOM Software S.A., Ogólnopolską Fundację Edukacji Komputerowej, Wydawnictwa Naukowo-Techniczne i Olimpiadę Informatyczną. Laureaci I, II i III miejsca otrzymali odpowiednio złote, srebrne i brązowe medale. Poniżej zestawiono listę wszystkich laureatów i finalistów:

(1) Marcin Michalski, III LO im. Marynarki Wojennej RP w Gdyni, laureat I miejsca, 410 pkt. (puchar Prezydenta RP dla najlepszego młodego informatyka roku; notebook — PROKOM; roczny abonament na książki — WNT)

(2) Bartosz Walczak, V LO im. Augusta Witkowskiego w Krakowie, laureat I miejsca, 310 pkt. (notebook — PROKOM)

(3) Filip Wolski, Gimnazjum nr 24 w Gdyni, laureat I miejsca, 280 pkt. (notebook — PROKOM)

(4) Michał Brzozowski, III LO im. Marynarki Wojennej RP w Gdyni, laureat II miejsca, 250 pkt. (notebook — PROKOM)

(5) Andrzej Chodor, IV LO im. Hanki Sawickiej w Kielcach, laureat II miejsca, 243 pkt. (drukarka laserowa — PROKOM)

(6) Łukasz Krupa, LO im. Króla Wł. Jagiełły w Dębicy, laureat II miejsca, 220 pkt. (drukarka laserowa — PROKOM)

(7) Michał Jaszczyk, XIII LO w Szczecinie, laureat II miejsca, 220 pkt. (drukarka laserowa — PROKOM)

(8) Szymon Acedański, VIII LO im. Marii Skłodowskiej-Curie w Katowicach, laureat II miejsca, 213 pkt. (drukarka laserowa — PROKOM)

(9) Michał Bartoszkiewicz, XIV LO im. Polonii Belgijskiej we Wrocławiu, laureat III miejsca, 194 pkt. (drukarka laserowa — PROKOM)

(10) Bartłomiej Romański, XIV LO im. Stanisława Staszica w Warszawie, laureat III miejsca, 180 pkt. (drukarka laserowa — PROKOM)

(11) Marcin Wielgus, I LO im. B. Nowodworskiego w Krakowie, laureat III miejsca, 170 pkt. (drukarka laserowa — PROKOM)

(12) Adam Narkiewicz, XIV LO im. Stanisława Staszica w Warszawie, laureat III miejsca, 160 pkt. (drukarka laserowa — PROKOM)

(13) Marcin Pilipczuk, XIV LO im. Stanisława Staszica w Warszawie, laureat III miejsca, 150 pkt. (drukarka atramentowa — PROKOM)

(14) Paweł Gosztyła, I LO im. Leona Kruczkowskiego w Tychach, laureat III miejsca, 150 pkt. (drukarka atramentowa — PROKOM)

(15) Andrzej Grzywocz, III LO im. Marynarki Wojennej RP w Gdyni, finalista z wyróżnieniem, 146 pkt. (drukarka atramentowa — PROKOM)

(16) Dawid Sieradzki, VIII LO im. Adama Mickiewicza w Poznaniu, finalista z wyróżnieniem, 142 pkt. (drukarka atramentowa — PROKOM)

(17) Piotr Hołubowicz, VIII LO im. Adama Mickiewicza w Poznaniu, finalista z wyróżnieniem, 140 pkt (drukarka atramentowa — PROKOM).

(18) Jan Prażuch, V LO im. Augusta Witkowskiego w Krakowie, finalista z wyróżnieniem, 136 pkt. (drukarka atramentowa — PROKOM)

(19) Jakub Łącki, III LO im. Marynarki Wojennej RP w Gdyni, finalista z wyróżnieniem, 135 pkt. (drukarka atramentowa — PROKOM)

(20) Szymon Wąsik, VIII LO im. Adama Mickiewicza w Poznaniu, finalista z wyróżnieniem, 120 pkt. (drukarka atramentowa — PROKOM)

(21) Adam Slaski, III LO im. Marynarki Wojennej RP w Gdyni, finalista z wyróżnieniem, 120 pkt. (drukarka atramentowa — PROKOM)

(22) Krzysztof Ślusarski, III LO im. Marynarki Wojennej RP w Gdyni, finalista z wyróżnieniem, 120 pkt. (drukarka atramentowa — PROKOM)

(23) Jan Kaczmarczyk, V LO im. Augusta Witkowskiego w Krakowie, finalista z wyróżnieniem, 120 pkt. (drukarka atramentowa — PROKOM)

(24) Tomasz Uliński, III LO im. Marynarki Wojennej RP w Gdyni, finalista z wyróżnieniem, 120 pkt. (drukarka atramentowa — PROKOM)

Lista pozostałych finalistów w kolejności alfabetycznej:

- Jarosław Apelski, XIV LO im. Stanisława Staszica w Warszawie
- Piotr Danilewski, V LO im. Augusta Witkowskiego w Krakowie
- Piotr Drajski, VI Liceum Ogólnokształcące w Gdyni
- Adrian Galewski, Zespół Szkół Technicznych w Wodzisławiu Śląskim
- Adam Iwanicki, I LO im. Marii Konopnickiej w Suwałkach
- Maciej Kalbarczyk, L. LO im. Ruy Barbosa w Warszawie
- Damian Klata, XXXI LO w Łódzi
- Tomasz Kłos, III LO im. Marynarki Wojennej RP w Gdyni
- Piotr Kucharski, V LO im. Ksiecia J. Poniatowskiego w Warszawie
- Tomasz Łakota,VIII LO im. Marii Skłodowskiej-Curie w Katowicach
- Piotr Mikulski, XIII LO w Szczecinie
- Adam Nowacki, I LO im. Adama Mickiewicza w Białymstoku
- Jarosław Osmański, IV LO im. Tadeusza Kościuszki w Toruniu
- Paweł Papis, VI Liceum Ogólnokształcące w Gdyni
- Krzysztof Pawłowski, XIV LO im. Stanisława Staszica w Warszawie
- Aleksander Piotrowski, VI LO im. J. i J. Śniadeckich w Bydgoszczy
- Maciej Popowicz, II LO im. Piastów Śląskich we Wrocławiu
- Krzysztof Porczyk, V LO im. Księcia J. Poniatowskiego w Warszawie
- Maciej Taczuk, V LO im. Augusta Witkowskiego w Krakowie
- Rafał Wojtak, Zespół Szkół Elektronicznych w Rzeszowie

Wszyscy uczestnicy finałów otrzymali książki ufundowane przez WNT. Wszyscy zawodnicy, którzy uzyskali tytuł finalisty otrzymali upominki ufundowane przez Ogólnopolską Fundację Edukacji Komputerowej. Większość finalistów otrzymała także książki podarowane przez Krajowy Fundusz na rzecz Dzieci.

Ogłoszono komunikat o powołaniu reprezentacji Polski w składach:

- Międzynarodowa Olimpiada Informatyczna oraz Olimpiada Informatyczna Centralnej Europy:

  (1) Marcin Michalski
  (2) Bartosz Walczak
  (3) Filip Wolski
  (4) Michał Brzozowski

  zawodnikami rezerwowymi zostali:

  (5) Andrzej Chodor
  (6) Łukasz Krupa
  (7) Michał Jaszczyk

- Bałtycka Olimpiada Informatyczna:

  (1) Marcin Michalski
  (2) Bartosz Walczak
  (3) Filip Wolski
  (4) Michał Brzozowski
  (5) Andrzej Chodor
  (6) Łukasz Krupa
  (7) Michał Jaszczyk
  (8) Szymon Acedański

  zawodnikami rezerwowymi zostali:

  (9) Michał Bartoszkiewicz
  (10) Bartłomiej Romański

- obóz czesko-polsko-słowacki: członkowie reprezentacji oraz zawodnicy rezerwowi powołani na Międzynarodową Olimpiadę Informatyczną,

- obóz rozwojowo-treningowy im. A. Kreczmara dla finalistów Olimpiady Informatycznej: reprezentanci na Międzynarodową Olimpiadę Informatyczną oraz laureaci i finaliści Olimpiady, którzy nie byli w ostatnim roku szkolnym w programowo najwyższych klasach szkół średnich.

Sekretariat wystawił łącznie 14 zaświadczeń o uzyskaniu tytułu laureata i 30 zaświadczeń o uzyskaniu tytułu finalisty lub finalisty z wyróżnieniem X Olimpiady Informatycznej.

Finaliści zostali poinformowani o decyzjach Senatów wielu szkół wyższych dotyczących przyjęć na studia z pominięciem zwykłego postępowania kwalifikacyjnego.

Komitet Główny wyróżnił za wkład pracy w przygotowanie finalistów Olimpiady Informatycznej następujących opiekunów naukowych:

- Michał Baczyński (Uniwersytet Śląski)
  - Szymon Acedański (laureat II stopnia)
- Grażyna Bauerek (Zespół Szkół Technicznych w Wodzisławiu Śląskim)
  - Adrian Galewski (finalista)
- Iwona Bujnowska (I LO im. A. Mickiewicza w Białymstoku)
  - Adam Nowacki (finalista)
- Piotr Cacko (L LO im. Ruy Barbosa w Warszawie)
  - Maciej Kalbarczyk (finalista)
- Andrzej Dyrek (V LO im. A. Witkowskiego w Krakowie)
  - Bartosz Walczak (laureat I stopnia)
  - Jan Prażuch (finalista z wyróżnieniem)
  - Jan Kaczmarczyk (finalista z wyróżnieniem)
  - Piotr Danilewski (finalista)
  - Maciej Taczuk (finalista)
- Urszula Dzwonowska (XIV LO im. St. Staszica w Warszawie)
  - Adam Narkiewicz (laureat III stopnia)
- Marek Gałaszewski (I LO im. M. Konopnickiej w Suwałkach)
  - Adam Iwanicki (finalista)
- Alina Gościniak (VIII LO im. A. Mickiewicza w Poznaniu)
  - Piotr Hołubowicz (finalista z wyróżnieniem)
  - Szymon Wąsik (finalista z wyróżnieniem)
- Domicela Góral (IV LO im. Hanki Sawickiej w Kielcach)
  - Andrzej Chodor (laureat II stopnia)
- Dorota Roman-Jurdzińska (II LO im. Piastów Śląskich we Wrocławiu)
  - Maciej Popowicz (finalista)

- Maciej Kosylak (XXXI LO w Łodzi)
  - Damian Klata (finalista)
- Bogna Lubańska (V LO im. Księcia J. Poniatowskiego w Warszawie)
  - Piotr Kucharski (finalista)
  - Krzysztof Porczyk (finalista)
- Dawid Matla (XIV LO im. Polonii Belgijskiej we Wrocławiu)
  - Michał Bartoszkiewicz (laureat III stopnia)
- Wanda Narloch (I LO im. L. Kruczkowskiego w Tychach)
  - Paweł Gosztyła (laureat III stopnia)
- Bartosz Nowierski (student Politechniki Poznańskiej)
  - Piotr Hołubowicz (finalista z wyróżnieniem)
  - Szymon Wąsik (finalista z wyróżnieniem)
  - Dawid Sieradzki (finalista z wyróżnieniem)
- Elżbieta Pławińska-Podkrólewicz (IV LO w Toruniu)
  - Jarosław Osmański (finalista)
- Wojciech Roszczyński (VIII LO im. A. Mickiewicza w Poznaniu)
  - Dawid Sieradzki (finalista z wyróżnieniem)
- Ryszard Szubartowski (III LO im. Marynarki Wojennej RP w Gdyni)
  - Marcin Michalski (laureat I stopnia)
  - Filip Wolski (laureat I stopnia)
  - Michał Brzozowski laureat II stopnia
  - Andrzej Grzywocz (finalista z wyróżnieniem)
  - Jakub Łącki (finalista z wyróżnieniem)
  - Adam Slaski (finalista z wyróżnieniem)
  - Krzysztof Ślusarski (finalista z wyróżnieniem)
  - Tomasz Uliński (finalista z wyróżnieniem)
  - Piotr Drajski (finalista)
  - Tomasz Kłos (finalista)
  - Paweł Papis (finalista)
- Michał Szuman (XIII LO w Szczecinie)
  - Michał Jaszczyk (laureat II stopnia)

  - Piotr Mikulski (finalista)

- Iwona Waszkiewicz (VI LO im. J. i J. Śniadeckich w Bydgoszczy)
  - Aleksander Piotrowski (finalista)

- Maria Wielgus (Akademickie Centrum Komputerowe w Krakowie)
  - Marcin Wielgus (laureat III stopnia)

Zgodnie z Rozporządzeniem MEN w sprawie olimpiad, wyróżnieni nauczyciele otrzymają nagrody pieniężne.

OBCHODY X-LECIA OLIMPIADY INFORMATYCZNEJ

W roku szkolnym 2002/2003 Olimpiada Informatyczna obchodziła X-lecie swego istnienia. W związku z tym jubileuszem Stanisław Waligórski — pierwszy przewodniczący Komitetu Głównego, jeden ze współtwórców Olimpiady Informatycznej, wielce zasłużony dla Olimpiady Informatycznej i całej edukacji informatycznej w Polsce — otrzymał z rąk Prezydenta RP, Aleksandra Kwaśniewskiego, Krzyż Kawalerski Orderu Odrodzenia Polski.

Medalami Komisji Edukacji Narodowej odznaczono Macieja M. Sysłę, założyciela Olimpiady Informatycznej, wieloletniego wiceprzewodniczącego Komitetu Głównego oraz nauczycieli, którzy przygotowali największą liczbę finalistów (licząc do IX Olimpiady): Andrzeja Dyrka i Iwonę Waszkiewicz.

Komitet Główny postanowił uświetnić tę rocznicę uroczystością (połączoną z ogłoszeniem wyników X Olimpiady), która odbyła się 12 kwietnia br. w Urzędzie Miasta Sopot. W uroczystości uczestniczyli:

- przedstawiciel Ministerstwa Edukacji Narodowej i Sportu, Podsekretarz Stanu — Adam Giersz,
- zasłużeni dla Olimpiady: Stanisław Waligórski, Bolesław Wojdyło,
- przedstawiciele zaprzyjaźnionych z Olimpiadą firm:
  - Prokom Software S.A. — Wiceprezes Krzysztof Wilski i Członek Zarządu Beata Stelmach,
  - Combidata — Prezes Krzysztof Koszewski,
  - WNT — redaktor Elżbieta Beuermann,
- przedstawiciele Sopotu z Prezydentem Miasta Jackiem Karnowskim,
- nauczyciele-opiekunowie naukowi finalistów,
- oraz złoci medaliści wszystkich poprzednich 9 olimpiad.

Wyrazy uznania dla Olimpiady oraz życzenia dalszych sukcesów przekazali przedstawiciele MENiS, współorganizatorów i władz miasta Sopot.

Wszystkim zaproszonym wręczono znaczki olimpiady informatycznej.

Zasłużeni działacze: Stanisław Waligórski, Maciej M. Sysło, Tadeusz Kuran, Krzysztof Święcicki, Bolesław Wojdyło, Andrzej Walat, Krystyna Kominek, Krzysztof Stencel i Marcin Kubica otrzymali pamiątkowe plakietki.

Przedstawicielom firm: Prokom Software S.A., Combidata Poland, WNT oraz OFEK, które wspierały organizację Olimpiady, wręczono dyplomy MENiS i pamiątkowe plakietki.

Wszystkim zasłużonym dla Olimpiady Informatycznej oraz gościom uroczystości wręczono medale X-lecia Olimpiady.

Warszawa, 13 czerwca 2003 roku

Regulamin Olimpiady Informatycznej

§1 WSTĘP

Olimpiada Informatyczna jest olimpiadą przedmiotową powołaną przez Instytut Informatyki Uniwersytetu Wrocławskiego, który jest organizatorem Olimpiady, zgodnie z zarządzeniem nr 28 Ministra Edukacji Narodowej z dnia 14 września 1992 roku (Dz. Urz. MEN nr 7 z 1992 r. poz. 31) z późniejszymi zmianami (Rozporządzenie Ministra Edukacji Narodowej i Sportu z dnia 29 stycznia 2002 r. w sprawie organizacji oraz sposobu przeprowadzenia konkursów, turniejów i olimpiad). W organizacji Olimpiady Instytut współdziała ze środowiskami akademickimi, zawodowymi i oświatowymi działającymi w sprawach edukacji informatycznej.

§2 CELE OLIMPIADY INFORMATYCZNEJ

(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.

§3 ORGANIZACJA OLIMPIADY

(1) Olimpiadę przeprowadza Komitet Główny Olimpiady Informatycznej.

(2) Olimpiada Informatyczna jest trójstopniowa.

(3) W Olimpiadzie Informatycznej 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) Integralną częścią rozwiązania zadania zawodów I, II i III stopnia jest, zgodnie z treścią zadania, program w języku programowania wybranym z listy języków ustalanej przez Komitet Główny corocznie przed rozpoczęciem zawodów i ogłaszanej w „Zasadach organizacji zawodów" na dany rok szkolny lub dane do oceny.

(6) Zawody I stopnia mają charakter otwarty i polegają na samodzielnym rozwiązywaniu przez uczestnika zadań ustalonych dla tych zawodów oraz przekazaniu rozwiązań w podanym terminie, w miejsce i w sposób określony w „Zasadach organizacji zawodów".

(7) Liczbę uczestników kwalifikowanych do zawodów II i III stopnia ustala Komitet Główny i podaje ją w „Zasadach organizacji zawodów".

(8) Prace są oceniane automatycznie za pomocą specjalnego oprogramowania narzędziowego. Jeśli rozwiązaniem zadania jest program, to program zawodnika jest uruchamiany na testach z przygotowanego zestawu. Czas działania programu jest ograniczony przez z góry zadany limit czasowy. Zestawy testów i limity czasowe są tajne do chwili zakończenia zawodów. Podstawą oceny jest zgodność sprawdzanego programu z podaną w treści zadania specyfikacją, poprawność wygenerowanego przez program wyniku oraz czas działania tego programu. Jeśli rozwiązaniem zadania jest plik z danymi, wówczas ocenia się jego poprawność i na tej podstawie przyznaje punkty.

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

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

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

(12) Prace zespołowe, niesamodzielne lub niezgodne z „Zasadami organizacji zawodów" nie będą brane pod uwagę.

(13) Tryb opracowywania zadań olimpijskich:

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

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

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

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

§4 KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

(1) Komitet Główny Olimpiady Informatycznej, zwany dalej Komitetem, powoływany przez organizatora na kadencję trzyletnią, jest odpowiedzialny za poziom merytoryczny i organizację zawodów. Komitet składa corocznie organizatorowi sprawozdanie z przeprowadzonych zawodów.

(2) Członkami Komitetu mogą być pracownicy naukowi, nauczyciele, pracownicy oświaty związani z kształceniem informatycznym oraz studenci informatyki.

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

(4) Komitet może w czasie swojej kadencji dokonywać zmian w swoim składzie.

(5) Komitet powołuje i rozwiązuje komitety okręgowe Olimpiady.

(6) Komitet:

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

(b) powołuje i odwołuje członków Jury Olimpiady, które jest odpowiedzialne za sprawdzenie zadań,

(c) udziela wyjaśnień w sprawach dotyczących Olimpiady,

(d) ustala listy laureatów i wyróżnionych uczestników oraz kolejność lokat,

(e) przyznaje uprawnienia i nagrody rzeczowe wyróżniającym się uczestnikom Olimpiady,

(f) wyłania uprawnionych do startu w Międzynarodowej Olimpiadzie Informatycznej i innych międzynarodowych zawodach informatycznych oraz ustala skład reprezentacji.

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

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

(9) Do organizacji zawodów II stopnia w miejscowościach, których nie obejmuje żaden Komitet Okręgowy, Komitet Główny powołuje Komisję Zawodów co najmniej trzy miesiące przed terminem rozpoczęcia zawodów.

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

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

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

(13) Komitet ma siedzibę w Ośrodku Edukacji Informatycznej i Zastosowań Komputerów w Warszawie. Ośrodek wspiera Komitet we wszystkich działaniach organizacyjnych zgodnie z Deklaracją przekazaną organizatorowi.

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

(15) 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.

(16) 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.

(17) 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) Kadencja komitetu wygasa wraz z kadencją Komitetu Głównego.

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

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

(5) Do organizacji zawodów II stopnia komitet okręgowy powołuje Komisję Zawodów co najmniej dwa miesiące przed terminem rozpoczęcia zawodów.

(6) Przewodniczący (albo jego zastępca) oraz sekretarz komitetu okręgowego mogą uczestniczyć w obradach Komitetu Głównego z prawem głosu.

§6 PRZEBIEG OLIMPIADY

(1) Komitet Główny rozsyła do szkół wymienionych w §3.3 oraz kuratoriów oświaty i koordynatorów edukacji informatycznej treść zadań I stopnia wraz z „Zasadami organizacji zawodów".

(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) Uczestnicy zawodów II stopnia, których wyniki zostały uznane przez Komitet Główny Olimpiady za wyróżniające, otrzymują na podstawie zaświadczenia wydanego przez Komitet, najwyższą ocenę z informatyki na zakończenie nauki w klasie, do której uczęszczają.

(2) Uczestnicy Olimpiady, którzy zostali zakwalifikowani do zawodów III stopnia są zwolnieni z egzaminu z przygotowania zawodowego z przedmiotu informatyka oraz (zgodnie z zarządzeniem nr 35 Ministra Edukacji Narodowej z dnia 30 listopada 1991 r. oraz rozporządzeniem Ministra Edukacji Narodowej i Sportu z dnia 21 marca 2001 r.) z części ustnej egzaminu dojrzałości z przedmiotu informatyka, jeżeli w klasie, do której uczęszczał zawodnik był realizowany ten przedmiot w zakresie pozwalającym na zdawanie matury z tego przedmiotu.

(3) Laureaci zawodów III stopnia, a także finaliści, są zwolnieni w części lub w całości z egzaminów wstępnych do tych szkół wyższych, których senaty podjęły odpowiednie uchwały zgodnie z przepisami ustawy z dnia 12 września 1990 r. o szkolnictwie wyższym (Dz.U. nr 65 poz. 385).

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

(5) Nauczyciel, którego praca przy przygotowaniu uczestnika Olimpiady zostanie oceniona przez Komitet Główny jako wyróżniająca otrzymuje nagrodę wypłacaną z budżetu Olimpiady.

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

(7) 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 będzie się ubiegał o pozyskanie środków finansowych z budżetu państwa, składając wniosek w tej sprawie do Ministra Edukacji Narodowej i Sportu i przedstawiając przewidywany plan finansowy organizacji Olimpiady na dany rok. Komitet 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 z przeprowadzonej Olimpiady w ciągu 2 miesięcy po jej zakończeniu i przedstawia je organizatorowi i Ministerstwu Edukacji Narodowej.

(3) Niniejszy regulamin może być zmieniony przez Komitet Główny tylko przed rozpoczęciem kolejnej edycji zawodów Olimpiady po zatwierdzeniu zmian przez organizatora i uzyskaniu aprobaty Ministerstwa Edukacji Narodowej.

Warszawa, 6 czerwca 2002 r.

Zasady organizacji zawodów w roku szkolnym 2002/2003

§1 WSTĘP

(1) Olimpiada Informatyczna jest olimpiadą przedmiotową powołaną przez Instytut Informatyki Uniwersytetu Wrocławskiego, który jest organizatorem Olimpiady zgodnie z zarządzeniem nr 28 Ministra Edukacji Narodowej z dnia 14 września 1992 roku (Dz. Urz. MEN nr 7 z 1992 r. poz. 31) z późniejszymi zmianami (zarządzenie Ministra Edukacji Narodowej nr 19 z dnia 20 października 1994 r., Dz. Urz. MEN nr 5 z 1994 r poz. 27).

§2 ORGANIZACJA OLIMPIADY

(1) Olimpiadę przeprowadza Komitet Główny Olimpiady Informatycznej.

(2) Olimpiada Informatyczna jest trójstopniowa.

(3) Olimpiada Informatyczna jest przeznaczona dla uczniów wszystkich typów szkół średnich dla młodzieży (z wyjątkiem szkół policealnych). W Olimpiadzie mogą również uczestniczyć uczniowie gimnazjów oraz — za zgodą Komitetu Głównego — uczniowie szkół podstawowych.

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

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

(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 280 uczestników, których rozwiązania zadań I stopnia zostaną ocenione najwyżej; do zawodów III stopnia — 40 uczestników, których rozwiązania zadań II stopnia zostaną ocenione najwyżej. Komitet Główny może zmienić podane liczby zakwalifikowanych uczestników co najwyżej o 20%.

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

(9) Terminarz zawodów:

zawody I stopnia — 7.10–4.11. 2002 r.
ogłoszenie wyników:
w witrynie Olimpiady — 7.12.2002 r.,
pocztą — 21.12.2002 r.
zawody II stopnia — 11–13.02.2003 r.
ogłoszenie wyników:
w witrynie Olimpiady — 22.02.2003 r.
pocztą — 5.03.2003 r.
zawody III stopnia — 8–12.04.2003 r.

§3 WYMAGANIA DOTYCZĄCE ROZWIĄZAŃ ZADAŃ ZAWODÓW I STOPNIA

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

- Poprzez witrynę Olimpiady o adresie: `www.oi.edu.pl` do godziny 12.00 (w południe) dnia 4 listopada 2002 r. Komitet Główny nie ponosi odpowiedzialności za brak możliwości przekazania rozwiązań przez witrynę w sytuacji nadmiernego obciążenia lub awarii serwisu. Odbiór przesyłki zostanie potwierdzony przez Komitet Główny zwrotnym listem elektronicznym (prosimy o zachowanie tego listu). Brak potwierdzenia może oznaczać, że rozwiązanie nie zostało poprawnie zarejestrowane. W tym przypadku zawodnik powinien przesłać swoje rozwiązanie przesyłką poleconą za pośrednictwem zwykłej poczty. Szczegóły dotyczące sposobu postępowania przy przekazywaniu zadań i związanej z tym rejestracji będą dokładnie podane w witrynie.
- Pocztą, przesyłką poleconą, na adres:

  Olimpiada Informatyczna
  Ośrodek Edukacji Informatycznej i Zastosowań Komputerów
  ul. Raszyńska 8/10
  02-026 Warszawa
  tel. (0-prefiks-22) 822 40 19, 668 55 33

  w nieprzekraczalnym terminie nadania do 4 listopada 2002 r. (decyduje data stempla pocztowego). Prosimy o zachowanie dowodu nadania przesyłki. Rozwiązania dostarczane w inny sposób nie będą przyjmowane. W przypadku jednoczesnego zgłoszenia rozwiązania przez Internet i listem poleconym, ocenie podlega jedynie rozwiązanie wysłane listem poleconym.

W takim przypadku konieczne jest również podanie w dokumencie zgłoszeniowym identyfikatora użytkownika użytego do zgłoszenia rozwiązań przez Internet.

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

(3) Prace niesamodzielne lub zbiorowe nie będą brane pod uwagę.

(4) Rozwiązanie każdego zadania składa się z programu (tylko jednego) w postaci źródłowej; imię i nazwisko uczestnika musi być podane w komentarzu na początku każdego programu.

(5) Nazwy plików z programami w postaci źródłowej muszą mieć jako rozszerzenie co najwyżej trzyliterowy skrót nazwy użytego języka programowania, to jest:

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

(6) Programy w C/C++ będą kompilowane w systemie Linux za pomocą kompilatora GCC v. 2.95. Programy w Pascalu będą kompilowane w systemie Linux za pomocą kompilatora FreePascala v. 1.0.6. Wybór polecenia kompilacji zależy od podanego rozszerzenia pliku w następujący sposób:

Dla c	`gcc -O2 -static -lm zadanie.c`
Dla cpp	`g++ -O2 -static -lm zadanie.cpp`
Dla pas	`ppc386 -O2 -XS zadanie.pas`

Pakiety instalacyjne tych kompilatorów (i ich wersje dla DOS/Windows) są dostępne w witrynie Olimpiady `www.oi.edu.pl`.

(7) Program powinien odczytywać dane wejściowe ze standardowego wejścia i zapisywać dane wyjściowe na standardowe wyjście.

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

(9) Uczestnik korzystający z poczty zwykłej przysyła:

- Jedną dyskietkę w formacie FAT (standard dla komputerów PC) zawierającą:
  - spis zawartości dyskietki oraz dane osobowe zawodnika w pliku nazwanym `SPIS.TRC`,
  - do każdego rozwiązanego zadania - program źródłowy.

Dyskietka nie powinna zawierać żadnych podkatalogów.

- Wypełniony dokument zgłoszeniowy (dostępny jako załącznik do niniejszego plakatu lub w witrynie internetowej Olimpiady). Gorąco prosimy o podanie adresu elektronicznego. Podanie adresu jest niezbędne do wzięcia udziału w procedurze reklamacyjnej opisanej w punktach 13, 14 i 15.

(10) Uczestnik korzystający z witryny olimpiady postępuje zgodnie z instrukcjami umieszczonymi w witrynie.

(11) W witrynie Olimpiady o adresie `www.oi.edu.pl` wśród Informacji dla zawodników znajdują się Odpowiedzi na pytania zawodników dotyczące Olimpiady. Ponieważ Odpowiedzi mogą zawierać ważne informacje dotyczące toczących się zawodów, prosimy wszystkich uczestników Olimpiady o regularne zapoznawanie się z ukazującymi się odpowiedziami. Dalsze pytania należy przysyłać poprzez witrynę Olimpiady. Komitet Główny może nie udzielić odpowiedzi na pytanie z ważnych przyczyn, m.in. gdy jest ono niejednoznaczne lub dotyczy sposobu rozwiązania zadania.

(12) Poprzez witrynę dostępne są narzędzia do sprawdzania rozwiązań pod względem formalnym. Szczegóły dotyczące sposobu postępowania są dokładnie podane w witrynie.

(13) Od dnia 25.11.2002 r. poprzez witrynę Olimpiady każdy zawodnik będzie mógł zapoznać się ze wstępną oceną swojej pracy. Wstępne oceny będą dostępne jedynie w witrynie Olimpiady i tylko dla osób, które podały adres elektroniczny.

(14) Do dnia 29.11.2002 r. (włącznie) poprzez witrynę Olimpiady każdy zawodnik będzie mógł zgłaszać uwagi do wstępnej oceny swoich rozwiązań. Reklamacji nie podlega jednak dobór testów, limitów czasowych, kompilatorów i sposobu oceny.

(15) Reklamacje złożone po 29.11.2002 r. nie będą rozpatrywane.

§4 UPRAWNIENIA I NAGRODY

(1) Uczestnicy zawodów II stopnia, których wyniki zostały uznane przez Komitet Główny za wyróżniające, otrzymują najwyższą ocenę z informatyki na zakończenie nauki w klasie, do której uczęszczają.

(2) Uczestnicy Olimpiady, którzy zostali zakwalifikowani do zawodów III stopnia, są zwolnieni z egzaminu dojrzałości (zgodnie z zarządzeniem nr 29 Ministra Edukacji Narodowej z dnia 30 listopada 1991 r.) lub z egzaminu z przygotowania zawodowego z przedmiotu informatyka. Zwolnienie jest równoznaczne z wystawieniem oceny najwyższej.

(3) Laureaci i finaliści Olimpiady są zwolnieni w części lub w całości z egzaminów wstępnych do tych szkół wyższych, których senaty podjęty odpowiednie uchwały zgodnie z przepisami ustawy z dnia 12 września 1990 roku o szkolnictwie wyższym (Dz. U. nr 65).

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

(5) Komitet Główny ustala skład reprezentacji Polski na XV Międzynarodową Olimpiadę Informatyczną w 2003 roku na podstawie wyników zawodów III stopnia i regulaminu tej Olimpiady Międzynarodowej. Szczegółowe zasady zostaną podane po otrzymaniu formalnego zaproszenia na XV Międzynarodową Olimpiadę Informatyczną.

(6) Nauczyciel (opiekun naukowy), który przygotował laureata Olimpiady Informatycznej, otrzymuje nagrodę przyznawaną przez Komitet Główny Olimpiady.

(7) Wyznaczeni przez Komitet Główny reprezentanci Polski na olimpiady międzynarodowe oraz finaliści, którzy nie są w ostatniej programowo klasie swojej szkoły, zostaną zaproszeni do nieodpłatnego udziału w IV Obozie Naukowo-Treningowym im. Antoniego Kreczmara, który odbędzie się w czasie wakacji 2003 r.

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

§5 PRZEPISY KOŃCOWE

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

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

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

(4) Uczniowie zakwalifikowani 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, wyżywienie i zwrot kosztów przejazdu.

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

Zawody I stopnia

Zawody I stopnia — opracowania zadań

Krzysztof Sikora — Treść zadania, Opracowanie
Marcin Stefaniak — Program

Ciągi bez zająknięć

Rozważamy ciągi liter. Powiemy, że ciąg $x_1x_2\ldots x_n$ zawiera **zająknięcie**, *jeśli napotykamy w nim dwa, następujące bezpośrednio po sobie, wystąpienia takiego samego podciągu. Tzn. jeśli dla pewnych i i j ($1 \leqslant i < j \leqslant \frac{n+i+1}{2}$) zachodzi: $x_i = x_j$, $x_{i+1} = x_{j+1}$, $\ldots$, $x_{j-1} = x_{2j-i-1}$.*

Interesują nas n-elementowe ciągi bez zająknięć o minimalnej liczbie liter.

Przykłady

Dla $n = 3$ wystarczą dwie litery, powiedzmy a i b. Mamy dokładnie dwa 3-elementowe ciągi bez zająknięć złożone z takich liter: aba i bab. Dla $n = 5$ potrzebne są już 3 różne litery. Na przykład abcab jest trójliterowym ciągiem bez zająknięć. W ciągu babab mamy dwa zająknięcia: ba i ab.

Zadanie

Napisz program, który:

- *wczyta długość ciągu n,*
- *obliczy n-elementowy ciąg bez zająknięć o minimalnej liczbie różnych liter,*
- *wypisze wynik.*

Wejście

W pierwszym wierszu standardowego wejścia zapisana jest jedna dodatnia liczba całkowita n, $1 \leqslant n \leqslant 10\,000\,000$.

Wyjście

Twój program powinien pisać na standardowe wyjście. W pierwszym wierszu powinna zostać wypisana jedna dodatnia liczba całkowita k, równa minimalnej liczbie różnych liter, które muszą wystąpić w n-elementowym ciągu nie zawierającym zająknięć.

W drugim wierszu należy wypisać obliczony ciąg bez zająknięć, jako słowo złożone z n małych liter alfabetu angielskiego, od litery `a` *do k-tej litery alfabetu. Jeżeli istnieje wiele takich ciągów, Twój program powinien wypisać jeden z nich.*

Możesz przyjąć, że 26 liter wystarczy.

Przykład

Dla danych wejściowych:

```
5
```

poprawnym wynikiem jest:

```
3
abcab
```

Rozwiązanie

Analiza problemu

Słowa bezkwadratowe i silnie bezsześcienne

Rozpoczniemy od kilku definicji, które są niezbędne do zrozumienia rozwiązania wzorcowego.

Definicja 1 Słowo nazywamy *bezkwadratowym*, jeśli nie zawiera podsłowa postaci xx, gdzie x jest niepustym, skończonym słowem.

Łatwo zauważyć, że nad alfabetem dwuliterowym ($\{a, b\}$) nie istnieje słowo bezkwadratowe dłuższe niż 3, ponieważ dowolny ciąg długości 4 zawiera aa lub bb albo jest postaci $abab$ lub $baba$. Z tego powodu dowolne bezkwadratowe słowo o długości większej niż 3 musi być słowem nad alfabetem co najmniej trzyliterowym. Norweski matematyk Axel Thue udowodnił, że istnieje nieskończenie długie słowo bezkwadratowe nad alfabetem trzyliterowym i pokazał, jak je znaleźć.

Definicja 2 Słowo nazywamy *silnie bezsześciennym*, jeśli nie zawiera podsłowa postaci xxa, gdzie x jest niepustym, skończonym słowem i a jest pierwszym symbolem x.

Oczywiście, jeśli słowo jest bezkwadratowe, to jest także silnie bezsześcienne.

Słowo Thuego-Morsa

Definicja 3 *Słowem Thuego-Morsa* nad alfabetem $\{a, b\}$ nazywamy ciąg $\alpha = (s_n)_{n\in\mathbb{N}}$ taki, że

$$s_n = \begin{cases} a & \text{jeśli } n \text{ ma parzystą liczbę jedynek w zapisie binarnym,} \\ b & \text{jeśli } n \text{ ma nieparzystą liczbę jedynek w zapisie binarnym.} \end{cases}$$

W konsekwencji powstaje nieskończone słowo nad alfabetem dwuelementowym

$$\alpha = abbabaabbaabab\ldots$$

Podamy teraz kilka własności słowa Thuego-Morsa. Dla prostoty dowodów załóżmy, że litery a i b są „przeciwne”, czyli $a = -b$ i $b = -a$. Łatwo zauważyć, patrząc na binarną reprezentacje n, że:

Własność 4 $s_{2n} = s_n$

Własność 5 $s_{2n+1} = -s_n$

Z własności 4 i 5 mamy:

Własność 6 *Jeśli $s_j = s_{j+1}$, to j jest nieparzyste.*

Dowód Załóżmy, że $s_j = s_{j+1}$ i j jest parzyste. Wtedy $s_j = s_{j/2}$ i $s_{j+1} = -s_{j/2}$, czyli $s_j \neq s_{j+1}$. Sprzeczność. ■

Teraz możemy łatwo pokazać, że:

Własność 7 *Nie istnieje j takie, że $s_j = s_{j+1} = s_{j+2}$.*

Z własności 4 wynika, że

$$(s_{2n})_{n\in\mathbb{N}} = \alpha,$$

a z własności 5, że

$$(-s_{2n+1})_{n\in\mathbb{N}} = \alpha.$$

Oznaczmy ciąg składający się z liter na pozycjach parzystych przez P, a ciąg składający się z liter na pozycjach nieparzystych przez N.

Własność 8 *W 5 kolejnych literach ciągu α istnieją 2 kolejne litery, które są takie same.*

Dowód Gdyby tak nie było, mielibyśmy jedną z dwóch sytuacji: *ababa* lub *babab*. Wtedy podciąg złożony z pierwszej, trzeciej i piątej litery byłby postaci *aaa* lub *bbb* i należał albo do P, albo do N, w zależności od parzystości położenie ciągu w słowie Thuego-Morsa. Teraz P lub N zawierałby 3 kolejne takie same litery, co jest niemożliwe. ■

Twierdzenie 9 *Słowo Thuego-Morsa jest silnie bezsześcienne, tzn. nie istnieją $i \geqslant 0$, $j > 0$ takie, że $s_{i+k} = s_{i+j+k}$ dla każdego k, $0 \leqslant k \leqslant j$.*

Dowód Załóżmy, że takie wartości i i j istnieją, wtedy:

1. j nie może być równe 1 (z własności 7).
2. j nie może być nieparzyste i większe niż 1, gdyż wtedy jest co najmniej 7 liter między i i $i + 2j$ (włącznie) i z własności 8 wynika, że istnieją dwie kolejne takie same litery. Z faktu, iż litery od i to $i + j$ są takie same jak od $i + j$ do $i + 2j$, mamy, że istnieją dwie różne takie pary, przesunięte o j. Ponieważ j jest nieparzyste, jedna z tych par zaczyna się na parzystej pozycji, co nie jest możliwe (z własności 6).
3. Zatem j musi być parzyste. Załóżmy, że jest minimalne. Teraz albo P, albo N spełnia równania z twierdzenia dla $j/2$ w miejsce j. Ponieważ j jest parzyste, mamy sprzeczność, bo j było minimalne. Nie może być także nieparzyste, co zostało udowodnione wcześniej. ■

Lemat 10 *Istnieje słowo bezkwadratowe β nad alfabetem czteroliterowym.*

Dowód Skonstruujmy alfabet mocy 4 o elementach będących parami symboli z alfabetu zdefiniowanego wcześniej:

$$\Sigma = \{[aa], [ab], [ba], [bb]\}.$$

Zdefiniujmy teraz ciąg $\beta = (d_n)_{n \in \mathbb{N}}$ taki, że

$$d_n = [s_n s_{n+1}] \text{ dla każdego } n \geqslant 0,$$

gdzie s_n to litery ciągu Thuego-Morsa. Ciąg β jest bezkwadratowy. Gdyby tak nie było, istniałoby podsłowo yy, w którym

$$y = d_j \dots d_{j+t-1} = d_{j+t} \dots d_{j+2t-1}$$

dla pewnego $t \geqslant 1$. Wtedy mielibyśmy

$$[s_j s_{j+1}] \dots [s_{j+t-1} s_{j+t}] = [s_{j+t} s_{j+t+1}] \dots [s_{j+2t-1} s_{j+2t}],$$

co daje $s_{j+i} = s_{j+t+i}$ dla $0 \leqslant i \ \leqslant t$. Wtedy słowo Thuego-Morsa zawiera

$$(s_j \dots s_{j+t-1})^2 s_j,$$

co jest sprzeczne z twierdzeniem 9. ■

Teraz pokażemy, jak przekształcić ciąg β w bezkwadratowy ciąg γ nad trzyliterowym alfabetem. Konstrukcja opiera się na następującej obserwacji: jeśli oznaczymy litery alfabetu Σ jako

$$[aa] = 1,\ [ab] = 2,\ [ba] = 3,\ [bb] = 4,$$

to łatwo można zobaczyć, że w β symbol 1 musi być przed 1 lub 2. W rzeczywistości 1 nie może występować po 1, gdyż β jest bezkwadratowe. Więc 1 musi być przed 2. Przeprowadzając podobne rozumowanie dla pozostałych symboli otrzymujemy:

Lemat 11 *W słowie β:*

- 1 *występuje po* 3 *i przed* 2.
- 4 *występuje po* 2 *i przed* 3.

Teraz, gdy zastąpimy wszystkie wystąpienia 1 i 4 w ciągu β przez 5, otrzymamy ciąg γ.

Twierdzenie 12 *Słowo γ jest bezkwadratowe.*

Dowód Gdyby tak nie było, istniałoby podsłowo yy.

Gdyby długość y była równa 1, yy byłoby postaci 55 (22 i 33 nie mogą się pojawić). To pociągałoby za sobą istnienie 11, 14, 41 lub 44 w ciągu β, co byłoby sprzeczne z lematem 11.

