MINISTERSTWO EDUKACJI NARODOWEJ I SPORTU
UNIWERSYTET WROCŁAWSKI
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

XII OLIMPIADA INFORMATYCZNA
2004/2005

Olimpiada Informatyczna jest organizowana przy współudziale

PROKOM SOFTWARE SA

WARSZAWA, 2005

MINISTERSTWO EDUKACJI NARODOWEJ I SPORTU
UNIWERSYTET WROCŁAWSKI
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

XII OLIMPIADA INFORMATYCZNA
2004/2005

WARSZAWA, 2005

Autorzy tekstów:

Szymon Acedański
Marek Cygan
prof. dr hab. Zbigniew Czech
dr hab. Krzysztof Diks
dr hab. Wojciech Guzicki
dr Przemysława Kanarek
dr Łukasz Kowalik
dr Marcin Kubica
Paweł Parys
Marcin Pilipczuk
Jakub Radoszewski
prof. dr hab. Wojciech Rytter
mgr Krzysztof Sikora
Piotr Stańczyk
Bartosz Walczak

Autorzy programów na dysku CD-ROM:

Michał Adamaszek
Karol Cwalina
Marek Cygan
mgr Wojciech Dudek
Adam Iwanicki
Jakub Radoszewski
dr Marcin Kubica
Tomasz Malesiński
Marcin Michalski
dr Marcin Mucha
mgr Krzysztof Onak
mgr Arkadiusz Paterek
Marcin Pilipczuk
Jakub Radoszewski
mgr Rafał Rusin
Piotr Stańczyk
mgr Marcin Stefaniak
mgr Paweł Wolff

Opracowanie i redakcja:

dr Przemysława Kanarek
Adam Iwanicki

Opracowanie i redakcja treści zadań:

dr Marcin Kubica
Paweł Parys

Skład:

Adam Iwanicki

Pozycja dotowana przez Ministerstwo Edukacji Narodowej i Sportu.

Druk książki został sfinansowany przez PROKOM SOFTWARE SA

ISBN 83–922946–0–2

Spis treści

Krzysztof Diks

Wstęp

Drogi Czytelniku!

Już po raz 12-ty trafia do zainteresowanych Olimpiadą Informatyczną „niebieska książeczka”. Mijający rok był dla Olimpiady rokiem szczególnym. Polska była organizatorem 17-tej Międzynarodowej Olimpiady Informatycznej. Wszystkie działania i wysiłki osób zaangażowanych w informatyczny ruch olimpijski były podporządkowane temu wydarzeniu. Jednym z naszych głównych celów było bardzo dobre przygotowanie reprezentacji Polski na Olimpiadę Międzynarodową. Cel ten staraliśmy się zrealizować między innymi przez dobór takich zadań na tegoroczne zawody krajowe, które pozwoliłyby wyłonić do reprezentacji najlepszych z najlepszych. Wydaje się, że cel ten został osiągnięty.

W XII Olimpiadzie Informatycznej wystartowało 923 uczniów. Na 17-tej Międzynarodowej Olimpiadzie Informatycznej oraz Środkowoeuropejskiej Olimpiadzie Informatycznej reprezentowała nas najlepsza czwórka z zawodów krajowych: Filip Wolski, Adam Gawarkiewicz, Daniel Czajka i Tomasz Kulczyński. W obu imprezach nasi reprezentanci spisali się znakomicie. Na 17-tej Międzynarodowej Olimpiadzie Informatycznej, startując wśród 276 zawodników z 70 krajów z całego świata, Filip Wolski zdobył złoty medal (10-te miejsce w świecie), Adam Gawarkiewicz i Tomasz Kulczyński zdobyli medale srebrne, natomiast Daniel Czajka zdobył medal brązowy. Na Olimpiadzie Środkowoeuropejskiej w pierwszej szóstce znaleźli się wszyscy nasi reprezentanci zajmując wśród 48 zawodników z 13 krajów następujące miejsca:

2. Tomasz Kulczyński, medal złoty
3. Filip Wolski, medal złoty
4. Adam Gawarkiewicz, medal złoty
6. Daniel Czajka, medal srebrny.