Gdyby długość y wynosiła co najmniej 2, wówczas każde wystąpienie 5 w y znajdowałoby się pomiędzy dwoma symbolami. Wtedy moglibyśmy dokonać jednoznacznej rekonstrukcji słowa (nazwijmy je x), z którego powstał y. Wówczas β zawierałoby xx, co jest sprzeczne z lematem 10. ■

Algorytm

Rozwiązanie wzorcowe rozpatruje 4 przypadki:

- dla $n = 1$ wielkość alfabetu wynosi 1, a poszukiwanym słowem jest a,
- dla $n = 2$ wielkość alfabetu wynosi 2, a poszukiwanym słowem jest ab,
- dla $n = 3$ wielkość alfabetu wynosi 2, a poszukiwanym słowem jest aba,
- dla $n \geqslant 4$ wielkość alfabetu wynosi 3, a poszukiwane słowo otrzymywane jest przez generowanie kolejnych liter ciągu Thuego-Morsa i przekształcanie ich zgodnie z regułami podanymi wcześniej.

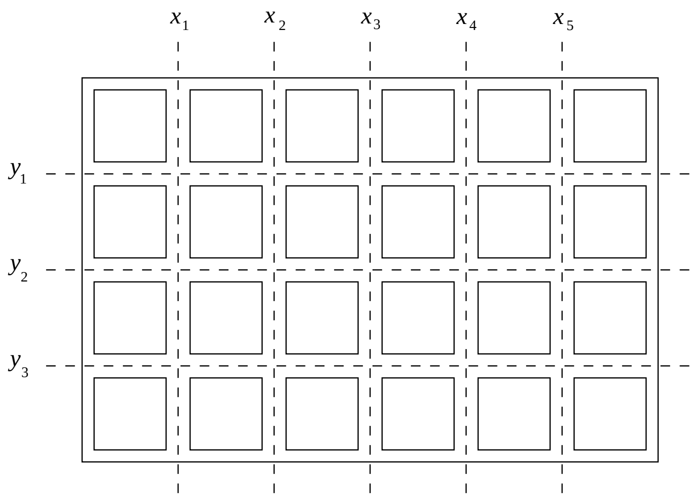
Algorytm działa w czasie $O(n)$ i wykorzystuje pamięć o stałym rozmiarze.

Testy

Zadanie sprawdzane było na zestawie 20 danych testowych, dla n równych 1, 2, 3, 4, 100, 300, 1000, 3000, 10000, 30000, 100000, 200000, 400000, 800000, 1000000, 2000000, 4000000, 6000000, 8000000 i 10000000.

Czekolada

Dana jest tabliczka czekolady złożona z $m \times n$ cząstek. Czekoladę należy połamać na pojedyncze cząstki. Kawałki czekolady możemy łamać wzdłuż pionowych i poziomych linii (zaznaczonych na rysunku liniami przerywanymi) wyznaczających podział czekolady na cząstki. Jedno przełamanie kawałka czekolady wzdłuż wybranej pionowej lub poziomej linii dzieli ten kawałek na dwa mniejsze. Każde przełamanie kawałka czekolady jest obarczone pewnym **kosztem** *wyrażającym się dodatnią liczbą całkowitą. Koszt ten nie zależy od wielkości łamanego kawałka, a jedynie od linii wzdłuż której łamiemy. Oznaczmy koszty łamania wzdłuż kolejnych pionowych linii przez x_1, x_2, ..., x_{m-1}, a wzdłuż poziomych linii przez y_1, y_2, ..., y_{n-1}. Koszt połamania całej tabliczki na pojedyncze cząstki to suma kosztów kolejnych przełamań. Należy obliczyć minimalny koszt połamania całej tabliczki na pojedyncze cząstki.*



Przykładowo, jeżeli połamiemy czekoladę przedstawioną na rysunku, najpierw wzdłuż linii poziomych, a następnie każdy z otrzymanych kawałków wzdłuż linii pionowych, to koszt takiego połamania wyniesie $y_1 + y_2 + y_3 + 4 \cdot (x_1 + x_2 + x_3 + x_4 + x_5)$.

Napisz program, który:

- *wczyta liczby $x_1, x_2, \ldots, x_{m-1}$ i $y_1, y_2, \ldots, y_{n-1}$,*
- *obliczy minimalny koszt połamania całej tabliczki na pojedyncze cząstki,*
- *wypisze wynik.*

Wejście

W pierwszym wierszu standardowego wejścia zapisane są dwie dodatnie liczby całkowite m i n oddzielone pojedynczym odstępem, $2 \leqslant m, n \leqslant 1\ 000$. W kolejnych $m-1$ wierszach zapisane *są liczby x_1, x_2, ..., x_{m-1}, po jednej w wierszu, $1 \leqslant x_i \leqslant 1\,000$. W kolejnych $n-1$ wierszach zapisane są liczby y_1, y_2, ..., y_{n-1}, po jednej w wierszu, $1 \leqslant y_i \leqslant 1\,000$.*

Wyjście

Twój program powinien pisać na standardowe wyjście. W pierwszym i jedynym wierszu Twój program powinien wypisać jedną liczbę całkowitą — minimalny koszt połamania całej tabliczki na pojedyncze cząstki.

Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
42
```

Rozwiązanie

Analiza

Rozwiązanie zadania opiera się na obserwacji, że lepiej najpierw łamać czekoladę wzdłuż linii o dużych kosztach po to, żeby wykonać takich łamań jak najmniej.

Zauważmy najpierw, że liczba przełamań jakie musimy wykonać nie zależy od sposobu połamania czekolady na cząstki. Fakt ten jest zapewne oczywisty dla wszystkich smakoszy czekolady.

Lemat 1 *Liczba przełamań koniecznych do połamania czekolady o wymiarach $m \times n$ nie zależy od sposobu połamania i wynosi $n \cdot m - 1$.*

Dowód Każde przełamanie zwiększa liczbę kawałków czekolady o 1. Ponieważ zaczynamy od czekolady w jednym kawałku, a kończymy mając $n \cdot m$ pojedynczych cząstek, więc wykonujemy $n \cdot m - 1$ przełamań. ■

Skoro liczba przełamań nie zależy od sposobu połamania czekolady, to powinniśmy tak łamać czekoladę, aby przełamania obarczone dużym kosztem pojawiały się jak najrzadziej. Intuicja podpowiada, że powinniśmy łamać czekoladę w kolejności od linii o dużych kosztach przełamania do linii o małych kosztach. Poniższy lemat potwierdza słuszność takiej intuicji.

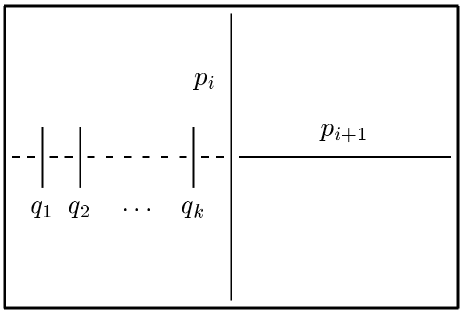
Lemat 2 *Istnieje rozwiązanie optymalne, w którym ciąg przełamań charakteryzuje się nierosnącymi kosztami.*

Dowód Załóżmy, że tak nie jest. Wybierzmy wówczas optymalny ciąg przełamań $(p_1, p_2, \ldots, p_{n \cdot m-1})$ o maksymalnej liczbie inwersji kosztów tych przełamań[1]. Mamy takie dwa kolejne przełamania p_i i p_{i+1}, że koszt p_{i+1} jest większy niż koszt p_i. Jeżeli jest kilka takich i, to wybierzmy największe. Tak więc, koszty przełamań p_{i+1}, p_{i+2}, $\ldots$, $p_{n \cdot m-1}$ tworzą ciąg nierosnący. Mamy kilka możliwych przypadków:

1. Przełamanie p_{i+1} leży poza obszarem kawałka przełamywanego przez p_i. Wówczas możemy zamienić miejscami przełamania p_i i p_{i+1} nie tracąc optymalności. Jednak ciąg kosztów przełamań $(p_1, \ldots, p_{i-1}, p_{i+1}, p_i, p_{i+2}, \ldots, p_{n \cdot m-1})$ ma więcej inwersji niż ciąg wyjściowy, co nie jest możliwe, a więc taki przypadek nie będzie miał miejsca.

2. Przełamanie p_{i+1} leży w obrębie kawałka przełamywanego przez p_i i oba przełamania są do siebie równoległe. Podobnie jak poprzednio, możemy wówczas zamienić miejscami przełamania p_i i p_{i+1} uzyskując również optymalny ciąg przełamać i to charakteryzujący się większą liczbą inwersji ciągu kosztów przełamań — sprzeczność, taki przypadek jest niemożliwy.

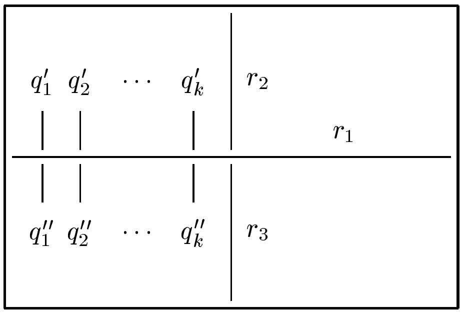
3. Przełamanie p_{i+1} leży w obrębie kawałka przełamywanego przez p_i i jest do niego prostopadłe. Sytuację tę przedstawia poniższy rysunek.



Wśród kolejnych przełamań musi się znajdować jedno lub więcej przełamań zaznaczonych na powyższym rysunku linią przerywaną. Ich liczba zależy od liczby przełamań $q_1, \ldots, q_k$ przecinających linię przerywaną. Zauważmy, że koszty przełamań $q_1, \ldots, q_k$ nie są większe niż koszt przełamania p_{i+1}.

Rozważmy trochę inny sposób połamania czekolady. Zastąpmy przełamania p_i, p_{i+1} oraz przełamania zaznaczone linią przerywaną przełamaniami r_1, r_2 i r_3 przedstawionymi na poniższym rysunku. Dodatkowo przełamania $q_1, \ldots, q_k$ zastępujemy przełamaniami $q'_1, q''_1, \ldots, q'_k, q''_k$.

[1] Liczba inwersji ciągu $(x_1, x_2, \ldots, x_n)$ to liczba takich par (i, j), że $i < j$ i $x_i > x_j$.



O ile zmienia się koszt połamania? Usuwamy przełamanie p_i oraz $k+2$ przełamań o takim koszcie co p_{i+1}, wprowadzamy natomiast jedno przełamanie o koszcie takim jak p_{i+1}, dwa przełamania o takim koszcie jak p_i oraz k dodatkowych przełamań o kosztach nie przekraczających kosztu p_{i+1}. Ponieważ koszt p_{i+1} jest większy niż koszt p_i, otrzymujemy więc rozwiązania lepsze od optymalnego. Jest to niemożliwe, a zatem przypadek taki nie będzie miał miejsca.

Reasumując, założenie, że lemat nie jest spełniony prowadzi do sprzeczności. Musi on więc być prawdziwy. ■

Pokazaliśmy, że istnieje rozwiązanie optymalne, w którym koszty kolejnych przełamań tworzą ciąg nierosnący, ale czy takie uporządkowanie wystarczy, aby połamanie było optymalne? Z poniższego lematu wynika, że tak.

Lemat 3 *Każde połamanie czekolady, w którym koszty kolejnych połamań tworzą ciąg nierosnący ma ten sam łączny koszt.*

Dowód Lemat ten można udowodnić przez indukcję ze względu na wielkość tabliczki czekolady.

1. Jeżeli tabliczka składa się tylko z jednej cząstki, to jest tylko jedno możliwe połamanie (puste), a więc lemat jest spełniony.

2. Załóżmy, że tabliczka jest większa. Rozważmy wszystkie linie łamania o maksymalnych kosztach. Przełamania wzdłuż tych linii wykonujemy oczywiście na początku. Powiedzmy, że mamy k pionowych i l poziomych takich linii. Dzielą one czekoladę na $(k+1)\cdot(l+1)$ prostokątnych kawałków czekolady.

   Potraktujmy przez chwilę te kawałki jak pojedyncze cząstki czekolady. Odpowiada to czekoladzie wielkości $(k+1)\times(l+1)$, w której wszystkie przełamania są obarczone takim samym kosztem. Na mocy lematu 1 każde połamanie takiej czekolady ma ten sam koszt i wymaga $(k+1)\cdot(l+1)-1$ przełamań, niezależnie od sposobu połamania.

   Zauważmy, że każde połamanie całej czekolady o nierosnących kosztach kolejnych przełamań składa się z przełamań o maksymalnym koszcie, oraz ze „scalenia” połamań pozostałych kawałków, z których każdy jest połamany w kolejności nierosnących kosztów przełamań. Potraktujmy teraz kawałki powstałe po wykonaniu przełamań o maksymalnym koszcie, jak mniejsze tabliczki czekolady.

> Z założenia indukcyjnego, każde połamanie takiego kawałka, o nierosnących kosztach kolejnych przełamań, ma ten sam koszt. Tak więc każde połamanie całej czekolady o nierosnących kosztach kolejnych przełamań ma ten sam koszt. ■

Z powyższego lematu wynika natychmiast następujące twierdzenie:

Twierdzenie 4 *Połamanie czekolady polegające na przełamywaniu jej wzdłuż linii łamania przez całą szerokość tabliczki, w kolejności nierosnących kosztów linii łamania, jest optymalnym połamaniem czekolady.*

Na tym twierdzeniu oprzemy rozwiązanie zadania.

Algorytm

Zadanie to możemy rozwiązać stosując programowanie zachłanne, korzystając z twierdzenia udowodnionego w poprzednim punkcie. Wystarczy, że posortujemy linie łamania wg kosztów, pamiętając które są pionowe, a które poziome. Następnie wykonujemy przełamania w kolejności nierosnących kosztów, przełamując za każdym razem przez całą szerokość tabliczki. Nie musimy pamiętać, jak dokładnie w danej chwili wygląda tabliczka czekolady. Wystarczy, że pamiętamy, ile pionowych i ile poziomych przełamań wykonaliśmy. Koszt przełamania przez całą szerokość tabliczki jest równy kosztowi pojedynczego przełamania pomnożonemu przez liczbę wykonanych przełamań w kierunku prostopadłym (plus jeden). Oto program implementujący takie rozwiązanie zadania:

```
const maxN = 1000;

type
  linia = record
    koszt : integer;
    kierunek : (poziomy, pionowy);
  end;
  dane = array[1..2 * maxN] of linia;

{ Procedura sortująca dane wg kosztów przełamań. }
procedure sort(var d: dane; n: integer);
...
var
  n, m, i, pionowe, poziome: integer;
  d: dane;
  koszt: longint;
begin
  { Wczytanie danych. }
  readln(m, n);
  for i := 1 to m - 1 do begin
    readln(d[i].koszt);
```

```
      d[i].kierunek := pionowy;
    end;
    for i := m to n + m - 2 do begin
      readln(d[i].koszt);
      d[i].kierunek := poziomy;
    end;

    { Obliczenie wyniku. }
    koszt := 0;
    sort(d, n + m - 2);
    pionowe := 0;
    poziome := 0;
    { Kolejne linie łamania, w kolejności nierosnących kosztów. }
    for i := n + m - 2 downto 1 do begin
      if d[i].kierunek = pionowy then begin
        { Pionowe przełamanie. }
        koszt := koszt + (poziome + 1) * d[i].koszt;
        pionowe := pionowe + 1;
      end else begin
        { Poziome przełamanie. }
        koszt := koszt + (pionowe + 1) * d[i].koszt;
        poziome := poziome + 1;
      end;
    end;

    { Wypisanie wyniku. }
    writeln(koszt);
  end;
```

Program ten można znaleźć na załączonej płytce. W rozwiązaniu tym dominujący koszt ma sortowanie danych. Dlatego też ma on złożoność czasową rzędu $\Theta((n+m)\cdot\log(n+m))$, a pamięciową rzędu $\Theta(n+m)$, co przy podanych w treści zadania ograniczeniach na dane jest akceptowalne. Można jednak trochę przyspieszyć to rozwiązanie.

Zauważmy, że linie łamania mają koszty całkowite z zakresu od 1 do 1 000. Zamiast sortować dane, wystarczy, że zliczymy, ile jest pionowych i poziomych linii łamania o określonych kosztach. Możemy wówczas obliczyć koszt przełamania wzdłuż wszystkich linii o tym samym koszcie za jednym zamachem. Złożoność takiego rozwiązania jest liniowa ze względu na sumę liczby linii przełamań i maksymalnego kosztu przełamania. Ponieważ zarówno m i n, jak i koszty przełamań są ograniczone w treści zadania przez 1 000, więc można przyjąć, że rozwiązanie to ma koszt stały, choć sama stała jest rzędu tysięcy operacji. Oto program realizujący to rozwiązanie, można go też znaleźć na załączonej płytce:

```
const
  MaxKoszt = 1000;

```

```
var
  pionowe, poziome: array [1..MaxKoszt] of integer;
  n, m, i, k, lpion, lpoz: integer;
  koszt: longint;

begin
  { Inicjacja struktury danych. }
  for i := 1 to MaxKoszt do begin
    pionowe[i] := 0;
    poziome[i] := 0;
  end;

  { Wczytanie danych. }
  readln(m, n);
  for i := 1 to m − 1 do begin
    readln(k);
    pionowe[k] := pionowe[k] + 1;
  end;
  for i := 1 to n − 1 do begin
    readln(k);
    poziome[k] := poziome[k] + 1;
  end;

  { Obliczenie wyniku. }
  koszt := 0;
  lpion := 0;
  lpoz := 0;
  for i := MaxKoszt downto 1 do begin
    koszt := koszt + (lpoz + 1) ∗ pionowe[i] ∗ i;
    lpion := lpion + pionowe[i];
    koszt := koszt + (lpion + 1) ∗ poziome[i] ∗ i;
    lpoz := lpoz + poziome[i];
  end;

  { Wypisanie wyniku. }
  writeln(koszt);
end;
```

Jak widać, oba przedstawione rozwiązania są dużo prostsze niż dowód twierdzenia, na którym się opierają. Choć dowód jest skomplikowany, to sama strategia zachłanna jest dosyć intuicyjna i zapewne część zawodników zaimplementowała ją bez dowodzenia jej poprawności. Z tego powodu zadanie okazało się być łatwe — 71% zgłoszonych rozwiązań zdobyło maksymalną liczbę punktów.

Testy

Rozwiązania tego zadania były sprawdzane na 10 testach, z czego 6 to testy poprawnościowe, a 4 to testy wydajnościowe. Testy poprawnościowe sprawdzały poprawność algorytmów na niedużych danych — czekoladach wielkości od 2×2 do 100×100. Testy wydajnościowe to czekolady wielkości około $1\,000 \times 1\,000$[2]. Testy te można znaleźć na załączonej płytce.

[2] Czekolada wielkości $1\,000 \times 1\,000$ kawałków, jak wykazują proste rachunki, powinna ważyć około 4 ton. To ci dopiero test wydajnościowy dla prawdziwego smakosza!

Łukasz Kaiser	**Łukasz Kaiser, Krzysztof Onak**	**Krzysztof Onak**
Treść zadania	Opracowanie	Program

Liczby Przesmyków

W zamierzchłych czasach żyło plemię Przesmyków. Byli to wybitni, jak na owe czasy, znawcy liczb. Do ich zapisu używali jedynie dwóch symboli, „+" i „−", których rytualne znaczenie jest wciąż badane przez historyków. Wiadomo, że Przesmycy umieli zapisywać wszystkie liczby naturalne $0, 1, 2, \ldots$ *Do zapisu liczb używali ciągów znaków „+" i „−", przy czym niektóre takie ciągi nie były wykorzystywane z przyczyn religijnych. Co roku kapłani ogłaszali, ile maksymalnie z rzędu znaków „−" może wystąpić w zapisach liczb. W zależności od roku ograniczenie to wynosiło od 1 do 113. Sposób zapisu liczb ustalano w następujący sposób: Wszystkie poprawne ciągi znaków „+" i „−" były ustawiane w kolejności od krótszych do dłuższych, a ciągi tej samej długości w porządku alfabetycznym (takim, jak w słowniku, przy czym „−" poprzedzał „+"). Tak uporządkowane ciągi reprezentowały kolejno liczby 0, 1, 2, ... Przykładowo, jeżeli nie można było używać więcej niż jednego znaku „−" z rzędu, to zapis liczb wyglądał następująco:*

0	$-$	*4*	$++$	*8*	$++-$	
1	$+$	*5*	$-+-$	*9*	$+++$	
2	$-+$	*6*	$-++$	*10*	$-+-+$	...
3	$+-$	*7*	$+-+$	*11*	$-++-$	

Wraz ze zmianą ograniczenia zmieniał się zapis liczb. Na przykład, gdy można było używać dwóch lub więcej znaków „−" z rzędu, liczba 2 była zapisywana jako „−−". Przysparza to dużo problemów współczesnym historykom.

Zadanie

Napisz program, który:

- *wczyta dwa ograniczenia na maksymalna liczbę znaków „−" z rzędu, które mogą pojawiać się w zapisach liczb, oraz zestaw liczb w zapisie Przesmyków dla pierwszego ograniczenia,*
- *przetłumaczy te liczby na zapis Przesmyków dla drugiego ograniczenia,*
- *wypisze liczby w nowym zapisie.*

Wejście

W pierwszym wierszu standardowego wejścia zapisane są trzy dodatnie liczby całkowite m_1, m_2 *i* n, *oddzielone pojedynczymi odstępami,* $1 \leqslant m_1, m_2 \leqslant 113$, $1 \leqslant n \leqslant 10$. *Liczba* m_1 *to ograniczenie na maksymalną liczbę znaków „−" z rzędu, które mogą pojawiać się w zapisach liczb w danych wejściowych. Liczba* m_2 *to ograniczenie na maksymalną liczbę znaków „−" z rzędu, które mogą pojawiać się w wypisywanych liczbach. Liczba* n *to liczba zapisów liczb, które należy przekształcić. W kolejnych* n *wierszach znajduje się* n *zapisów liczb, po jednym w wierszu. Każdy z tych zapisów nie przekracza* $1\,000$ *znaków.*

Wyjście

Twój program powinien pisać na standardowe wyjście. Powinien on wypisać w kolejnych wierszach kolejne liczby z danych wejściowych przetłumaczone na zapis Przesmyków przy ograniczeniu m_2 na maksymalną liczbę znaków „$-$” z rzędu.

Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
++
--
-+-
```

Rozwiązanie

Analiza problemu

Zauważmy, że problem można sprowadzić do konwersji liczb z zapisu używanego przez Przesmyków na liczby w zwykłym zapisie oraz w drugą stronę. Ustalmy więc, że największa dozwolona liczba minusów pod rząd w zapisie Przesmyków to m (zakładamy, że m jest dodatnie) i spróbujmy znaleźć metodę konwersji.

Policzmy najpierw liczbę ciągów złożonych ze znaków „+” i „$-$” długości n zawierających co najwyżej m znaków „$-$” pod rząd. Oznaczmy tę liczbę L_n. Widzimy od razu, że

$$L_0 = 1,\ L_1 = 2.$$

Aby policzyć liczbę L_n, rozważmy położenie pierwszego plusa w dowolnym ciągu długości n. Może on się znajdować na pozycjach $1, 2, \ldots, m+1$, bo gdyby znajdował się dalej, to występowałoby więcej niż m minusów pod rząd. Zauważmy, że dobrych ciągów, w których pierwszy znak „+” znajduje się na pozycji k, jest dokładnie tyle, ile jest dobrych ciągów zaczynających się za tym znakiem, czyli L_{n-k}. Stąd widzimy, że

$$L_n = L_{n-1} + L_{n-2} + \ldots + L_{n-(m+1)}.$$

Widać, że można ten wzór uprościć, zapisując go dla L_{n-1} i odejmując stronami. Wówczas mamy

$$L_n = 2 \cdot L_{n-1} - L_{n-m-2}.$$

Powyższy wzór można również otrzymać bezpośrednio rozumując tak: każdy ciąg długości n jest przedłużeniem ciągu długości $n-1$ o „+” lub „$-$” na początku. Jedyne złe ciągi, jakie się pojawiają, to ciągi mające $m+1$ minusów na początku a poza tym dobre, a tych jest L_{n-m-2}. Stąd również wynika powyższy wzór.

Załóżmy teraz, że chcemy zamienić liczbę w zapisie Przesmyków na zwykłą liczbę w zapisie dziesiętnym. Jeśli ma ona długość n w zapisie Przesmyków, to z treści zadania wiemy, że jest ona większa od wszystkich liczb mniejszej długości i dodatkowo należy stwierdzić, na jakiej pozycji leksykograficznie znajduje się ona wśród ciągów długości n. Niech $N(n,s)$ oznacza numer, który w porządku leksykograficznym wśród ciągów długości n zajmuje ciąg s. Załóżmy, że s ma pierwszy znak „+" na pozycji k oraz oznaczmy przez s' podciąg s rozpoczynający się od pozycji $k+1$. Pokażemy, że

$$N(n,s) = N(n-k,s') + L_{n-k-1} + L_{n-k-2} + \ldots + L_{n-m-1}.$$

Zauważmy, że w porządku leksykograficznym dla długości n najpierw występują ciągi mające na początku m minusów i pierwszy plus na pozycji $m+1$. Tych ciągów jest L_{n-m-1} i jeśli trafiliśmy na taki ciąg, to wystarczy policzyć pozycję reszty (za pierwszym plusem) dla odpowiedniej długości. Jeśli mamy ciąg o k minusach na początku, to poza policzeniem pozycji reszty trzeba dodać liczbę wszystkich ciągów, które miały więcej minusów, czyli $k+1$ minusy, $k+2$ minusy, itd. aż do m minusów, czyli

$$L_{n-k-1} + L_{n-k-2} + \ldots + L_{n-m-1}.$$

Dzięki temu umiemy już policzyć, jakiej zwykłej liczbie odpowiada ciąg kodujący liczbę Przesmyków — ciąg s długości n odpowiada oczywiście liczbie

$$L_1 + L_2 + \ldots + L_{n-1} + N(n,s).$$

Problemem pozostaje jedynie odwrócenie tego algorytmu, czyli zapisanie ciągu odpowiadającego danej liczbie. Zauważmy, że z powyższego wzoru wynika, że odejmując od danej liczby c kolejne liczby L_k dla $k = 1, 2, \ldots$ tak długo aż po odjęciu kolejnej liczby L_n otrzymamy liczbę ujemną, pozwala nam wyznaczyć n — długość ciągu „+" i „−" odpowiadającego c oraz liczbę $N(n,s)$ dla szukanego ciągu s. Jednak ponieważ

$$N(n,s) = N(n-k,s') + L_{n-k-1} + L_{n-k-2} + \ldots + L_{n-m-1}$$

to w zupełnie analogiczny sposób, odejmując liczby $L_{n-m-1}, L_{n-m}, \ldots$ aż do uzyskania liczby ujemnej, wyznaczamy pozycję k pierwszego znaku „+" w ciągu s oraz liczbę $N(n-k,s')$ dla szukanej reszty s'. Powtarzając to, dochodzimy do ciągów długości 1 i korzystając z tego, że

$$N(1,+) = 1, \; N(1,-) = 0$$

wyznaczamy szukany ciąg s.

Rozwiązanie wzorcowe

W programie wzorcowym liczby L_i są obliczane leniwie, to znaczy dopiero wtedy, gdy są potrzebne, oraz są oczywiście zapamiętywane. Aby je obliczać, trzeba zaimplementować własną arytmetykę. Obliczenia dokonywane są w systemie o podstawie 2^{30}, co gwarantuje szybkie działanie, jednak akceptowane były również rozwiązania wykorzystujące mniejsze podstawy arytmetyki.

Do obliczania liczby $N(n,s)$ dla ciągu znaków $z_1 z_2 \dots z_k$ zastosowano następujące uproszczenie podanego w analizie wzoru:

$$N(z_1 z_2 \dots z_k) = N(z_2 \dots z_k) + \begin{cases} -L_{k-m-2}, & \text{gdy } z_1 = -; \\ L_{k-1} - L_{k-m-2}, & \text{gdy } z_1 = +, \end{cases}$$

który w programie przekłada się na pętlę obliczającą $N(n,s)$.

Przy odwracaniu najpierw jest liczona długość szukanego ciągu, potem stosowana jest następująca metoda. Jeśli $N(z_1 z_2 \dots z_k) < L_{k-1} - L_{k-m-2}$, to $z_1 =$ „$-$" i $N(z_2 \dots z_k) = N(z_1 z_2 \dots z_k) + L_{k-m-2}$. W przeciwnym przypadku $z_1 =$ „$+$" i $N(z_2 \dots z_k) = N(z_1 z_2 \dots z_k) - L_{k-1} + L_{k-m-2}$.

Dla pojedynczego wczytanego ciągu długości k program działa w czasie $O(k^2)$.

Inne rozwiązania

Możliwe są nieco inne metody rozwiązania. Najprostsza metoda polegająca na wyliczeniu wszystkich ciągów miała złożoność wykładniczą i nie była dobrym rozwiązaniem, jednak i tak otrzymywała około 20% punktów. Warto jednak wymienić możliwe usprawnienia. Po pierwsze stosując metodę podaną przy analizie problemu warto pamiętać nie liczby L_i, tylko sumy $L_1 + L_2 + \dots + L_i$. Można dla danego m od razu policzyć tablicę tych liczb, chociaż liczenie ich leniwie jest oszczędniejsze.

Testy

Ciągi w plikach testowych są w większości przypadków losowe. W każdym pliku można znaleźć także ciąg złożony z samych plusów i leksykograficznie pierwszy ciąg dla pewnej ustalonej długości. Testowano ponadto kilka brzegowych przypadków.

nr testu	m_1	m_2	n	zakres długości
1	5	7	6	$[1;10]$
2	6	3	5	$[15;20]$
3	16	13	5	$[40;60]$
4	15	30	10	$[90;100]$
5	21	20	10	$[145;155]$
6	20	23	10	$[180;220]$
7	46	27	10	$[400;500]$
8	100	104	10	$[500;1000]$
9	113	1	10	$[1000;1000]$
10	1	113	10	$[999;1000]$

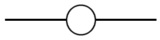
Kolumna *zakres długości* podaje orientacyjnie przedział, w którym są zawarte długości wszystkich ciągów wejściowych.

Marcin Kubica **Marcin Kubica**
Treść zadania, Opracowanie Program

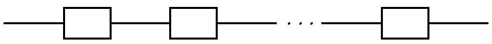
Płytki drukowane

Firma Bajtel rozpoczyna produkcję elektronicznych układów szeregowo-równoległych. Każdy taki układ składa się z części elektronicznych, połączeń między nimi oraz dwóch połączeń doprowadzających prąd. Układ szeregowo-równoległy może składać się z:

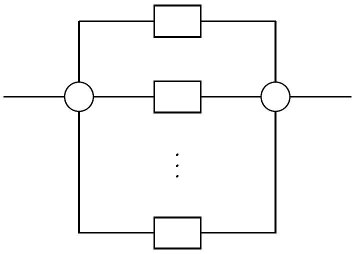
- *pojedynczej części,*



- *kilku mniejszych układów szeregowo-równoległych połączonych szeregowo,*



- *dwóch części rozgałęziających łączących równolegle kilka mniejszych układów szeregowo-równoległych.*



Układy są montowane na dwustronnych płytkach drukowanych. Problem polega na ustaleniu, które połączenia powinny znaleźć się na górnej, a które na dolnej stronie płytki. Ze względów technicznych, jak najwięcej połączeń powinno znaleźć się na dolnej stronie płytki, jednak do każdej części musi dochodzić przynajmniej jedno połączenie znajdujące się na górnej stronie płytki.

Zadanie

Napisz program, który:

- *wczyta opis układu szeregowo-równoległego,*
- *obliczy minimalną liczbę połączeń, jakie muszą znajdować się na górnej stronie płytki,*
- *wypisze wynik.*

Wejście

Ze standardowego wejścia należy wczytać opis układu szeregowo-równoległego. Opis ten ma postać rekurencyjną:

- *jeśli pierwszy wiersz opisu zawiera wielką literę* `S` *oraz dodatnią liczbę całkowitą n ($2 \leqslant n \leqslant 10\,000$) oddzielone pojedynczym odstępem, to opisywany układ składa się z n mniejszych układów połączonych szeregowo, opisanych w kolejnych wierszach,*
- *jeśli pierwszy wiersz opisu zawiera wielką literę* `R` *oraz dodatnią liczbę całkowitą n ($2 \leqslant n \leqslant 10\,000$) oddzielone pojedynczym odstępem, to opisywany układ składa się z n mniejszych układów połączonych równolegle (za pomocą dwóch części rozgałęziających), opisanych w kolejnych wierszach,*
- *wiersz zawierający jedynie wielką literę* `X` *oznacza układ złożony tylko z pojedynczej części.*

Łączna liczba liter `X` *pojawiających się w opisie układu nie przekracza $10\,000\,000$, a głębokość rekurencji w opisie nie przekracza 500.*

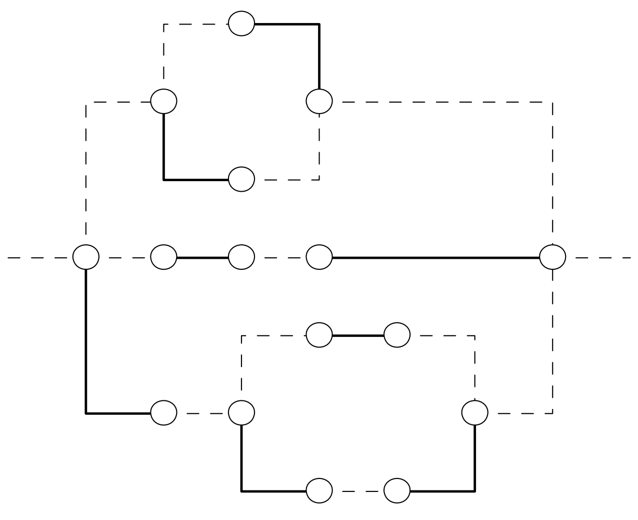
Wyjście

Twój program powinien pisać na standardowe wyjście. W pierwszym wierszu powinna zostać wypisana jedna liczba całkowita, równa minimalnej liczbie połączeń, jakie muszą znaleźć się na górnej stronie płytki.

Przykład

Dla danych wejściowych:

```
R 3
S 2
X
R 2
S 2
X
X
S 2
X
X
S 3
X
X
X
R 2
X
X
```



Schemat układu szeregowo-równoległego dla danych z przykładu. Ciągłą linią zaznaczono połączenia znajdujące się na górnej stronie płytki.

poprawnym wynikiem jest:
8

Rozwiązanie

Układy szeregowo-równoległe mają wyraźnie rekurencyjną strukturę. Narzuca się pytanie, czy można to zadanie rozwiązać rekurencyjnie, zgodnie ze strukturą układu? Niestety rozwiązanie zadania dla podukładów nie daje wystarczających informacji. Oprócz wyniku dla podukładu istotne jest jeszcze, czy połączenia doprowadzające prąd do podukładu znajdują się na górnej czy dolnej stronie płytki. Ponieważ każdy układ ma dwa połączenia doprowadzające prąd, a każde z nich może być po górnej lub dolnej stronie płytki, mamy razem cztery różne sposoby podłączenia podukładu. Możemy uogólnić trochę opisane zadanie i rozwiązać je stosując programowanie dynamiczne. Dla każdego podukładu obliczamy cztery liczby — minimalną liczbę połączeń w podukładzie, jakie muszą być umieszczone na górnej stronie płytki, przy założeniu, że:

- oba połączenia doprowadzające prąd znajdują się na górnej stronie płytki,
- lewe połączenie doprowadzające prąd znajduje się na górnej stronie płytki (o prawym nic nie zakładamy),
- prawe połączenie doprowadzające prąd znajduje się na górnej stronie płytki (o lewym nic nie zakładamy),
- bez żadnych założeń.

Taką czwórkę będziemy oznaczać przez $x = (x_{0,0}, x_{0,1}, x_{1,0}, x_{1,1})$, gdzie 1 w indeksie oznacza, iż zakładamy, że odpowiednie połączenie doprowadzające prąd jest na górnej stronie płytki, a 0 oznacza brak takiego założenia. W programie możemy takie czwórki reprezentować jako tablice:

```
type wynik = array[0..1, 0..1] of longint;
```

Zauważmy, że zachodzą nierówności: $x_{0,0} \leqslant x_{0,1} \leqslant x_{1,1}$, $x_{0,0} \leqslant x_{1,0} \leqslant x_{1,1}$. Jeśli więc obliczymy czwórkę odpowiadającą całemu układowi, to szukany wynik będzie równy $x_{0,0}$.

Czwórka odpowiadająca pojedynczemu elementowi to $(1, 1, 1, 2)$. Odpowiada jej następująca stała:

```
const pojedynczy : wynik = ((1, 1), (1, 2));
```

Zauważmy, że szeregowe łączenie układów jest operacją łączną, tzn. układy A, B i C połączone szeregowo możemy traktować jak szeregowe połączenie układu A i szeregowego połączenia układów B i C lub jak układ szeregowy złożony z A i B połączony dalej z C. Dzięki temu szeregowe połączenie wielu układów możemy sprowadzić do szeregowego łączenia par układów.

Rozważmy dwa układy, A i B, połączone szeregowo. Musimy wziąć pod uwagę dwa przypadki: połączenie łączące te dwa układy jest albo na górnej, albo na dolnej stronie płytki. Jeśli x jest czwórką obliczoną dla układu A, a y jest czwórką obliczoną dla układu B, to czwórkę z obliczoną dla szeregowego połączenia A i B możemy wyznaczyć w następujący sposób:

$$z_{0,0} = \min(x_{0,1} + y_{1,0} - 1, x_{0,0} + y_{0,0})$$

$$z_{0,1} = \min(x_{0,1} + y_{1,1} - 1, x_{0,0} + y_{0,1})$$

$$z_{1,0} = \min(x_{1,1} + y_{1,0} - 1, x_{1,0} + y_{0,0})$$

$$z_{1,1} = \min(x_{1,1} + y_{1,1} - 1, x_{1,0} + y_{0,1})$$

Zależność tę możemy zaimplementować w następujący sposób:

```
function min(x, y : longint) : longint;
begin
  if x < y then return x else return y;
end;

function szeregowo(x, y : wynik) : wynik;
var tymczasowa : wynik;
begin
  tymczasowa[0, 0] := min(x[0,1] + y[1,0] − 1, x[0,0] + y[0,0]);
  tymczasowa[0, 1] := min(x[0,1] + y[1,1] − 1, x[0,0] + y[0,1]);
  tymczasowa[1, 0] := min(x[1,1] + y[1,0] − 1, x[1,0] + y[0,0]);
  tymczasowa[1, 1] := min(x[1,1] + y[1,1] − 1, x[1,0] + y[0,1]);
  return tymczasowa;
end;
```

Układy połączone równolegle przetwarzamy w dwóch krokach. Najpierw obliczamy czwórkę liczb charakteryzujących układ równoległy bez elementów rozgałęziających — odpowiednie elementy czwórki opisują sytuację, gdy choć jeden składowy układ ma lewe/prawe połączenie doprowadzające prąd na górnej stronie płytki. Następnie uwzględniamy elementy rozgałęziające. Dzięki temu, w pierwszym kroku możemy obliczać wynik, dokładając kolejne układy składowe. Jeżeli x jest czwórką obliczoną dla układu A, a y jest czwórką obliczoną dla układu B, to czwórkę z obliczoną dla równoległego połączenia A i B (bez elementów rozgałęziających) możemy określić w następujący sposób:

$$z_{0,0} = x_{0,0} + y_{0,0}$$

$$z_{0,1} = \min(x_{0,1} + y_{0,0}, x_{0,0} + y_{0,1})$$

$$z_{1,0} = \min(x_{1,0} + y_{0,0}, x_{0,0} + y_{1,0})$$

$$z_{1,1} = \min(x_{1,1} + y_{0,0}, x_{0,1} + y_{1,0}, x_{1,0} + y_{0,1}, x_{0,0} + z_{1,1})$$

Zależność tę implementuje następująca funkcja:

```
function równolegle(x, y: wynik) : wynik;
var tymczasowa : wynik;
begin
```

```
   tymczasowa[0, 0] := x[0, 0] + y[0, 0];
   tymczasowa[0, 1] := min(x[0, 1] + y[0, 0], x[0, 0] + y[0, 1]);
   tymczasowa[1, 0] := min(x[1, 0] + y[0, 0], x[0, 0] + y[1, 0]);
   tymczasowa[1, 1] := min(
      min(x[0, 0] + y[1, 1], x[1, 1] + y[0, 0]),
      min(x[1, 0] + y[0, 1], x[0, 1] + y[1, 0])
   );
   return tymczasowa;
end;
```

Załóżmy, że x opisuje równoległe złożenie wszystkich składowych układów. Wówczas możemy uwzględnić elementy rozgałęziające korzystając z następujących zależności:

$$y_{0,0} = x_{1,1}$$

$$y_{0,1} = x_{1,0} + 1$$

$$y_{1,0} = x_{0,1} + 1$$

$$y_{1,1} = x_{0,0} + 2$$

Implementuje to następująca funkcja:

```
function rozgałęzienia(x : wynik) : wynik;
var tymczasowa : wynik;
begin
   tymczasowa[0, 0] := x[1, 1];
   tymczasowa[0, 1] := x[1, 0] + 1;
   tymczasowa[1, 0] := x[0, 1] + 1;
   tymczasowa[1, 1] := x[0, 0] + 2
   return tymczasowa;
end;
```

Zauważmy, że dane wejściowe mają taki format, że nie musimy wczytywać do pamięci opisu układu. Możemy w trakcie wczytywania na bieżąco obliczać czwórki liczb odpowiadających układom. Realizuje to następująca funkcja:

```
function układ: wynik;
var
   ch : char;
   i, n : integer;
   tymczasowa : wynik;
begin
   wczytaj(ch);
   if ch = 'X' then begin
      return pojedynczy;
   end else begin
      wczytaj(n);
      if ch = 'R' then begin
```

```
        tymczasowa := układ();
        for i := 1 to n − 1 do
          tymczasowa := równolegle(tymczasowa, układ());
        return rozgałęzienia(tymczasowa);
      end else begin
        tymczasowa := układ();
        for i := 1 to n − 1 do
          tymczasowa := szeregowo(tymczasowa, układ());
        return tymczasowa;
      end;
    end;
  end;
```

Aby rozwiązać zadanie, pozostaje nam obliczyć czwórkę liczb odpowiadającą całemu układowi i wypisać jej pierwszy element.

```
begin
  wypisz(układ()[0, 0]);
end;
```

Zastanówmy się, jaka jest złożoność tego algorytmu. Zauważmy, że obliczenie wyniku dla układu, na podstawie wyników obliczonych dla jego układów składowych, wymaga liczby operacji proporcjonalnej do liczby układów składowych. Tak więc złożoność czasowa całego algorytmu jest rzędu $\Theta(n)$, gdzie n to liczba elementów w układzie. Dzięki temu, że nie trzymamy w pamięci opisu całego układu i obliczamy wyniki dla podukładów na bieżąco, w trakcie wczytywania ich opisów, złożoność pamięciowa jest takiego rzędu, jak głębokość rekurencji w wywołaniach funkcji *układ*, a ta nie przekracza 500.