Na podkreślenie zasługują też wyniki naszej drużyny „młodzieżowej” na Olimpiadzie Krajów Bałtyckich. Na tych zawodach Polskę reprezentowała 6-tka najlepszych zawodników z olimpiady krajowej, którzy jeszcze w przyszłym roku mają szansę na start w zawodach olimpijskich. Wszyscy nasi reprezentanci (Filip Wolski, Marcin Skotniczny, Adam Gawarkiewicz, Piotr Świgoń, Tomasz Kulczyński i Piotr Szmigiel) wrócili z medalami, a Filip Wolski został zwycięzcą całej imprezy.

Żaden z tych sukcesów nie byłby możliwy bez zaangażowania wielu osób związanych z Olimpiadą: członków Komitetu Głównego, członków komitetów okręgowych, jurorów, pracowników sekretariatu, pracowników instytucji i firm współorganizujących i wspierających Olimpiadę — Ministerstwa Edukacji Narodowej i Sportu, Uniwersytetu Wrocławskiego, firmy Prokom Software S.A. oraz Wydawnictw Naukowo-Technicznych. Wszystkim gorąco dziękuję.

Na koniec chciałbym wyrazić moje uznanie dla pani dr Przemki Kanarek i Adama Iwanickiego za wysiłek włożony w redakcję tego tomu. Z pewnością docenisz to drogi

Czytelniku zgłębiając tajniki rozwiązań niełatwych olimpijskich zadań. Wyrażam też nadzieję, że zawarte w tym tomie opisy i dołączone rozwiązania wzorcowe przyczynią się do wykreowania kolejnego pokolenia znakomitych młodych polskich informatyków, przyszłych medalistów międzynarodowych konkursów informatycznych.

Krzysztof Diks

Sprawozdanie z przebiegu XII Olimpiady Informatycznej 2004/2005

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

ORGANIZACJA ZAWODÓW

W roku szkolnym 2004/2005 odbyły się zawody XII 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.

18 października 2004 r. rozesłano plakaty zawierające zasady organizacji zawodów I stopnia oraz zestaw 5 zadań konkursowych do 3843 szkół i zespołów szkół młodzieżowych ponadgimnazjalnych oraz do wszystkich kuratorów i koordynatorów edukacji informatycznej. Zawody I stopnia rozpoczęły się dnia 25 października 2004 r. Ostatecznym terminem nadsyłania prac konkursowych był 22 listopada 2004 roku.

Zawody II i III stopnia były dwudniowymi sesjami stacjonarnymi, poprzedzonymi jednodniowymi sesjami próbnymi. Zawody II stopnia odbyły się w sześciu okręgach: Warszawie, Wrocławiu, Toruniu, Katowicach, Krakowie i Rzeszowie oraz w Sopocie, w dniach 8–10.02.2005 r., natomiast zawody III stopnia odbyły się w ośrodku firmy Combidata Poland S.A. w Sopocie, w dniach 5–9.04.2005 r.

Uroczystość zakończenia XII Olimpiady Informatycznej odbyła się w dniu 09.04.2005 r. w Sali Posiedzeń Urzędu Miasta w Sopocie.

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)
mgr Anna Beata Kwiatkowska (IV LO im. T. Kościuszki, w Toruniu)
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 w Toruniu)
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.

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:
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 w Toruniu)
zastępca przewodniczącego:
dr Mirosława Skowrońska (Uniwersytet Mikołaja Kopernika w Toruniu)
sekretarz:
dr Barbara Klunder (Uniwersytet Mikołaja Kopernika w Toruniu)
członkowie:
mgr Anna Beata 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 w Gliwicach)
zastępca przewodniczącego:
mgr inż. Sebastian Deorowicz (Politechnika Śląska w Gliwicach)
sekretarz:
mgr inż. Adam Skórczyński (Politechnika Śląska w Gliwicach)
członkowie:
dr inż. Mariusz Boryczka (Uniwersytet Śląski w Sosnowcu)
mgr inż. Marcin Ciura (Politechnika Śląska w Gliwicach)
mgr inż. Marcin Szołtysek (Politechnika Śląska w Gliwicach)
mgr Jacek Widuch (Politechnika Śląska w Gliwicach)
mgr Wojciech Wieczorek (Uniwersytet Śląski w Sosnowcu)