Testy

Rozwiązania zawodników były sprawdzane na testach od małych do bardzo dużych wymiarów. Testy te można znaleźć na załączonym dysku. W tabelce poniżej podano liczby elementów w testowych układach. Dwa pierwsze testy to niewielkie testy sprawdzające poprawność rozwiązań. Kolejne dwa testy sprawdzają zachowanie rozwiązań dla układów o głębokim zagnieżdżeniu rekurencyjnym opisów — odpowiednio 300 i 500. Test nr 5 to test, w którym relatywnie niewiele połączeń powinno znaleźć się na górnej stronie płytki. Kolejne cztery testy to testy wydajnościowe — losowe dane coraz większych rozmiarów. Ostatni test to duży zestaw danych opisujących mocno rozgałęziony układ.

nr testu	liczba elementów
1	20
2	22
3	7841
4	13841
5	85828
6	100000
7	1000000
8	10000000
9	10000000
10	9973426

Mimo, iż zadanie to wydaje się stosunkowo proste, to wielu zawodnikom sprawiło trudności. Spośród wszystkich zadań pierwszego etapu, zawodnicy nadesłali najmniej rozwiązań właśnie tego zadania. Mniej więcej połowa rozwiązań przeszła testy poprawnościowe, natomiast tylko nieliczne rozwiązania przeszły testy wydajnościowe. W rezultacie, zdecydowana większość rozwiązań dostała nie więcej niż 50% punktów, a nieliczna część rozwiązań otrzymała pełną liczbę punktów. Jak widać na przykładzie tego zadania, poprawne rozwiązanie zadania to dopiero połowa sukcesu. Pełen sukces gwarantuje dopiero rozwiązanie poprawne i efektywne.

Łukasz Kowalik Treść zadania, Opracowanie

Paweł Wolff Program

Przemytnicy

Bajtocja słynie z bogatych złóż złota, dlatego przez długie lata kwitła sprzedaż tego kruszcu do sąsiedniego królestwa, Bitlandii. Niestety powiększająca się ostatnio dziura budżetowa zmusiła króla Bitlandii do wprowadzenia zaporowych ceł na metale i minerały. Handlarze przekraczający granicę muszą zapłacić 50% wartości przewożonego ładunku. Bajtockim kupcom grozi bankructwo. Na szczęście bajtoccy alchemicy opracowali sposoby pozwalające zamieniać pewne metale w inne. Pomysł kupców polega na tym, aby z pomocą alchemików zamieniać złoto w pewien tani metal, a następnie, po przewiezieniu go przez granicę i zapłaceniu niewielkiego cła, znowu otrzymywać z niego złoto. Niestety alchemicy nie znaleźli sposobu na zamianę dowolnego metalu w dowolny inny. Może się więc zdarzyć, że proces otrzymania danego metalu ze złota musi przebiegać wielostopniowo i że na każdym etapie uzyskiwany będzie inny metal. Alchemicy każą sobie słono płacić za swoje usługi i dla każdego znanego sobie procesu zamiany metalu A w metal B wyznaczyli cenę za przemianę 1 kg surowca. Handlarze zastanawiają się, w jakiej postaci należy przewozić złoto przez granicę oraz jaki ciąg procesów alchemicznych należy zastosować, aby zyski były możliwie największe.

Zadanie

Pomóż uzdrowić bajtocką gospodarkę! Napisz program, który:

- *Wczyta tabelę cen wszystkich metali, a także ceny przemian oferowanych przez alchemików.*
- *Wyznaczy taki ciąg metali $m_0, m_1, \ldots, m_k$, że:*
  - *$m_0 = m_k$ to złoto,*
  - *dla każdego $i = 1, 2, \ldots, k$ alchemicy potrafią otrzymać metal m_i z metalu m_{i-1}, oraz*
  - *koszt wykonania całego ciągu procesów alchemicznych dla 1 kg złota powiększony o płacone na granicy cło (50% ceny 1 kg najtańszego z metali m_i, dla $i = 0, 1, \ldots, k$) jest najmniejszy z możliwych.*

  Zakładamy, że podczas procesów alchemicznych waga metali nie zmienia się.
- *Wypisze koszt wykonania wyznaczonego ciągu procesów alchemicznych powiększony o płacone na granicy cło.*

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna dodatnia liczba całkowita n oznaczająca liczbę rodzajów metali, $1 \leqslant n \leqslant 5\,000$. W wierszu o numerze $k+1$, dla $1 \leqslant k \leqslant n$, znajduje się nieujemna parzysta liczba całkowita p_k — cena 1 kg metalu oznaczonego numerem k, $0 \leqslant p_k \leqslant 10^9$. Przyjmujemy, że złoto ma numer 1. W wierszu *o numerze $n + 2$ znajduje się jedna nieujemna liczba całkowita m równa liczbie procesów przemiany znanych alchemikom, $0 \leqslant m \leqslant 100\ 000$. W każdym z kolejnych m wierszy znajdują się po trzy liczby naturalne, pooddzielane pojedynczymi odstępami, opisujące kolejne procesy przemiany. Trójka liczb a, b, c oznacza, że alchemicy potrafią z metalu o numerze a otrzymywać metal o numerze b i za zamianę 1 kg surowca każą sobie płacić c bajtalarów, $1 \leqslant a, b \leqslant n$, $0 \leqslant c \leqslant 10\ 000$. Uporządkowana para liczb a i b może się pojawić w danych co najwyżej jeden raz.*

Wyjście

Twój program powinien pisać na standardowe wyjście. W pierwszym wierszu powinna zostać wypisana jedna liczba całkowita — koszt wykonania wyznaczonego ciągu procesów alchemicznych powiększony o płacone na granicy cło.

Przykład

Dla danych wejściowych:

```
4
200
100
40
2
6
1 2 10
1 3 5
2 1 25
3 2 10
3 4 5
4 1 50
```

poprawnym wynikiem jest:

```
60
```

Rozwiązanie

Abstrakcyjne sformułowanie problemu

Nie trzeba wielkiej przenikliwości, żeby odkryć, że na dane do tego zadania można patrzeć jak na pewien graf. Wierzchołkami w tym grafie są metale m_1, m_2, ..., m_n. Graf jest skierowany, tzn. wierzchołki są połączone strzałkami. Nasz graf zawiera krawędź od wierzchołka u do wierzchołka v, gdy alchemicy potrafią przemieniać metal u w metal v. Każdej takiej krawędzi przypisano liczbę całkowitą równą cenie przemiany. Liczby, które przypisujemy krawędziom grafu są często nazywane *wagami krawędzi*. W tym przypadku wierzchołki również mają wagi, równe połowie ceny odpowiedniego metalu.

Ścieżka w grafie skierowanym to dowolny ciąg wierzchołków v_0, v_1, …, v_k taki, że dla każdego $i = 1, \ldots, k$ graf zawiera krawędź $v_{i-1} \to v_i$. *Wagą ścieżki* nazywamy sumę wag jej krawędzi. Często zamiast pisać „ścieżka o najmniejszej wadze” będziemy używać określenia *najkrótsza ścieżka*.

Teraz możemy zacząć zastanawiać się, jak rozwiązać zadanie. Chcemy zamieniać złoto w pewien metal m_i, przewozić je przez granicę, a następnie znowu otrzymywać złoto. Taką operację należy przeprowadzać w sposób jak najmniej kosztowny. Spróbujemy następującego podejścia: dla każdego metalu m_i policzymy, jak najtaniej przemieniać złoto w m_i oraz jak najtaniej przemieniać m_i w złoto. Koszty obu przemian powiększone o cło za metal m_i dadzą nam koszt całej operacji i będziemy mogli wybrać tę najtańszą.

Co to oznacza w języku teorii grafów? Proces przemiany jednego metalu w drugi to po prostu pewna ścieżka w naszym grafie. Koszt takiego procesu to nic innego jak waga ścieżki. Dla każdego wierzchołka m_i musimy więc znaleźć:

- najkrótszą ścieżkę od m_1 (złoto) do m_i,
- najkrótszą ścieżkę od m_i do m_1.

Potem wystarczy już tylko znaleźć taki metal m_k $(1 \leqslant k \leqslant n)$, że suma wag obu ścieżek dla m_k i wagi wierzchołka m_k jest najmniejsza. Zauważmy, że może się zdażyć, że najtaniej jest przewozić przez granicę złoto!

Droga do celu

Widzimy teraz, że aby rozwiązać zadanie musimy umieć znajdować najkrótsze ścieżki w grafie. Możemy użyć w tym celu algorytmu Dijkstry. Był on już wielokrotnie prezentowany w opracowaniach zadań olimpijskich ([4] s. 52, [8] s. 108–109), nie będziemy go więc omawiać. Jego opis wraz z dokładną analizą złożoności można znaleźć także w książce Cormena, Leisersona i Rivesta ([15]).

Istotną cechą algorytmu Dijkstry jest fakt, że znajduje on najkrótsze ścieżki (w sensie sumy wag) od jednego wybranego wierzchołka nazywanego *źródłem* do wszystkich pozostałych wierzchołków grafu. Za wierzchołek źródłowy możemy przyjąć m_1. A więc połowę pracy mamy już za sobą! Teraz wystarczy znaleźć najkrótsze ścieżki *od* wszystkich wierzchołków do m_1. Moglibyśmy uruchamiać algorytm Dijkstry jeszcze n razy, za każdym razem za źródło przyjmując inny wierzchołek. Okazuje się jednak, że i tym razem wystarczy tylko *jedno* uruchomienie algorytmu Dijkstry. Należy tylko wcześniej odwrócić wszystkie krawędzie w grafie. Innymi słowy, w nowym, *odwróconym grafie* istnieje krawędź od wierzchołka u do wierzchołka v, gdy w pierwotnym grafie mieliśmy krawędź od v do u. Najkrótsza ścieżka od m_1 do v w grafie odwróconym będzie najkrótszą ścieżką od v do m_1 w pierwotnym grafie. Ponownie uruchamiamy więc algorytm Dijkstry ze źródłem w wierzchołku m_1, tyle że w grafie odwróconym.

Złożoność czasowa i pamięciowa

Złożoność czasowa przedstawionego algorytmu jest zdominowana przez złożoność algorytmu Dijkstry (który jest uruchamiany dwukrotnie), ponieważ wszystkie pozostałe operacje wykonywane przez algorytm (łącznie z wczytaniem danych i wypisaniem wyniku) zajmują czas $O(n+m)$. W programie wzorcowym zastosowano implementację algorytmu Dijkstry działającą w czasie $O(n^2)$. Złożoność pamięciowa naszego algorytmu wynosi natomiast $O(n+m)$.

Inne rozwiązania

Inne rozwiązanie o tej samej złożoności

Rozwiązanie o takiej samej złożoności, jak rozwiązanie wzorcowe, możemy otrzymać uruchamiając algorytm Dijkstry tylko raz, ale dla nieco innego grafu. Oznaczmy wierzchołki tego grafu przez $m_1, m_2, \ldots, m_n$ oraz $m'_1, m'_2, \ldots, m'_n$. Przemiana metalu o numerze a w metal o numerze b kosztująca c bajtalarów będzie reprezentowana przez dwie krawędzie:

- krawędź z m_a do m_b,
- krawędź z m'_a do m'_b

obydwie o wadze c. Ponadto, dla każdego $k = 1, 2, \ldots, n$, między wierzchołkami m_k i m'_k dodajemy krawędź o wadze równej cłu za przewóz metalu o numerze k. W tak zdefiniowanym grafie wystarczy znaleźć ścieżkę o najmniejszej wadze od wierzchołka m_1 do m'_1.

Na powyższą konstrukcję możemy patrzeć jak na sumę dwóch grafów używanych w rozwiązaniu wzorcowym, do której dodajemy krawędzie łączące odpowiadające sobie wierzchołki. Oba rozwiązania są więc w istocie bardzo podobne.

Dalsze optymalizacje

Algorytm Dijkstry korzysta ze struktury danych nazywanej kolejką priorytetową. Jest to struktura przechowująca zbiór elementów (w naszym przypadku zbiór wierzchołków grafu). Z każdym elementem związany jest klucz (w naszym przypadku liczba całkowita). Struktura umożliwia dodawanie elementów, zwrócenie elementu o najmniejszym kluczu i usunięcie go ze zbioru. Udostępnia także operację zmniejszenia wartości klucza dla wybranego elementu. W programie wzorcowym kolejka priorytetowa została zaimplementowana jako zwykła tablica. Jeśli jednak zastosujemy do tego celu kopce binarne, nasz algorytm będzie działał w czasie $O((m+n)\log n)$, co jest istotnym przyspieszeniem. Stosując jeszcze bardziej zaawansowane struktury danych możemy otrzymać nawet czas $O(n\log n + m)$, chociaż nie jest jasne, czy dla danych zawartych w testach takie rozwiązanie działałoby najszybciej. Wszystkie te warianty implementacji są doskonale opisane w literaturze, gorąco zachęcamy do ich studiowania!

Rozwiązanie o złożoności $O(n^3)$

Inne rozwiązanie zadania mogłoby opierać się na algorytmie Floyda-Warshalla (zobacz np. [1] s. 117-118 albo w książce [15] s. 627) znajdującym najkrótsze ścieżki między *wszystkimi* parami wierzchołków. To rozwiązanie jest o tyle kuszące, że posiada wyjątkowo prostą implementację. Niestety jego złożoność czasowa wynosi $O(n^3)$, natomiast złożoność pamięciowa jest rzędu $O(n^2)$. Takie rozwiązania nie poradziłyby sobie z większością testów.

Testy

Testy 1a, 1b, 1c, 2, 3, 4 miały za zadanie jedynie sprawdzać poprawność rozwiązań. W szczególności przy danych z testu 2 najtaniej jest przewozić przez granicę złoto.

Testy 5–10 miały na celu sprawdzenie wydajności rozwiązań. Graf z testu 5 jest pojedynczym cyklem skierowanym, w teście 6 zapisano dwa cykle przebiegające wszystkie wierzchołki, test 7 zawiera graf pełny z losowymi wagami, natomiast testy 8–10 zawierają duże, losowo utworzone grafy. W tabeli poniżej podano rozmiary testów.

nr testu	n	m
1a	1	0
1b	2	1
1c	3	2
2	3	6
3	5	15
4	7	17
5	2000	2000
6	1000	2000
7	100	9900
8	1000	5000
9	2000	20000
10	5000	100000

Zawody II stopnia

Zawody II stopnia — opracowania zadań

Zbigniew Czech	**Michał Adamaszek, Krzysztof Onak**
Treść zadania, Opracowanie	Programy

Mastermind II

Będziemy rozważali ciągi, które spełniają następujące warunki:

- *długość ciągu wynosi u,*
- *elementami ciągu są cyfry z zakresu 1–9,*
- *wyrazy ciągu się nie powtarzają.*

Pojedynczy ciąg będziemy nazywali **układem**.

Mając dane dwa układy, ich zgodność oceniamy podając dwie liczby. Pierwsza z nich (kolumna A w przykładzie) to suma cyfr, które występują w obu układach i znajdują się na tej samej pozycji w obu ciągach, natomiast druga (kolumna B) to suma cyfr, które występują w obu układach, ale znajdują się na różnych pozycjach.

Mamy dane u układów i podane oceny ich zgodności z pewnym nieznanym układem. Należy podać nieznany układ. Poniżej przedstawiono przykładowe dane i wynik dla $u = 3$.

A	B			
4	0	4	9	7
0	10	6	7	4
0	5	9	4	1

nieznany układ:

4	1	6

Zadanie

Napisz program, który:

- *wczyta opis układów i oceny ich zgodności,*
- *znajdzie układ spełniający warunki zadania,*
- *wypisze wynik.*

Wejście

Twój program powinien czytać opis ze standardowego wejścia. W pierwszym wierszu zapisana jest jedna liczba całkowita u, $1 \leqslant u \leqslant 9$. W kolejnych u wierszach opisane są podane układy cyfr oraz ocena ich zgodności z pewnym nieznanym układem, po jednym w wierszu. W każdym z tych wierszy zapisanych jest po $u + 2$ nieujemnych liczb całkowitych pooddzielanych pojedynczymi odstępami. Pierwsza i druga liczba są oceną zgodności danego układu z nieznanym układem. Ostatnie u liczb to różne cyfry z zakresu 1–9 tworzące dany układ.

Wyjście

Twój program powinien wypisać na standardowe wyjście u różnych cyfr z zakresu 1–9 tworzących poszukiwany układ, oddzielonych pojedynczymi odstępami.

Możesz założyć, że dla danych testowych istnieje co najmniej jedno rozwiązanie. Jeżeli dla danych wejściowych istnieje wiele pasujących układów, Twój program powinien wypisać tylko jeden z nich.

Przykład

Dla danych wejściowych:

```
3
4 0 4 9 7
0 10 6 7 4
0 5 9 4 1
```

poprawnym wynikiem jest:

```
4 1 6
```

Rozwiązanie

Nieznany układ składa się z niepowtarzających się cyfr z zakresu 1–9. Liczba wszystkich możliwych takich układów wynosi $9 \cdot 8 \cdot \ldots \cdot (9-u+1) = 9!/(9-u)!$. Układy te, to tzw. wariacje bez powtórzeń. Zadanie można rozwiązać generując wszystkie możliwe wariacje i sprawdzając dla każdej z nich, czy jest ona zgodna z układami wejściowymi. Ten sposób rozwiązania został zaimplementowany w programie `mas1.cpp`. Łatwo widać, że złożoność takiego rozwiązania jest $O(9!u^2/(9-u)!)$. Omówione rozwiązanie można ulepszyć, przerywając obliczenia w momencie znalezienia poszukiwanej wariacji. Ponadto w trakcie budowania wariacji, tj. po każdorazowym jej rozszerzeniu o kolejną pozycję, można uaktualniać bieżące parametry zgodności budowanej wariacji z układami wejściowymi. Pozwala to na zaprzestanie dalszego rozszerzania bieżącej wariacji i przejście do następnej w momencie, gdy parametry zgodności z którymś z układów wejściowych przekroczą wymagane w danych wejściowych wartości. Ulepszenia te zostały zastosowane w programie `mas2.cpp`. Badania eksperymentalne pokazały, że dla przygotowanych danych testowych rozwiązanie to jest istotnie szybsze od poprzedniego rozwiązania.

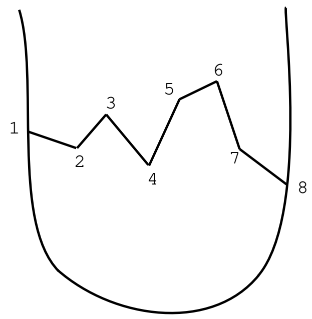
Testy

Dane testowe zawarte w plikach `mas1.in`–`mas10.in` zostały wygenerowane w sposób losowy.

Tomasz Waleń | **Arkadiusz Paterek**
Treść zadania, Opracowanie | Program

Autostrady

Bajtocja leży na półwyspie. Już od czasów króla Bitola podstawową formą komunikacji w Bajtocji jest transport kolejowy. Król Bitol wybudował jedną super szybką linię kolejową łączącą wschodnie i zachodnie wybrzeże półwyspu. Linia kolejowa przechodzi przez wszystkie miasta Bajtocji, wyznaczając ich numerację — pierwsze miasto na linii ma numer 1, a ostatnie n. Miasto nr 1 leży na zachodnim, a nr n na wschodnim wybrzeżu.



Rys. 1: Sieć kolejowa Bajtocji.

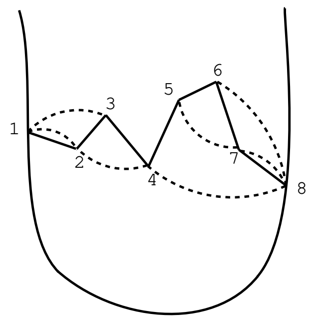
W ostatnich latach, dzięki ministrowi Bajterowiczowi, gospodarka Bajtocji rozwinęła się bardzo gwałtownie i obecna sieć komunikacyjna wymaga szybkiej modernizacji. Król Bajtol zarządził (w ramach kolejnego planu 2^3*-letniego) budowę wielu autostrad. Każda z autostrad ma łączyć bezpośrednio dwa wybrane miasta Bajtocji. Ze względu na to, że każda autostrada będzie budowana przez oddzielną agencję rządową i na każdej będzie obowiązywał inny rodzaj winiet, zdecydowano, że autostrady nie mogą przecinać się same ze sobą, ani też nie mogą przecinać linii kolejowej. Stąd jedyną możliwością jest zbudowanie autostrad po północnej lub południowej stronie linii kolejowej. Na rysunku 2 przedstawiono przykładowy plan autostrad (autostrady są zaznaczone łukami, a linia kolejowa to łamana składająca się z odcinków).*

Najjaśniejszy król Bajtol zdecydował już jakie pary miast mają zostać połączone autostradami. Każda z autostrad opisana jest przez parę miast, które ma łączyć. Twoim zadaniem jest ustalenie dla danego zestawu połączeń, które z autostrad powinny leżeć na północ od linii kolejowej, a które na południe. Pamiętaj jednak, że autostrady nie mogą się wzajemnie przecinać, ani też przecinać linii kolejowej.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia informację o planowanych autostradach,*
- *wyznaczy rozmieszczenie autostrad (lub stwierdzi, że nie da się ich zbudować),*



Rys. 2: Przykładowy plan autostrad łączących miasta: 1–2, 1–3, 2–4, 5–7, 4–8, 7–8, 6–8.

- *zapisze wynik na standardowym wyjściu.*

Limit pamięci dla tego zadania wynosi 32 MB.

Wejście

W pierwszym wierszu standardowego wejścia zapisane są dwie liczby całkowite — liczba miast n i liczba planowanych autostrad k, $1 \leqslant n, k \leqslant 20\,000$. W kolejnych k wierszach zapisane są pary miast, które mają zostać połączone autostradami. W wierszu $i + 1$ zapisane są dwie liczby całkowite p_i, q_i oddzielone pojedynczym odstępem — numery miast, które ma połączyć i-ta autostrada, $1 \leqslant p_i < q_i \leqslant n$. Pary miast w danych wejściowych nie powtarzają się.

Wyjście

Twój program powinien wypisać na standardowe wyjście plan budowy autostrad lub pojedyncze słowo `NIE`*, jeśli nie jest możliwe zbudowanie wszystkich autostrad. Jeśli budowa autostrad jest możliwa, to na standardowe wyjście należy wypisać k wierszy. W i-tym wierszu należy wypisać jedną wielką literę, odpowiednio* `N` *— jeśli autostrada łącząca miasta p_i i q_i ma zostać zbudowana na północ od linii kolejowej — lub* `S` *— jeśli na południe od linii kolejowej. Jeśli istnieje wiele możliwych rozwiązań, Twój program powinien wypisać tylko jedno z nich.*

Przykład

Dla danych wejściowych:

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

```
6 8
```

poprawnym wynikiem jest:

```
N
N
S
S
S
N
N
```

Rozwiązanie

Zadanie ma bardzo prosty odpowiednik w algorytmach grafowych. Każdej autostradzie należy przypisać jeden wierzchołek grafu, dwa wierzchołki grafu połączone są krawędzią wtedy i tylko wtedy, gdy nie mogą zostać umieszczone po tej samej stronie linii kolejowej. Taki graf nazywamy grafem przecięć.

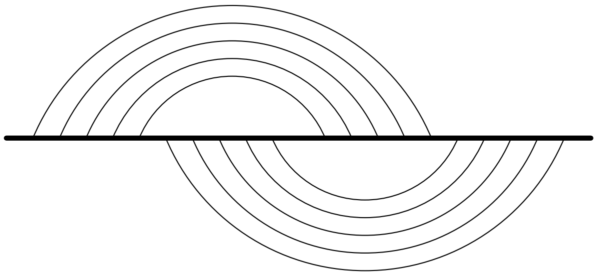
Mając tak zbudowany graf, problem rozmieszczenia autostrad sprowadza się do pokolorowania grafu dwoma kolorami (lub stwierdzenia, że nie jest to możliwe).

Problem kolorowania grafu dwoma kolorami można rozwiązać stosując np. przeszukiwanie grafu w głąb (DFS). Przykładowa implementacja tego algorytmu z użyciem stosu ma postać:

```
function DFS(v0);
begin
  stos := {v0};
  kolor[v0] := "biały";
  { wszystkie pozostałe wierzchołki mają nieustalony kolor }
  while stos ≠ ∅ do
    v := stos.pop();
    for u ∈ sąsiedzi(v) do
      if kolor[u] = "?" then
        begin
          kolor[u] := { kolor przeciwny do koloru v };
          stos.push(u);
        end;
      else
        if kolor[u]=kolor[v] then
          { Błąd, nie można rozmieścić autostrad };
  end;
end;
```

Niestety graf przecięć może być dosyć duży. Może zdarzyć się, że będzie miał nawet $O(k^2)$ krawędzi, gdzie k to liczba autostrad. Stąd cały algorytm miałby w pesymistycznym przypadku koszt czasowy $O(k^2)$.

Na rysunku 3 przedstawione są dane dla których graf przecięć jest bardzo duży.



Rysunek 3: Przykładowe dane dla których graf przecięć ma $O(k^2)$ krawędzi

Rozwiązanie wzorcowe

Należy zauważyć, że nie zawsze konieczne jest przeglądanie całego grafu. Niekiedy wystarczy przeglądnięcie części krawędzi oraz późniejsza weryfikacja, czy opuszczone krawędzie nie powodują błędu.

Na tym spostrzeżeniu opiera się rozwiązanie wzorcowe, jednak aby efektywnie operować na grafie przecięć, potrzebna jest struktura danych udostępniająca następujące operacje:

- *wstaw*(l, r) — wstawienie do struktury danych autostrady o końcach l i r,
- *usuń*(l, r) — usunięcie ze struktury autostrady o końcach l i r,
- *najdalejNaPrawo*(l, r) — zwraca autostradę (a, b) o prawym końcu b położonym najbardziej na prawo i taką, że lewy koniec spełnia nierówność $l < a < r$,
- *najdalejNaLewo*(l, r) — operacja analogiczna do *najbardziejNaPrawo*, z tym, że poszukiwana jest autostrada o minimalnym lewym końcu i takim, że prawy koniec spełnia nierówność $l < b < r$.

Wszystkie powyższe operacje można zaimplementować w czasie $O(\log k)$ używając drzew zrównoważonych (np. AVL lub czerwono-czarnych). Ponieważ w algorytmie wszystkie operacje wstawiania odbywają się na początku, stąd zamiast drzew zrównoważonych można też użyć odpowiednio przygotowanych zwykłych drzew BST.

```
{ s — struktura danych zawierająca wszystkie autostrady }
{ na początku wszystkie wierzchołki mają nieustalony kolor }
for v0 ∈ V do if kolor[v0] = "?" then begin
    stos := {v0};
    kolor[v0] := "biały";
    while stos ≠ ∅ do begin
        v := stos.pop();
        repeat
            u := s.najdalejNaPrawo(v.a, v.b);
            if { u przecina v } then begin
                s.usuń(u);
```

```
            stos.push(u);
            kolor[u] := { kolor przeciwny do koloru v };
         end;
      until { u nie przecina v };

      { analogiczne wyszukiwanie przy użyciu operacji najdalejNaLewo() }
   end;
end;
{ Weryfikacja odpowiedzi }
```

Weryfikacja odpowiedzi na końcu algorytmu jest konieczna, ponieważ znalezione rozwiązanie może nie być poprawne (właśnie dzięki krawędziom, które zostały ominięte poprzez usuwanie wierzchołków ze struktury). Na szczęście taka sytuacja może zajść jedynie, gdy nie istnieje rozwiązanie zgodne z warunkami zadania. Weryfikację odpowiedzi można wykonać w czasie $O(k)$.

Nowy algorytm ma koszt $O(k \log k)$, gdyż można co najwyżej wykonać k operacji *usuń* oraz co najwyżej $2k$ operacji *najdalejNaPrawo* i *najdalejNaLewo*.

Testy

Zadanie sprawdzane było na zestawie 12 danych testowych. Testy numer 1 i 1a oraz 10 i 10a były zgrupowane. Dla testów numer 1a i 10a nie było możliwe wybudowanie autostrad zgodnie z warunkami zadania.

nr testu	n	k
1	12	8
1a	6	3
2	10000	14
3	200	22
4	1000	1001
5	10000	9948
6	20000	19957
7	20000	10219
8	20000	19802
9	20000	11018
10	20000	9030
10a	20000	9031

Krzysztof Sikora — Treść zadania, Opracowanie
Wojciech Dudek — Program

Trójmian[1]

Rozważmy trójmian $(x^2+x+1)^n$*. Interesują nas współczynniki* c_i *rozwinięcia tego trójmianu:*

$$c_0 + c_1x + c_2x^2 + \cdots + c_{2n}x^{2n}$$

Na przykład, $(x^2 + x + 1)^3 = 1 + 3x + 6x^2 + 7x^3 + 6x^4 + 3x^5 + x^6$.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia zestawy danych, w których są liczby* n *oraz* i,
- *dla każdego zestawu obliczy resztę z dzielenia przez 3 współczynnika* c_i *stojącego przy* x^i *w rozwinięciu trójmianu* $(x^2+x+1)^n$,
- *dla każdego zestawu wypisze na standardowe wyjście obliczoną resztę.*

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita k *określająca liczbę zestawów danych,* $1 \leqslant k \leqslant 10\,000$*. Po niej następuje* k *zestawów danych, po jednym w wierszu. W każdym zestawie danych zapisane są dwie nieujemne liczby całkowite* n *oraz* i *oddzielone pojedynczym odstępem,* $0 \leqslant n \leqslant 1\,000\,000\,000\,000\,000$, $0 \leqslant i \leqslant 2n$.

Wyjście

Na standardowe wyjście należy wypisać k *wierszy. Wiersz* j*-ty powinien zawierać jedną dodatnią liczbę całkowitą będąca resztą z dzielenia* c_i *przez* 3 *dla liczb z* j*-tego zestawu.*

Przykład

Dla danych wejściowych:

```
5
2 0
7 4
4 5
5 3
8 15
```

[1] W rzeczywistości w tym zadaniu nie mamy do czynienia z trójmianami. Przyjęliśmy tę nazwę przez analogię do dwumianu Newtona $(a+b)^n$.

poprawnym wynikiem jest:

```
1
2
1
0
2
```

Rozwiązanie

Niech $R(m,n)$ będzie współczynnikiem (modulo 3) stojącym przy x^n w rozwinięciu wielomianu $W_m(x) = (x^2+x+1)^m$ oraz niech $S(n) = \sum_{i=0}^{n} 3^i$.
Zachodzą następujące własności dla $i \in \mathbb{N}$:

$$S(i) + 3^{i+1} = S(i+1) \tag{1}$$

$$3 \cdot S(i) + 1 = S(i+1) \tag{2}$$

$$3^i \leqslant S(i) < 3^{i+1} < S(i+1) \tag{3}$$

$$3^{i+1} = 2 \cdot S(i) + 1 \tag{4}$$

oraz dla $m = k + l$ gdzie $k, l \in \mathbb{N}$

$$R(m,n) = \sum_{i=0}^{n} R(k,i) \cdot R(l, n-i). \tag{5}$$

Z faktu, że współczynniki trójmianu $x^2 + x + 1$ są symetryczne, wynika symetria współczynników wielomianu $W_m(x)$. Mamy zatem

$$R(m,n) = R(m, 2m-n). \tag{6}$$

Współczynniki $R(m,n)$ tworzą figurę podobną do trójkąta Sierpińskiego. Można zauważyć, że w naszym trójkącie znajdują się dwa rodzaje wzajemnie rekurencyjnych wzorów, nazwijmy je X i Y, które można zobaczyć na rysunkach 1 i 2.

Program wzorcowy wykorzystuje właśnie tę własność. Pozwala ona w co najwyżej dwóch krokach zredukować problem do przypadku trzykrotnie mniejszego. Otrzymujemy więc następujące równanie na złożoność czasową i pamięciową obliczenia dla pojedyńczego zestawu danych

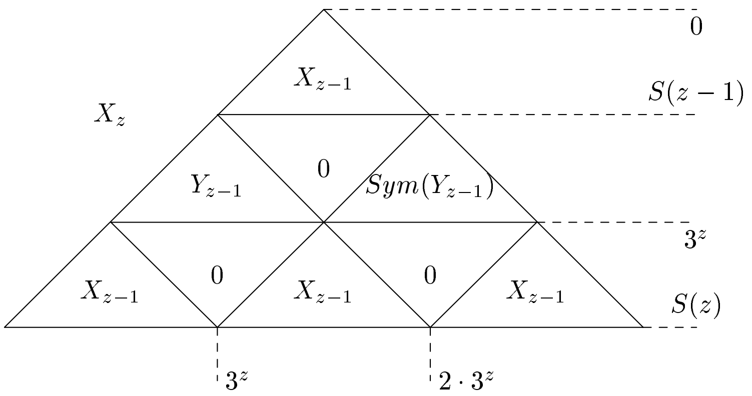
$$T(n) = T(\lfloor \frac{n}{3} \rfloor) + O(1).$$

Rozwiązując powyższą rekurencję otrzymujemy, że $T(n) = O(\log n)$. Zatem cały program działa w czasie $O(k \log n)$ i wykorzystuje $O(\log n)$ pamięci (stos dla rekurencji).

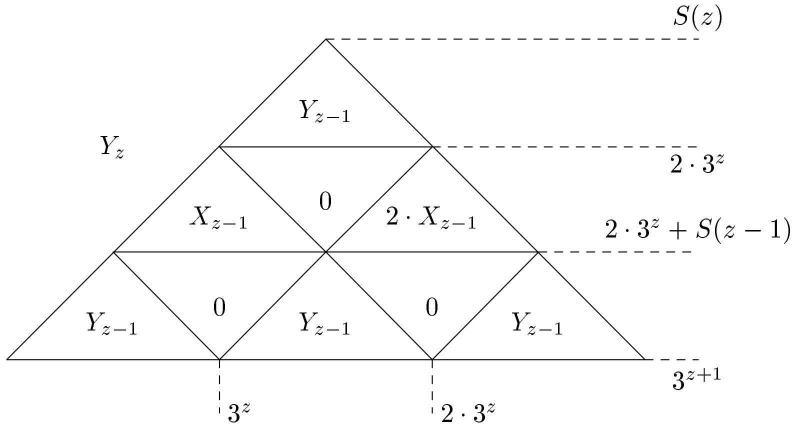
Na koniec udowodnimy zależność między X i Y. Pomocne do tego będą dwie własności, których udowodnienie pozostawiamy Czytelnikowi jako pouczające ćwiczenie.

$$R(3^i, x) = \begin{cases} 1 & \text{dla } x \text{ postaci } 0,\ 3^i \text{ lub } 2 \cdot 3^i; \\ 0 & \text{wpp.} \end{cases} \tag{7}$$

$$R(2 \cdot 3^i, x) = \begin{cases} 1 & \text{dla } x = 0 \text{ lub } x \text{ postaci } 4 \cdot 3^i; \\ 2 & \text{dla } x \text{ postaci } 3^i \text{ lub } 3 \cdot 3^i; \\ 0 & \text{wpp.} \end{cases} \tag{8}$$



Rysunek 1: Wzór X_z



Rysunek 2: Wzór Y_z

Nie będziemy rozpatrywać wszystkich przypadków, gdyż są one podobne i dowodzi się je analogicznie. Pokażemy jeden wybrany.

Lemat 1 *Prawy trójkąt w trzeciej warstwie figury X_z to X_{z-1}.*

Dowód Trzecia warstwa zaczyna się od wiersza 3^z, a prawy trójkąt od kolumny $2 \cdot 3^z$. Otrzymujemy więc

$$R(m,n) = R(3^z + m', 2 \cdot 3^z + n'),$$

gdzie $m' < S(z-1)$ i $n' < 3^z$. Następnie korzystamy z własności 5 i 7:

$$R(m,n) = R(m', 2 \cdot 3^z + n') + R(m', 2 \cdot 3^z + n' - 3^z) + R(m', 2 \cdot 3^z + n' - 2 \cdot 3^z) = R(m', n').$$

Pierwszy i drugi składnik są równe 0. Wynika to z własności 4 i z faktu, że stopień $W_{S(z-1)}(x)$ jest nie większy niż $2 \cdot S(z-1)$. ■

Testy

Zadanie sprawdzane było na zestawie 10 danych testowych.

- `tro1.in` — pierwszych 14 linii trójkąta
- `tro2.in` — pierwszych 27 linii trójkąta
- `tro3.in` — $k = 1000, m = 1001$, n losowe
- `tro4.in` — $k = 1000$, m i n małe losowe
- `tro5.in` — $k = 5000$, $m \leqslant 10^7$
- `tro6.in` — $k = 3000$, $m \leqslant 10^9$
- `tro7.in` — $k = 5000$, $m \leqslant 10^{11}$
- `tro8.in` — $k = 10000$, $m \leqslant 10^{15}$, $n = S(i)\ mod\ (2m)$
- `tro9.in` — $k = 10000$, $m \leqslant 10^{15}$

Rafał Rusin	Rafał Rusin, Paweł Wolff	Paweł Wolff
Treść zadania	Opracowanie	Program

Kafelki

Majster Bajtazar wraz ze swym pomocnikiem Bajtolinim układają kafelki w łazience państwa Bajtockich. Elementem dekoracyjnym w łazience ma być poziomy pas złożony z rozmaitych wzorzystych kafli, szerokości n kafli i wysokości jednego kafla. Pani Bajtocka powiedziała Bajtoliniemu, że kafelki tworzące poziomy pas muszą być ułożone tak, żeby tworzyły wzór powtarzający się co k kafli. Ledwo pani Bajtocka wyszła, przyszedł pan Bajtocki i powiedział Bajtoliniemu, że kafelki tworzące poziomy pas muszą być ułożone tak, żeby tworzyły wzór powtarzający się co l kafli. Biedny Bajtolini przyszedł do Bajtazara po radę:

> *— Mistrzu Bajtazarze, to jak mam w końcu ułożyć kafelki? Czy wzór ma się powtarzać co k, czy co l kafli?*
>
> *— Nasz klient, nasz pan! Musisz ułożyć kafelki tak, żeby wzór powtarzał się zarówno co k, jak i co l kafli. Ponadto musisz użyć jak największej liczby różnych kafli, tak aby wzór nie był zbyt monotonny. A teraz już nie filozofuj, tylko do roboty!*

Bajtolini zgłupiał do reszty. Pomóż mu!

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia liczby n, k i l,*
- *obliczy liczbę różnych kafli, jakich należy użyć,*
- *wypisze wynik na standardowe wyjście.*

Wejście

Na standardowym wejściu znajdują się trzy liczby całkowite n, k i l, odpowiednio w pierwszym, drugim i trzecim wierszu. Liczby te spełniają zależności $1 \leqslant n \leqslant 10^{500}$, $1 \leqslant k, l \leqslant n$. Uwaga: liczby k i l nie muszą być dzielnikami n.

Wyjście

Twój program powinien wypisać na standardowe wyjście (w pierwszym i jedynym wierszu) jedną liczbę całkowitą — maksymalną liczbę różnych kafli, jakich należy użyć do udekorowania łazienki pasem długości n tak, żeby wzór powtarzał się zarówno co k, jak i l kafli.

Przykład

Dla danych wejściowych:
10
5
7
poprawnym wynikiem jest:
2

Rozwiązanie

Problem postawiony w zadaniu sprowadza się do znalezienia największej liczby różnych liter, jakie mogą wystąpić w słowie złożonym z n-liter o okresach k i l (jednocześnie). W dalszym ciągu tę liczbę (przy ustalonych k, l) będziemy oznaczać przez $w(n)$. Z lematu o okresowości (udowodnionego poniżej) wiemy, że gdy $n \geqslant k+l-\mathrm{NWD}(k,l)$, to $\mathrm{NWD}(k,l)$ jest okresem każdego takiego słowa i dodatkowo istnieje takie słowo długości n, które nie posiada żadnego krótszego okresu, zatem oczywiście w takim przypadku $w(n) = \mathrm{NWD}(k,l)$.

Dalej pokażemy, że jeśli n spełnia zależność $k,l \leqslant n \leqslant k+l-\mathrm{NWD}(k,l)$, to prawdziwy jest wzór

$$w(n) = k + l - n.$$

Dla ustalenia uwagi przyjmijmy, że $k \leqslant l$. Zauważmy, że w przypadkach, gdy $n = l$ i $n = k+l-\mathrm{NWD}(k,l)$, powyższy wzór jest prawidłowy. W pierwszym przypadku mamy do czynienia po prostu ze słowem k-okresowym, stąd wynikiem jest k, natomiast w drugim przypadku działa lemat o okresowości, który mówi, że nasze słowo musi być $\mathrm{NWD}(k,l)$-okresowe, więc wynikiem jest $\mathrm{NWD}(k,l)$.

Teraz pokażemy, że $w(n) - w(n+1) \leqslant 1$ dla $n \geqslant l$. Rozważmy dowolne n-literowe słowo o okresach k, l. Niech $\{a_1, a_2, a_3, \ldots, a_n\}$ będzie zbiorem pozycji w tym słowie. a_1 będzie odpowiadać pozycji pierwszej litery w słowie, a_2 drugiej, itd. Wprowadźmy graf nieskierowany, w którym wierzchołkami są elementy a_i z tego zbioru. Połączmy krawędziami ze sobą te pozycje, na których muszą znajdować się identyczne litery ze względu na okresy k i l tego słowa. Zatem w grafie będą krawędzie:

$$(a_1, a_{k+1}), (a_2, a_{k+2}), (a_3, a_{k+3}), \ldots, (a_{n-k}, a_n),$$

$$(a_1, a_{l+1}), (a_2, a_{l+2}), (a_3, a_{l+3}), \ldots, (a_{n-l}, a_n).$$

Zauważmy, że pozycje należące do tej samej spójnej składowej muszą odpowiadać identycznym literom. Natomiast pozycje należące do różnych spójnych składowych mogą odpowiadać różnym literom.

Jak łatwo zauważyć $w(n)$ jest liczbą spójnych składowych w powyższym grafie. Wystarczy z każdą spójną składową skojarzyć inną literę i otrzymane w ten sposób słowo będzie k- i l-okresowe, a także będzie zawierać maksymalną możliwą liczbę liter.

Natomiast analogiczny graf, zbudowany na zbiorze $(n+1)$ pozycji (dodatkowym wierzchołkiem jest a_{n+1}) musi dodatkowo zawierać dwie krawędzie:

$$(a_{(n+1)-k}, a_{n+1}), (a_{(n+1)-l}, a_{n+1}).$$

Jeśli wierzchołki $a_{(n+1)-k}, a_{(n+1)-l}$ były w różnych spójnych składowych poprzedniego grafu, to w tym przypadku zostaną połączone w jedną spójną składową, do której dojdzie jeszcze wierzchołek a_{n+1}. Pozostałe spójne składowe pozostaną bez zmian, co oznacza, że liczba spójnych składowych zmniejszy się o jeden. Natomiast jeśli wierzchołki $a_{(n+1)-k}, a_{(n+1)-l}$ w poprzednim grafie były w tej samej spójnej składowej, to w tym grafie do owej spójnej składowej jedynie dojdzie wierzchołek a_{n+1}. (Pozostałe spójne składowe znów pozostaną bez zmian.) Oznacza to, że w tym przypadku liczba spójnych składowych nie zmieni się. To dowodzi, że istotnie $w(n) - w(n+1) \leqslant 1$.

Jednak z uwagi na to, że $w(l) = k$ oraz $w(k+l-\mathrm{NWD}(k,l)) = \mathrm{NWD}(k,l)$, musi zachodzić

$$w(n) - w(n+1) = 1 \text{ dla } n = l, l+1, l+2, \ldots, k+l-\mathrm{NWD}(k,l)-1,$$

co uzasadnia prawdziwość wzoru

$$w(n) = \begin{cases} k+l-n & \text{gdy } l \leqslant n \leqslant k+l-\mathrm{NWD}(k,l) \\ \mathrm{NWD}(k,l) & \text{gdy } n > k+l-\mathrm{NWD}(k,l) \end{cases}$$

Lemat 1 (o okresowości)

(1) *Słowo w o okresach k i l i długości $|w|$ co najmniej $k+l-\mathrm{NWD}(k,l)$ jest również* $\mathrm{NWD}(k,l)$ *okresowe.*

(2) *Zawsze istnieje pewne słowo x o okresach k i l, które nie posiada okresu mniejszego niż* $\mathrm{NWD}(k,l)$.

Dowód

(1) Zastosujemy indukcję ze względu na długość słowa:

- Gdy $k = l$, wtedy $\mathrm{NWD}(k,l) = k$, czyli słowo jest $\mathrm{NWD}(k,l)$-okresowe.
- Przypadek, gdy $k \neq l$. Załóżmy bez straty ogólności, że $k > l$. Rozpatrzmy litery na pozycjach i ze zbioru $1, 2, 3, \ldots, l-\mathrm{NWD}(k,l)$ (wszystkie pozycje liter w słowie to $1, 2, 3, \ldots, |w|$). Zauważmy, że każda litera na pozycji i jest identyczna z literą na pozycji $i+k$, ponieważ słowo jest k-okresowe. A z kolei litera na pozycji $i+k$ jest identyczna z literą na pozycji $(i+k)-l = i+(k-l)$, ponieważ słowo jest l-okresowe. Należy zauważyć, że dla każdej rozpatrywanej pozycji $i \in 1, 2, 3, \ldots, l-\mathrm{NWD}(k,l)$ pozycja $i+k$ występuje w naszym słowie, ponieważ z założenia ma ono długość co najmniej $k+l-\mathrm{NWD}(k,l)$. Oczywiście każda pozycja $(i+k)-l$ też występuje w słowie, ponieważ $l < k$, a pozycje i oraz $i+k$ występują w słowie. Z tych rozważań wynika, że prefiks u słowa w o długości $(l-\mathrm{NWD}(k,l))+(k-l) = k-\mathrm{NWD}(k,l)$ jest $(k-l)$-okresowy. Ponadto jest on l-okresowy, gdyż całe słowo w jest l-okresowe. Dodatkowo u jest długości co najmniej $(k-l)+l-\mathrm{NWD}(k-l,l) = k-\mathrm{NWD}(k,l)$, zatem wszystkie warunki do zastosowania założenia indukcyjnego są spełnione (istotne jest również, że słowo u jest krótsze niż w). A więc stosując to założenie indukcyjne otrzymujemy, że u jest także $\mathrm{NWD}(k-l,l)$-okresowe. Z kolei z własności NWD wynika, że gdy $k > l$, to $\mathrm{NWD}(k,l) = \mathrm{NWD}(k-l,l)$, czyli słowo u o długości

$k - \mathrm{NWD}(k,l)$ jest $\mathrm{NWD}(k,l)$-okresowe. Na koniec zauważmy, że długość słowa u równa $k - \mathrm{NWD}(k,l)$ jest zawsze większa lub równa l, gdy $k > l$. Dowód tego faktu pozostawiamy czytelnikowi. Z tego wynika, że prefiks słowa w o długości l jest $\mathrm{NWD}(k,l)$-okresowy. Do tego dotychczas dążyliśmy.

Teraz rozpatrzmy litery w słowie w na pozycjach $i \in l{+}1, l{+}2, l{+}3, \ldots, |w|-(k-l)$. Będziemy postępować podobnie, jak poprzednio. Zauważmy, że litery na pozycjach i, $i - l$ są jednakowe, ponieważ w jest l-okresowe oraz pozycja $i - l$ występuje w słowie dla każdego rozpatrywanego i. Ponadto litery na pozycjach $i - l$ i $(i - l) + k = i + (k - l)$ są identyczne, ze względu na k-okresowość słowa w. Również każda pozycja $(i - l) + k$ występuje w słowie w dla rozpatrywanych $i \in l + 1, l + 2, l + 3, \ldots, |w| - (k - l)$. Z tego wynika, że podsłowo v, składające się z kolejnych liter słowa w na pozycjach $l + 1, l + 2, l + 3, \ldots, |w|$ jest $(k - l)$-okresowe. Słowo v jest długości co najmniej $k - \mathrm{NWD}(k,l)$, ponieważ słowo w jest długości co najmniej $k + l - \mathrm{NWD}(k,l)$, a v jest krótsze o l liter. Analogicznie, jak poprzednio, możemy zastosować założenie indukcyjne do v, gdyż jest ono $(k - l)$- i l-okresowe oraz jego długość jest co najmniej $(k - l) + l - \mathrm{NWD}(k - l, l)$. A więc v jest $\mathrm{NWD}(k - l, l) = \mathrm{NWD}(k,l)$-okresowe.

A więc po wielkich trudach udowodniliśmy, że słowo w na pozycjach 1, 2, 3, $\ldots$, l jest $\mathrm{NWD}(k,l)$-okresowe oraz na pozycjach $l + 1$, $l + 2$, $l + 3$, $\ldots$, $|w|$ jest $\mathrm{NWD}(k,l)$-okresowe. Pozostaje zauważyć, że ponieważ w jest l-okresowe, więc na pozycjach i oraz $i + l$, dla $i \in 1..\mathrm{NWD}(k,l)$, występują te same litery, więc całe słowo w jest $\mathrm{NWD}(k,l)$-okresowe.

(2) Niech słowo z będzie dowolnym słowem długości $\mathrm{NWD}(k,l)$, składającym się z parami różnych liter. Wtedy szukane słowo x jest prefiksem nieskończonego słowa $z^\infty = zzz\ldots$

Łatwo zauważyć, że takie słowo jest k- i l-okresowe i nie posiada żadnego krótszego okresu. ■

Implementacja

Algorytm zastosowany w rozwiązaniu wzorcowym sprowadza się do obliczenia $\mathrm{NWD}(k,l)$, rozpoznania, który z dwóch przypadków z powyższego wzoru na $w(n)$ zachodzi, oraz obliczenia wartości $w(n)$.

Ze względu na konieczność wykonywania operacji arytmetycznych na liczbach przekraczających zakres standardowych typów całkowitoliczbowych, konieczna jest implementacja własnej arytmetyki.

W rozwiązaniu wielkie liczby reprezentowane są jako tablica cyfr dziesiętnych, a do obliczania NWD z liczb k, l zastosowano tzw. binarny algorytm obliczania NWD. Złożoność tego rozwiązania wynosi $O(m^2)$, gdzie przez m rozumiemy ograniczenie na liczbę cyfr liczb n, k, l.

Inne rozwiązania

Zadanie można rozwiązać używając przeszukiwania w głąb lub wszerz grafu nie skierowanego o n wierzchołkach. Każdy wierzchołek o numerze i należy połączyć krawędzią z $i+k$, $i+l$. Rozwiązaniem jest liczba spójnych składowych. Jednak to rozwiązanie jest bardzo nieefektywne.

Rozwiązania błędne

Podstawowymi błędami w rozwiązaniach zawodników były:

1. Nieprawdziwe założenie, że jeśli dane słowo jest k- i l-okresowe, to jest także $\mathrm{NWD}(k,l)$-okresowe, bez względu na długość tego słowa. Takie rozwiązanie daje niepoprawny wynik w przypadku, gdy $n < k + l - \mathrm{NWD}(k,l)$.
2. Brak implementacji arytmetyki wielkich liczb.
3. Błędy w implementacji (np. arytmetyki wielkich liczb).

Testy

Poniżej znajduje się opis testów.

- `kaf1.in` — liczby n, k, l mieszczą się w zakresie typu `integer`; odpowiedzią jest $\mathrm{NWD}(k,l)$, gdyż zachodzi $n = k + l - \mathrm{NWD}(k,l)$.
- `kaf2.in` — liczby n, k, l mieszczą się w zakresie typu `integer`; zachodzi przypadek $n < k + l - \mathrm{NWD}(k,l)$.
- `kaf3.in` — liczby n, k, l mieszczą się w zakresie typu `integer`; $n = l$, więc odpowiedzią jest k.
- `kaf4.in` — tak jak w teście nr 2.
- `kaf5.in` — liczby n, k, l mieszczą się w zakresie typu `longint`; odpowiedzią jest $\mathrm{NWD}(k,l)$, gdyż zachodzi $n \geqslant k + l - \mathrm{NWD}(k,l)$.
- `kaf6.in` — liczby n, k, l mieszczą się w zakresie typu `longint`; zachodzi przypadek $n < k + l - \mathrm{NWD}(k,l)$; ponadto dysproporcja między liczbami k i l pozwala wykryć nieefektywne implementacje algorytmu obliczania NWD, np. używające tylko odejmowania zamiast dzielenia z resztą, czyli stosujące formułę $\mathrm{NWD}(a,b) = \mathrm{NWD}(a-b,b)$ dla $a > b$, zamiast $\mathrm{NWD}(a,b) = \mathrm{NWD}(a \text{ div } b, b)$.
- `kaf7.in` — liczby n, k, l mieszczą się w zakresie typu `longint`; zachodzi przypadek $n < k + l - \mathrm{NWD}(k,l)$.
- `kaf8.in` — tak jak w teście nr 7.
- `kaf9.in` — liczby n, k, l mają około 200 cyfr; odpowiedzią jest $\mathrm{NWD}(k,l)$, gdyż zachodzi $n \geqslant k + l - \mathrm{NWD}(k,l)$.

- `kaf10.in` — liczby n, k mają około 200 cyfr; zachodzi przypadek $n < k + l - \mathrm{NWD}(k, l)$.
- `kaf11.in` — liczby n, l mają około 100 cyfr, a liczba k – 9 cyfr; poza tym test ten jest podobny do testu nr 6.
- `kaf12.in` — liczby n, k, l mają około 500 cyfr; zachodzi przypadek $n < k + l - \mathrm{NWD}(k, l)$.

Krzysztof Sikora — Treść zadania, Opracowanie
Rafał Rusin — Program

Połączenia

Ministerstwo Infrastruktury Bajtocji postanowiło stworzyć program pozwalający szybko obliczać długości tras między dowolnymi miastami. Nie byłoby w tym nic dziwnego, gdyby nie fakt, iż mieszkańcy Bajtocji nie zawsze szukają najkrótszej trasy. Zdarza się, że pragną dowiedzieć się o k-tą co do długości najkrótszą trasę. Dopuszczamy zapętlenia tras, tzn. takie trasy, na których miasta powtarzają się.

Przykład

Jeśli między dwoma miastami istnieją 4 trasy o długościach: 2, 4, 4 i 5, to najkrótsze połączenie ma długość 2, drugie co do długości 4, trzecie 4, a czwarte 5.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis sieci dróg Bajtocji oraz zapytania dotyczące długości tras przejazdu,*
- *obliczy i wypisze na standardowe wyjście odpowiedzi do wczytanych zapytań.*

Wejście

W pierwszym wierszu standardowego wejścia zapisane są dwie dodatnie liczby całkowite n i m oddzielone pojedynczym odstępem, $1 \le n \le 100$, $0 \le m \le n^2 - n$. Są to odpowiednio liczba miast w Bajtocji oraz liczba dróg łączących miasta. Miasta są ponumerowane od 1 do n.

W każdym z kolejnych m wierszy znajdują się po trzy liczby całkowite oddzielone pojedynczymi odstępami: a, b i d, $a \ne b$, $1 \le d \le 500$. Każda taka trójka opisuje jedną, jednokierunkową drogę długości d umożliwiającą przejechanie z miasta a do b. Dla każdych dwóch miast istnieje co najwyżej jedna droga umożliwiająca przejazd w danym kierunku.

W kolejnym wierszu znajduje się jedna liczba całkowita q, $1 \le q \le 10\,000$, oznaczająca liczbę zapytań. W kolejnych q wierszach są zapisane zapytania, po jednym w wierszu. Każde zapytanie to trzy liczby całkowite oddzielone pojedynczymi odstępami: c, d i k, $1 \le k \le 100$. Zapytanie takie dotyczy długości k-tej najkrótszej trasy z miasta c do miasta d.

Wyjście

Twój program powinien wypisywać odpowiedzi na wczytane zapytania na standardowe wyjście, po jednej odpowiedzi w wierszu. W i-tym wierszu powinna zostać wypisana odpowiedź na i-te zapytanie — jedna liczba całkowita równa szukanej długości trasy lub `-1`, gdy taka trasa nie istnieje.

Przykład

Dla danych wejściowych:

```
5 5
1 2 3
2 3 2
3 2 1
1 3 10
1 4 1
8
1 3 1
1 3 2
1 3 3
1 4 2
2 5 1
2 2 1
2 2 2
1 1 2
```

poprawnym wynikiem jest:

```
5
8
10
-1
-1
3
6
-1
```

Rozwiązanie

Problem można sprowadzić do problemu grafowego. Mamy dany graf skierowany z wagami na krawędziach. Należy znaleźć długość k-tej najkrótszej trasy między dwoma wierzchołkami. Z uwagi na to, iż stawianych jest wiele zapytań, rozsądne jest skonstruowanie algorytmu, który znajduje wszystkie możliwe odpowiedzi na starcie (być może dość dużym nakładem pracy), a dopiero po ich obliczeniu odpowiada na pytania, korzystając z wcześniej obliczonych informacji. Rozwiązanie wzorcowe jest uogólnieniem algorytmu Floyda-Warshalla dla dowolnego $k \geq 1$, który oblicza długości k najkrótszych tras między wszystkimi parami wierzchołków.

Definicja 1 Multizbiór o elementach ze zbioru $\{1, 2, \ldots, \infty\}$ nazywamy *p-zbiorem*.

Przyjmijmy założenie, że $a + \infty = \infty$ dla dowolnego $a \in \{1, 2, \ldots, \infty\}$.

Definicja 2 Dla p-zbiorów A i B

$$A + B = \{(a + b) : a \in A, b \in B\}.$$

Definicja 3 Dla p-zbioru A i $k \in \{1, 2, \ldots, \infty\}$

$$k * A = \underbrace{A + A + \cdots + A}_{k}.$$

Definicja 4 Dla p-zbioru A

$$A^* = \bigcup_{i=1}^{\infty} i * A.$$

Definicja 5 Dla p-zbioru A

$$A \downarrow k$$

oznacza p-zbiór k najmniejszych elementów z A.

W oryginalnym algorytmie Floyda-Warshalla obliczamy tablice d^l (w rzeczywistości obliczenia wykonywane są tylko na jednej tablicy, ale dla czytelniejszego opisu będziemy używać tej notacji), których element $d^l[i,j]$ oznacza długość najkrótszej ścieżki z wierzchołka i do wierzchołka j wykorzystującej wierzchołki ze zbioru $\{1, 2, \ldots, l\}$. Niech $D^l[i,j]$ będzie p-zbiorem, którego q-ty najmniejszy element oznacza długość q-tej najkrótszej trasy od wierzchołka i do wierzchołka j wykorzystującej wierzchołki ze zbioru $\{1, 2, \ldots, l\}$.

Algorytm

Mamy dany graf skierowany $G = (V, E)$ z wagami na krawędziach. Niech

$$D^0[i,j] := \begin{cases} \{\text{długość krawędzi z } i \text{ do } j\}, & \text{gdy taka krawędź istnieje;} \\ \emptyset & \text{wpp.} \end{cases}$$

Dla $l = 1, 2, \ldots, n$ wykonujemy kroki:

- $D^l[l,l] := (D^{l-1}[l,l])^*$,
- dla każdej pary $i, j \in \{1, 2, \ldots, n\}$

  $D^l[i,j] := D^{l-1}[i,j] \cup (D^{l-1}[i,l] + D^{l-1}[l,j]) \cup (D^{l-1}[i,l] + D^l[l,l] + D^{l-1}[l,j]).$

Twierdzenie 6 *Element q-ty co do wielkości (q-ty najmniejszy) w multizbiorze $D^n[i,j]$ jest długością q-tej najkrótszej trasy z wierzchołka i do wierzchołka j.*

Dowód Indukcja ze względu na l.

- $l = 0$: $D^0[i,j]$ dla $i, j \in \{1, 2, \ldots, n\}$ oznacza p-zbiór zawierający długości najkrótszych ścieżek pomiędzy wierzchołkami i i j niewykorzystujących żadnych wierzchołków pośrednich między nimi. W p-zbiorach są zatem tylko długości krawędzi grafu.
- $l \in \{1, 2, \ldots, n\}$: Dla $i, j \in \{1, 2, \ldots, n\}$ mamy dane p-zbiory $D^{l-1}[i,j]$ zawierające długości ścieżek pomiędzy wierzchołkami i i j, które używają jako wierzchołki pośrednie tylko wierzchołki $\{1, 2, \ldots, l-1\}$.

  Krok pierwszy algorytmu: $D^l[l,l] := (D^{l-1}[l,l])^*$.

Lemat 7 $D^l[l,l]$ *jest multizbiorem długości ścieżek s o początku i końcu w wierzchołku l, które wykorzystują tylko wierzchołki ze zbioru* $\{1,2,\ldots,l\}$

Dowód Weźmy dowolną trasę s. Można podzielić ją jednoznacznie na podtrasy $s_1, s_2, \ldots, s_q$ o początkach i końcach w l w taki sposób, żeby żadna z nich nie zawierała wierzchołka l jako wierzchołka pośredniego. Multizbiór długości ścieżek rozpadających się na q podtras to $q * D^{l-1}[l,l]$, a zatem $(D^{l-1}[l,l])^*$ jest multizbiorem długości wszystkich rozpatrywanych ścieżek. ■

Drugi krok algorytmu: dla każdej pary $i,j \in \{1,2,\ldots,n\}$ obliczamy $D^l[i,j] := D^{l-1}[i,j] \cup (D^{l-1}[i,l] + D^{l-1}[l,j]) \cup (D^{l-1}[i,l] + D^l[l,l] + D^{l-1}[l,j])$. (Zauważmy, że zastosowanie tego przypisania dla $i = j = l$ nie zmienia obliczonej już wartości $D^l[l,l]$.)

Lemat 8 $D^l[i,j]$ *jest multizbiorem długości ścieżek o początku w wierzchołku i i końcu w wierzchołku j* ($i,j \in 1,2,\ldots,n$)*, które wykorzystują wierzchołki* $\{1,2,\ldots,l\}$.

Dowód Weźmy dowolną trasę s. Rozpatrzmy wszystkie możliwe przypadki występowania w niej wierzchołka l poza początkiem i końcem trasy:

– nie występuje — multizbiór długości takich tras to $D^{l-1}[i,j]$,

– występuje dokładnie raz — multizbiór długości tras tej postaci to $D^{l-1}[i,l] + D^{l-1}[l,j]$,

– występuje przynajmniej dwa razy — p-zbiór długości tras spełniających ten warunek to $D^{l-1}[i,l] + D^l[l,l] + D^{l-1}[l,j]$. ■

Udowodniliśmy tym samym, że żadna trasa nie jest pominięta przez algorytm. ■

Przedstawiony algorytm operuje na nieskończonych multizbiorach. Jednak należy zauważyć, że wystarczy pamiętać jedynie k najmniejszych elementów w każdym z używanych p-zbiorów (najlepiej w kolejności posortowanej).

Operację $A \cup B$ dla p-zbiorów A, B można wykonać w czasie $O(k)$. Wystarczy scalić posortowane elementy z A, B, biorąc tylko k początkowych wartości.

Operację $A + B$ dla p-zbiorów A, B można wykonać w czasie $O(k \lg k)$, używając kopca binarnego. Załóżmy, że $A = \{a_1, a_2, \ldots, a_k\}$, $B = \{b_1, b_2, \ldots, b_k\}$ oraz $a_i < a_{i+1}$ i $b_i < b_{i+1}$ dla $i \in \{1,2,\ldots,k-1\}$. Na początku wkładamy do kopca elementy $a_1 + b_1, a_2 + b_1, \ldots, a_k + b_1$. Następnie wykonujemy k razy:

- znajdujemy najmniejszy element w kopcu (niech to będzie $a_i + b_j$), usuwamy go z kopca i wkładamy do wektora wynikowego na kolejną pozycję,

- wkładamy do kopca $a_i + b_{j+1}$ (jeżeli taki element istnieje).

Gdyby element $a_i + b_j$ nie był najmniejszym elementem poza wektorem wynikowym, to istniałby element $a_p + b_q$ spoza wektora wynikowego mniejszy od niego. Wtedy wszystkie elementy $a_p + a_r$, dla $r \leq q$, byłyby mniejsze od $a_i + b_j$ oraz któryś z nich znajdowałby się w kopcu, gdyż dla każdego α istnieje β takie, że $a_\alpha + b_\beta$ jest w kopcu. W takiej sytuacji powinien on zostać wyjęty z kopca, a nie został. Sprzeczność, czyli $a_i + b_j$ jest najmniejszym elementem.

Lemat 9 *Dla dowolnego p-zbioru A i liczby k*

$$\Big(\bigcup_{i=1}^{\infty} i * A\Big) \downarrow k = \Big(\bigcup_{i=1}^{k} i * A\Big) \downarrow k.$$

Dowód Niech x będzie najmniejszym elementem z A. Wystarczy zauważyć, że najmniejszy element $(k+j) * A$, dla $j \in \{1, 2, \ldots, \infty\}$, jest równy $(k+j) \cdot x$ i co najmniej k elementów należących do $\bigcup_{i=1}^{k} i * A$ jest od niego mniejszych. Są to elementy $i \cdot x$, dla $i \in \{1, 2, \ldots, k\}$. ■

Operację $A^* \downarrow k$ da się wykonać w czasie $O(k^2 \lg k)$, wykorzystując powyższy fakt, poprzez bezpośrednie wykonywanie operacji $A \cup B$ oraz $A + B$.

Górne ograniczenie na maksymalną długość k-tej najkrótszej trasy w grafie o n wierzchołkach i krawędziach z wagami $\leq d$ wynosi dnk. Wynika to z faktu, że najkrótsza trasa może składać się z n krawędzi, każdej o długości d. Kolejne najkrótsze trasy ($k > 1$) mogą składać się z trasy $(k-1)$-najkrótszej plus kolejnych n krawędzi.

Cały algorytm działa w czasie $O(nk^2 \lg k + n^3 k \lg k)$.

Testy

Zadanie sprawdzane było na zestawie 10 danych testowych:

- `pol1.in` — $n = 1, m = 0, q = 1$, poprawnościowy;
- `pol2.in` — $n = 6, m = 8, q = 100$, poprawnościowy;
- `pol3.in` — $n = 3, m = 6, q = 30$, poprawnościowy;
- `pol4.in` — $n = 21, m = 27, q = 25$, poprawnościowy;
- `pol5.in` — $n = 8, m = 9, q = 100$, poprawnościowy;
- `pol6.in` — $n = 6, m = 14, q = 10000$, sprawdza odporność na dużą liczbę pytań;
- `pol7.in` — $n = 40, m = 41, q = 10000$, sprawdza odporność na dużą liczbę pytań;
- `pol8.in` — $n = 40, m = 40, q = 10000$, sprawdza górne ograniczenie na długość k-tej trasy;
- `pol9.in` — $n = 30, m = 243, q = 10000$, wydajnościowy;
- `pol10.in` — $n = 30, m = 870, q = 10000$, wydajnościowy.

Zawody III stopnia

Zawody III stopnia — opracowania zadań

Krzysztof Onak — Treść zadania, Opracowanie
Marcin Stefaniak — Program

Gra w dzielniki

Pszczółka Maja i Gucio czasami grają w następującą grę. Maja wymyśla liczbę naturalną k z przedziału od 1 do pewnej ustalonej liczby naturalnej n. Następnie Gucio zadaje pytania postaci „Czy k jest podzielne przez m?”, gdzie m to dodatnia liczba całkowita. Maja po każdym takim pytaniu odpowiada `TAK` lub `NIE`. Gucio chce w jak najmniejszej liczbie pytań dowiedzieć się, jaką liczbę Maja miała na myśli.

Zadanie

Napisz program, który po skompilowaniu razem z odpowiednim modułem grającym będzie grał jako Gucio. Na potrzeby tego zadania otrzymasz **uproszczony** *moduł grający, który pozwoli Ci przetestować swoje rozwiązanie.*

Opis interfejsu modułu grającego w Pascalu / C

Twój program powinien komunikować się ze „światem zewnętrznym” tylko i wyłącznie poprzez wywołania funkcji i procedur modułu grającego (`maja.pas` w Pascalu, `maja.h` w C/C++). Oznacza to, że nie wolno otwierać żadnych plików ani też korzystać ze standardowego wejścia/wyjścia.

```
(* Pascal *)
function gramy_dalej: longint;
function czy_podzielna_przez(m : longint) : boolean;
procedure zgaduj(k : longint);

/* C/C++ */
int gramy_dalej();
int czy_podzielna_przez(int m);
void zgaduj(int k);
```

Twój program powinien być zdolny rozegrać z Mają wiele partii podczas jednego uruchomienia. Aby zainicjować rozgrywkę, należy użyć funkcji `gramy_dalej()`, której wynikiem jest n — górne ograniczenie na wymyśloną przez Maję liczbę. Liczba n spełnia ograniczenia $1 \leq n \leq 1\ 000\ 000$. Jeśli nie ma już więcej rozgrywek do rozegrania, wynikiem funkcji jest 0.

Następnie twój program może zadawać pytania za pomocą funkcji `czy_podzielna_przez(m)`. Parametr m musi spełniać warunek $1 \leq m \leq n$.

Aby zakończyć rozgrywkę trzeba spróbować zgadnąć sekretną liczbę Mai za pomocą procedury `zgaduj(k)`. Parametr k powinien spełniać $1 \leq k \leq n$. Można próbować tylko raz; po próbie odgadnięcia liczby Mai można zainicjować następną rozgrywkę.

Rozwiązanie

Uproszczona gra

Zanim przejdziemy do problemu, który mamy rozwiązać, rozpatrzmy uproszczoną grę, w której Maja wymyśla liczbę naturalną k także z przedziału od 1 do n, ale dodatkowo zakładamy, że $k = 1$ lub k jest liczbą pierwszą. W takiej grze dobrą strategią byłoby sprawdzanie, czy k dzieli się przez kolejne liczby pierwsze z ustalonego przedziału, dopóki nie okazałoby się, że k jest jedną z liczb pierwszych albo nie dzieli się przez żadną z nich. W pesymistycznym przypadku musimy zatem zadać tyle pytań, ile jest liczb pierwszych w $[1; n]$. Jak się potem okaże, dokładnie w takiej samej liczbie pytań — w pesymistycznym przypadku — możemy rozwiązać nasz wyjściowy problem.

Wyznaczanie liczb pierwszych — sito Eratostenesa

Jak możemy przypuszczać po wstępnej analizie, liczby pierwsze będą odgrywać w rozwiązaniu zadania ważną rolę, dlatego zapoznamy się teraz z klasycznym algorytmem ich wyznaczania, zwanym *sitem Eratostenesa*. Zakładamy, że interesują nas liczby pierwsze z zakresu $[1; n]$. Bierzemy najpierw kolekcję wszystkich liczb naturalnych od 2 do n. Następnie, dopóki nie opróżnimy kolekcji, postępujemy w następujący sposób: wybieramy z kolekcji najmniejszą liczbę k — musi to być liczba pierwsza, gdyż nie dzieli się przez mniejsze liczby pierwsze — i usuwamy z kolekcji wszystkie wielokrotności k. Jeśli na przykład $n = 10$, to wykonujemy, co następuje:

Krok	k	Liczby pierwsze	Kolekcja
1.			2, 3, 4, 5, 6, 7, 8, 9, 10
2.	2	2	3, 5, 7, 9
3.	3	2, 3	5, 7
4.	5	2, 3, 5	7
5.	7	2, 3, 5, 7	

Warto zauważyć, że aby znaleźć wszystkie liczby pierwsze nie większe niż n, wystarczy z kolekcji wyrzucić wielokrotności liczb pierwszych mniejszych lub równych $\sqrt{n}$. Wszystkie liczby, które pozostaną, są już na pewno pierwsze, bo jeśli liczba jest złożona, to ma dzielnik mniejszy lub równy pierwiastkowi z niej.

Korzystając z tej obserwacji, możemy znacznie efektywniej zaimplementować sito:

```
var
  pierwsza : array[2..n] of boolean;
  i, j : integer;
begin
  for i := 2 to n do
    pierwsza[i] := true;
  i := 2;
  while i * i ⩽ n do begin
```

```
    if pierwsza[i] do begin
      j := 2 * i;
      while j ⩽ n do begin
        pierwsza[j] := false;
        j := j + i;
      end;
    end;
    i := i + 1;
  end;
end;
```

Po zakończeniu działania algorytmu w tablicy *pierwsza* na pozycjach będącymi liczbami pierwszymi znajdują się wartości **true**, a pod pozostałymi — **false**.

Właściwa gra

Przedstawimy teraz algorytm rozwiązujący nasz początkowy problem. Testujemy podzielność k przez kolejne liczby pierwsze, poczynając od najmniejszej. Jeśli w pewnym momencie okazuje się, że k jest podzielne przez liczbę pierwszą p, to sprawdzamy podzielność przez p^2, p^3 i tak dalej, dopóki odpowiedź nie będzie negatywna. Dodatkowo przerywamy testowanie, gdy wiadomo, że pozytywna odpowiedź sprawiłaby, że liczba k musiałaby być większa od n, np. jeśli wiemy, że liczba całkowita k wybrana z przedziału $[1;5]$ jest podzielna przez 2^2, to nie może być podzielna ani przez 2^3, ani przez 3, ani przez 5.

Okazuje się, że postępując w ten sposób, nie zadamy nigdy więcej pytań niż wynosi liczba liczb pierwszych w przedziale $[1;n]$. Prawdziwy jest bowiem następujący fakt:

Fakt 1 *Jeśli m jest całkowite i $1 \leqslant m \leqslant 500\,000$, to w przedziale $(m; 2*m]$ znajduje się liczba pierwsza.*

Skąd to wiemy? Wystarczy napisać prosty program, który znajduje wszystkie liczby pierwsze mniejsze lub równe 1 000 000, a następnie upewnia się, że nie ma takiej pary kolejnych liczby pierwszych p_1 i p_2, że $2*p_1 < p_2$.

Jeśli okazuje się, że liczba k jest podzielna przez p, to problem sprowadza się do znalezienia liczby k' z przedziału $[1;\lfloor\frac{n}{p}\rfloor]$ takiej, że $k = pk'$. Ponieważ $p \geqslant 2$, wystarczy, aby w przedziale $(\lfloor\frac{n}{2}\rfloor; 2\lfloor\frac{n}{2}\rfloor] \subseteq (\lfloor\frac{n}{2}\rfloor; n] \subseteq (\lfloor\frac{n}{p}\rfloor; n]$ znajdowała się liczba pierwsza q. Jeśli nawet będziemy musieli sprawdzać podzielność k' przez p, to nie będziemy już musieli dzielić k' przez q, a to pozwala nam zmieścić się w limicie pytań.

Dodatkowo zauważmy, że jeśli Maja ma ustaloną z góry liczbę k, to, stosując tą strategię, dla większości liczb z przedziału $[1;n]$ odgadniemy ją dosyć szybko, bo większość liczb ma małe dzielniki pierwsze.

Przypadek ogólny

Przy rozwiązaniu zadania skorzystaliśmy wyraźnie z ograniczenia na n. A co, gdyby n mogło być większe niż 1 000 000? Czy wtedy także nasz algorytm zawsze zadawałby co najwyżej tyle pytań, ile jest liczb pierwszych w przedziale $[1;n]$? Odpowiedź okazuje się pozytywna, gdyż prawdziwe jest rozszerzenie wcześniejszego faktu na wszystkie dodatnie liczby całkowite:

Fakt 2 *Jeśli m jest całkowite dodatnie, to w przedziale $(m; 2*m]$ znajduje się liczba pierwsza.*

Fakt 2 jest znany jako postulat Bertranda lub twierdzenie Czebyszewa. W pierwszej połowie XIX wieku hipotezę, że tak jest, postawił Józef Bertrand (1822–1900), który sprawdził ją dla $m < 3\,000\,000$. W roku 1850 Pafnutij Czebyszew (1821–1894) udowodnił, że tak jest rzeczywiście.

Testy

Programy zawodników grały w dzielniki dziesięciokrotnie z programem jury, który za każdym razem miał zadaną z góry liczbę n i przedział $[a;b]$ $(1 \leqslant a \leqslant b \leqslant n)$, z którego miała pochodzić liczba k. Program nie ustalał jednak od razu liczby $k \in [a;b]$, lecz odpowiadał negatywnie na pytania o podzielność tak długo, jak to tylko było możliwe. Odpowiedź pozytywna padała tylko, jeśli odpowiedź negatywna albo prowadziłaby do sprzeczności, albo k musiałoby być spoza przedziału $[a;b]$.

numer testu	n	przedział $[a;b]$
1	1000	$[100;500]$
2	1000000	$[5;10000]$
3	1000000	$[1;1]$
4	1000	$[10;1000]$
5	4000	$[20;2000]$
6	8000	$[40;4000]$
7	16000	$[80;8000]$
8	100000	$[180;18000]$
9	200000	$[88000;90000]$
10	500000	$[199000;200000]$

Adam Borowski	Marcin Stefaniak	Krzysztof Onak
Treść zadania	Opracowanie	Program

Skarb

Król Bajtazar ukrywał w swym zamku skarb, a miejsce jego ukrycia trzymał w tajemnicy. Kiedy jednak wyruszał na wojnę bał się, że może zginąć i skarb przepadnie. Wybrał więc zaufanych strażników i każdemu powierzył część informacji potrzebnych do odnalezienia skarbu. Następnie rozkazał strażnikom zejść do lochów rozciągających się pod zamkiem (każdemu w inne miejsce) i chodzić po lochach **metodą prawej ręki**. *Lochy to podziemne komnaty połączone korytarzami. Korytarze nie przecinają się poza komnatami. Korytarz może przebiegać pod innymi korytarzami. Metoda prawej ręki polega na tym, że strażnik po wejściu do komnaty wychodzi z niej pierwszym korytarzem po prawej stronie. Strażnicy rozpoczynają wędrówkę przy wejściach do korytarzy, zatem może się zdarzyć, że wielu strażników zaczyna wędrówkę, wychodząc z tej samej komnaty, jeśli tylko ruszają różnymi korytarzami.*

Król wie, że dopóki nie powróci z wojny lub nie polegnie, każdy ze strażników będzie wiernie wykonywał jego rozkazy. Jest jednak świadom, że gdy tylko dwóch (lub więcej) strażników spotka się w komnacie, to nie omieszka wymienić się wszystkimi posiadanymi informacjami dotyczącymi skarbu. Strażnicy nie są samolubni i wymieniają informacje, nawet, jeśli któryś z nich nie dowie się w ten sposób niczego nowego. Jeśli strażnicy rozpoczynają wędrówkę w tej samej komnacie, to od razu wymieniają się informacjami, które początkowo znają. Gdy jednak strażnicy mijają się w korytarzach, to nie rozmawiają ze sobą.

Król zastanawia się, czy gdy wróci szczęśliwie z wojny, skarb będzie nadal bezpieczny. Chce on wiedzieć, którzy strażnicy mogą poznać wszystkie informacje potrzebne do odnalezienia skarbu.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis lochów, początkowe położenie strażników, komnaty, do których pójdą najpierw, oraz informacje dotyczące skarbu, jakie każdy ze strażników początkowo zna,*
- *wyznaczy strażników, którzy mogą kiedyś poznać wszystkie informacje potrzebne do odnalezienia skarbu,*
- *wypisze numery tych strażników na standardowe wyjście.*

Wejście

W pierwszym wierszu standardowego wejścia zapisana jest jedna liczba całkowita n. Jest to liczba komnat w lochach, $2 \leqslant n \leqslant 100$. Komnaty są ponumerowane od 1 do n. W kolejnych n wierszach opisane są korytarze łączące komnaty. W wierszu $i+1$ opisane są korytarze wychodzące z komnaty nr i, w kolejności zgodnej z ruchem wskazówek zegara. W każdym z tych wierszy znajdują się liczby całkowite pooddzielane pojedynczymi odstępami. Pierwsza z *tych liczb, k_i, to liczba korytarzy wychodzących z komnaty nr i, $1 \leqslant k_i \leqslant n-1$. Dalej w tym samym wierszu zapisanych jest $2k_i$ liczb całkowitych — każdy z wychodzących korytarzy jest opisany dwiema liczbami całkowitymi. Pierwsza z liczb opisujących korytarz to numer komnaty, do której on prowadzi. Druga z tych liczb to jego długość, liczba całkowita z zakresu od 1 do 100. Korytarze są dwukierunkowe, tzn. jeżeli z komnaty a wychodzi korytarz długości l prowadzący do komnaty b, to z komnaty b wychodzi korytarz długości l prowadzący do komnaty a. Każda para komnat może być połączona co najwyżej jednym korytarzem. Strażnik potrzebuje na przejście korytarzem dokładnie tyle czasu, ile wynosi jego długość. Zakładamy, że czas spędzany przez strażników w komnatach jest pomijalny.*

W wierszu $n+2$ zapisane są dwie liczby całkowite k i l, $1 \leqslant k \leqslant 100$, $1 \leqslant l \leqslant 100$, gdzie k to liczba strażników, a l to liczba informacji potrzebnych do odnalezienia skarbu. Strażnicy są ponumerowani od 1 do k. Informacje dotyczące skarbu są ponumerowane od 1 do l. W kolejnych k wierszach opisani są strażnicy, w wierszu $i+n+2$ opisany jest strażnik nr i. W każdym z tych wierszy zapisane są liczby całkowite pooddzielane pojedynczymi odstępami. Pierwsza liczba w wierszu to numer komnaty, w której początkowo znajduje się strażnik. Druga liczba to numer komnaty, do której strażnik ruszy jako pierwszej. Trzecia liczba, m_i, to liczba informacji dotyczących skarbu, które strażnik nr i początkowo zna, $0 \leqslant m_i \leqslant l$. Dalej w wierszu znajduje się m_i liczb całkowitych — numery informacji znanych początkowo strażnikowi nr i.

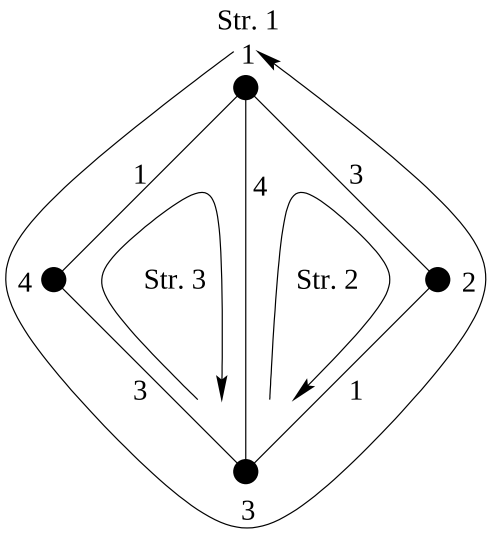
Wyjście

W pierwszym wierszu standardowego wyjścia Twój program powinien wypisać jedną liczbę całkowitą — liczbę strażników, którzy mogą kiedyś poznać wszystkie informacje potrzebne do odnalezienia skarbu. W następnych wierszach powinny znaleźć się uporządkowane rosnąco numery tych strażników, po jednym w wierszu.

Przykład

Dla danych wejściowych:

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



poprawnym wynikiem jest:

```
2
2
3
```

Rozwiązanie

W zadaniu „Skarb" naszym celem jest wyznaczenie strażników, którzy spotykając się z innymi podczas wędrówki po labiryncie, zdobędą wszystkie dostępne informacje o skarbie. W treści zadania określony jest deterministyczny (jednoznaczny) sposób wędrówki strażników (tzw. metoda prawej ręki), a w danych wejściowych zawarty jest kompletny opis labiryntu, początkowe pozycje strażników i znane przez nich strzępki informacji.