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 w Krakowie)

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

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

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

Jury Olimpiady Informatycznej

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

Szymon Acedański
Michał Adamaszek
Karol Cwalina
Marek Cygan
Tomasz Idziaszek
Adam Iwanicki
Tomasz Malesiński
Marcin Michalski
mgr Krzysztof Onak
Paweł Parys
mgr Arkadiusz Paterek
Marcin Pilipczuk
Jakub Radoszewski
mgr Krzysztof Sikora
Piotr Stańczyk
mgr Marcin Stefaniak

Bartosz Walczak
mgr Tomasz Waleń
Szymon Wąsik
Marek Żylak

ZAWODY I STOPNIA

W XII Olimpiadzie Informatycznej wzięło udział 923 zawodników.

Decyzją Komitetu Głównego do zawodów zostało dopuszczonych 21 uczniów gimnazjów. Byli to uczniowie z następujących gimnazjów:

• Gimnazjum nr 24	Gdynia	2 uczniów
• Publiczne Gimnazjum	Grójec	2 uczniów
• Katolickie Gimnazjum SPSK	Białystok	1 uczeń
• 50 Gimnazjum	Bydgoszcz	1 uczeń
• Gimnazjum nr 7	Chorzów	1 uczeń
• Gimnazjum im. Św. Królowej Jadwigi	Kielce	1 uczeń
• Gimnazjum nr 23	Kraków	1 uczeń
• Gimnazjum nr 1	Kurów	1 uczeń
• Gimnazjum nr 11	Łódź	1 uczeń
• Publiczne Gimnazjum	Łysa Góra	1 uczeń
• Gimnazjum nr 1	Pasłęk	1 uczeń
• Gimnazjum nr 30	Poznań	1 uczeń
• Sportowe Gimnazjum nr 9	Tychy	1 uczeń
• Gimnazjum Przymierza Rodzin im. Jana Pawła II	Warszawa	1 uczeń
• Społeczne Gimnazjum nr 2	Warszawa	1 uczeń
• Publiczne Gimnazjum	Węgrów	1 uczeń
• Gimnazjum nr 40	Wrocław	1 uczeń
• Gimnazjum nr 3	Zabrze	1 uczeń
• Gimnazjum nr 2 im. A. Asnyka	Zielona Góra	1 uczeń

Kolejność województw pod względem liczby uczestników była następująca:

mazowieckie	136
małopolskie	132
pomorskie	127
śląskie	100
kujawsko-pomorskie	68
dolnośląskie	55
podkarpackie	44
wielkopolskie	43
łódzkie	41
lubelskie	32
podlaskie	30
świętokrzyskie	29
zachodniopomorskie	26

lubuskie	22
opolskie	19
warmińsko-mazurskie	19

W zawodach I stopnia najliczniej reprezentowane były szkoły:

III LO im. Marynarki Wojennej RP	Gdynia	66 uczniów
V LO im. A. Witkowskiego	Kraków	57
XIV LO im. St. Staszica	Warszawa	46
VI LO im. W. Sierpińskiego	Gdynia	30
VI LO im. J. i J. Śniadeckich	Bydgoszcz	20
VIII LO im. M. Skłodowskiej–Curie	Katowice	17
VIII LO im. A. Mickiewicza	Poznań	15
XIV LO im. Polonii Belgijskiej	Wrocław	14
VI LO im. Jana Kochanowskiego	Radom	11
XIII LO	Szczecin	11
I LO im. S. Żeromskiego	Kielce	10
Publiczne LO nr 2 im. M. Konopnickiej	Opole	9
I LO im. A. Mickiewicza	Białystok	8
IV LO im. T. Kościuszki	Toruń	8
III LO im. A. Mickiewicza	Wrocław	8
I LO im. St. Staszica	Lublin	7
LO WSIiZ	Rzeszów	7
V Liceum Ogólnokształcące	Bielsko-Biała	6
II LO im. Jana III Sobieskiego	Kraków	6
X LO im. prof. S. Banacha	Toruń	6
XVIII LO im. Jana Zamoyskiego	Warszawa	6
I LO im. C. K. Norwida	Bydgoszcz	5
I LO im. St. Dubois	Koszalin	5
XXVII LO im. T. Czackiego	Warszawa	5
V LO im. K. Kieślowskiego	Zielona Góra	5
III LO im. S. Batorego	Chorzów	4
IX LO im. C. K. Norwida	Częstochowa	4
V Liceum Ogólnokształcące	Gdańsk	4
I LO im. B. Krzywoustego	Głogów	4
Katolickie LO Zakonu Ojców Pijarów	Kraków	4
I LO im. Władysława Jagiełły	Krasnystaw	4
I LO im. T. Kościuszki	Łomża	4
ZSO nr 1 im. Jana Długosza	Nowy Sącz	4
II LO im. Królowej Jadwigi	Pabianice	4
Ponadgimnazjalne IV LO im. M. Kopernika	Rzeszów	4
I LO im. B. Krzywoustego	Słupsk	4
Gimnazjum i Liceum Akademickie	Toruń	4
VI LO im. T. Reytana	Warszawa	4
I Liceum Ogólnokształcące	Zielona Góra	4