To wszystko wystarcza, by napisać prosty program symulujący przedstawioną sytuację i tym sposobem rozwiązać zadanie. Niestety, to rozwiązanie jest mało efektywne, bo dla niektórych danych wejściowych musi upłynąć wiele kroków symulacji nim wymiana informacji pomiędzy strażnikami się ustabilizuje. W końcu to nastąpi, bo strażnicy nic nie zapominają, a wiedza na temat skarbu występuje w skończonej liczbie porcji. Chwilę, gdy to następuje, nazwiemy zakończeniem.

Trasy strażników są cykliczne

Aby uprościć rozwiązanie, trzeba zauważyć parę prostych faktów. Jak już wspomnieliśmy, sposób wędrówki strażników jest ustalony jednoznacznie, według metody prawej ręki. Przypomnijmy, że oznacza to, iż strażnik po wejściu do komnaty skręca w korytarz następny po swojej prawej stronie. Jest to łatwa metoda obchodzenia różnego rodzaju lochów i labiryntów, gdyż badacz z niej korzystający w każdej chwili może zawrócić i wrócić do punktu startu, poruszając się wg metody lewej ręki[1].

Chociaż korytarze pomiędzy komnatami można równie dobrze przechodzić w obie strony, to od tej pory będziemy rozważać trasy strażników będące ciągiem korytarzy wraz z wyraźnie określonym kierunkiem przejścia. Dlatego przez „korytarze" rozumieć będziemy skierowane odcinki tras strażników pomiędzy sąsiednimi komnatami.

Następny korytarz na trasie strażnika jest jednoznacznie wyznaczony na postawie poprzedniego korytarza. Co więcej, poprzedni korytarz też jest jednoznacznie wyznaczony na podstawie następnego (metodą lewej ręki). Wobec tego trasa strażnika nie może się schodzić w jednym korytarzu, tak jak nie może się rozdwajać. Wyrażając się precyzyjnie i matematycznie, przyporządkowanie każdemu korytarzowi następnego jest permutacją zbioru korytarzy.

Wobec tego strażnik, chodząc metodą prawej ręki, porusza się po cyklu w tym sensie, że po pewnym czasie wraca do tych samych korytarzy i komnat. W trakcie tego cyklu strażnik może przechodzić przez jedną komnatę wiele razy, o ile tylko wchodzi tam różnym korytarzem.

[1] W ten sposób można oszczędzić na nici Ariadny, ale za to nie ma gwarancji obejścia całego labiryntu.

Rozważmy dwóch strażników. Wiemy, że wędrują oni nieustannie po zamkniętych trasach o okresach, powiedzmy, l_1 i l_2. Wobec tego ich pozycje powtarzają się z okresem $NWW(l_1, l_2)$ (ta są liczby całkowite). Zatem jeśli dwaj strażnicy spotykają się choć raz, to potem będą się spotykać jeszcze wiele razy. Jak wiadomo, strażnicy podczas spotkań wymieniają się swoją wiedzą na temat skarbu, więc na zakończenie będą wiedzieć to samo.

Graf spotkań

Rozważmy graf, którego wierzchołkami są strażnicy, a krawędzie łączą pary strażników, którzy spotykają się podczas wędrówki po labiryncie. Strażnicy połączeni w tym grafie mają taką samą wiedzę na zakończenie, a zatem strażnicy w jednej spójnej składowej będą mieli taką samą wiedzę na zakończenie. Strażnicy nic nie zapominają, więc wszyscy będą znali te kawałki informacji, które zna choć jeden z nich. Z drugiej strony, nie będą znali nic więcej, bo niby skąd mieliby się więcej dowiedzieć.

Szkic rozwiązania

Oto szkic rozwiązania: wpierw obliczamy pary spotykających się strażników (o tym za chwile), a potem znajdujemy spójne składowe w tym grafie. Możemy to zrobić jednym ze standardowych sposobów: przechodząc graf w głąb lub wszerz, albo skorzystać ze struktury zbiorów rozłącznych (Find-Union). Dla każdej spójnej składowej, sumujemy zbiory informacji znanych początkowo strażnikom z tej składowej. Strażnicy z tej składowej mogą znaleźć skarb, jeżeli w wyniku otrzymamy pełen zbiór informacji.

Kiedy dwaj strażnicy się spotykają?

Załóżmy, że dwaj strażnicy poruszają się po cyklicznych trasach o okresach odpowiednio l_1 i l_2. Strażnicy mogą się spotykać tylko i wyłącznie w komnacie. Załóżmy, że przy pierwszym obejściu trasy odwiedzają oni pewną ustaloną komnatę w chwilach odpowiednio $t_{1,i}$, $1 \leq i \leq k_1$ oraz $t_{2,i}$, $1 \leq i \leq k_2$ (strażnicy mogą odwiedzać tą samą komnatę wiele razy). Spotykają się oni wtedy i tylko wtedy, gdy

$$al_1 + t_{1,i} = bl_2 + t_{2,j}$$

dla pewnych liczb całkowitych a, b, i, j, $1 \leq i \leq k_1$, $1 \leq j \leq k_2$. A to jest równoważne temu, że

$$al_1 - bl_2 = t_{2,j} - t_{1,i}.$$

Lewa strona tej równości przyjmuje dla różnych a i b wszystkie wielokrotności $\mathrm{NWD}(l_1, l_2)$[2] (fakt ten znany jest pod nazwą podstawowego twierdzenia arytmetyki). Wobec tego istnieje rozwiązanie tej równości wtedy i tylko wtedy, gdy

$$\mathrm{NWD}(l_1, l_2) | t_{2,j} - t_{1,i}$$

[2]Największy wspólny dzielnik można obliczyć znanym algorytmem Euklidesa.

lub inaczej

$$t_{1,i} \equiv t_{2,j} \pmod{\mathrm{NWD}(l_1, l_2)}$$

dla pewnych $1 \leq i \leq k_1$, $1 \leq j \leq k_2$. Możemy to sprawdzić w czasie linowym ze względu na $i + j$, jeśli użyjemy tablicy bitowej długości $\mathrm{NWD}(l_1, l_2)$.

Aby sprawdzić, czy dwaj strażnicy się mogą spotkać:

- wyznaczamy trasy obu strażników,
- obliczamy największy wspólny dzielnik długości tych tras,
- obliczamy momenty, w których strażnicy odwiedzają poszczególne komnaty na swych trasach,
- dla każdego wierzchołka wspólnego dla obu tras sprawdzamy wyżej objaśnioną metodą, czy strażnicy się tam spotykają.

Analiza złożoności algorytmu

Szacując złożoność obliczeniową, będziemy uwzględniali wielkość danych: liczbę komnat n, liczbę strażników k oraz liczbę informacji l. Dodatkowo przyda się maksymalna długość korytarza między komnatami s.

Efektywna implementacja rozwiązania wymaga stablicowania tego, który korytarz jest następny wg metody prawej ręki. Wystarczy, że dla każdej komnaty a będziemy pamiętać, dokąd pójdzie następnie strażnik, jeśli przyszedł z komnaty b. Taką informację można przygotować w czasie proporcjonalnym do liczby krawędzi w grafie, która z kolei jest rzędu $O(n^2)$.

Sprawdzenie jednej pary strażników zabiera czas rzędu $O(n^2)$, o ile mamy daną czystą tablicę bitową wielkości $O(n^2 s)$, bo przez tyle można ograniczyć okresy tras strażników. Inicjalizacja tej tablicy zajmuje czas $O(n^2 s)$, ale jest wykonywana tylko raz. Wszystkich par strażników jest k^2, więc sprawdzenie ich wszystkich zajmie czas rzędu $O(n^2 k^2 + n^2 s)$.

Znalezienie spójnych składowych grafu spotkań strażników i wyznaczenie sum zbiorów ich informacji nie przekracza czasu rzędu $O(kl)$.

Ostatecznie złożoność czasowa tego algorytmu wynosi $O(n^2 k^2 + n^2 s + kl)$. Złożoność pamięciową można oszacować przez $O(n^2 + kl + n^2 s)$.

Testy

Do sprawdzenia rozwiązań przygotowano 10 testów. Oto ich opisy:

- `ska1.in` — mały test poprawnościowy;
- `ska2.in` — mały test poprawnościowy;
- `ska3.in` — łańcuch trójkątów, sąsiednie stykają się w jednym punkcie;
- `ska4.in` — trójkąty stykające się w jednym punkcie, każdy strażnik chodzi po jednym trójkącie;

- `ska5.in` — dwa cykle proste stykające się w wielu wierzchołkach;
- `ska6.in` — graf dwudzielny;
- `ska7.in` — jak 4, ale każdy strażnik odwiedza wszystkie wierzchołki;
- `ska8.in` — graf „półpełny”;
- `ska9.in` — graf pełny;
- `ska10.in` — graf pełny.

Krzysztof Onak **Marcin Stefaniak**
Treść zadania, Opracowanie Program

Sumy

Mamy dany zbiór dodatnich liczb całkowitych A. Rozważmy teraz zbiór nieujemnych liczb całkowitych A' taki, że liczba x należy do A' wtedy i tylko wtedy, gdy x jest sumą pewnych elementów z A (elementy mogą się powtarzać). Na przykład, jeśli $A = \{2, 5, 7\}$, to do zbioru A' należą np. liczby 0 (suma 0 elementów), 2, 4 (2 + 2) i 12 (5 + 7 lub 7 + 5 lub 2 + 2 + 2 + 2 + 2 + 2), a nie należą liczby 1 i 3.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis zbioru A oraz ciąg liczb b_i,*
- *dla każdej liczby b_i stwierdzi, czy należy ona do zbioru A',*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu znajduje się jedna liczba całkowita n — liczba elementów w zbiorze A, $1 \leqslant n \leqslant 5\,000$. Kolejne n wierszy zawiera elementy zbioru A, po jednym w wierszu. W wierszu $i + 1$ zapisana jest jedna dodatnia liczba całkowita a_i, $1 \leqslant a_i \leqslant 50\,000$. $A = \{a_1, a_2, \ldots, a_n\}$, $a_1 < a_2 < \ldots < a_n$.

W wierszu o numerze $n + 2$ znajduje się jedna liczba całkowita k, $1 \leqslant k \leqslant 10\,000$. Kolejne k wierszy zawiera po jednej liczbie całkowitej z zakresu od 0 do 1 000 000 000, są to odpowiednio liczby b_1, b_2, …, b_k.

Wyjście

Wyjście powinno składać się z k wierszy. Wiersz o numerze i powinien zawierać słowo `TAK`*, jeśli $b_i \in A'$, a słowo* `NIE` *w przeciwnym przypadku.*

Przykład

Dla danych wejściowych:

```
3
2
5
7
6
0
```

```
1
4
12
3
2
```

poprawnym wynikiem jest:

```
TAK
NIE
TAK
TAK
NIE
TAK
```

Rozwiązanie

W pierwszej chwili po przeczytaniu zadania może nasunąć się przypuszczenie, że aby znaleźć rozwiązanie, należy sięgnąć po metody teorii liczb. Tymczasem program wzorcowy korzysta z technik znanych z teorii grafów. Pokażemy teraz, jak sprowadzić problem z treści zadania do pewnego innego — klasycznego już zresztą — problemu.

Zacznijmy od prostej obserwacji, że jeśli weźmiemy pewną liczbę a ze zbioru A oraz $x \in A'$, to także liczby $x + a$, $x + 2a$, $x + 3a$, $\ldots$ należą do zbioru A'.

Podzielmy wszystkie liczby naturalne (zero także traktujemy jako liczbę naturalną) ze względu na to, jaką dają resztę z dzielenia przez a. Oznaczmy przez A_i ($i \in \{0, 1, \ldots, a-1\}$) zbiór wszystkich liczb, które podzielone przez a dają resztę i. Jeśli np. $a = 3$, to dostajemy zbiory $A_0 = \{0, 3, 6, 9, \ldots\}$, $A_1 = \{1, 4, 7, 10, \ldots\}$ i $A_2 = \{2, 5, 8, 11, \ldots\}$.

Dla każdego zbioru A_i zachodzi własność, że jeśli jakaś liczba z tego zbioru należy do zbioru A', to wszystkie większe od niej w A_i też należą do A'. Jest to prosty i oczywisty wniosek z wcześniejszej obserwacji. Zatem w każdym zbiorze A_i, który ma element wspólny ze zbiorem A', istnieje liczba p_i taka, że dla każdego $x \in A_i$, x należy do A' wtedy i tylko wtedy, gdy $x \geqslant p_i$. Dla zbiorów A_i, których przecięcie z A' jest puste, przyjmujemy $p_i = \infty$, rozszerzając na nie tym samym podaną przed chwilą własność.

Zauważmy, że gdybyśmy znali liczby p_i, odpowiedź na pytanie, czy liczba naturalna x należy do A', byłaby już niesłychanie prosta. Najpierw sprawdzając, jaka jest reszta z dzielenia x przez a, dowiadywalibyśmy się, do którego ze zbiorów A_i wpada x, a wtedy pozostawałoby porównanie x z odpowiadającą temu zbiorowi wartością p_i. Dla pojedynczej liczby x można to w naszym przypadku zrobić w czasie stałym. Skoncentrujemy się teraz na pokazaniu, jak obliczyć p_i.

Skonstruujemy graf, w którym wierzchołkami będą zbiory A_i. Dla każdej liczby a_j i każdego wierzchołka A_i dodajemy krawędź z A_i do $A_{(i+a_j) \bmod a}$ o długości a_j. Krawędź ta opisuje fakt, że dodając do ustalonej liczby ze zbioru A_i liczbę a_j, trafiamy do zbioru $A_{(i+a_j) \bmod a}$, a nowo powstała liczba jest większa o a_j. Okazuje się, że w tak stworzonym grafie długość najkrótszej ścieżki (jeśli takiej nie ma, to przyjmujemy ∞) z A_0 do A_i jest właśnie poszukiwaną wartością p_i. Wynika to z faktu, że długość najkrótszej ścieżki jest równocześnie najmniejszą liczbą należącą do zbioru $A_i \cap A'$, a to z kolei jest wnioskiem z przedstawionej interpretacji krawędzi w tym grafie.

Pozostaje zatem do rozwiązania standardowy problem policzenia w grafie najkrótszych ścieżek z ustalonego wierzchołka do wszystkich pozostałych przy założeniu, że wagi krawędzi są nieujemne. Rozwiązuje się go najczęściej przy pomocy algorytmu Dijkstry, który już wielokrotnie pojawiał w rozwiązaniach zadań olimpijskich. Dokładny opis algorytmu można znaleźć na przykład w książce [15].

Zastanówmy się teraz, jak efektywnie zaimplementować nasze rozwiązanie. Przede wszystkim na wartość a warto wybrać najmniejsze a_i, jakie mamy w zbiorze A, minimalizując tym samym liczbę wierzchołków w grafie. Ponadto, jeśli w zbiorze A istnieją dwie różne liczby o tej samej reszcie modulo a, to można bez zmiany zbioru A' wyrzucić z A większą z nich, ponieważ jest ona sumą mniejszej i nieujemnej wielokrotności a. Tym sposobem zredukujemy zbiór A do co najwyżej a liczb i ostatecznie otrzymamy graf o $O(a\min(n,a))$ krawędziach i a wierzchołkach.

Pozostaje problem wyboru sposobu, w jaki zrealizujemy algorytm Dijkstry. Niech V będzie liczbą wierzchołków w grafie, a E liczbą krawędzi. Jeśli zdecydujemy się na zwykłe pamiętanie w tablicy wierzchołków jeszcze nierozpatrzonych, to koszt czasowy wynosi $O(V^2+E)$. Druga standardowa metoda realizacji algorytmu Dijkstry to z wykorzystaniem kopca binarnego, wówczas działamy w czasie $O(V+E\log V)$. W naszym przypadku dostajemy odpowiednio czasy działania $O(n+a^2)$ i $O(n+a\log a\min(n,a))$. W szczególnych warunkach jedna z implementacji może być gorsza od drugiej. Można by więc w zależności od przypadku, z którym mamy do czynienia, wybierać jedną z implementacji. Można wyliczyć, że dla $n \leqslant \frac{a}{\log a}$ z asymptotycznego punktu widzenia bardziej opłaca się korzystać z kopca. Jednak, jeśli zdecydowalibyśmy się na rozwiązanie hybrydowe, to trzeba by zaimplementować obie metody realizacji algorytmu Dijkstry, co mogłoby być zbyt czasochłonne podczas zawodów. Można na tę sytuację spojrzeć w ten sposób, że stosując implementację z kopcem binarnym, nie tracimy nigdy więcej niż $\log a$ z naszego oszacowania, co w naszym przypadku nie jest dużą wartością. Zatem ostatecznie program wzorcowy oblicza liczby p_i, korzystając z kopca binarnego, w czasie $O(n+a\log a\min(n,a))$.

Testy

Rozwiązania zawodników były oceniane za pomocą zestawu 15 testów:

- `sum1.in` — $n = 1$, $k = 100$;
- `sum1a.in` — $n = 5$, $k = 10$;
- `sum2.in` — $n = 20$, $k = 100$;
- `sum2a.in` — $n = 10$, $k = 100$;
- `sum3.in` — $n = 50$, $k = 100$;
- `sum4.in` — $n = 120$, $k = 100$;
- `sum5.in` — $n = 500$, $k = 1000$;

- `sum6.in` — $n = 4000$, $k = 1000$;
- `sum7.in` — $n = 5000$, $k = 1000$;
- `sum7a.in` — $n = 2$, $k = 1000$;
- `sum8.in` — $n = 5000$, $k = 1000$;
- `sum8a.in` — $n = 10$, $k = 1000$;
- `sum9.in` — $n = 100$, $k = 1000$;
- `sum9a.in` — $n = 100$, $k = 1000$;
- `sum10.in` — $n = 20$, $k = 1000$.

Pary testów 1 i 1a, 2 i 2a, 7 i 7a, 8 i 8a, 9 i 9a były zgrupowane.

Marcin Stefaniak
Treść zadania, Opracowanie

Krzysztof Onak
Program

Kryształ

Bajtoccy fizycy prowadzą badania nad kryształami bitanium. Atomy w kryształach bitanium tworzą sieć w kształcie kwadratowej kraty o prostopadłych osiach. Kryształy te mają dwie prostopadłe osie. Każdy atom w krysztale wiruje „w miejscu" wokół jednej z osi kryształu, w jedną lub w drugą stronę. Jako że mamy dwie osie i dwa kierunki wirowania, każdy atom może być w danej chwili w jednym z czterech stanów.

Jeśli dwa atomy wirują wzdłuż tej samej osi, ale w przeciwnych kierunkach, to mówimy, że wirują one przeciwnie. Dwa atomy sąsiadujące ze sobą wzdłuż osi kryształu lub po przekątnej mogą tworzyć parę stabilną lub niestabilną. Jeśli takie atomy wirują przeciwnie, to mówimy, że tworzą parę niestabilną. W przeciwnym wypadku mówimy, że tworzą parę stabilną.

Bajtoccy fizycy potrafią tak umieścić kwadratowy kryształ bitanium w spolaryzowanym polu bitomagnetycznym, że atomy na brzegach kryształu tworzą pary stabilne i nie zmieniają swoich stanów, a ponadto atomy położone na brzegu kryształu, symetrycznie względem jego środka, wirują przeciwnie. Teoria mówi, że w takim krysztale w każdej chwili musi się gdzieś znajdować niestabilna para atomów. Nie wiadomo tylko gdzie, ponieważ wewnątrz kryształu atomy bezustannie zmieniają swoje stany.

Ostatnio opracowano technikę „zamrażania" atomów w krysztale. Zamrożony atom nie zmienia już swojego stanu. Nie da się przewidzieć w jakim stanie zostanie zamrożony, ale można ten stan poznać po jego zamrożeniu. Umieszczenie kryształu w polu bitomagnetycznym zamraża atomy znajdujące się na brzegu kryształu. Dodatkowo można zamrozić pewne atomy w krysztale, niestety nie wszystkie. W kwadratowym krysztale bitanium o wymiarach $n \times n$ atomów można zamrozić dodatkowo (poza brzegiem) co najwyżej $3n$ atomów.

Zadanie

Napisz program sterujący aparaturą w laboratorium. Twój program powinien:

- *pobrać z aparatury informacje o stanach atomów na brzegu kryształu,*
- *sterować zamrażaniem atomów i badać stany zamrożonych atomów,*
- *doprowadzić do zamrożenia niestabilnej pary atomów i podać współrzędne takiej pary.*

Na potrzeby tego zadania, otrzymasz **uproszczony** *moduł sterujący aparaturą, który pozwoli Ci przetestować swoje rozwiązanie.*

Opis interfejsu aparatury

Twój program powinien komunikować się ze „światem zewnętrznym" tylko i wyłącznie poprzez wywołania funkcji i procedur modułu (`krysztal.pas` *w Pascalu,* `krysztal.h` *w C/C++). Oznacza to, że nie wolno otwierać żadnych plików ani też korzystać ze standardowych strumieni wejścia/wyjścia.*

```
(* Pascal *)
function rozmiar: integer;
```

```
function zamroz (x, y : integer) : integer;
procedure niestabilna (x1, y1, x2, y2 : integer);
```

```
/* C/C++ */
int rozmiar(void);
int zamroz (int x, int y);
void niestabilna (int x1, int y1, int x2, int y2);
```

Twój program powinien najpierw uruchomić aparaturę wywołując funkcję `rozmiar`*, której wynikiem jest rozmiar kryształu* n*,* $3 \leqslant n \leqslant 10\,000$*. Badany kryształ ma wymiary* $n \times n$ *atomów. Atomy w krysztale identyfikujemy za pomocą współrzędnych całkowitych* (x, y)*,* $1 \leqslant x, y \leqslant n$*.*

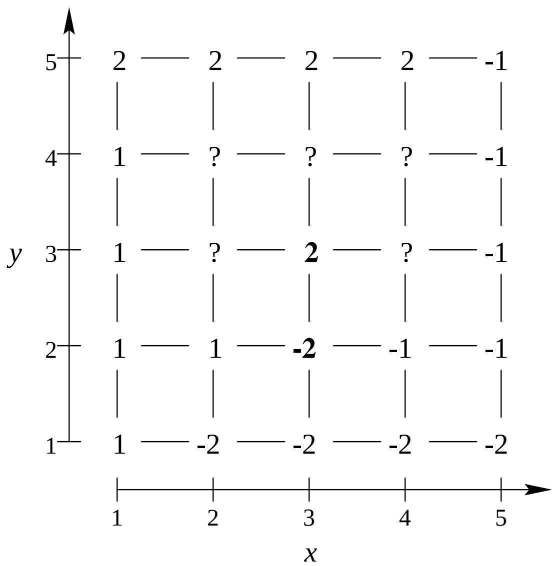
Wywołanie funkcji `zamroz(`x`,` y`)` *(dla* $1 \leqslant x, y \leqslant n$*) powoduje zamrożenie atomu o współrzędnych* (x, y)*, a wynikiem funkcji jest stan zamrożonego atomu. Stany atomów są reprezentowane przez liczby* -2*,* -1*, 1 lub 2. Wartość bezwzględna liczby określa oś wokół której wiruje atom, a jej znak określa kierunek wirowania atomu. (Zwróć uwagę, że nie ma znaczenia, która oś jest reprezentowana przez liczby 1 i* -1*, a która przez 2 i* -2*, oraz który kierunek wirowania jest reprezentowany przez liczby ujemne, a który przez dodatnie.) Atomy na brzegu kryształu w momencie uruchomienia aparatury są już zamrożone. Zamrożenie już zamrożonego atomu nie zmienia jego stanu. Można co najwyżej* $3n$ *razy wywołać funkcję* `zamroz` *dla atomów położonych we wnętrzu kryształu. Dla atomów położonych na brzegu kryształu można ją wywoływać dowolnie wiele razy.*

Po wyznaczeniu niestabilnej pary $(x_1, y_1), (x_2, y_2)$*, Twój program powinien wywołać procedurę* `niestabilna (`x_1`,` y_1`,` x_2`,` y_2`)`*. Wywołanie tej procedury kończy działanie programu. Jeśli w krysztale jest wiele par niestabilnych atomów, to Twój program powinien wskazać jedną z nich.*

Przykład

Interakcja z programem może wyglądać następująco:

```
rozmiar() = 5
zamroz (1, 1) = 1
zamroz (1, 2) = 1
zamroz (1, 3) = 1
zamroz (1, 4) = 1
zamroz (1, 5) = 2
zamroz (2, 5) = 2
zamroz (3, 5) = 2
zamroz (4, 5) = 2
zamroz (3, 3) = 2
zamroz (2, 2) = 1
zamroz (4, 2) = -1
zamroz (3, 2) = -2
niestabilna (3, 2, 3, 3)
```



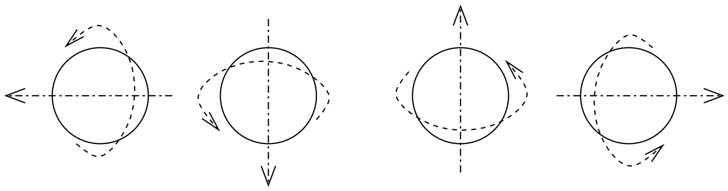
Rozwiązanie

Trudności zadania

Zadanie „Kryształ” sprawiło wiele trudności zawodnikom. Żadnemu finaliście nie udało się go rozwiązać choćby w połowie dobrze! Stało się tak z trzech powodów. Po pierwsze, w tym samym dniu zawodów zmierzyć się można było jeszcze z dwoma niełatwymi zadaniami — łatwo było przewidzieć, że nie trzeba rozwiązać wszystkich zadań, by zająć eksponowane miejsce. Po drugie, prawdziwą treść problemu ukryto w mętnej historyjce „fizycznej”, a nieszablonowa forma zadania — gra w pytania i odpowiedzi — utrudniała zrozumienie sedna sprawy. Po trzecie i ostatnie, lecz nie najmniej ważne, nie było to zwyczajne zadanie programistyczne — opierało się na matematycznym zadaniu. I, jak to z matematycznymi zadaniami często bywa, posiada ono zadziwiająco łatwe rozwiązanie, na które niełatwo wpaść.

Gra w strzałki

Na początek sformułujmy zadanie „Kryształ” w wygodny dla nas sposób. W treści zadania mówi się o krysztale (kwadratowej siatce) atomów, które wirują wokół osi poziomej lub pionowej. Kierunek kręcenia się atomu oznaczamy strzałką, skierowaną wzdłuż osi tak, by patrząc w jej kierunku atom wirował zgodnie z ruchem wskazówek zegara (patrz rys. 1).



Rysunek 1: Cztery możliwe kierunki wirowania atomów.

Wyobraźmy sobie taką grę: na każdym polu kwadratowej tablicy $N \times N$ wpisano jedną z czterech strzałek ($\leftarrow, \uparrow, \rightarrow, \downarrow$). Przeciwległe pola na brzegu planszy zawierają przeciwnie skierowane strzałki (własność **antysymetrii** brzegu planszy). Możemy

- (za darmo) poznać zawartość pól na brzegu planszy
- pytać się o wartości wpisane na polach wewnątrz planszy

Naszym celem jest znaleźć parę sąsiednich pól (w sensie króla szachowego) z przeciwnymi strzałkami, zadając możliwie mało pytań (nie więcej niż $3N$ razy)

Dla wygody, możemy przypisać strzałkom liczby ($\uparrow = 2, \downarrow = -2, \rightarrow = 1, \leftarrow = -1$) tak, by przeciwnym strzałkom odpowiadały liczby przeciwne (różniące się tylko znakiem).

Stabilność i niestabilność

Zgodnie z terminologią treści zadania, jeśli dwa sąsiednie pola zawierają przeciwne wartości, będziemy nazywać je parą **niestabilną**, w przeciwnym wypadku będziemy mówić o parze **stabilnej**. Podobnie będziemy mówić o krawędzi pomiędzy sąsiednimi polami, że jest stablina lub niestabilna.

Zbiór pól na planszy (np. cykl lub prostokąt) nazywamy stabilnym, gdy każda para sąsiednich pól jest stabilna. Na odwrót, wystarczy choć jedna para niestabilna w danym zbiorze, byśmy powiedzieli, że jest niestabilny.

Przykład

Oto przykładowa plansza na początku gry. Znaki zapytania oznaczają pola zakryte, o które możemy pytać. Zauważmy, że brzeg planszy tworzy cykl stabilny, spełniający własność antysymetrii.

$$\begin{bmatrix} \rightarrow & \downarrow & \rightarrow & \rightarrow & \uparrow \\ \downarrow & ? & ? & ? & \uparrow \\ \downarrow & ? & ? & ? & \uparrow \\ \downarrow & ? & ? & ? & \uparrow \\ \downarrow & \leftarrow & \leftarrow & \uparrow & \leftarrow \end{bmatrix} \equiv \begin{bmatrix} 1 & -2 & 1 & 1 & 2 \\ -2 & ? & ? & ? & 2 \\ -2 & ? & ? & ? & 2 \\ -2 & ? & ? & ? & 2 \\ -2 & -1 & -1 & 2 & -1 \end{bmatrix}$$

Gdybyśmy odkryli wszystkie pola, mogło by się okazać, że plansza wygląda np. tak (wyróżniono parę niestabilną):

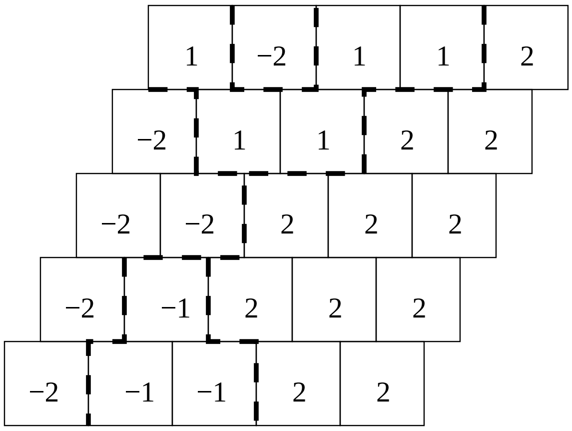
$$\begin{bmatrix} \uparrow & \leftarrow & \uparrow & \uparrow & \rightarrow \\ \leftarrow & \uparrow & \uparrow & \rightarrow & \rightarrow \\ \leftarrow & \underline{\leftarrow} & \underline{\rightarrow} & \rightarrow & \rightarrow \\ \leftarrow & \downarrow & \rightarrow & \rightarrow & \rightarrow \\ \leftarrow & \downarrow & \downarrow & \rightarrow & \downarrow \end{bmatrix} \equiv \begin{bmatrix} 1 & -2 & 1 & 1 & 2 \\ -2 & 1 & 1 & 2 & 2 \\ -2 & \mathbf{-2} & \mathbf{2} & 2 & 2 \\ -2 & -1 & 2 & 2 & 2 \\ -2 & -1 & -1 & 2 & -1 \end{bmatrix}$$

Zauważmy, że chociaż przeciwległe pola z brzegu planszy zawierają przeciwne wartości, to wewnątrz niej nie musi to być prawda. Mimo to, jakkolwiek byśmy nie wypełnili wnętrza, zawsze będzie ono niestabilne. Dlaczego tak się dzieje? W treści zadania wspomniane było, że „mówi tak teoria” — a my to zaraz zwyczajnie udowodnimy.

Dowód istnienia pary niestabilnej

Wpierw pokażemy dowód, który nie będzie od razu się przekładał na efektywny algorytm grający.

Nasza plansza jest kwadratową tablicą, w której za sąsiednie pola uznajemy te, które stykają się bokami lub rogami. „Kopnijmy” ją troszeczkę tak, by żadne cztery pola nie schodziły się w jednym wierzchołku (rys. 2). Czytelnicy zaznajomieni z grą „Hex” z łatwością dostrzegą podobieństwo otrzymanej siatki do tamtej gry. Chociaż niektóre sąsiednie pola przestały stykać się po „kopnięciu”, nadal będziemy w stanie dowieść, że istnieje para sąsiednich pól o wartościach przeciwnych.

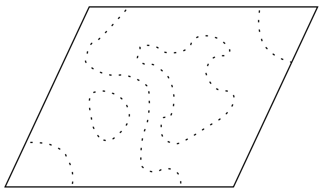


Rysunek 2: Plansza à la „Hex”. Zaznaczono granice między obszarami.

Użyjemy metody podobnej do dowodu braku remisów w grze „Hex” — rozważmy brzegi między obszarami złożonymi z pól tej samej wartości.

Cała plansza do „Hexa” rozpada się na cztery obszary O_1, O_{-1}, O_2, O_{-2}, każdy złożony z pól o tej samej wartości (rys. 2). Chcemy pokazać, że stykają się obszary O_{-1} i O_1 lub O_{-2} i O_2. Załóżmy, że tak nie jest, a dojdziemy do sprzeczności.

W każdym wierzchołku siatki planszy schodzą się najwyżej 3 pola. Co więcej, w żadnym wierzchołku nie mogą się schodzić trzy różne obszary, bo wtedy wśród tej trójki obszarów znalazłyby się dwa przeciwne. Wobec tego brzegi pomiędzy obszarami nie krzyżują się ani też się nie rozdwajają, a tylko składają się z pętli lub ścieżek kończących się na brzegu (rys. 3)



Rysunek 3: Możliwy kształt brzegu między obszarami — pętle i ścieżki.

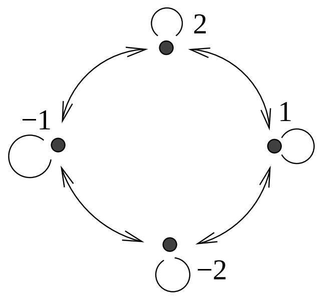
Dla ustalenia uwagi, spójrzmy na brzeg między obszarami 1 i 2. Składa się on z krawędzi typu $(1,2)$, które tworzą pętle i ścieżki. Każda ścieżka ma oba końce na brzegu planszy. Wobec tego liczba krawędzi typu $(1,2)$ na brzegu planszy jest parzysta.

Tymczasem w naszym przykładzie jest dokładnie jedno miejsce, w którym na brzegu pole z 1 styka się z polem oznaczonym 2. I to jest właśnie ten powód, dla którego nie da się wypełnić wnętrza planszy tak, by nie istniała tam para sąsiednich liczb przeciwnych.

Dlaczego liczba krawędzi $(1, 2)$ na brzegu jest nieparzysta?

Pozostaje pokazać, że liczba krawędzi rodzaju $(1, 2)$ na brzegu jest nieparzysta. Istotnie tak jest, co wynika z antysymetrii brzegu planszy.

Załóżmy, że brzeg planszy jest cyklem stabilnym. Nie ma tam krawędzi $(1, -1)$ ani $(2, -2)$. Możliwe krawędzie pomiędzy sąsiednimi polami przedstawia graf G na rysunku 4. Zauważmy, że ten graf przejść przypomina okrąg. Nie przypadkiem zresztą



Rysunek 4: Graf przejść pomiędzy wartościami pól.

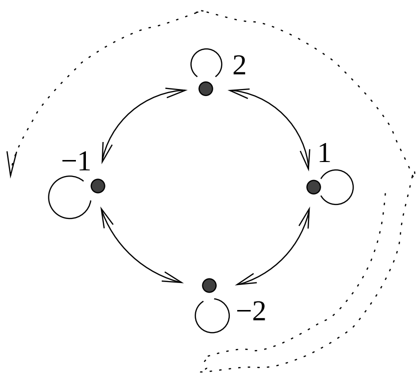
wierzchołki — jak pamiętamy, odpowiadające strzałkom — położone są zgodnie z ich kierunkami.

Wyobraźmy sobie podróż wokół planszy po cyklu brzegowym. Rozważmy trajektorię, jaką wyznacza w grafie G. Kiedy obejdziemy połowę brzegu, to trafimy na pole przeciwne w G do wyjściowego. Czyli wykonaliśmy $k + \frac{1}{2}$ pełnych obrotów po okręgu w G, gdzie k jest pewną liczbą całkowitą. Drugie pół podróży obchodzimy symetrycznie do pierwszej połówki, też wykonując $k + \frac{1}{2}$ obrotów, na dodatek w tą samą stronę co poprzednio. Łącznie wykonujemy nieparzystą liczbę $2k + 1$ obrotów w grafie przejść. A to oznacza, że każdą z krawędzi-ćwiartek okręgu w G przeszliśmy nieparzystą liczbę razy. W szczególności, krawędź rodzaju $(1, 2)$ występuje na cyklu brzegowym nieparzystą liczbę razy.

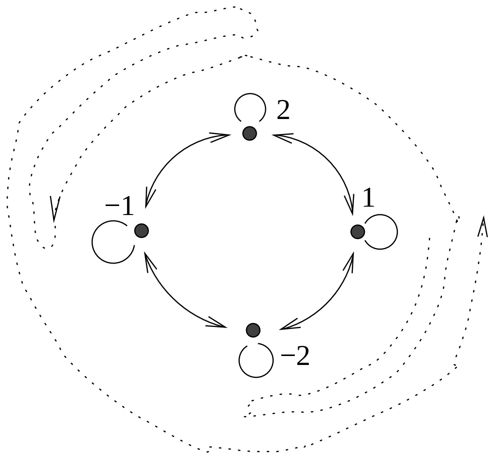
W naszym przykładzie, poruszając się od lewego górnego rogu zgodnie z ruchem wskazówek zegara, wyznaczamy w grafie G trajektorię z rysunków 5 (pół obrotu) i 6 (cały obrót). Zauważ, że po każdej z krawędzi $(\pm 1, \pm 2)$ przechodzimy nieparzystą liczbę razy.

Podsumowanie dowodu

Podsumujmy: dowodzimy nie wprost — z założenia o stabilności planszy wyprowadzamy absurd. Sprzeczność polega na tym, że z jednej strony liczba krawędzi pomiędzy polami o wartościach 1 i 2 na brzegu planszy musi być nieparzysta (co jest konsekwencją założenia o antysymetrii), a z drugiej strony wręcz przeciwnie — parzysta. Wobec tej sprzeczności, istnieje na planszy para niestabilna, co należało dowieść.



Rysunek 5: Przykład: pół obwodu.



Rysunek 6: Przykład: cały obwód.

Algorytm

Od dowodu istnienia pary niestabilnej do efektywnego algorytmu jej znajdowania droga niedaleka.

Moglibyśmy poszukiwać pary niestabilnej, przechodząc wzdłuż ścieżek rozgraniczających obszary 1 i 2 (lub innych). Niestety, takie rozwiązanie jest zbyt kosztowne, ponieważ ścieżki te mogą być zdecydowanie dłuższe niż $3N$, np. gdy brzegi obszarów okręcają się spiralnie wokół środka planszy. Potrzebujemy zatem sprytniejszego rozwiązania.

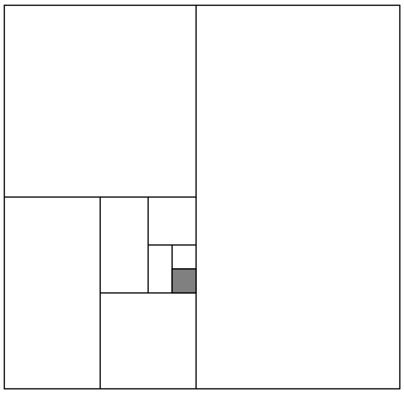
Rozwiązanie wzorcowe bardzo przypomina algorytm wyszukiwania binarnego — przeszukiwany obszar dzielimy na dwie części i wybieramy jedną, w której powinniśmy szukać dalej. Aby móc dokonać tego wyboru, użyjemy następującego kryterium.

Twierdzenie 1 *Rozważmy prostokątny fragment naszej planszy. Jeśli brzeg tego prostokąta jest stabilny i cykl brzegowy zawiera nieparzystą liczbę krawędzi rodzaju* $(1,2)$, *to ten fragment planszy jest niestabilny.*

Poprawność tego kryterium wynika z poprzedniego dowodu; jak się zaraz okaże, można też pokazać to przez indukcję, przez podział, co w istocie stanowi sedno algorytmu.

Opis algorytmu

1. Niech P będzie prostokątem obejmującym całą planszę.
2. Sprawdź, czy na brzegu P jest para niestabilna, jeśli tak to koniec.
3. Wzdłuż cyklu brzegowego P krawędź typu $(1,2)$ powinna występować nieparzystą liczbę razy — jeśli tak nie jest, to złe dane wejściowe.
4. Dopóki P jest większym prostokątem niż kwadrat 2×2:
   (a) podziel P na dwa mniej więcej równe prostokąty P_1 i P_2 (dzielimy na przemian poziomo i pionowo);
   (b) zapytaj o pola na linii podziału $P_1 \cap P_2$ — jeśli jest tam para niestabilna, to koniec;
   (c) policz, ile razy pojawia się krawędź typu $(1,2)$ na cyklach brzegowych P_1 i P_2;
   (d) za P podstaw ten z prostokątów P_1, P_2, który zawiera nieparzystą liczbę krawędzi $(1,2)$ na swoim cyklu brzegowym.
5. W kwadracie 2×2, w ktorym krawędź $(1,2)$ pojawia się nieparzystą liczbę razy, musi istnieć para sąsiednich liczb przeciwnych, więc znajdujemy ją i kończymy.

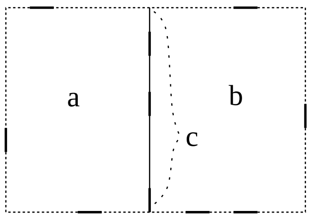


Rysunek 7: Tak dzielimy i przeszukujemy planszę.

Uzasadnienie poprawności algorytmu

Niezmiennikiem pętli algorytmu jest fakt, że bieżący prostokąt ma nieparzystą liczbę krawędzi $(1,2)$ na cyklu brzegowym, a w konsekwencji musi zawierać parę niestabilną.

Gdy dzielimy prostokąt P na dwa mniejsze prostokąty P_1, P_2, to dokładnie jeden z nich będzie zawierał nieparzystą liczbę krawędzi $(1,2)$. Jeśli bowiem oznaczymy (rys. 8) przez a oraz b liczbę krawędzi $(1,2)$ wzdłuż brzegu P_1 i P_2, a przez c liczbę krawędzi $(1,2)$ wzdłuż linii podziału $P_1 \cap P_2$, to widzimy, że liczba tych krawędzi wzdłuż brzegu oryginalnego prostokąta wynosi $a+b-2c$. Wiemy, że ta liczba jest nieparzysta, wobec tego $a+b$ jest nieparzyste, a więc albo a, albo b jest nieparzyste. Oczywiście, jesteśmy w stanie policzyć te liczby, znając wartości pól na brzegu wyjściowego prostokąta i wzdłuż linii podziału.



Rysunek 8: Przykład podziału prostokąta ($a = 6$, $b = 7$, $c = 3$).

Jasne, że algorytm się zatrzyma, bo w każdym kroku albo znajdujemy parę niestabilną, albo zamieniamy prostokąt na mniejszy. Gdy dojdziemy do prostokąta 2×2, na pewno znajdziemy tam parę niestabilną. Istotnie, ten prostokąt nie może być stabilny, bo wszystkie stabilne kwadraty 2×2 zawierają parzyste liczby krawędzi każdego typu na swoim cyklu brzegowym:

$$
\begin{array}{ccc}
1 & - & 1\\
| & & |\\
1 & - & 1
\end{array}
\quad
\begin{array}{ccc}
1 & - & 2\\
| & & |\\
1 & - & 1
\end{array}
\quad
\begin{array}{ccc}
1 & - & 2\\
| & & |\\
1 & - & 2
\end{array}
\quad
\begin{array}{ccc}
1 & - & 2\\
| & & |\\
2 & - & 1
\end{array}
$$

To, że na początkowej planszy krawędź $(1,2)$ występuje wzdłuż brzegu nieparzystą liczbę razy, wynika z założenia zadania o antysymetrii wartości pól na brzegu (już to pokazywaliśmy wcześniej). Zadanie można by istotnie uprościć, gdyby zrezygnować z założenia o antysymetrii, a zamiast niego powiedzieć wprost, że ta krawędź na brzegu występuje nieparzystą liczbę razy. Wtedy na pewno łatwiej byłoby wpaść na rozwiązanie, mając jak na tacy podany niezmiennik algorytmu.

Zauważmy też, że z tego algorytmu automatycznie wynika dowód istnienia pary pól o wartościach przeciwnych, częściowo tylko pokrywający się z poprzednim.

Analiza złożoności

Pozostaje zanalizować złożoność tego algorytmu. Kluczową sprawą jest tu liczba zadawanych pytań, która nie może przekroczyć $3N$. Nadmieńmy tylko, że złożoność czasowa i pamięciowa jest proporcjonalna do liczby zadanych pytań. Niech A_n będzie maksymalną liczbą pytań, które zada nasz algorytm dla kwadratu o boku n. Jasne, że A_n jest też maksymalną liczbą pytań dla wszystkich prostokątów o wymiarach mniejszych niż $n \times n$. Algorytm zadaje pytania o wartości pól wzdłuż linii podziałów, jak na rys. 7

Pokażemy przez indukcję po n, że $A_n \leq 3n$. Dla $n = 2$ o nic nie musimy pytać, więc $A_2 = 0 \leq 3 \cdot 2$.

Dla $n > 2$, zobaczmy co się dzieje po dwóch iteracjach pętli algorytmu. Kwadratową z początku planszę o boku n dzielimy na pół poziomo, a powstały prostokąt (nie większy niż $n \times \lceil \frac{n}{2} \rceil$) pionowo, otrzymując w najgorszym razie kwadrat o boku $\lceil \frac{n}{2} \rceil$. Przy tych podziałach wykonaliśmy nie więcej niż $n - 2 + (\lceil \frac{n}{2} \rceil - 2)$ zapytań. Z założenia indukcyjnego $A_{\lceil \frac{n}{2} \rceil} \leq 3\lceil \frac{n}{2} \rceil$, a zatem

$$A_n \leq (n-2) + (\left\lceil \frac{n}{2} \right\rceil - 2) + A_{\lceil \frac{n}{2} \rceil} \leq \frac{3}{2}n - 3 + 3\left\lceil \frac{n}{2} \right\rceil \leq 3n.$$

Uogólnienia

Warto odnotować, że to zadanie można uogólniać na różne sposoby.

Dowolny kształt siatki

Zamiast tablicy kwadratowej można użyć innej siatki — wystarczy, by graf sąsiedztwa był płaską siatką trójkątów. Aby wprowadzić założenie o antysymetrii brzegu, potrzebujemy tego, by brzeg tej siatki był środkowo symetryczny; jeśli tak nie jest, możemy zastąpić założenie o antysymetrii brzegu podobnym albo mocniejszym.

Więcej wymiarów

Można użyć większej liczby wymiarów (na przykład trzech).

Niech T będzie n-wymiarową triangulacją kuli n-wymiarowej. Wtedy brzeg T jest $(n-1)$-wymiarową triangulacją sfery $(n-1)$ wymiarowej.

Dla $n = 2$ jest to siatka trójkątów wypełniająca „koło", dla $n = 3$ jest to „kula" podzielona na czworościany. W ogólnym przypadku składniki (trójkąty, czworościany) T nazywamy sympleksami n-wymiarowymi. Każdy sympleks n-wymiarowy ma $n+1$ wierzchołków.

Załóżmy, że w każdym wierzchołku T umieszczono jedną z wartości $\{\pm 1, \pm 2, \ldots, \pm n\}$. Mówimy, że krawędź T jest niestabilna, gdy jej wierzchołki mają przeciwne wartości, w przeciwnym przypadku ta krawędź jest stabilna. Wtedy:

- jeśli wartości na brzegu T są rozłożone antysymetrycznie i stabilnie, to na brzegu T istnieje nieparzysta liczba sympleksów z wartościami wierzchołków $\{1, 2, \ldots, n\}$;
- jeśli na brzegu T istnieje nieparzysta liczba sympleksów z wartościami wierzchołków $\{1, 2, \ldots, n\}$, to istnieje w T krawędź niestabilna
- i możemy znaleźć tą krawędź metodą analogiczną do wyszukiwania binarnego.

Więcej wartości

Możemy rozważyć więcej wartości. Na przykład, możemy użyć jako wartości 8 strzałek $\{\leftarrow,\nwarrow,\uparrow,\nearrow,\rightarrow,\searrow,\downarrow,\swarrow\}$, ale wtedy musimy ogólniej spojrzeć na warunek będący niezmiennikiem pętli. Nie da się go pozostawić w postaci „nieparzysta liczba krawędzi pewnego typu na brzegu". Zamiast tego trzeba użyć dokładniejszego warunku, jak np. „liczba obrotów strzałek na cyklu brzegowym jest niezerowa". Alternatywnie można narzucić większe ograniczenia na pojęcie stabilności par, np. za stabilne uznać pary strzałek odchylonych o nie więcej niż $45°$.

Ciągłość

Twierdzenia tu pokazywane są dyskretnymi odpowiednikami twierdzeń dotyczących ciągłych przekształceń. Dziedzina matematyki, która bada takie własności, nazywa się **topologią**. Niektóre wyniki uzyskane tymi metodami są niezmiernie ciekawe. Na przykład:

- zawsze na Ziemi istnieją dwa punkty antypodyczne o tej samej temperaturze i ciśnieniu,
- kanapkę z masłem i serem da się podzielić jednym płaskim cięciem na dwie równe połówki.

Dla naszego problemu ciągły analog znaleźć można w następującym twierdzeniu:

Twierdzenie 2 *Niech K będzie kulą (domkniętą) n-wymiarową o środku w 0, a S — jej brzegiem. Niech $f\colon K \to K$ będzie ciągłym przekształceniem kuli K w siebie takim, że $f(x) = -f(-x)$ dla $x \in S$. Wtedy istnieje $x_0 \in K$ takie, że $f(x_0) = 0$.*

Jak się okazuje, metoda naszego algorytmu nadaje się czasem do znajdowania rozwiązań równania $f(x) = p$ dla funkcji ciągłej $f\colon R^n \to R^n$. Na przykład, dla $n = 2$ zamykamy obszar potencjalnych rozwiązań pętlą C. Patrzymy na to, ile razy obraz $f(C)$ „owija się" wokół punktu p — ta wartość nazywa się **stopniem** f na C względem p. Jeżeli ten stopień różni się od zera, to wiadomo, że gdzieś wewnątrz C musi istnieć rozwiązanie $f(x) = p$. Wtedy wystarczy podzielić C na dwie części i sprawdzić tym samym sposobem, wewnątrz której z nich mamy dalej szukać. Pojęcie stopnia przekształcenia można uogólnić na więcej wymiarów.

Jak sprawdzić, czy punkt należy do wielokąta?

Metoda opisana powyżej jest szczególnie efektywna, gdy brzeg obszaru C jest łamaną i zakładamy, że dla każdego odcinka $\overline{AB} \in C$ tej łamanej, jego obraz $f(\overline{AB})$ leży po tej samej stronie punktu p, co odcinek łączący obrazy jego końców $\overline{f(A)f(B)}$. Wtedy wystarczy zbadać wartości f w wierzchołkach łamanej i policzyć, ile razy łamana $f(C)$ obraca się wokół p. A tego można dokonać bardzo zgrabnie, pamiętając liczbę obrotów z dokładnością do ćwierć obrotu — szczegóły pozostawiamy Czytelnikowi jako ćwiczenie.

W szczególności, gdy f jest funkcją identycznościową, otrzymujemy ładny i szybki algorytm sprawdzający, czy punkt p należy do wielokąta C. Algorytm ten działa w czasie liniowym ze względu na liczbę krawędzi wielokąta.

Testy

Do sprawdzania rozwiązań zawodników użyto 10 testów, każdy wart 10 punktów. Testy różniły się swoim charakterem i rozmiarami.

nr testu	rodzaj testu	n
1	1	6
2	2	50
3	2	300
4	2	1000
5	3	100
6	3	5000
7	3	10000
8	4	500
9	4	2000
10	4	10000

Rodzaje testów:

1. Ustalone stany kryształów wczytane z pliku.
2. Stany kryształów wylosowane na początku działania programu. Losowanie jest deterministyczne dzięki ustaleniu zarodka (ang. *seed*) generatora liczb losowych. Przy losowaniu stanu nowego kryształu staramy się, aby konfliktował (był niestabilny) z jak najmniejszą liczbą już ustalonych sąsiadów.
3. Brzeg losowany na początku działania programu. Dalsze działanie dobierane zależnie od pytań programu zawodnika w sposób podobny do tego z wcześniejszego punktu. Program działa deterministycznie.
4. Kwadratowa spirala umieszczona gdzieś na planszy. Jedyne niestabilne pary znajdują się w jej środku.

Andrzej Gąsienica-Samek Treść zadania | **Marcin Stefaniak** Opracowanie | **Krzysztof Onak** Program

Małpki

Na drzewie wisi n małpek ponumerowanych od 1 do n. Małpka z nr 1 trzyma się gałęzi ogonkiem. Pozostałe małpki albo są trzymane przez inne małpki, albo trzymają się innych małpek, albo jedno i drugie równocześnie. Każda małpka ma dwie przednie łapki, każdą może trzymać co najwyżej jedną inną małpkę (za ogon). Rozpoczynając od chwili 0, co sekundę jedna z małpek puszcza jedną łapkę. W ten sposób niektóre małpki spadają na ziemię, gdzie dalej mogą puszczać łapki (czas spadania małpek jest pomijalnie mały).

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis tego, w jaki sposób małpki się trzymają oraz w jakiej kolejności puszczają łapki,*
- *dla każdej małpki obliczy, kiedy spadnie ona na ziemię,*
- *wypisze wynik na standardowe wyjście.*

Wejście

Pierwszy wiersz standardowego wejścia zawiera dwie dodatnie liczby całkowite n i m, $1 \leqslant n \leqslant 200\,000$, $1 \leqslant m \leqslant 400\,000$. Liczba n oznacza liczbę małpek, a liczba m czas (w sekundach) przez jaki obserwujemy małpki. Kolejne n wierszy zawiera opis sytuacji początkowej. W wierszu $k+1$ ($1 \leqslant k \leqslant n$) znajdują się dwie liczby całkowite oznaczające numery małpek trzymanych przez małpkę nr k. Pierwsza z tych liczb to numer małpki trzymanej lewą łapką, a druga — prawą. Liczba -1 oznacza, że małpka ma wolną łapkę. Kolejne m wierszy opisuje wyniki obserwacji małpek. W i-tym spośród tych wierszy ($1 \leqslant i \leqslant m$) znajdują się dwie liczby całkowite. Pierwsza z nich oznacza numer małpki, a druga numer łapki (1 — lewa, 2 — prawa), którą małpka puszcza w chwili $i-1$.

Wyjście

Twój program powinien wypisać na standardowe wyjście dokładnie n liczb całkowitych, po jednej w wierszu. Liczba w wierszu i powinna oznaczać chwilę, w której małpka nr i spadła na ziemię, lub być równa -1, jeśli małpka nie spadła na ziemię w trakcie obserwacji.

Przykład

Dla danych wejściowych:

```
3 2
```

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

poprawnym wynikiem jest:

```
-1
1
1
```

Rozwiązanie

Zadanie „Małpki" z pewnością jest jednym z najciekawszych zadań tegorocznej edycji Olimpiady Informatycznej. Jego krótka i zgrabna treść prowadzi do niebanalnego problemu informatycznego. Rozwiązywać je można dwojako: zwyczajnie albo sposobem z rodzaju tych, które J. Bentley w swych „Perełkach programowania" określił mianem rozwiązań „aha!" — wystarczy bowiem krótka chwila olśnienia, by wymyślić efektywny i prosty algorytm dla tego problemu.

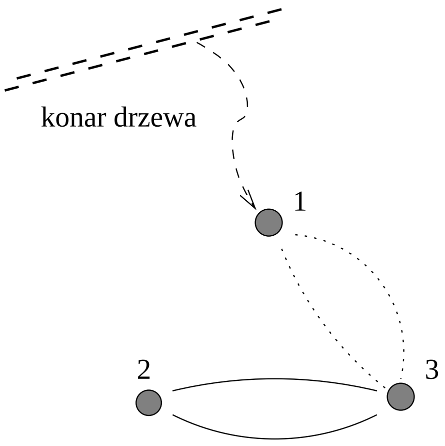
Graf uchwytów

Zacznijmy od przedstawienia tego problemu w terminologii grafów. Małpki trzymające się łapkami interpretujemy jako graf uchwytów G, którego wierzchołkami są małpki, a krawędziami łapki. Każda krawędź łączy małpkę trzymającą oraz małpkę trzymaną. Taki graf jest tak naprawdę multigrafem, bowiem dwa wierzchołki mogą być połączone więcej niż jedną krawędzią. Zauważmy, że z punktu widzenia „trzymania się" małpek nie jest istotne, do kogo należy trzymająca małpki rączka, zatem krawędzie tego grafu są nieskierowane.

Liczba małpek wynosi n. Wiadomo, że każda małpka ma dwie łapki do chwytania innych małpek. Wobec tego liczba wierzchołków grafu G wynosi n, a liczba krawędzi co najwyżej $2n$. To znaczy, że graf G jest rzadki, gdyż liczba krawędzi jest liniowa względem liczby wierzchołków. Fakt ten wpływa na złożoność obliczeniową rozwiązań.

Puszczenie łapki odpowiada po prostu usunięciu krawędzi z grafu G. Na podstawie danych wejściowych możemy łatwo zidentyfikować krawędzie usuwane w kolejnych chwilach. Chcemy wyznaczyć dla każdego wierzchołka moment, w którym traci on połączenie z wyróżnionym wierzchołkiem 1, reprezentującym małpkę trzymającą się gałęzi drzewa.

Graf uchwytów dla przykładowych danych z treści zadania zamieszczony jest na rysunku 1. Krawędzie usuwane w trakcie obserwacji zaznaczono linią kropkowaną. Widać, że po usunięciu obydwu krawędzi pomiędzy wierzchołkami 1 i 3, od wierzchołka 1 odłączają się wierzchołki 2 i 3.



Rysunek 1: Przykład grafu uchwytów

Pierwsze podejście

Naturalnym rozwiązaniem, jakie zapewne przychodzi każdemu do głowy w pierwszej kolejności, jest zwykła symulacja procesu usuwania krawędzi. W każdym kroku symulacji musimy znaleźć wierzchołki, które utraciły połączenie z wierzchołkiem nr 1.

Najprościej w każdym kroku wyliczać, które wierzchołki należą do spójnej składowej wierzchołka nr 1 i sprawdzać, które wierzchołki przestały w tym kroku w niej być. Obliczanie spójnej składowej można wykonać przeszukiwaniem grafu w głąb (DFS) lub wszerz (BFS), które to standardowe algorytmy nieraz już były opisywane na łamach „niebieskiej książeczki". Pesymistyczny koszt czasowy przeszukiwania jest liniowy względem liczby krawędzi, czyli jest rzędu $O(n)$. Ponieważ wykonujemy to w każdym kroku (a jest tych kroków m), całkowity koszt tego algorytmu jest rzędu $O(nm)$.

Czy można szybciej decydować, które wierzchołki odpadają? Oczywiście, jeśli rozłączane wierzchołki leżą już poza spójną składową wierzchołka 1, to nic się nie dzieje, ale w przeciwnym przypadku musimy wiedzieć, czy po usunięciu danej krawędzi rozspójnia nam wierzchołki, innymi słowy — czy ta krawędź jest mostem. Niestety, nie znamy żadnego algorytmu, który by pozwalał szybko odpowiadać na te pytania i w konsekwencji określać wierzchołki tracące połączenie w kolejnej chwili. Jest to o tyle ciekawe, że tego rodzaju algorytm przydałby się do zarządzania pamięcią, pozwalając zwalniać fragmenty pamięci natychmiast po tym, jak już są niepotrzebne.

Odśmiecanie

Przypomnijmy, jak działa zarządzanie pamięcią w różnych językach programowania.

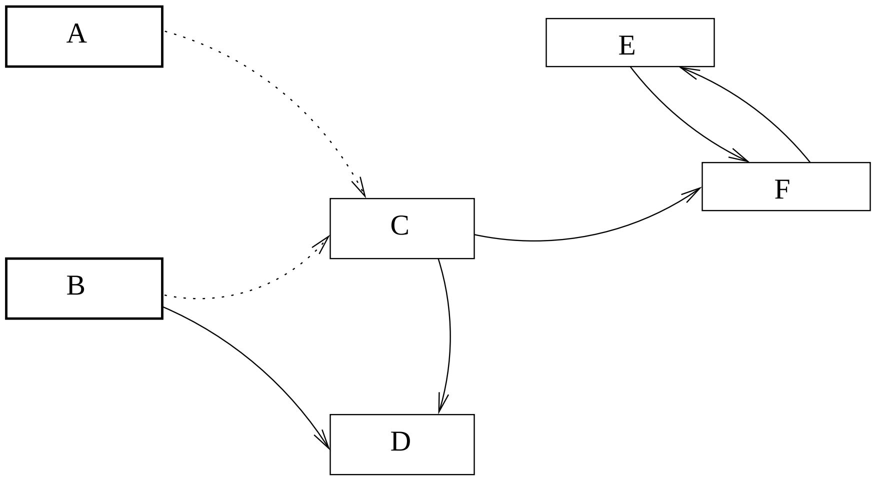
Program w trakcie działania od czasu do czasu przydziela sobie fragmenty pamięci określonej wielkości. Będziemy je nazywać obiektami, chociaż nie muszą mieć wiele wspólnego z tradycyjnym programowaniem obiektowym. Kiedy obiekt nie jest już potrzebny (ale nie wcześniej!), wypada zwolnić zajmowaną przezeń pamięć; jeśli o tym zapomnimy, grozi nam „wyciek pamięci", który prowadzi do zaburzeń działania programu. Na przykład pewien popularny system operacyjny tracił przy każdym nowym utworzonym programie, procesie czy wątku ponad sto bajtów pamięci, przez co nie dało się go używać przez dłuższy czas bez restartu.

Kto i jak ma decydować o zwalnianiu pamięci? Istnieją dwa (a w zasadzie to nawet trzy) główne podejścia.

- programista sam dba o zwalnianie pamięci (Pascal, C, C++),
- środowisko wykonania programu samo usuwa niepotrzebne obiekty (tak jest np. w języku Java),
- nigdy nie zwalniamy pamięci, nasze programy są małe i krótkie (niektóre języki „skryptowe").

Wadą pierwszego podejścia jest możliwość popełnienia błędu, w dodatku trudnego do wykrycia. Automatyczne usuwanie śmieci redukuje kłopoty programisty, ale nie zapewnia takiej wydajności, jaką potencjalnie mogłoby osiągnąć „ręczne" zwalnianie pamięci. O trzecim podejściu nie warto wspominać.

Jak działają automatyczne śmieciarki (ang. *garbage collector*)? Otóż, bardzo podobnie do naszych małpek. W każdym obiekcie pewne pola zawierają odniesienia do innych obiektów — to będą nasze „łapki", trzymające inne obiekty, tym razem już skierowane. Rzecz jasna, śmieciarka musi być w stanie określić, które to konkretnie komórki pamięci zawierają istotne adresy. Zazwyczaj wyznacza to na podstawie danych o typie obiektu. Niektóre obiekty (np. globalne) są nieusuwalne, podobnie jak małpka nr 1 przyczepiona do gałęzi drzewa.



Rysunek 2: Przykładowy graf obiektów w pamięci

Rysunek 2 ilustruje przykładowy graf obiektów w pamięci. Wyróżnione obiekty A i B są globalne. Gdy usunięte zostaną odniesienia pokazane kropkowanymi strzałkami, wtedy niepotrzebne są obiekty C, E i F.

Prostą metodą znajdowania nieużywanych obiektów jest zliczanie referencji. Gdyby w danym obiekcie przechowywać liczbę odniesień do niego, i modyfikować ją za każdym razem, gdy odniesienia do tego obiektu pojawiają się lub znikają, to możemy usunąć obiekt, gdy liczba odwołań spadnie do zera. To rozwiązanie nie sprawdza się, gdy nieużywane obiekty tworzą cykle obiektów wzajemnie podtrzymujących się przy życiu.

Aby wyznaczyć wszystkie niepotrzebne obiekty, śmieciarka przeszukuje graf obiektów, startując od tych nieusuwalnych obiektów, i zaznacza odwiedzone. Po zakończeniu przeszukiwania, obiekty nie zaznaczone są nieosiągalne dla programu, a zatem zbędne, i możemy je usunąć. Przy okazji można obiekty poprzesuwać w pamięci, by ją zdefragmentować — dzięki temu łatwiej jest przydzielać później pamięć.

Takie przeszukiwanie całego grafu obiektów jest stosunkowo długą operacją, blokującą działanie reszty programu. Nie znamy metody, która by potrafiła wszystko odśmiecić na bieżąco, tak jak nie potrafimy rozwiązać „Małpek" w czasie liniowym prostą symulacją.

Rozwiązanie: czas odwrócony

Jak dotąd przekonywałem, że nie da się tego zadania zrobić efektywnie. To prawda, o ile wciąż myślimy o symulowaniu zachowania małpek na drzewie. Aby rozwiązać to zadanie, musimy spojrzeć na nie z innej strony. Podobnie jak Neo, bohater filmu „Matrix", musimy uwolnić nasz umysł od symulacji. Neo, jak wiadomo, jest świetnym hakerem i potrafi zatrzymywać czas. My zaś jesteśmy lepsi, bo potrafimy czas odwrócić. Aha!

Z łatwością możemy wczytać wszystkie dane wejściowe do pamięci i przeprowadzić proces odwrotny: zaczynając od końcowego grafu uchwytów, dodawać w nim krawędzie.

Dodając krawędź sprawdzamy, czy powoduje to przyłączenie pewnych wierzchołków do wierzchołka o numerze 1. Dzieje się tak wtedy i tylko wtedy, gdy jeden z wierzchołków krawędzi jest już w składowej spójnej wierzchołka nr 1, a drugi nie. Wtedy przyłączona zostaje cała składowa spójna drugiego wierzchołka, możemy ją wyznaczyć w czasie proporcjonalnym do liczby jej wierzchołków. Ponieważ każdy wierzchołek może zostać przyłączony do wierzchołka nr 1 najwyżej raz, łączny koszt tych operacji jest liniowy ze względu na liczbę wierzchołków. Złożoność tego algorytmu, zarówno czasowa jak i pamięciowa, wynosi $O(n)$.

A oto fragment programu wzorcowego w języku C++ wykonującego te obliczenia (pominięto wczytywanie danych i ich wstępną obróbkę):

```
const int NIGDY = -1;  // małpka na drzewie, nigdy nie spadła
const int ZIEMIA = -2; // małpka leży na ziemi

int n, m;

int uchwyt[MAXM][2]; // kto kogo puścił w danej chwili czasu
vector<int> krawedzie[MAXN];  // krawędzie grafu uchwytów
```

```
int czas[MAXN];  // czas, w którym spadła dana małpka

// startujemy od małpki v i nadajemy wszystkim małpkom, z którymi
// jest ona związana i które leżą na ziemi, czas odwiedzin t
void dfs(int v, int t) {
  if (czas[v] != ZIEMIA) return;
  czas[v] = t;
  int sasiadow = krawedzie[v].size();
  for(int i = 0; i < sasiadow; ++i)
    dfs(krawedzie[v][i], t);
}

// obliczamy czasy upadku
void przetworz() {
  // małpki, które w ogóle nie spadły
  dfs(0, NIGDY);
  // dodajemy kolejne uchwyty
  for(int i = m - 1; i >= 0; --i) {
    int v = uchwyt[i][0], w = uchwyt[i][1];
    // czy nowe małpki połączyły się z drzewem?
    if (czas[v] == ZIEMIA && czas[w] != ZIEMIA) {
      dfs(v, i);
    } else if (czas[w] == ZIEMIA && czas[v] != ZIEMIA) {
      dfs(w, i);
    }
    // dodajemy krawędź w grafie
    krawedzie[v].push_back(w);
    krawedzie[w].push_back(v);
  }
}
```

Struktura zbiorów rozłącznych

Podobne rozwiązanie można też zaimplementować przy użyciu struktury zbiorów rozłącznych (Find-Union). Jest to znana i nieskomplikowana struktura danych, umożliwiająca w czasie niemal stałym wykonanie następujących operacji:

- sprawdzenie, do którego zbioru należy element (`find`),
- połączenie dwóch zbiorów w jeden (`union`).

Struktura Find-Union jest lasem (zbiorem drzew), w którym każdy element zna swojego ojca. Operacja `find` znajduje reprezentanta zbioru – jest nim korzeń drzewa zawierającego dany element. Operacja `union` podczepia mniejsze drzewo pod większe. Dodatkowo dla większej wydajności wykonuje się tzw. kompresję ścieżek, tzn. przy operacji `find` podczepia się przejrzane elementy bezpośrednio pod korzeń drzewa.

W naszym zadaniu korzystamy ze struktury Find-Union do przechowywania informacji o trzymających się grupach małpek. Podobnie jak poprzednio, przetwarzamy krawędzie grafu uchwytów przeciwnie do upływu czasu. Dodając krawędź, łączymy operacją `union` odpowiednie zbiory małpek. Używając operacji `find` dowiemy się przy tym, czy przyłączamy grupę małpek do małpki nr 1. Jeśli tak, musimy ustawić odpowiedni czas upadku tej grupy małpek.

Nie możemy, jak poprzednio, wykonać przeszukiwania grafu uchwytów (bo go nie mamy), by ustawić czas upadku każdej małpki z przyłączanej grupy. Struktura zbiorów rozłącznych ma postać drzewa, w którym każdy element zna swojego rodzica, a nie na odwrót, wobec czego nie da się szybko znaleźć wszystkich elementów z danej grupy. Na szczęście możemy się bez tego obejść, ustawiając czas upadku tylko w korzeniu drzewa przyłączanej grupy. Wtedy czas upadku małpki to pierwszy ustawiony czas na ścieżce od tej małpki do reprezentanta grupy. Musimy o tym pamiętać, gdy wykonujemy kompresję ścieżki.

A oto fragmenty kodu C++ definiujące strukturę zbiorów rozłącznych wzbogaconych o czas upadku:

```
struct {
  int ojciec;
  int pozycja;   // "rząd wielkości" grupy
  int czas;
} zbiory[MAXN];

int find(int v) {
  int ojciec = zbiory[v].ojciec;
  if (ojciec != v) {
    int r = find(ojciec);
    if (zbiory[v].czas == ZIEMIA)
      zbiory[v].czas = zbiory[ojciec].czas;
    zbiory[v].ojciec = r;
  }
  return zbiory[v].ojciec;
}

// zakładamy ze v, w są reprezentantami zbiorów
void union(int v, int w) {
  if (v == w) return;
  if (zbiory[v].pozycja > zbiory[w].pozycja) {
    zbiory[w].ojciec = v;
  } else {
    zbiory[v].ojciec = w;
    if (zbiory[v].pozycja == zbiory[w].pozycja)
      ++zbiory[w].pozycja;
  }
}
```

Struktura Find-Union ma to do siebie, że górne oszacowanie złożoności obliczeniowej jej operacji wyraża się dziwną funkcją α, odwrotnością funkcji Ackermanna. Nie wnikając w szczegóły matematyczne, możemy śmiało powiedzieć, że jest to funkcja bardzo, ale to bardzo wolno rosnąca, tak że praktycznie może zostać potraktowana jako stała. Dlatego, chociaż oszacowanie złożoności czasowej tego algorytmu wynosi $O(n + m\,\alpha(m,n))$, takie rozwiązanie działa wyraźnie szybciej niż poprzednie. Złożoność pamięciowa wynosi $O(n)$, w dodatku z niedużą stałą.

Testy

Do oceniania rozwiązań przygotowano 10 losowych testów. Testy losowano tak długo, aż w wyniku było wiele różnych czasów upadku na ziemię. Rozwiązanie liniowe przechodziło wszystkie testy, a rozwiązanie kwadratowe tylko cztery pierwsze.

nr testu	n
1	10
2	100
3	500
4	1000
5	10000
6	20000
7	50000
8	100000
9	150000
10	200000

Paweł Parys — Treść zadania, Opracowanie
Krzysztof Onak — Program

Tasowanie

Bajtazar ma talię złożoną z n kart, które lubi tasować. Pozycje kart w talii są ponumerowane od 1 do n. Bajtazar doszedł w tasowaniu do takiej wprawy, że za każdym razem wychodzi mu to tak samo, tzn. karta z pozycji k ($1 \leqslant k \leqslant n$) przechodzi zawsze na tę samą pozycję a_k. Takie tasowanie powtarza l razy. Na koniec karta z pozycji k znajduje się na pozycji b_k.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia liczby n i l oraz ciąg liczb (b_k),*
- *wyznaczy ciąg liczb (a_k),*
- *wypisze go na standardowe wyjście.*

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie dodatnie liczby całkowite n i l ($1 \leqslant n, l \leqslant 1\,000\,000$). W kolejnych n wierszach znajdują się kolejne elementy ciągu (b_k), po jednym w wierszu. W wierszu $k+1$ znajduje się liczba całkowita b_k — końcowa pozycja karty z pozycji k, $1 \leqslant b_k \leqslant n$.

Wyjście

Twój program powinien wypisać na standardowe wyjście n liczb całkowitych — kolejne elementy ciągu (a_k), po jednym w wierszu. W k-tym wierszu powinna się znajdować liczba a_k — pozycja karty z pozycji k po jednokrotnym tasowaniu. Możesz założyć, że dla danych testowych zawsze istnieje szukany ciąg (a_k). Jeśli jest wiele takich ciągów, Twój program powinien wypisać jeden z nich.

Przykład

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

Rozwiązanie

Najpierw wprowadzimy pewne pojęcia, które będą pomocne w zrozumieniu rozwiązania. Zamiast o tasowaniach będziemy mówić o *permutacjach*. Permutacja jest to funkcja ze zbioru $\{1, 2, \ldots, n\}$ na siebie, która każdą wartość przyjmuje dokładnie raz. Permutację można *składać*, tzn. powtarzać tasowanie. Zamiast o składaniu l razy można mówić o *podnoszeniu do potęgi* l. *Pierwiastkiem stopnia* l danej permutacji g nazwiemy taką permutację f, że f złożone ze sobą l razy daje g. W zadaniu szukamy więc pierwiastka stopnia l z danej permutacji. Podnoszenie permutacji do potęgi zachowuje się podobnie jak zwykłe potęgowanie, w szczególności: f^{a+b} jest tym samym, co f^a złożone z f^b, a f^{ab} to to samo, co $(f^a)^b$.

Cyklem w permutacji nazwiemy taki ciąg elementów $a_1, a_2, \ldots, a_k$, że $f(a_1) = a_2$, $f(a_2) = a_3, \ldots, f(a_{k-1}) = a_k$, $f(a_k) = a_1$. Dla rozwiązania tego zadania kluczowe jest rozważanie cykli naszej permutacji. Łatwo zauważyć, że cykl w początkowej permutacji przechodzi po l-krotnym złożeniu na jeden cykl lub kilka cykli tej samej długości. Ponadto, jeśli cykl jest długości k, to po k-krotnym złożeniu przechodzi on na identyczność (każdy element przechodzi na siebie). Zatem, jeśli $a \equiv b (\text{mod } k)$, to czy składamy a razy, czy b razy, dostajemy to samo.

Aby zobaczyć, co się może dziać, rozważmy najpierw pewien przykład. Ustalmy $l = 6$. Cykl długości cztery rozpada się na dwa cykle długości dwa. Również cykl długości dwanaście rozpada się na cykle długości dwa — tym razem jest ich sześć. Natomiast cykl długości dwa rozpada się na dwa cykle długości jeden. A więc po pierwsze: może się zdarzyć tak, że cykl danej długości k może powstać tylko z cyklów większej długości, a nie z cyklów długości k. Po drugie: ten sam efekt można uzyskać z różnych początkowych permutacji.

Na potrzeby tego rozwiązania wprowadzimy pojęcie *upierwszacza*, które pojawia się tutaj w sposób naturalny. Liczbę k nazwiemy upierwszaczem a względem b, jeśli k dzieli a oraz $\frac{a}{k}$ i b są względnie pierwsze. Jeśli a i b są dodatnie, to istnieje najmniejszy taki względny upierwszacz większy od zera. Oznaczmy go przez $\mathrm{NWU}(a, b)$. (Uwaga: $\mathrm{NWU}(a, b)$ i $\mathrm{NWU}(b, a)$ to co innego!) Zauważmy ponadto, że każdy upierwszacz dzieli się przez NWU. Popatrzmy na to pojęcie ze strony rozkładu na liczby pierwsze. Jeśli $a = p_1^{\alpha_1} \ldots p_k^{\alpha_k} r_1^{\gamma_1} \ldots r_m^{\gamma_m}$ oraz $b = q_1^{\beta_1} \ldots q_l^{\beta_l} r_1^{\delta_1} \ldots r_m^{\delta_m}$, przy czym wszystkie p, q, r są różnymi liczbami pierwszymi oraz wszystkie α, β, γ, δ są dodatnimi liczbami całkowitymi, to wtedy $\mathrm{NWU}(a, b) = r_1^{\gamma_1} \ldots r_m^{\gamma_m}$.

Rozwiązanie opiera się na następującym fakcie: Permutacja posiada (co najmniej jeden, być może więcej) pierwiastek stopnia l wtedy i tylko wtedy, gdy dla każdego $k > 0$ liczba cykli długości k jest podzielna przez $\mathrm{NWU}(l, k)$.

Dowiedziemy najpierw implikacji w prawo. Zakładamy, że istnieje rozwiązanie — pewien pierwiastek naszej permutacji. Skupmy uwagę na jednym z cykli. Ma on pewną długość k. Przy l krotnym złożeniu permutacji rozpada się on na $\mathrm{NWD}(l, k)$ cykli. Każdy z powstałych cykli będzie długości $k' = \frac{k}{\mathrm{NWD}(k,l)}$. Zauważmy, że $\mathrm{NWD}(l, k)$ jest upierwszaczem l względem k' (ponieważ $\frac{k}{\mathrm{NWD}(k,l)}$ i $\frac{l}{\mathrm{NWD}(k,l)}$ są względnie pierwsze). A zatem $\mathrm{NWU}(l, k')$ dzieli $\mathrm{NWD}(l, k)$. Oznacza to, że liczba powstających cykli długości k' jest podzielna przez $\mathrm{NWU}(l, k')$. Cykle danej długości k' mogą powstać z kilku różnych cykli początkowych, ale ich liczba i tak jest podzielna przez $\mathrm{NWU}(l, k')$.

Pora teraz na dowód w drugą stronę. Sam dowód jednak nie wystarczy — musimy wiedzieć, jak skonstruować rozwiązanie. Popatrzmy najpierw na cykle długości k względnie pierwszej z l. Z podstawowych faktów teorii liczb wynika, że w tym przypadku istnieje liczba naturalna p taka, że $pl \equiv 1 (\mathrm{mod}\ k)$. (Liczbę taką nazywamy *odwrotnością*.) W miarę łatwo można też taką liczbę wyznaczyć. Zauważmy, że nasz cykl podniesiony do potęgi p jest jednocześnie pierwiastkiem stopnia l. Jest tak dlatego, że cykl podniesiony do potęgi p, a następnie do l to to samo, co cykl do potęgi pl, a to to samo, co cykl do potęgi 1, czyli cykl wyjściowy, gdyż zmiany wykładnika o k nie zmieniają wyniku.

Teraz zajmijmy się przypadkiem ogólnym. Weźmy cykle ustalonej długości k, jest ich m. Niech $l' = \frac{l}{\mathrm{NWU}(l,k)}$. Wiadomo przy tym, że $\mathrm{NWU}(l,k)|m$ (z założenia) oraz że l' i k są względnie pierwsze (z definicji upierwszacza). Dla każdego cyklu długości k możemy policzyć cykl będący jego pierwiastkiem stopnia l' (jak wyżej). Bierzemy więc grupę $\mathrm{NWU}(l,k)$ cykli długości k. Dla każdego z nich obliczamy jego pierwiastek stopnia l'. Następnie „przeplatamy" $\mathrm{NWU}(l,k)$ otrzymanych cykli-pierwiastków. Oznacza to, że najpierw bierzemy po jednym elemencie z każdego cyklu, potem bierzemy po następnym elemencie z każdego cyklu, itd. Powstanie w ten sposób cykl taki, że podniesiony do potęgi $\mathrm{NWU}(l,k)$ rozpada się na nasze cykle-pierwiastki, a zatem jest on pierwiastkiem stopnia $\mathrm{NWU}(l,k)l' = l$ dla grupy cykli.

A więc postępujemy tak:

1. Znajdujemy cykle w permutacji i zliczamy, ile jest cykli danej długości.
2. Dopóki nie przetworzymy wszystkich cykli, wykonujemy, co następuje:
   (a) Bierzemy dowolną, jeszcze nie przetworzoną, grupę $\mathrm{NWU}(l,k)$ cykli pewnej długości k.
   (b) Liczymy p takie, że $l'p \equiv 1 (\mathrm{mod}\ k)$, gdzie $l' = \frac{l}{\mathrm{NWU}(l,k)}$.
   (c) Podnosimy każdy z tych cykli do potęgi p.
   (d) „Przeplatamy" te cykle między sobą.

Pozostaje jeszcze parę szczegółów. Liczenie odwrotności l' modulo k można zaimplementować efektywnie za pomocą algorytmu Euklidesa (i tak w innych sytuacjach powinno się to robić). Tutaj jednak w zupełności wystarczy sprawdzanie wszystkich liczb od 1 do $k-1$ — złożoność całego programu i tak się nie pogarsza. Trzeba już tylko wiedzieć, jak liczyć $\mathrm{NWU}(a,b)$. Będziemy ciągle dzielić a przez $\mathrm{NWD}(a,b)$ (w pętli) aż do momentu, gdy $\mathrm{NWD}(a,b) = 1$. Operacje, takie jak określanie długości cykli, podnoszenie cykli do potęgi oraz „przeplatanie" cykli, można bez większego problemu, po chwili zastanowienia, zaimplementować tak, aby złożoność całego programu była liniowa względem n.

Testy

Zadanie było sprawdzone na zestawie 12 danych testowych, przy czym testy 1, 1a i 1b były zgrupowane. Testy powstawały w ten sposób, że losowano permutację ustalonej długości n i podnoszono ją do ustalonej potęgi l. Poniższa tabelka przedstawia parametry n i l dla poszczególnych testów.

nr testu	n	l
1	20	2
1a	20	3
1b	1	1000000
2	50	49
3	200	100
4	1000	36
5	2000	102
6	100000	53625
7	500000	42452
8	1000000	900001
9	1000000	13
10	1000000	223100

XIV Międzynarodowa Olimpiada Informatyczna, Yong-In 2002

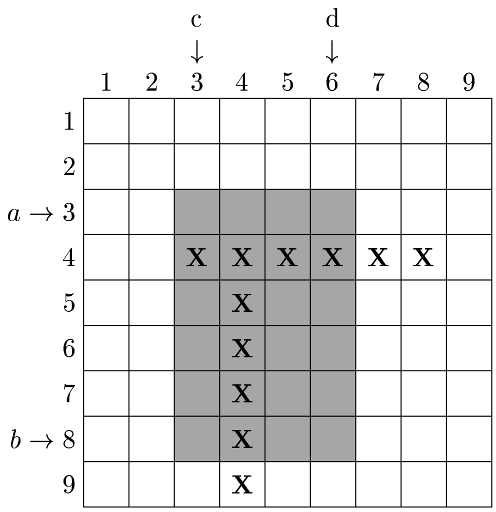
XIV Międzynarodowa Olimpiada Informatyczna — treści zadań

Krzysztof Diks, Marcin Kubica, Tomasz Waleń
Tłumaczenie

Kije-samobije (Two rods)

ZADANIE

Kijem-samobijem nazywamy poziomy lub pionowy ciąg co najmniej dwóch sąsiednich kratek na kratownicy. Na kratownicy $N \times N$ znajdują się dwa kije samobije, poziomy i pionowy. Na rysunku 1 kije są zaznaczone za pomocą znaków **X**. *Kije mogą być tej samej lub różnej długości, ponadto mogą mieć wspólną kratkę. Na rysunku 1 kratka $(4,4)$ może być interpretowana jako należąca do jednego kija lub do dwóch kijów. W takim przypadku przyjmujemy, że kratka należy do obu kijów. Zatem górnym końcem kija pionowego jest kratka $(4,4)$, a nie $(5,4)$.*



Rys. 1: Przykładowe położenie kijów

Na początku nie znamy położenia kijów na kratownicy. Twoim zadaniem jest napisanie programu, który je zlokalizuje. Poziomy kij nazywamy ROD1, natomiast pionowy ROD2. Każda kratka jest reprezentowana przez parę współrzędnych (r, c) — (wiersz, kolumna), a górny lewy róg kratownicy ma współrzędne $(1,1)$. Każdy kij ma dwa końce i jest reprezentowany przez ich współrzędne $[(r_1, c_1), (r_2, c_2)]$. Na rysunku 1 ROD1, to $[(4,3), (4,8)]$, natomiast ROD2 to $[(4,4), (9,4)]$.