Najliczniej reprezentowane były miasta:

Gdynia	101 uczniów	Gdańsk	11
Warszawa	91	Zielona Góra	10
Kraków	81	Bielsko-Biała	9
Wrocław	34	Chorzów	8
Bydgoszcz	28	Łomża	7
Katowice	24	Słupsk	7
Poznań	23	Konin	6
Toruń	22	Koszalin	6
Rzeszów	20	Nowy Sącz	6
Kielce	19	Tarnów	6
Szczecin	18	Głogów	5
Łódź	14	Gorzów Wlkp.	5
Opole	14	Grudziądz	5
Białystok	13	Olsztyn	5
Częstochowa	13	Puławy	5
Lublin	13	Rybnik	5
Radom	13		

Zawodnicy uczęszczali do następujących klas:

do klasy I	gimnazjum	1 uczeń
do klasy II	gimnazjum	5 uczniów
do klasy III	gimnazjum	15 uczniów
do klasy I	szkoły średniej	132 uczniów
do klasy II	szkoły średniej	339 uczniów
do klasy III	szkoły średniej	412 uczniów
do klasy IV	szkoły średniej	2 uczniów
do klasy V	szkoły średniej	10 uczniów

7 zawodników nie podało informacji do której klasy uczęszczają.

W zawodach I stopnia zawodnicy mieli do rozwiązania pięć zadań: „Bankomat", „Punkty", „Samochodziki", „Skarbonki", „Skoczki".

W wyniku zastosowania procedury sprawdzającej wykryto niesamodzielne rozwiązania. Komitet Główny, w zależności od indywidualnej sytuacji, nie brał tych rozwiązań pod uwagę lub zdyskwalifikował zawodników, którzy je nadesłali.

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

- **BAN** — Bankomat

	BAN	
	liczba zawodników	czyli
100 pkt.	219	23,7%
75–99 pkt.	71	7,1%
50–74 pkt.	31	3,4%
1–49 pkt.	331	35,8%
0 pkt.	73	7,9%
brak rozwiązania	198	21,5%

- **PUN** — Punkty

	PUN	
	liczba zawodników	czyli
100 pkt.	33	3,6%
75–99 pkt.	65	7,0%
50–74 pkt.	75	8,1%
1–49 pkt.	237	25,7%
0 pkt.	198	21,5%
brak rozwiązania	315	34,1%

- **SAM** — Samochodziki

	SAM	
	liczba zawodników	czyli
100 pkt.	271	29,4%
75–99 pkt.	18	1,9%
50–74 pkt.	81	8,8%
1–49 pkt.	217	23,5%
0 pkt.	213	23,1%
brak rozwiązania	123	13,3%

- **SKA** — Skarbonki

	SKA	
	liczba zawodników	czyli
100 pkt.	446	48,3%
75–99 pkt.	52	5,6%
50–74 pkt.	35	3,8%
1–49 pkt.	236	25,6%
0 pkt.	69	7,5%
brak rozwiązania	85	9,2%

- **SKO** — Skoczki

	SKO	
	liczba zawodników	czyli
100 pkt.	237	25,7%
75–99 pkt.	31	3,4%
50–74 pkt.	16	1,7%
1–49 pkt.	53	5,7%
0 pkt.	254	27,5%
brak rozwiązania	332	36,0%

W sumie za wszystkie 5 zadań konkursowych:

SUMA	liczba zawodników	czyli
500 pkt.	17	1,8%
375–499 pkt.	174	18,9%
250–374 pkt.	140	15,2%
1–249 pkt.	516	55,9%
0 pkt.	75	8,2%

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 335 zawodników, którzy osiągnęli w zawodach I stopnia wynik nie mniejszy niż 246 pkt.