To zadanie wymaga użycia biblioteki funkcji dla odczytu danych, znalezienia rozwiązania oraz wypisania wyniku. Długość boku kratownicy otrzymuje się za pomocą funkcji `gridsize`, *którą Twój program musi wywołać na początku każdego testu. Do zlokalizowania kijów samobijów można korzystać jedynie z funkcji bibliotecznej* `rect(a, b, c, d)`, *która bada prostokąt $[a,b] \times [c,d]$ (zacieniowany obszar na rysunku 1), gdzie $a \leqslant b$ i $c \leqslant d$. (Nie zapomnij o takiej właśnie kolejności parametrów.) Jeśli w zadanym prostokącie znajduje się chociaż jedna kratka* *należąca do któregokolwiek kija-samobija, to wynikiem* `rect` *jest 1, w przeciwnym wypadku jej wynikiem jest 0. Tak więc w przykładzie wynikiem* `rect(3, 8, 3, 6)` *jest 1. Twoim zadaniem jest napisanie programu znajdującego dokładne położenie kijów samobijów za pomocą ograniczonej liczby wywołań* `rect`*.*

Wynik powinien zostać podany za pomocą `report(`r_1`,` c_1`,` r_2`,` c_2`,` p_1`,` q_1`,` p_2`,` q_2`)` *gdzie ROD1 to* $[(r_1, c_1), (r_2, c_2)]$*, natomiast ROD2 to* $[(p_1, q_1), (p_2, q_2)]$*. Wywołanie* `report` *kończy wykonanie twojego programu. Pamiętaj, że ROD1 jest kijem poziomym, a ROD2 pionowym, oraz kratka* (r_1, c_1) *jest lewym końcem poziomego kija ROD1. Kratka* (p_1, q_1) *jest górnym końcem kija ROD2. Stąd* $r_1 = r_2$*,* $c_1 < c_2$*,* $p_1 < p_2$ *i* $q_1 = q_2$*. Jeśli wywołasz funkcję* `report` *z argumentami nie spełniającymi tych wymagań, to otrzymasz komunikat o błędzie na standardowym wyjściu.*

Wymagania

- *Do danych wejściowych możesz odwoływać się tylko za pomocą funkcji bibliotecznych* `gridsize` *i* `rect`*.*
- N*, rozmiar kratownicy, spełnia nierówności* $5 \leqslant N \leqslant 10\,000$*.*
- *Dla każdego testu liczba wywołań* `rect` *powinna być nie większa niż 400 razy. Wywołanie* `rect` *więcej niż 400 razy spowoduje przerwanie wykonywania Twojego programu.*
- *Twój program musi wywołać* `rect` *więcej niż raz, natomiast funkcja* `report` *musi zostać wywołana dokładnie raz.*
- *Niepoprawne wywołanie* `rect` *(na przykład z argumentami opisującymi prostokąt sięgający poza kratownicę) spowoduje przerwanie wykonywania Twojego programu.*
- *Twój program nie może odwoływać się (czytać/pisać) do żadnych plików ani korzystać ze standardowego wejścia/wyjścia.*

Biblioteka

FreePascal Library (`prectlib.ppu`, `prectlib.o`)

```
function gridsize: LongInt;
function rect(a, b, c, d : LongInt) : LongInt;
procedure report(r1, c1, r2, c2, p1, q1, p2, q2 : LongInt);
```

Instrukcja: *Aby skompilować program* `rods.pas` *umieść w kodzie źródłowym następujące polecenie*

```
uses prectlib;
```

i wykonaj

```
fpc -So -O2 -XS rods.pas
```

Program `prodstool.pas` *jest przykładem użycia biblioteki dla FreePascala.*

GNU C/C++ Library (`crectlib.h`, `crectlib.o`)

```
int gridsize();
int rect(int a, int b, int c, int d);
void report(int r1, int c1, int r2, int c2,
int p1, int q1, int p2, int q2);
```

Instrukcja: *Aby skompilować program* `rods.c` *umieść w kodzie źródłowym następujące polecenie*

```
#include "crectlib.h"
```

i wykonaj

```
gcc -O2 -static rods.c crectlib.o -lm
```

lub (dla C++)

```
g++ -O2 -static rods.cpp crectlib.o -lm
```

Program `crodstool.c` *jest przykładem użycia biblioteki dla GNU C/C++.*
Uwaga: *Dla C/C++ w środowisku RHIDE nie zapomnij ustawić w* **Option | Linker** *konfiguracji na* `crectlib.o`*.*

Eksperymenty

Żeby poeksperymentować z biblioteką musisz utworzyć plik tekstowy `rods.in`*. Plik ten musi zawierać dokładnie trzy wiersze. Wiersz pierwszy zawiera jedną liczbę całkowitą N — rozmiar kratownicy. Wiersz drugi zawiera współrzędne kija ROD1 — r_1 c_1 r_2 c_2 — gdzie (r_1, c_1) jest lewym końcem ROD1. Trzeci wiersz zawiera współrzędne kija ROD2 — p_1 q_1 p_2 q_2, gdzie (p_1, q_1) jest górnym końcem ROD2.*

Po wykonaniu programu zakończonego wywołaniem `report` *zostanie utworzony plik wynikowy* `rods.out`*. Plik ten zawiera liczbę wywołań funkcji* `rect` *oraz współrzędne końców kijów, które Twój program podał jako parametry* `report`*. Jeśli były błędy lub nastąpiło naruszenie wymagań odnoszących się do wywołań funkcji bibliotecznych, to plik* `rods.out` *będzie zawierał stosowne komunikaty o błędach.*

Dialog pomiędzy Twoim programem a biblioteką jest odnotowany w pliku `rods.log`*. Plik* `rods.log` *zawiera opis kolejnych wywołań funkcji* `rect` *w postaci „*`k : rect(`a`, `b`, `c`, `d`) = ` ans*”, co oznacza, że w k-tym wywołaniem funkcji* `rect` *było* `rect(`a`, `b`, `c`, `d`)` *z wynikiem* ans*.*

Przykład wejścia i wyjścia

`rods.in`:

```
9
4 3 4 8
4 4 9 4
```

`rods.out`:

```
20
4 3 4 8
4 4 9 4
```

Punktacja

Jeśli Twój program naruszy którekolwiek z ograniczeń (np. liczba wywołań funkcji `rect` *przekroczy 400) lub wyjście (lokalizacja kijów samobijów) będzie niepoprawne, dostaniesz 0 punktów.*

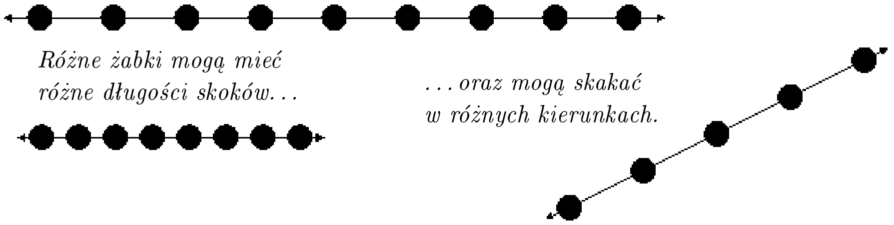
Jeśli Twój program odpowiedział poprawnie, wówczas Twoja punktacja zależy od liczby wywołań funkcji `rect` *(dla każdego testu). Dla każdego testu, jeśli liczba wywołań* `rect` *była co najwyżej 100, dostajesz 5 punktów za test. Jeśli liczba wywołań wynosiła od 101 do 200, dostajesz 3 punkty. Jeśli zaś liczba wywołań rect była w przedziale od 201 do 400, dostajesz 1 punkt.*

Krzysztof Diks, Marcin Kubica, Tomasz Waleń
Tłumaczenie

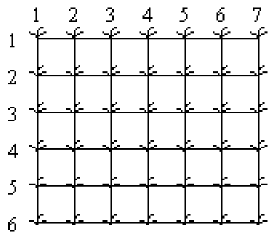
Kłopotliwe żabsko (The troublesome frog)

Zadanie

Cheonggaeguri, *mała żabka, jest dobrze znana w Korei ze swej uciążliwości. Zasłużyła sobie ona na to, skacząc nocą po poletkach ryżowych i rozdeptując sadzonki ryżu. Rano, widząc, które sadzonki zostały rozdeptane, chcesz odnaleźć trasę żabki, która dokonała największych zniszczeń. Żabki zawsze skaczą po poletku wzdłuż linii prostej, wykonując skoki tej samej długości.*

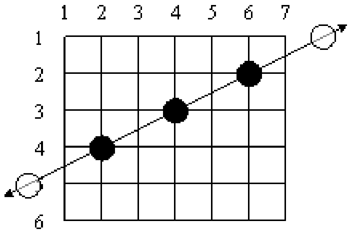


Na Twoim poletku sadzonki ryżu rosną w punktach o współrzędnych całkowitych, tak jak to pokazano na rys. 1, a uciążliwe żabsko przemierza zawsze całe poletko, wskakując na nie z jednej strony i opuszczając je z drugiej strony, tak jak to pokazano na rys. 2.

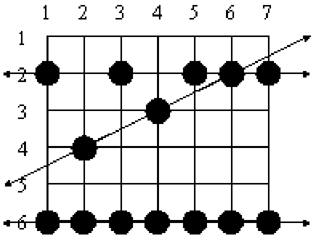


Rys. 1: Sadzonki ryżu

Po poletku może skakać wiele żabek, przeskakując z sadzonki na sadzonkę. Każdy skok kończy się na sadzonce, która zostaje rozdeptana, tak jak to pokazano na rys. 3. Zauważ, że niektóre sadzonki mogą w ciągu nocy zostać rozdeptane wielokrotnie. Oczywiście linie przedstawiające trasy żabek nie są widoczne, tak samo poza poletkiem żabki nie zostawiają żadnych śladów — w przypadku sytuacji przedstawionej na rys. 3 widoczne byłyby tylko ślady przedstawione na rys. 4.

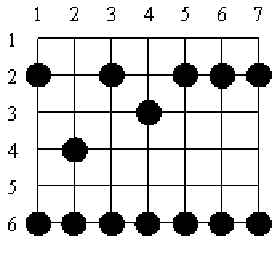


Rys. 2: Przykładowa trasa żabki



Rys. 3: Trasy żabek

Na podstawie rys. 4 można odtworzyć wszystkie trasy, jakimi żabki mogły się przemieszczać przez Twoje poletko. Interesują nas tylko te żabki, które przemierzając poletko ryżowe wylądowały na nim przynajmniej trzykrotnie. Taką trasę będziemy nazywać trasą żabki. W tym przypadku, wszystkie trzy trasy przedstawione na rys. 3 są trasami żabek (istnieją również inne możliwe trasy żabek). Trasa wzdłuż kolumny nr 1 mogłaby być trasą żabki ze skokiem długości 4, gdyby nie to, że zawiera tylko dwie rozgniecione sadzonki, a więc nie interesuje nas. Podobnie, ukośna trasa przechodząca przez sadzonki: w 2 rzędzie i 3 kolumnie, 3 rzędzie i 4 kolumnie, oraz 6 rzędzie i 7 kolumnie, obejmuje trzy sadzonki, ale nie istnieje taka długość skoku żabki, która obejmowałaby te trzy sadzonki i prowadziłaby przez rozdeptane sadzonki, dlatego też nie jest to trasa żabki. Zwróć uwagę na to, że prosta prowadząca przez trasę żabki



Rys. 4: Ślady żabek

może przechodzić przez dodatkowe rozdeptane sadzonki, które nie należą do trasy żabki (na rys. 4, spójrz na sadzonkę o współrzędnych (2, 6) na poziomej prostej biegnącej wzdłuż 2-go rzędu), a niektóre rozdeptane sadzonki mogą nie należeć do żadnej trasy żabki.

Twoje zadanie polega na napisaniu programu, który wyznaczy maksymalną liczbę rozgniecionych sadzonek należących do trasy jednej żabki (biorąc maksimum ze wszystkich możliwych tras żabek). Na rys. 4 wynik jest równy 7, odpowiada on trasie idącej wzdłuż rzędu nr 6.

Wejście

Twój program powinien czytać ze standardowego wejścia. Pierwszy wiersz zawiera dwie liczby całkowite R i C, oznaczające odpowiednio liczbę wierszy i kolumn na Twoim poletku, $1 \leqslant R, C \leqslant 5\ 000$. Drugi wiersz zawiera jedną liczbę całkowitą N, równą liczbie rozdeptanych sadzonek, $3 \leqslant N \leqslant 5\ 000$. Każdy z pozostałych N wierszy zawiera po dwie liczby całkowite, numer wiersza ($1 \leqslant$ numer wiersza $\leqslant R$) i numer kolumny ($1 \leqslant$ numer kolumny $\leqslant C$), w których znajduje się rozdeptana sadzonka, oddzielone pojedynczym odstępem. Każda rozdeptana sadzonka jest wymieniona tylko raz.

Wyjście

Twój program ma wypisywać na standardowe wyjście. Wyjście powinno się składać z jednego wiersza zawierającego pojedynczą liczbę całkowitą — liczbę rozdeptanych sadzonek należących do trasy żabki, która poczyniła największe szkody, jeśli istnieje przynajmniej jedna trasa żabki, natomiast w przeciwnym przypadku 0.

Przykłady wejścia i wyjścia

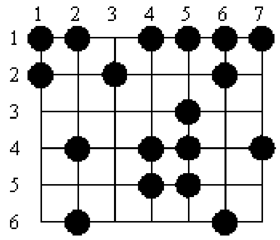
Przykład 1

Wejście:

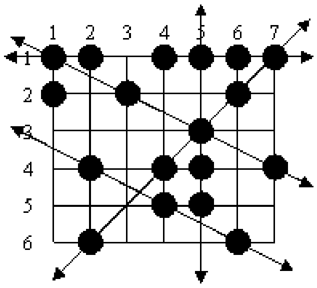
```
6 7
14
2 1
6 6
4 2
2 5
2 6
2 7
3 4
6 1
6 2
2 3
6 3
6 4
6 5
6 7
```

Wyjście:

```
7
```



Rys. 5: Ślady żabek



Rys. 6: Maksymalna liczba sadzonek rozdeptanych przez żabkę wynosi 4

Przykład 2

Wejście (Przykład z rys. 5)

```
6 7
18
1 1
6 2
3 5
1 5
4 7
1 2
1 4
1 6
1 7
2 1
```

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

Wyjście:

```
4
```

Ocena

Jeśli w danym teście Twój program wypisze poprawny wynik w zadanym czasie, to otrzyma komplet punktów za ten test. W przeciwnym przypadku otrzyma 0 punktów.

Krzysztof Diks, Marcin Kubica, Tomasz Waleń
Tłumaczenie

Przystanki autobusowe (Bus terminals)

Zadanie

Miasto Yong-In planuje budowę sieci autobusowej składającej się z N przystanków umiejscowionych na skrzyżowaniach. Yong-In to bardzo nowoczesne miasto, stąd jego mapa jest siatką składającą się z kwadratowych kwartałów o jednakowych rozmiarach. Dwa przystanki zostaną wybrane jako centra komunikacyjne H_1 i H_2. Centra będą połączone ze sobą bezpośrednią linią, a każdy z $N-2$ pozostałych przystanków zostanie połączony bezpośrednio albo z H_1, albo z H_2 (ale nie jednocześnie do obydwu). Nie ma bezpośrednich połączeń pomiędzy przystankami, które nie są centrami przesiadkowymi.

Odległość pomiędzy dwoma przystankami jest równa długości najkrótszej drogi pomiędzy nimi biegnącej ulicami Yong-In. Przystanki są reprezentowane przez pary współrzędnych (x,y). Jeśli pierwszy przystanek ma współrzędne (x_1,y_1), a drugi (x_2,y_2), to odległość pomiędzy nimi wynosi $|x_1-x_2|+|y_1-y_2|$. Jeśli przystanki A i B są połączone bezpośrednio z tym samym centrum komunikacyjnym H, to długość drogi, którą należy przejechać z A do B, jest równa sumie odległości pomiędzy A i H oraz pomiędzy H i B. Jeśli przystanki są bezpośrednio połączone z różnymi centrami komunikacyjnymi, np. A z H_1 i B z H_2, to długość drogi, którą należy pokonać między nimi jest sumą odległości A i H_1, H_1 i H_2 oraz H_2 i B.

Planujący sieć autobusową chcą zapewnić, żeby każdy obywatel dojechał do dowolnego punktu docelowego najszybciej jak się da. Oznacza to, że należy wybrać centra komunikacyjne w taki sposób, żeby najdłuższa droga pomiędzy dowolnymi dwoma przystankami była jak najkrótsza.

Wybór centrów P jest lepszy od wyboru Q, jeśli najdłuższa droga prowadząca pomiędzy dwoma przystankami przy wyborze P jest krótsza od takiej drogi przy wyborze Q.

Wejście

Twój program powinien czytać ze standardowego wejścia. Pierwszy wiersz zawiera jedną dodatnią liczbę całkowitą N, $2 \leqslant N \leqslant 500$, równą liczbie przystanków autobusowych. W każdym z pozostałych N wierszy znajdują się współrzędne jednego przystanku, odpowiednio współrzędna x i współrzędna y. Współrzędne są dodatnimi liczbami całkowitymi $\leqslant 5\,000$. Żadne dwa przystanki nie znajdują się w tym samym miejscu.

Wyjście

Twój program powinien pisać na standardowe wyjście. Wyjście składa się z jednego wiersza zawierającego jedną dodatnią liczbę całkowitą, równą długości najdłuższej drogi pomiędzy dwoma przystankami dla najlepszego wyboru centrów komunikacyjnych.

Przykłady wejścia i wyjścia

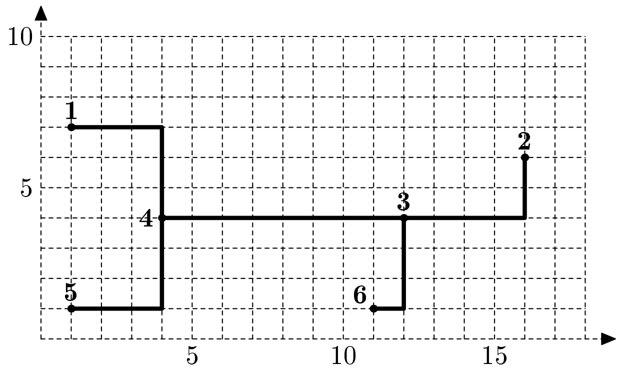
Przykład 1

Wejście:

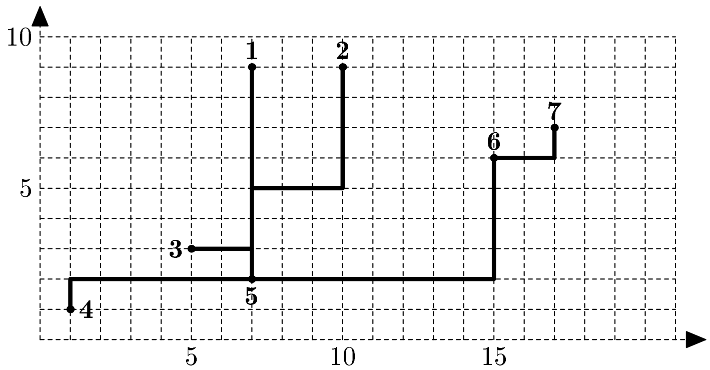
```
6
1 7
16 6
12 4
4 4
1 1
11 1
```

Wyjście:

```
20
```



Rys. 1: Sieć dla przykładu 1



Rys. 2: Sieć dla przykładu 2

Przykład 2

Wejście:

```
7
7 9
10 9
5 3
1 1
7 2
15 6
17 7
```

Wyjście:

```
25
```

Rysunki 1 i 2 przedstawiają sieci autobusowe dla powyższych danych. Jeśli w przykładzie 1, przystanki 3 i 4 zostaną wybrane jako centra komunikacyjne, to najdłuższa droga prowadzi od przystanku 2 do 5 lub od 2 do 1. Nie istnieje lepszy wybór centrów, tak więc prawidłową odpowiedzią jest w tym przypadku 20.

Dla sieci z przykładu 2, jeśli za centra zostaną wybrane przystanki 5 i 6, to najdłuższa droga prowadzi między przystankami 2 i 7. Jest to najlepszy wybór, a zatem prawidłową odpowiedzią jest 25.

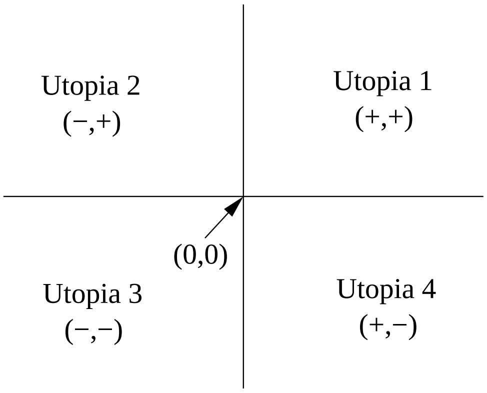
Punktacja

Jeśli Twój program wypisze poprawną odpowiedź na test, w zadanym ograniczeniu czasowym, wówczas zdobywasz maksymalną liczbę punktów za ten test. W przeciwnym przypadku dostajesz 0 punktów.

Krzysztof Diks, Marcin Kubica, Tomasz Waleń
Tłumaczenie

Podzielona Utopia (Utopia divided)

Dawno temu straszna wojna zniszczyła przepiękną Utopię. W jej wyniku kraj został podzielony na cztery obszary. Obszary te są wyznaczone przez dwie linie proste (granice): północ-południe i wschód-zachód. Przecięcie tych dwóch linii uznano za punkt $(0,0)$. Wszystkie cztery obszary rościły sobie prawo do nazwy Utopia, ale z biegiem czasu zaczęto je nazywać Utopia 1 (obszar północno-wschodni), Utopia 2 (obszar północno-zachodni), Utopia 3 (obszar południowo-zachodni) i Utopia 4 (obszar południowo-wschodni). Punkt w obszarze jest wyznaczany przez swoje „odległości" w kierunku wschodnim i północnym od punktu $(0,0)$. „Odległości" mogą być ujemne. Tak więc punkty w obszarze Utopia 2 są parami (ujemna, dodatnia), w Utopia 3 — (ujemna, ujemna), w Utopia 4 — (dodatnia, ujemna), natomiast w Utopia 1 — (dodatnia, dodatnia).



Głównym problemem, z jakim spotykają się mieszkańcy, jest brak możliwości przekraczania granic. Szczęśliwie genialny uczestnik Olimpiady Informatycznej pochodzący z Utopii wynalazł bezpieczny sposób teleportacji. Do tego celu są potrzebne są liczby kodowe, z których każda może być użyta tylko raz. Twoim zadaniem jest wykonać ciąg teleportacji z punku $(0,0)$ przez obszary Utopii w zadanym porządku. Nie jest ważne, w którym miejscu obszaru wylądujesz, ale musisz wykonać kolejno N lądowań, każde w zadanym obszarze. Może się zdarzyć, że w tym samym obszarze masz wylądować 2 lub więcej razy pod rząd. Po opuszczeniu punktu $(0,0)$ nie wolno Ci nigdy wylądować na granicy obszarów.

Otrzymujesz na wejściu ciąg $2N$ liczb kodowych i masz zapisać je jako ciąg N par kodowych, umieszczając znak + lub − przed każdą z liczb. Jeśli w danym miejscu znajdujesz się w punkcie (x,y) i używasz pary kodowej $(+u,-v)$, to zostaniesz teleportowany do punktu $(x+u,y-v)$. Masz do dyspozycji $2N$ liczb i możesz je dowolnie połączyć w pary, pisząc + lub − przed każdą z liczb.

Załóżmy, że dany jest ciąg liczb kodowych 7, 5, 6, 1, 3, 2, 4, 8 i masz odwiedzić kolejno obszary o numerach 4, 1, 2, 1. Ciąg par kodowych $(+7,-1)$, $(-5,+2)$, $(-4,+3)$,

$(+8, +6)$ pozwoli Ci się przemieścić z punktu $(0, 0)$ kolejno do punktów $(7, -1)$, $(2, 1)$, $(-2, 4)$, $(6, 10)$. Punkty te znajdują się odpowiednio w obszarach Utopia 4, Utopia 1, Utopia 2, Utopia 1.

Zadanie

Dane jest $2N$ różnych liczb kodowych oraz ciąg N numerów obszarów wskazujących miejsca lądowań. Z podanych liczb zbuduj ciąg par kodowych umożliwiający przemieszczanie się przez obszary w zadanej kolejności.

Wejście

Twój program powinien czytać dane ze standardowego wejścia. Pierwszy wiersz zawiera dodatnią liczbę całkowitą N ($1 \leqslant N \leqslant 10\,000$). Drugi wiersz zawiera $2N$ różnych całkowitych liczb kodowych ($1 \leqslant$ liczba kodowa $\leqslant 100\,000$) pooddzielanych pojedynczymi odstępami. Ostatni wiersz zawiera ciąg N numerów obszarów, z których każdy jest równy 1, 2, 3 lub 4.

Wyjście

Twój program powinien pisać na standardowe wyjście. Wyjście składa się z N wierszy, każdy zawierający parę liczb kodowych, każda poprzedzona znakiem (+/−). Pary liczb są kodami prowadzącymi teleportowanego przez zadany ciąg obszarów. Uwaga: między znakiem a liczbą nie może być znaku odstępu, a po pierwszej liczbie kodowej musi być pojedynczy odstęp.

W przypadku wielu rozwiązań Twój program może wypisać dowolne z nich. Jeśli nie ma rozwiązania, Twój program powinien wypisać tylko jedną liczbę — 0.

Przykłady wejścia i wyjścia

Przykład 1

Wejście:

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

Wyjście:

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

Przykład 2

Wejście:

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

Wyjście:

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

Punktacja

Jeśli Twój program poda poprawną odpowiedź dla testu, w podanym ograniczeniu czasowym, zdobywasz maksymalną liczbę punktów za ten test, wpp. za taki test otrzymujesz 0 punktów.

Krzysztof Diks, Marcin Kubica, Tomasz Waleń
Tłumaczenie

Szeregowanie zadań (Batch scheduling)

Zadanie

Dany jest ciąg N zadań, które mają być przetworzone przez komputer. Zadania są ponumerowane od 1 do N. Ciąg zadań należy podzielić na jeden lub więcej wsadów, gdzie każdy wsad składa się z kolejnych zadań w danym ciągu. Przetwarzanie zaczyna się w chwili 0. Wsady są przetwarzane po kolei, zaczynając od pierwszego (tzn. jeśli wsad b zawiera zadania o numerach mniejszych niż wsad c, to wsad b jest przetwarzany przed wsadem c). Zadania tworzące wsad są przetwarzane przez komputer po kolei. W momencie zakończenia wsadu komputer natychmiast wypisuje wyniki wszystkich składających się nań zadań. Moment ogłoszenia wyników zadania i to moment zakończenia ostatniego zadania ze wsadu zawierającego i.

Każdy wsad wymaga czasu S pracy komputera na swoje przygotowanie. Dla każdego zadania i znana jest jego waga F_i oraz czas przetwarzania T_i. Jeśli wsad składa się z zadań x, $x+1$, ..., $x+k$ i jego przetwarzanie zaczyna się w chwili t, to wyniki przetwarzania wszystkich zadań tworzących ten wsad są wypisywane w chwili $t+S+(Tx+Tx+1+\cdots+Tx+k)$. Zwróć uwagę, że komputer wypisuje równocześnie wyniki wszystkich zadań tworzących jeden wsad. Jeśli wyniki przetwarzania zadania i są wypisywane w chwili O_i, to koszt jego przetwarzania wynosi $O_i \times F_i$. Załóżmy na przykład, że mamy 5 zadań, czas przygotowania wsadu wynosi $S = 1$, $(T_1, T_2, T_3, T_4, T_5) = (5, 5, 10, 14, 14)$ oraz $(F_1, F_2, F_3, F_4, F_5) = (3, 2, 3, 3, 4)$. Jeśli ciąg zadań zostanie podzielony na trzy wsady $\{1, 2\}$, $\{3\}$, $\{4, 5\}$, to wyniki tych zadań będą wypisywane odpowiednio w chwilach $(O_1, O_2, O_3, O_4, O_5) = (5, 5, 10, 14, 14)$, a koszty przetworzenia zadań wyniosą odpowiednio $(15, 10, 30, 42, 56)$. Całkowity koszt podziału ciągu zadań na wsady to suma kosztów przetworzenia wszystkich zadań. Dla powyższego przykładowego podziału ciągu zadań całkowity koszt wynosi 153.

Twoim zadaniem jest napisanie programu, który na podstawie czasu przygotowania wsadu, oraz ciągu zadań wraz z ich czasami przetwarzania i wagami, obliczy minimalny całkowity koszt podziału ciągu zadań.

Wejście

Twój program powinien wczytywać dane ze standardowego wejścia. Pierwszy wiersz zawiera liczbę zadań N, $1 \leqslant N \leqslant 10\,000$. Drugi wiersz zawiera liczbę całkowitą S równą czasowi przygotowania wsadu, $0 \leqslant S \leqslant 50$. Kolejne N wierszy zawiera informacje o zadaniach 1, 2, ..., N, w takiej właśnie kolejności. Na początku każdego z tych wierszy znajduje się liczba całkowita T_i, $1 \leqslant T_i \leqslant 100$, czyli czas przetwarzania zadania i. Po niej występuje liczba całkowita F_i, $1 \leqslant F_i \leqslant 100$, waga zadania i.

Wyjście

Twój program powinien pisać na standardowe wyjście. Wyjście składa się z jednego wiersza, który zawiera jedną liczbę całkowitą: minimalny całkowity koszt podziału ciągu zadań na wsady.

Przykłady wejścia i wyjścia

Przykład 1

Wejście:

```
2
50
100 100
100 100
```

Wyjście:

```
45000
```

Przykład 2

Wejście:

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

Wyjście:

```
153
```

Przykład 2 to przykład z treści zadania.

Uwaga

W każdym z testów całkowity koszt dowolnego podziału ciągu zadań nie przekracza $2^{31} - 1$.

Punktacja

Jeśli w danym teście Twój program wypisze poprawny wynik w zadanym czasie, to otrzymasz komplet punktów za ten test. W przeciwnym przypadku otrzymasz 0 punktów.

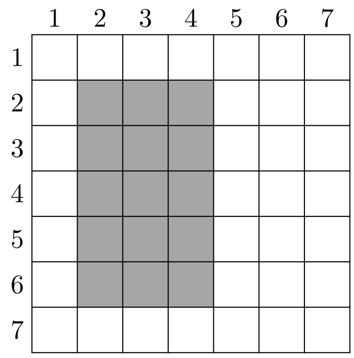
Krzysztof Diks, Marcin Kubica, Tomasz Waleń
Tłumaczenie

XOR

Problem

Twoim zadaniem jest napisanie aplikacji dla telefonów komórkowych. Telefony mają czarno-białe ekrany. Współrzędne x, pikseli na ekranie, rosną od lewej do prawej, natomiast współrzędne y od góry do dołu — zobacz rysunki poniżej. Aplikacja wykorzystuje wiele rysunków, z których nie wszystkie są tego samego rozmiaru. Zamiast pamiętać obrazki w pamięci telefonu, chcesz tworzyć je za pomocą biblioteki graficznej wbudowanej w telefon. Możesz założyć, że na początku rysowania ekran jest cały biały. Biblioteka graficzna zawiera tylko jedną operację `XOR(`L`,`R`,`T`,`B`)`*, która zmienia kolory pikseli na przeciwne w obrębie prostokąta o lewym górnym rogu w (L,T) i dolnym prawym rogu w (R,B), gdzie L oznacza* left *(lewy), T —* top *(góra), R —* right *(prawy), B —* bottom *(dół). Uwaga: w niektórych innych bibliotekach graficznych kolejność argumentów jest inna.*

Dla przykładu rozważmy rysunki 1, 2 i 3. Zastosowanie operacji `XOR(2,4,2,6)` *do białego ekranu daje obrazek z rysunku 1. Zastosowanie* `XOR(3,6,4,7)` *do obrazu z rysunku 1, daje obraz z rysunku 2, a po zastosowaniu* `XOR(1,3,3,5)` *do obrazu z rysunku 2 otrzymujemy obraz z rysunku 3.*

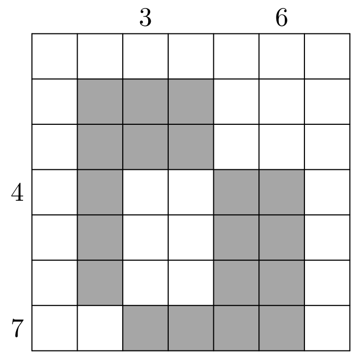


Rys. 1: Ekran po pierwszej operacji

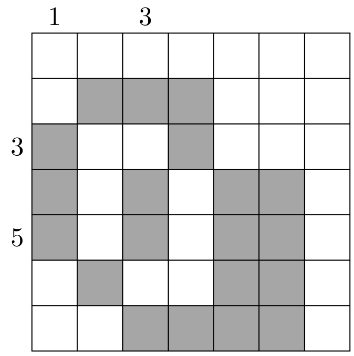
Mając dany zbiór czarno-białych obrazków, Twoim zadaniem jest wygenerowanie każdego z nich z początkowo białego ekranu za pomocą jak najmniejszej liczby wywołań `XOR`*. Dane są pliki wejściowe opisujące obrazki, dla których masz dostarczyć pliki zawierające parametry wywołań* `XOR` *(a nie program tworzący te pliki).*

Wejście

Masz danych 10 egzemplarzy problemów w plikach tekstowych od `xor1.in` *do* `xor10.in`*. Każdy plik ma następującą strukturę. Pierwszy wiersz zawiera jedną liczbę całkowitą N,*



Rys. 2: Ekran po drugiej operacjach



Rys. 3: Ekran po trzeciej operacjach

$5 \leqslant N \leqslant 2\,000$, oznaczającą że obrazek ma wymiary $N \times N$. Pozostałe wiersze pliku odpowiadają kolejnym wierszom obrazka z góry na dół. Każdy z tych wierszy zawiera N liczb całkowitych — wartości pikseli w wierszu, z lewa na prawo. Każda z tych liczb jest równa 0 lub 1, gdzie 0 oznacza kolor biały, a 1 czarny.

Wyjście

Masz dostarczyć 10 plików wejściowych odpowiadających plikom wejściowym. Pierwszy wiersz zawiera napis

`#FILE xor` I

gdzie liczba I jest numerem odpowiedniego pliku wejściowego. Drugi wiersz zawiera liczbę całkowitą K — liczbę wywołań operacji `XOR` opisanych w tym pliku. W kolejnych K wierszach zapisano parametry kolejnych wywołań `XOR`. Każdy taki wiersz zawiera 4 liczby całkowite — parametry L, R, T, B (w tej kolejności) wywołania `XOR`.

Przykład wejścia i wyjścia

Wejście (`xor0.in`*):*

```
7
0 0 0 0 0 0 0
0 1 1 1 0 0 0
1 0 0 1 0 0 0
1 0 1 0 1 1 0
1 0 1 0 1 1 0
0 1 0 0 1 1 0
0 0 1 1 1 1 0
```

Wyjście (`xor0.out`*):*

```
#FILE xor 0
3
2 4 2 6
3 6 4 7
1 3 3 5
```

Ocena

Jeśli:

- *podany przez Ciebie ciąg operacji* `XOR` *nie daje zadanego obrazka, lub*
- *liczba wywołań* `XOR` *podana w pliku wyjściowym jest różna od K, lub*
- *w pliku wyjściowym K jest większe od 40 000, lub*
- *plik wyjściowy zawiera takie wywołania* `XOR`*, że $L > R$ lub $T > B$, lub*
- *plik wyjściowy zawiera wywołanie* `XOR`*, w którym parametry nie są dodatnimi liczbami całkowitymi, lub*
- *plik wyjściowy zawiera wywołanie* `XOR`*, w którym pewien parametr jest większy niż N,*

to dostajesz 0 punktów za test. W przeciwnym wypadku Twoja punktacja jest obliczana według następującego wzoru:

$$1 + 9\,\frac{\textit{Liczba wywołań w najlepszym rozwiązaniu podanym przez zawodnika}}{\textit{Liczba wywołań w Twoim rozwiązaniu}}$$

Wynik za każdy test jest zaokrąglany do pierwszego miejsca po przecinku. Łączny wynik jest zaokrąglany do najbliższej liczby całkowitej.

Przypuśćmy, że zgłosiłeś rozwiązanie ze 121 wywołaniami `XOR`*. Jeśli to jest najlepsze ze wszystkich zgłoszonych rozwiązań, to zdobywasz 10 punktów. Jeśli najlepszym rozwiązaniem zgłoszonym przez zawodników jest 98 wywołań* `XOR`*, to dostaniesz 8,3 punktów, czyli $1+9*\frac{98}{121}$ ($= 8{,}289\ldots$) zaokrąglone do pierwszego miejsca po przecinku.*

IX Bałtycka Olimpiada Informatyczna, Tartu 2003

IX Bałtycka Olimpiada Informatyczna — treści zadań

Marcin Mucha, Piotr Sankowski
Tłumaczenie

Alokacja rejestrów (Regs)

Współczesne procesory mają ograniczoną liczbę **rejestrów** *— jednostek alokacji ogólnego przeznaczenia, które są znacząco szybsze niż pamięć główna. Operacje wykonujące obliczenia (np. dodawanie, mnożenie itp.) wymagają, aby ich argumenty znajdowały się w rejestrach i zwracały wynik w rejestrze.*

W tym zadaniu zajmujemy się problemem **alokacji rejestrów** *dla obliczenia wartości wyrażenia. Kompilator reprezentuje wyrażenia jako drzewo. Liście drzewa odpowiadają wartościom, które muszą być wczytane z pamięci głównej. Węzły pośrednie w drzewie (nie będące liśćmi) odpowiadają operacjom i każdy z nich ma tyle dzieci, ile argumentów ma ta operacja. Oczywiste jest, że wartości wszystkich argumentów muszą być dostępne, zanim operacja może być wykonana.*

Ponieważ liczba dostępnych rejestrów jest ograniczona, kompilator musi zdecydować, które wyniki pośrednie przechowywać w rejestrach (są dostępne niezwłocznie, gdy są potrzebne), a które przechowywać w pamięci głównej (muszą być załadowane z powrotem do rejestrów, gdy są potrzebne). Może się okazać, że warto także zmienić porządek wyznaczania argumentów dla operacji (właśnie dlatego większość języków wysokiego poziomu nie gwarantuje żadnego porządku obliczeń).

Twoim zadaniem jest napisanie programu, który dla otrzymanego drzewa wyrażenia, znajdzie plan alokacji rejestrów oraz kolejność obliczeń o minimalnym całkowitym koszcie.

Wejście

Pierwszy wiersz pliku wejściowego `regs.in` *zawiera liczbę rejestrów,* $N (1 \leq N \leq 100)$*. Drugi wiersz zawiera dwie liczby całkowite: koszt wczytania wartości z pamięci głównej do rejestru* C_l $(1 \leq C_l \leq 100)$ *oraz koszt zapamiętania wartości rejestru w pamięci głównej* C_s $(1 \leq C_s \leq 100)$*. Reszta pliku wejściowego zawiera opis drzewa wyrażenia, zaczynając od korzenia:*

- *pierwszy wiersz zawiera liczbę dzieci węzła,* K_x $(0 \leq K_x \leq 10$ *i* $K_x \leqslant N)$*;*
- *jeżeli* $K_x = 0$*, to ten węzeł jest liściem i opis jest zakończony;*
- *jeżeli* $K_x > 0$*, to jest to węzeł pośredni i:*
  - *następny wiersz zawiera jedną liczbę całkowitą: koszt operacji reprezentowanej przez węzeł,* C_x $(1 \leq C_x \leq 100)$*;*
  - *po tym następuje opis* K_x *poddrzew zgodnie z tym samym schematem.*

Węzły są ponumerowane od 1 do M *w kolejności, w jakiej występują w pliku wejściowym. Możesz założyć, że* $M \leq 10\,000$*.*

Wyjście

Pierwszy wiersz pliku wynikowego `regs.out` *musi zawierać minimalny koszt obliczenia wartości wyrażenia. Reszta pliku musi zawierać po jednym wierszu dla każdego węzła pośredniego w drzewie wyrażenia. Każdy z tych wierszy musi zawierać dwie liczby całkowite: pierwszą powinien być numer węzła drzewa do obliczenia, drugą powinna być 1, jeżeli wynik ma być przechowany w rejestrze, lub 0, jeżeli ma być przechowany w pamięci głównej (co zwiększa koszt całkowity o* C_s*).*

Operacje muszą być wypisane w kolejności, w której powinny być wykonane tak, aby zagwarantować, że całkowity koszt obliczania wartości wyrażenia będzie najmniejszy możliwy, przy spełnionych następujących założeniach:

- *wartość węzła może być wyznaczona dopiero po wyznaczeniu wartości jego wszystkich dzieci;*
- *wszystkie argumenty wykonywanej operacji, które nie znajdują się w rejestrach muszą być wczytane z pamięci głównej (wczytanie każdego kosztuje* C_l*);*
- *rejestry zawierające argumenty wykonywanej operacji mogą być niezwłocznie użyte ponownie, w szczególności można w jednym z nich przechować wynik tej operacji.*

Jeżeli istnieje więcej niż jeden plan o minimalnym koszcie, wypisz dowolny z nich.

Przykład

Dla pliku wejściowego `regs.in`*:*

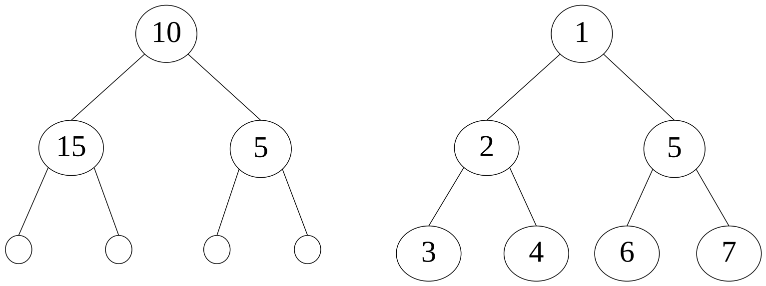
```
2
3 2
2
10
2
15
0
0
2
5
0
0
```

poprawnym wynikiem jest plik wyjściowy `regs.out`*:*

```
47
2 0
5 1
1 1
```

Uwagi

Rysunek poniżej ilustruje plik wejściowy podany powyżej: obydwa drzewa odpowiadają drzewu wyrażenia. Drzewo z lewej strony pokazuje koszt wyznaczenia wartości węzłów pośrednich, a drzewo po prawej pokazuje numerację węzłów.



Rys. 1: Koszt planu obliczeń z pliku wynikowego powyżej wynosi
$(C_l + C_l + 15 + C_s) + (C_l + C_l + 5) + (C_l + 10) = 47$.

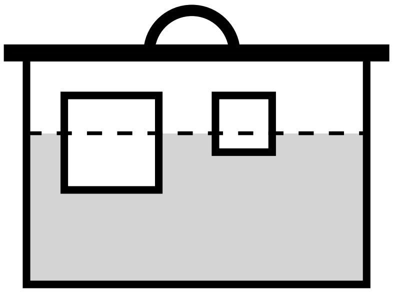
Uwagi

Otrzymasz program `REGSCHECK`*, który sprawdza poprawność (ale nie optymalność) pliku* `regs.out` *odpowiadającego plikowi wejściowemu* `regs.in`*. Program ten zwraca opisowe komunikaty błędów.*

Marcin Mucha, Piotr Sankowski
Tłumaczenie

Beczka (Barrel)

Bajtazar nalał trochę kompotu ze stołówki do beczki i wrzucił do niej sześcianiki o różnych rozmiarach i gęstościach, po czym położył na beczce pokrywkę i docisnął tak, aby dotykała brzegu beczki. Udało się! Teraz jest ciekaw, jaki poziom kompotu ustalił się w beczce.



Napisz program, który pomoże Bajtazarowi obliczyć poziom kompotu w beczce. Bajtazar wie, że:

- *gęstość kompotu wynosi 1.0,*
- *wpływ powietrza można pominąć,*
- *sześcianiki mieszczą się całkowicie w beczce,*
- *sześcianiki nie obracają się ani nie dotykają,*
- *kompot w zasadzie jest wiśniowy.*

Wejście

Twój program powinien czytać dane z pliku `barrel.in`*. Pierwszy wiersz tego pliku zawiera trzy liczby rzeczywiste — pole powierzchni podstawy beczki S ($0 < S \leq 1\ 000$), wysokość beczki H ($0 < H \leq 1\ 000$) i objętość kompotu w beczce V ($0 < V \leq S \cdot H$). Następny wiersz zawiera liczbę N wrzuconych sześcianików ($0 < N \leq 1\ 000$). W kolejnych N wierszach znajdują się opisy sześcianików. Każdy wiersz opisuje jeden sześcianik i zawiera dwie liczby rzeczywiste — długość jego krawędzi L ($0 < L \leq 1\ 000$) i jego gęstość D ($0 < D \leq 10$).*

Wyjście

Twój program powinien zapisać wynik do pliku `barrel.out`*. Plik ten powinien składać się z pojedynczego wiersza zawierającego liczbę rzeczywistą — poziom kompotu w beczce. Wynik nie powinien się różnić od poprawnej wartości o więcej niż 10^{-4}.*

Przykład

Dla pliku wejściowego `barrel.in`*:*

```
100 10 500
1
1 0.5
```

poprawnym wynikiem jest plik wyjściowy `barrel.out`*:*

```
5.0050
```

Marcin Mucha, Piotr Sankowski
Tłumaczenie

Gangi (Gangs)

Chicago lat dwudziestych — scena gansterskich batalii. Dwaj gangsterzy, którzy kiedyś się już spotkali, są albo dobrymi przyjaciółmi, albo śmiertelnymi wrogami. Gangsterzy przestrzegają następujących zasad:

1. *Przyjaciel mojego przyjaciela jest moim przyjacielem.*
2. *Wróg mojego wroga jest moim przyjacielem.*

Dwaj gangsterzy należą do jednego gangu wtedy i tylko wtedy, gdy są przyjaciółmi.

Jesteś pracownikiem chicagowskiej policji. Twoim zadaniem jest obliczenie maksymalnej możliwej liczby gangów w Chicago na podstawie wiedzy policji o stosunkach pomiędzy poszczególnymi gangsterami.

Wejście

Twój program powinien czytać dane z pliku `gangs.in`*. W pierwszym wierszu pliku wejściowego znajduje się liczba N ($2 \leqslant N \leqslant 1\,000$) gangsterów znanych policji. Gangsterzy mają numery od 1 do N. Drugi wiersz zawiera liczbę M ($1 \leqslant M \leqslant 5\,000$) znanych policji faktów dotyczących gangsterów.*

Każdy z kolejnych M wierszy zawiera opis jednego faktu. Każdy fakt jest postaci `F` *p q lub* `E` *p q (trzy składniki rozdzielone pojedynczymi odstępami), gdzie $1 \leqslant p < q \leqslant N$ są numerami gangsterów. Litera* `F` *oznacza, że o p i q wiadomo, że są przyjaciółmi, litera* `E`*, że wrogami.*

Możesz założyć, że wejście jest spójne, tzn. dwaj gangsterzy nie mogą być jednocześnie przyjaciółmi i wrogami.

Wyjście

Twój program powinien zapisać wynik do pliku `gangs.out`*. Plik ten powinien składać się z pojedynczego wiersza zawierającego maksymalną możliwą liczbę gangów.*

Przykład

Dla pliku wejściowego `gangs.in`*:*

```
6
4
E 1 4
F 3 5
F 4 6
E 1 2
```

poprawnym wynikiem jest plik wyjściowy `gangs.out`*:*

```
3
```

Uwaga: *Trzy gangi w powyższym przykładzie to $\{1\}$, $\{2, 4, 6\}$ i $\{3, 5\}$.*

Marcin Mucha, Piotr Sankowski
Tłumaczenie

Klejnoty (Gems)

Firma Zabawki-Z-Klejnotów poprosiła Cię o wykonanie następującego zadania.

Otrzymałeś spójny acykliczny graf, tzn. nie zawiera on pętli, a zbiór wierzchołków jest połączony krawędziami w taki sposób, że z każdego wierzchołka można dojść do wszystkich pozostałych, przechodząc po krawędziach.

Firma Zabawki-Z-Klejnotów zamierza produkować modele takich grafów. Wierzchołki będą zrobione z klejnotów, a krawędzie ze złotego drutu. Wymagane jest, aby sąsiednie wierzchołki były zrobione z klejnotów różnego rodzaju. Dla każdej liczby całkowitej p istnieje dokładnie jeden rodzaj klejnotów w cenie p.

Twoim zadaniem jest napisanie programu wyznaczającego minimalną całkowitą cenę klejnotów potrzebną do zrobienia takiego modelu. Mógłby to zrobić Bajtazar, ale jest teraz w delegacji.

Wejście

Pierwszy wiersz pliku wejściowego `gems.in` *zawiera jedną liczbę całkowitą N ($1 \leqslant N \leqslant 10\,000$), jest to liczba wierzchołków. Wierzchołki ponumerowane są od 1 do N. Kolejne $N-1$ wierszy w pliku opisuje krawędzie, jedną w wierszu. Każdy z tych wierszy zawiera parę liczb całkowitych A i B oddzielonych odstępem ($1 \leqslant A, B \leqslant N$, $A \neq B$). Taka para reprezentuje krawędź łączącą wierzchołki A i B.*

Wyjście

Pierwszy i jedyny wiersz pliku wynikowego `gems.out` *musi zawierać jedną liczbę całkowitą: minimalny całkowity koszt klejnotów potrzebnych do zrobienia modelu.*

Przykład

Dla pliku wejściowego `gems.in`*:*

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

poprawnym wynikiem jest plik wyjściowy `gems.out`*:*

```
11
```

Marcin Mucha, Piotr Sankowski
Tłumaczenie

Lampy (Lamps)

Wokół głównej komnaty zamkowej rozwieszone są lampy. Lampy ponumerowane są liczbami $1, 2, \ldots, N$. Lampa może być zapalona bądź zgaszona. W każdej sekundzie lampa o numerze i zmienia swój stan (zapalona/zgaszona), jeżeli lampa o numerze $i+1$ jest zapalona. Wyjątkiem jest lampa o numerze N, która zmienia swój stan, jeżeli lampa o numerze 1 jest zapalona.

Twoim zadaniem jest napisać program, który dla zadanego początkowego stanu lamp obliczy stan lamp po M sekundach.

Wejście

Twój program powinien czytać dane z pliku `lamps.in`. *W pierwszym wierszu pliku wejściowego znajdują się dwie liczby całkowite: N ($0 < N \leqslant 10^6$) i M ($0 \leqslant M \leqslant 10^9$). Następnych N wierszy zawiera opis stanu początkowego kolejnych lamp, zaczynając od lampy 1. Pojedynczy wiersz zawiera opis jednej lampy:* `0` *oznacza, że lampa ta jest zgaszona,* `1` *— że jest zapalona.*

Wyjście

Twój program powinien zapisać wynik do pliku `lamps.out`. *Plik ten powinien składać się z N wierszy opisujących stany lamp po upływie M sekund. Sposób opisu stanów lamp ma być taki sam, jak w pliku wejściowym.*

Przykład

Dla pliku wejściowego `lamps.in`*:*

```
3 1
0
0
1
```

poprawnym wynikiem jest plik wyjściowy `lamps.out`*:*

```
0
1
1
```

Marcin Mucha, Piotr Sankowski
Tłumaczenie

Tablica (Table)

Dla danej całkowitej liczby M zbuduj kwadratową tablicę o N wierszach i N kolumnach ($2 \leq N \leq 10$) zapełnioną cyframi dziesiętnymi, spełniającą następujący warunek: N-cyfrowa liczba stworzona przez cyfry w każdym wierszu tablicy (od lewej do prawej), w każdej kolumnie tablicy (od góry do dołu) i w obydwu głównych przekątnych (od góry do dołu) musi być wielokrotnością M, nie może się zaczynać cyfrą 0 i nie może się powtarzać w tablicy.

Dla przykładu, dla $M = 2$ poprawną tablicą dla $M = 2$ może być

2	3	4
5	6	6
8	2	0

Poniższe tablice są niepoprawne (dla $M = 2$):

4

ponieważ $N < 2$;

2	0
4	8

ponieważ liczby w ostatniej kolumnie i na jednej z głównych przekątnych zaczynają się cyfrą 0;

2	3	4
5	8	8
2	0	2

ponieważ liczba 482 występuje dwukrotnie w tablicy.

Zadanie to nie zawsze ma rozwiązanie. Dla przykładu jest ono nierozwiązywalne dla $M = 10$.

Wejście

Otrzymałeś pliki testowe `TABLE`x`.IN` *($1 \leq x \leq 10$), każdy z jedną wartością M.*

Wyjście

Musisz znaleźć poprawną tablicę dla każdego przypadku testowego i zapisać ją do odpowiadającego mu pliku wynikowego `TABLE`x`.OUT` *($1 \leq x \leq 10$). Pierwszy wiersz pliku musi zawierać N, liczbę wierszy i kolumn w tablicy. Wiersz numer $i+1$ ($1 \leq i \leq N$) musi zawierać elementy i-tego wiersza tablicy zapisane jako N cyfr oddzielonych spacjami.*

Przykład

Dla pliku wejściowego `table.in`*:*

```
2
```

poprawnym wynikiem jest plik wyjściowy `table.out`*:*

```
3
2 3 4
5 6 6
8 2 0
```

Uwagi

Wiadomo, że każdy przypadek testowy posiada co najmniej jedno rozwiązanie.

Punktacja

Otrzymasz zero punktów za test, jeżeli nie dostarczysz odpowiedzi do niego albo jeżeli nie będzie spełniony któryś z wymienionych powyżej warunków.

W innym wypadku otrzymasz liczbę punktów, która zostanie wyznaczona na podstawie następującej formuły:

$$\textit{maksymalna liczba punktów za test} \cdot \frac{N_{\textit{najmniejsze wśród odpowiedzi zawodników}}}{N_{\textit{Twoje}}},$$

zaokrąglone w dół do najbliższej liczby całkowitej. W związku z tym powinieneś spróbować znaleźć jak najmniejszą poprawną tablicę spełniającą zadane warunki.

X Olimpiada Informatyczna Europy Środkowej, 2003

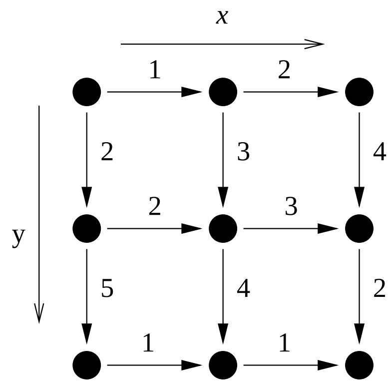
X Olimpiada Informatyczna Europy Środkowej — treści zadań

Marcin Kubica, Krzysztof Stencel
Tłumaczenie

Kwadrat (Square)

Dany jest graf. Wszystkie jego wierzchołki leżą na kwadratowej kracie o wymiarze $N \times N$ ($1 \leqslant N \leqslant 2003$). Każdy punkt kraty jest zajęty przez dokładnie jeden wierzchołek. Każdy wierzchołek jest połączony krawędziami ze swoimi sąsiadami po prawej i poniżej, o ile tacy istnieją. Krawędziom przypisano wagi w ($1 \leqslant w \leqslant 500\,000$). Każda ścieżka zaczynająca się w górnym lewym rogu (tzn. w $v_{1,1}$) prowadząca do wierzchołka $v_{x,y}$ ma tę samą wagę. Waga ścieżki to suma wag tworzących ją krawędzi.

W poniższym grafie, dla $N = 3$, każda ścieżka prowadząca z wierzchołka $v_{1,1}$ do wierzchołka $v_{2,2}$ ma wagę 4. Jedyna ścieżka z $v_{1,1}$ do $v_{1,2}$ ma wagę 2.



Dana jest liczba całkowita L ($1 \leqslant L \leqslant 2\,000\,000\,000$). Twoim zadaniem jest wyszukanie wierzchołka $v_{x,y}$ takiego, że ścieżka z $v_{1,1}$ do $v_{x,y}$ ma dokładnie wagę L. Wagi nie są bezpośrednio znane Twojemu programowi. Musi on o nie spytać za pośrednictwem biblioteki. Twój program może zadać jedynie 6667 pytań o wagi.

Biblioteka udostępnia następujące funkcje:

- `getN()` *i* `getL()` *zwracają odpowiednio wartości N i L.*
- `getWeight(x, y, direction)` *zwraca wagę krawędzi prowadzącej z $v_{x,y}$ w prawo (*`direction=0`*) lub w dół (*`direction=1`*).*

Gdy znajdziesz wierzchołek $v_{x,y}$ taki, że ścieżka z $v_{1,1}$ do $v_{x,y}$ ma wagę dokładnie L, powinieneś wywołać `solution(x, y)`*. Jeśli szukany wierzchołek nie istnieje, wywołaj* `solution(-1, -1)`*. Twój program zostanie automatycznie przerwany po wywołaniu* `solution`*. Jeśli wywołasz* `getWeight` *więcej niż 6667 razy lub Twoje rozwiązanie będzie niepoprawne, dostaniesz 0 punktów za dany test.*

W tym zadaniu nie ma plików wejściowych ani wyjściowych.

Funkcje biblioteczne

C/C++

`int getN(void)`
`int getL(void)`
`int getWeight(int x, int y, int direction)`
`void solution(int x, int y)`

Pascal

`function getN: Longint`
`function getL: Longint`
`function getWeight(x, y, direction: Longint): Longint`
`procedure solution(x, y: Longint)`

W katalogu `~/ceoi` *lub* `c:\ceoi` *znajdziesz przykładową implementację biblioteki, jednak Twój program będzie oceniany za pomocą innej implementacji.*

Na potrzeby testowania programu stwórz plik `square.in`*, który będzie czytany przez dostarczoną bibliotekę. Pierwszy wiersz w pliku* `square.in` *musi zawierać liczby całkowite N i L. Każdy z kolejnych N wierszy musi zawierać po $N-1$ liczb całkowitych — wagi poziomych krawędzi. Kolejne $N-1$ wierszy musi zawierać po N liczb całkowitych — są to wagi pionowych krawędzi.*

Bibliotekę należy dołączyć za pomocą `#include "square_lib.h"` *lub* `uses square_lib`*.*

Zauważ, że przykładowa biblioteka udostępniona w celach testowych jest wolna i pamięciożerna. Dzieje się tak, ponieważ musi ona wczytać plik wejściowy do pamięci. Jednakże możesz założyć, że biblioteka sprawdzaczki nie zużywa w ogóle pamięci i czasu procesora.

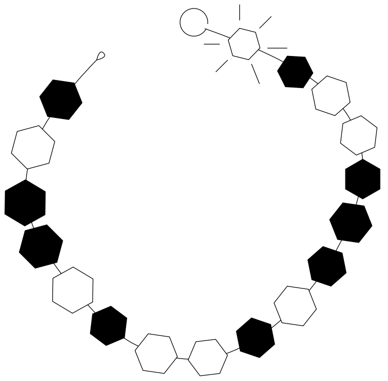
Przykładowe wejście dla biblioteki

square.in	Protokół
3 4	getN() → 3
1 2	getL() → 4
2 3	getWeight(1,1,0) → 1
1 1	getWeight(2,1,1) → 3
2 3 4	solution(2,2)
5 4 2	

Marcin Kubica, Krzysztof Stencel
Tłumaczenie

Naszyjnik (The Pearl Necklace)

Za górami za lasami żyły sobie dwa klany krasnali: rudych i zielonych. W trakcie wspólnej wyprawy w głąb Bajtogóry ekspedycja rudych i zielonych krasnali znalazła naszyjnik złożony w większości z bezwartościowych czarnych i białych szklanych koralików. Na końcu naszyjnika znajduje się jednak drogocenny brylant.



Każdy z klanów krasnali chciałby zawłaszczyć brylant. Krasnale postanowiły rozwiązać konflikt w sposób pokojowy, grając w następującą grę:

Krasnale są ponumerowane od 1 do N. Każdy z krasnali ma dwie (publicznie znane) listy numerów krasnali: białą listę i czarną listę. (Listy poszczególnych krasnali mogą być różne.) Każda z tych list może zawierać numery zarówno rudych jak i zielonych krasnali. W trakcie gry naszyjnik jest przekazywany zgodnie z następującymi regułami: Gdy krasnal dostaje naszyjnik, zdejmuje z niego pierwszy koralik. Jeśli koralik ten jest biały, to przekazuje pozostałą część naszyjnika wybranemu przez siebie krasnalowi ze swojej białej listy (może to być on sam). Jeśli koralik jest czarny, to naszyjnik przechodzi w ręce wybranego przez niego krasnala z jego czarnej listy. Na początku gry losuje się, który krasnal otrzymuje naszyjnik jako pierwszy.

Na końcu naszyjnik składa się tylko z brylantu. Krasnal, który otrzyma taki naszyjnik, zatrzymuje go dla swego klanu, a gra się kończy.

Napisz program, który pomoże zielonym krasnalom zdobyć brylant. Użyj biblioteki opisanej poniżej. Możesz założyć, że rude krasnale są wredne i grają optymalnie.

Biblioteka

Masz do dyspozycji bibliotekę zawierającą następujące funkcje:

- `getNext()` — *należy ją wywołać, gdy ruch wykonuje rudy krasnal. Wynikiem jest numer krasnala, w którego ręce przechodzi naszyjnik.*
- `setNext(d)` — *należy ją wywołać, gdy ruch wykonuje zielony krasnal. Parametr* `d` *określa numer krasnala, w którego ręce należy przekazać naszyjnik.*

- `finish()` — *należy ją wywołać, gdy gra jest skończona. To wywołanie zakończy twój program.*

Do testowania twojego programu otrzymasz testową wersję biblioteki. Kod źródłowy biblioteki znajdziesz w plikach `pearls_lib.h` *i* `pearls_lib.pas`*, które znajdują się w katalogach* `/home/ceoi/` *i* `c:\ceoi`*. W tej wersji biblioteki rude krasnale zawsze przekazują naszyjnik pierwszemu krasnalowi z odpowiedniej listy.*

Specyfikacja dla programów w C/C++

Użyj dyrektywy `#include "pearls_lib.h"` *w celu dołączenia biblioteki udostępniającej następujące funkcje:*

```
int getNext(void);
void setNext(int d);
void finish(void);
```

Specyfikacja dla programów w Pascalu

Użyj dyrektywy `uses pearls_lib;` *w celu dołączenia biblioteki udostępniającej nastepujące funkcje:*

```
function getNext:Integer;
procedure setNext(d:Integer);
procedure finish;
```

Wejście

Pierwszy wiersz pliku wejściowego `pearls.in` *zawiera początkową długość naszyjnika* L $(1 \leq L \leq 1\,000)$*, liczbę krasnali* N $(1 \leq N \leq 1\,000)$ *oraz* F *— numer pierwszego krasnala, który otrzyma naszyjnik jako pierwszy* $(1 \leq F \leq N)$*. Zwróć uwagę, że* $1 \leqslant i \leqslant N$ *dla każdego numeru krasnala* i*.*

Drugi wiersz zawiera L *znaków opisujących naszyjnik. Każdy z pierwszych* $L-1$ *znaków jest albo literą* `B`*, albo literą* `W`*.* `B` *reprezentuje czarny koralik, a* `W` *biały koralik. Ostatnim znakiem jest litera* `D` *reprezentująca brylant.*

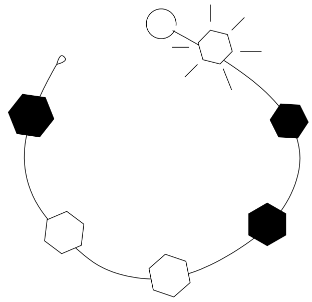
Kolejnych N *wierszy opisuje krasnale.* i*-ty z tych wierszy opisuje krasnala nr* i*. Każdy z tych wierszy zaczyna się od liczby określającej kolor krasnala: 0 dla zielonego i 1 dla rudego. Następna liczba* L_B *to długość czarnej listy danego krasnala* $(1 \leq L_B \leq 20)$*, po której następują numery krasnali z tej listy. Po tej liście następuje liczba* L_W *— długość białej listy danego krasnala* $(1 \leq L_W \leq 20)$*, po której następuje* L_W *numerów krasnali z białej listy.*

Wyjście

Brak.

Przykład

pearls.in	*Wywołania biblioteki*
6 4 2	setNext(1)
BWWBBD	setNext(4)
0 1 2 1 4	getNext() -> 1
0 2 1 3 1 1	setNext(2)
1 1 4 1 4	setNext(1)
1 2 2 3 1 1	finish()



Ocena

Jeśli nie wywołasz `finish()`*, lub jeśli wywołasz* `setNext(d)`*, gdy nie będzie to kolej zielonego krasnala lub gdy krasnal* `d` *nie występuje na liście, lub jeśli wywołasz* `getNext()`*, gdy nie będzie to kolej rudego krasnala, otrzymasz zero punktów. Zero punktów otrzymasz również, jeśli w chwili wywołania* `finish()` *brylant będzie w rękach rudego krasnala lub gdy naszyjnik będzie zawierał cokolwiek oprócz brylantu. Tylko, gdy zielony krasnal będzie miał na końcu brylant, otrzymasz pełną punktację. Każdy z testów jest tak dobrany, że jest to możliwe.*

Marcin Kubica, Krzysztof Stencel
Tłumaczenie

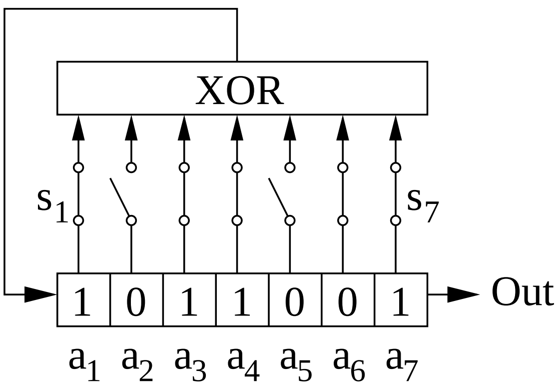
Rejestr przesuwający (Shift Register)

Rejestr komputera zawiera N bitów. Rejestr przesuwający to specjalny rodzaj rejestru, którego bity można łatwo przesuwać o jedną pozycję.

Za pomocą takiego rejestru można w następujący sposób generować liczby pseudolosowe: Rejestr przesuwający na początku zawiera bity $a_1, a_2, \ldots, a_N$*. W każdym cyklu zegara na wyjściu rejestru pojawia się skrajnie prawy bit* a_N*. Pozostałe bity są przesuwane o jedną pozycję w prawo. Bit na pierwszej pozycji przyjmuje nową wartość* a'_1 *określoną w następujący sposób: Każdy bit rejestru jest podłączony do bramki XOR poprzez przełącznik (porównaj rysunek poniżej), tzn. dla każdego bitu mamy przełącznik* s_i *(równy 1 lub 0), który określa, czy wartość bitu* a_i *jest przekazywana do bramki XOR. Niech* $k_i = s_i \cdot a_i$*. Nowa wartość* a'_1 *jest równa wyjściu bramki XOR:* $XOR(k_1, k_2, \ldots, k_N)$*. (Uwaga: Jeśli liczba jedynek w ciągu* $k_1, k_2, \ldots, k_N$ *jest nieparzysta, to wartością* $XOR(k_1, k_2, \ldots, k_N)$ *jest 1, w przeciwnym przypadku wartością jest 0.)*

Oto formalna definicja:

$$\begin{aligned} a'_1 &= XOR(k_1, k_2, \ldots, k_N) \\ a'_i &= a_{i-1} \text{ dla } 2 \leqslant i \leqslant N \\ \textit{wyjście} &= a_N \end{aligned}$$



cykl	a_1	a_2	a_3	a_4	a_5	a_6	a_7	*output*
0	1	0	1	1	0	0	1	-
1	0	1	0	1	1	0	0	1
2	1	0	1	0	1	1	0	0
3	1	1	0	1	0	1	1	0
4	0	1	1	0	1	0	1	1
5	0	0	1	1	0	1	0	1
6	1	0	0	1	1	0	1	0
7	1	1	0	0	1	1	0	1
8	0	1	1	0	0	1	1	0
9	1	0	1	1	0	0	1	1
10	0	1	0	1	1	0	0	1
11	1	0	1	0	1	1	0	0
12	1	1	0	1	0	1	1	0
13	0	1	1	0	1	0	1	1
14	0	0	1	1	0	1	0	1

W powyższym przykładzie wartość a_1 w cyklu 1 jest obliczana następująco:
$XOR(1 \cdot 1, 0 \cdot 0, 1 \cdot 1, 1 \cdot 1, 0 \cdot 0, 1 \cdot 0, 1 \cdot 1) = 0$.

Znasz $2N$ wartości wyjściowych takiego rejestru przesuwającego. Na podstawie tych wartości powinieneś spróbować wyznaczyć wartości przełączników s_i.

Wejście

Pierwszy wiersz pliku wejściowego `register.in` *zawiera rozmiar rejestru N ($1 \leq N \leq 750$). Drugi wiersz zawiera $2N$ liczb 0 i/lub 1 — pierwszych $2N$ wyjść rejestru przesuwającego.*

Wyjście

Plik wyjściowy `register.out` *zawiera dokładnie jeden wiersz. Jeśli istnieje ustawienie przełączników, przy którym rejestr wytworzy zadane wartości wyjściowe, wypisz wartości s_i jednego z takich ustawień przełączników, zaczynając od s_1. Jeśli nie ma takiego ustawienia, wypisz tylko liczbę -1.*

Przykłady

register.in	register.out
7 1 0 0 1 1 0 1 0 1 1 0 0 1 1	1 0 1 1 0 1 1

register.in	register.out
3 0 0 0 1 1 1	-1

Marcin Kubica, Krzysztof Stencel
Tłumaczenie

Wieże Hanoi (Towers of Hanoi)

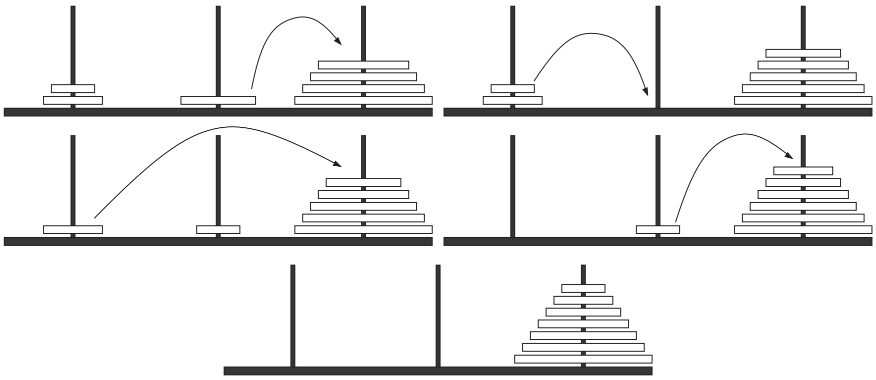
Zapewne spotkałeś się już z problemem **wież Hanoi**. *Drewniane krążki różnych rozmiarów są nałożone na trzy słupki. Wszystkie krążki na tym samym słupku są uporządkowane wg rozmiaru z największym krążkiem na spodzie. Celem jest przeniesienie całej wieży z jednego słupka na drugi słupek. W jednym ruchu można przenieść tylko jeden krążek i nie wolno kłaść większego krążka na mniejszy.*

Zgodnie ze starodawną legendą, tybetańscy mnisi od tysięcy lat próbują rozwiązać ten problem dla 47 krążków. Ponieważ jednak wymaga to przynajmniej $2^{47}-1$ ruchów, a mnisi nie mieli żadnej strategii, więc wszystko pomieszali, przestrzegając jednak zasad. Chcieliby teraz przełożyć wszystkie krążki na dowolny ze słupków, wykonując minimalną liczbę ruchów. Jednak zgodnie ze złożonymi ślubami, mogą wykonywać jedynie ruchy zgodne z zasadami. Chcieliby wiedzieć, na który słupek najlepiej jest przełożyć krążki i ilu co najmniej wymaga to ruchów.

Napisz program, który rozwiązuje problem mnichów. Twój program powinien radzić sobie z dowolną liczbą krążków N ($0 < N \leq 100\,000$). Liczby pojawiające się w trakcie obliczeń mogą być bardzo duże. Dlatego też mnisi są zainteresowani jedynie liczbą ruchów modulo $1\,000\,000$.

Przykład

Poniższy przykład można rozwiązać w 4 ruchach.



Wejście

Pierwszy wiersz pliku `hanoi.in` *zawiera liczbę krążków N. Drugi wiersz zawiera trzy liczby całkowite s_1, s_2, s_3 ($0 \leqslant s_1, s_2, s_3 \leqslant N$ oraz $s_1 + s_2 + s_3 = N$). Są to liczby krążków na słupkach. Wiersze od 3-go do 5-go zawierają rozmiary krążków na poszczególnych słupkach. Dokładniej, $(i+2)$-gi wiersz pliku wejściowego zawiera liczby całkowite $m_{i,1}, \ldots, m_{i,s_i}$*

określające rozmiary krążków na słupku i, przy czym $1 \leqslant m_{i,j} \leqslant N$*. Krążki są podane w kolejności od dołu do góry, tzn.* $m_{i,1} > m_{i,2} > \ldots > m_{i,s_i}$*. Pustemu słupkowi odpowiada pusty wiersz. Każdy z N krążków ma inny rozmiar. Wszystkie liczby są pooddzielane pojedynczymi odstępami.*

Wyjście

Pierwszy wiersz pliku wyjściowego `hanoi.out` *powinien zawierać liczbę* $d \in \{1, 2, 3\}$ *— numer słupka, na który można przełożyć krążki w minimalnej liczbie ruchów. Drugi wiersz powinien zawierać liczbę M — wymaganą liczbę ruchów modulo 1 000 000.*

Przykład

hanoi.in	hanoi.out
7 2 1 4 2 1 3 7 6 5 4	3 4

Dane testowe

Twój program zostanie sprawdzony na dwudziestu różnych plikach wejściowych. W poniższej tabelce podano pierwsze wiersze tych plików, tzn. liczby krążków N.

Test	1	2	3	4	5	6	7	8	9	10	11	12
N	5	10	15	20	50	100	150	200	1000	2000	3000	4000

Test	13	14	15	16	17	18	19	20
N	30000	40000	50000	60000	70000	80000	90000	100000

Marcin Kubica, Krzysztof Stencel
Tłumaczenie

Wycieczka (Trip)

Ala i Bob chcą pojechać na wycieczkę. Każde z nich zaplanowało trasę, która jest listą miast do odwiedzenia w określonej kolejności. Każde z miast może występować na trasie wielokrotnie.

Chcą podróżować razem, więc muszą uzgodnić wspólną trasę. Żadne z nich nie chce zmieniać kolejności miast na swojej trasie ani też dodawać nowych miast. Nie mają więc wyboru — muszą usunąć niektóre miasta ze swoich tras. Oczywiście wspólna trasa musi być jak najdłuższa.

W zwiedzanym przez nich regionie jest dokładnie 26 miast. Na listach są one oznaczone za pomocą małych liter alfabetu angielskiego od „a” do „z”.

Wejście

Plik wejściowy `trip.in` *składa się z dwóch wierszy: pierwszy wiersz to lista Ali, a drugi to lista Boba. Każda z list zawiera od 1 do 80 małych liter bez odstępów między nimi.*

Wyjście

Plik wyjściowy `trip.out` *powinien zawierać wszystkie trasy, które spełniają podane powyżej warunki, ale żadna trasa nie może pojawić się więcej niż raz. Każdą trasę należy wypisać w oddzielnym wierszu. Istnieje co najmniej jedna niepusta taka trasa, ale nie ma więcej niż 1 000 różnych takich tras. Kolejność tras w pliku wyjściowym nie ma znaczenia.*

Przykład

trip.in	trip.out
abcabcaa acbacba	ababa abaca abcba acbca acaba acaca acbaa

Marcin Kubica, Krzysztof Stencel
Tłumaczenie

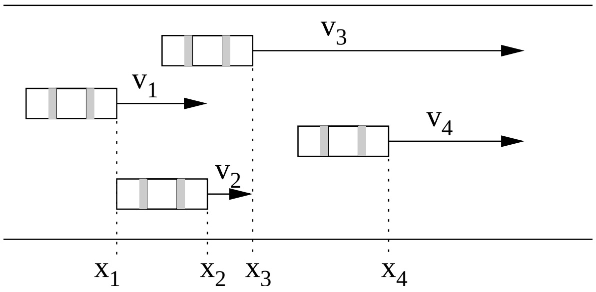
Wyścig (The Race)

W dorocznych zawodach nadświetlnych statków kosmicznych o puchar Lorda Wejdera wystartowało N statków. Statek numer i jest tak zbudowany, że przyspiesza do swej maksymalnej prędkości V_i w czasie 0 i dalej porusza się z tą prędkością. Ze względu na dotychczasowe wyniki, każdy statek startuje w punkcie znajdującym się X_i kilometrów za linią startową.

Trasa wyścigu jest nieskończenie długa. Ze względu na duże prędkości statków, trasa wyścigu ma kształt linii prostej. Na tej prostej statki mogą bez problemów wyprzedzać się, nie przeszkadzając sobie nawzajem.

Wielu widzów nie zorientowało się jeszcze, że wynik wyścigu można z góry przewidzieć. Twoim zadaniem jest przekonać ich o tym poprzez podanie, ile razy statki będą się wyprzedzać, i podanie pierwszych 10 000 wyprzedzeń w porządku chronologicznym.

Możesz założyć, że każdy statek startuje z innego punktu startowego, co więcej, nigdy nie zdarzy się tak, aby więcej niż dwa statki były w tej samej chwili w tym samym punkcie trasy.



Wejście

W pierwszym wierszu pliku `therace.in` *znajduje się liczba statków N ($0 < N \leqslant 250\,000$). W kolejnych N wierszach podano charakterystyki statków, po jednej w wierszu. Wiersz numer $i+1$ zawiera dwie liczby całkowite X_i i V_i — punkt startowy i prędkość statku numer i ($0 \leqslant X_i \leqslant 1\,000\,000$, $0 < V_i < 100$). Charakterystyki statków są podane w kolejności pozycji startowych, tzn. $X_1 < X_2 < \ldots < X_N$. Pozycja startowa to liczba kilometrów za linią startową, skąd statek startuje. Prędkość jest podana w kilometrach na sekundę.*

Wyjście

Pierwszy wiersz pliku `therace.out` *powinien zawierać liczbę wyprzedzeń statków* **modulo 1 000 000**. *Poprzez podanie liczby wyprzedzeń modulo 1 000 000, wykazujesz swoją znajomość wyniku, nie psując przy tym zabawy mniej inteligentnym widzom.*

Kolejne wiersze powinny zawierać opisy wyprzedzeń w porządku chronologicznym, po jednym opisie w wierszu. Jeśli będzie więcej niż 10 000 wyprzedzeń, wypisz jedynie pierwsze 10 000 z nich, w przeciwnym przypadku wypisz je wszystkie. Każdy z tych wierszy powinien zawierać dwie liczby całkowite i i j — oznaczają one, że statek nr i wyprzedzi statek nr j. Jeśli kilka *wyprzedzeń ma miejsce równocześnie, należy je uporządkować według miejsca na trasie, tzn. wyprzedzenie bliższe linii startowej powinno powinno być wypisane najpierw.*

Uwagi

Jeśli podasz poprawną liczbę wyprzedzeń, dostaniesz 40% punktów za test. Za poprawne podanie pierwszych 10 000 wyprzedzeń otrzymasz pozostałe 60% punktów. Każda z tych dwóch części jest oceniana niezależnie, pod warunkiem, że program poprawnie zakończy działanie w zadanym czasie.

Przykład

therace.in	therace.out
4 0 2 2 1 3 8 6 3	2 3 4 1 2

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] A. V. Aho, J. E. Hopcroft, J. D. Ullman. *Projektowanie i analiza algorytmów komputerowych.* PWN, Warszawa, 1983.

[11] L. Banachowski, A. Kreczmar. *Elementy analizy algorytmów.* WNT, Warszawa, 1982.

[12] L. Banachowski, A. Kreczmar, W. Rytter. *Analiza algorytmów i struktur danych.* WNT, Warszawa, 1987.

[13] L. Banachowski, K. Diks, W. Rytter. *Algorytmy i struktury danych.* WNT, Warszawa, 1996.

[14] J. Bentley. *Perełki oprogramowania.* WNT, Warszawa, 1992.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest. *Wprowadzenie do algorytmów.* WNT, Warszawa, 1997.

[16] *Elementy informatyki. Pakiet oprogramowania edukacyjnego.* Instytut Informatyki Uniwersytetu Wrocławskiego, OFEK, Wrocław–Poznań, 1993.

[17] *Elementy informatyki: Podręcznik (cz. 1), Rozwiązania zadań (cz. 2), Poradnik metodyczny dla nauczyciela (cz. 3).* Pod redakcją M. M. Sysły, PWN, Warszawa, 1996.

[18] G. Graham, D. Knuth, O. Patashnik. *Matematyka konkretna.* PWN, Warszawa, 1996.

[19] D. Harel. *Algorytmika. Rzecz o istocie informatyki.* WNT, Warszawa, 1992.

[20] J. E. Hopcroft, J. D. Ullman. *Wprowadzenie do teorii automatów, języków i obliczeń.* PWN, Warszawa, 1994.

[21] W. Lipski. *Kombinatoryka dla programistów.* WNT, Warszawa, 1989.

[22] W. Lipski, W. Marek. *Analiza kombinatoryczna.* PWN, Warszawa, 1986.

[23] E. M. Reingold, J. Nievergelt, N. Deo. *Algorytmy kombinatoryczne.* WNT, Warszawa, 1985.

[24] K. A. Ross, C. R. B. Wright. *Matematyka dyskretna.* PWN, Warszawa, 1996.

[25] R. Sedgewick. *Algorithms.* Addison-Wesley, 1988.

[26] M. M. Sysło. *Algorytmy.* WSiP, Warszawa, 1998.

[27] M. M. Sysło. *Piramidy, szyszki i inne konstrukcje algorytmiczne.* WSiP, Warszawa, 1998.

[28] M. M. Sysło, N. Deo, J. S. Kowalik. *Algorytmy optymalizacji dyskretnej z programami w języku Pascal.* PWN, Warszawa, 1993.

[29] R. J. Wilson. *Wprowadzenie do teorii grafów.* PWN, Warszawa, 1998.

[30] N. Wirth. *Algorytmy + struktury danych = programy.* WNT, Warszawa, 1999.

Niniejsza publikacja stanowi wyczerpujące źródło informacji o zawodach X Olimpiady Informatycznej, przeprowadzonych w roku szkolnym 2002/2003. Książka zawiera informacje dotyczące organizacji, regulaminu oraz wyników zawodów. Zawarto także opis rozwiązań wszystkich zadań konkursowych. Do książki dołączony jest dysk CD-ROM zawierający wzorcowe rozwiązania i testy do wszystkich zadań Olimpiady.

Książka zawiera też zadania z XIV Międzynarodowej Olimpiady Informatycznej, IX Bałtyckiej Olimpiady Informatycznej i X Olimpiady Informatycznej Europy Środkowej.

X 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.

Olimpiada Informatyczna
jest organizowana przy współudziale

PROKOM
SOFTWARE SA

ISBN 83-917700-4-4