Trzech zawodników nie stawiło się na zawody. W zawodach II stopnia uczestniczyło 332 zawodników.

Zawody II stopnia odbyły się w dniach 8–10 lutego 2005 r. w sześciu stałych okręgach oraz w Sopocie:

- w Warszawie — 57 zawodników z następujących województw:
  - łódzkie (1)
  - mazowieckie (45)
  - podlaskie (10)
  - warmińsko-mazurskie (1)
- we Wrocławiu — 51 zawodników z następujących województw:
  - dolnośląskie (16)
  - lubuskie (3)
  - łódzkie (3)
  - opolskie (5)
  - wielkopolskie (16)

  - zachodniopomorskie (8)
- w Krakowie — 50 zawodników z następujących województw:
  - małopolskie (50)
- w Toruniu — 42 zawodników z następujących województw:
  - kujawsko-pomorskie (38)
  - mazowieckie (2)
  - warmińsko-mazurskie (1)
  - wielkopolskie (1)
- w Gliwicach — 37 zawodników z następujących województw:
  - opolskie (1)
  - śląskie (30)
  - świętokrzyskie (6)
- w Rzeszowie — 28 zawodników z następujących województw:
  - lubelskie (3)
  - małopolskie (11)
  - podkarpackie (13)
  - świętokrzyskie (1)
- w Sopocie — 67 zawodników z następujących województw:
  - pomorskie (64)
  - śląskie (1)
  - zachodniopomorskie (2)

W zawodach II stopnia najliczniej reprezentowane były szkoły:

V LO im. A. Witkowskiego	Kraków	45 uczniów
III LO im. Marynarki Wojennej RP	Gdynia	40
XIV LO im. St. Staszica	Warszawa	22
VI LO im. J. i J. Śniadeckich	Bydgoszcz	18
VI LO im. W. Sierpińskiego	Gdynia	16
VIII LO im. A. Mickiewicza	Poznań	11
VIII LO im. M. Skłodowskiej-Curie	Katowice	9
XIII LO	Szczecin	7
III LO im. A. Mickiewicza	Wrocław	7
I LO im. A. Mickiewicza	Białystok	6
IV LO im. T. Kościuszki	Toruń	6
I LO im. C. K. Norwida	Bydgoszcz	5
I LO im. S. Żeromskiego	Kielce	5

Publiczne LO nr 2 im. M. Konopnickiej	Opole	5
II LO im. Jana III Sobieskiego	Kraków	4
LO WSIiZ	Rzeszów	4
X LO im. prof. Stefana Banacha	Toruń	4
XVIII LO im. Jana Zamoyskiego	Warszawa	4
XIV LO im. Polonii Belgijskiej	Wrocław	4
Katolickie LO Ojców Pijarów	Kraków	3
I LO im. St. Staszica	Lublin	3
I LO im. M. Kopernika	Łódź	3
Ponadgimnazjalne IV LO im. M. Kopernika	Rzeszów	3
I LO im. B. Krzywoustego	Słupsk	3
XXVII LO im. T. Czackiego	Warszawa	3

Najliczniej reprezentowane były miasta:

Gdynia	56	uczniów	Szczecin	7	uczniów
Kraków	54		Kielce	6	
Warszawa	38		Opole	5	
Bydgoszcz	24		Chorzów	3	
Wrocław	14		Częstochowa	3	
Katowice	12		Gdańsk	3	
Poznań	12		Lublin	3	
Toruń	12		Łódź	3	
Białystok	8		Mielec	3	
Rzeszów	8		Słupsk	3	

W dniu 8 lutego odbyła się sesja próbna, na której zawodnicy rozwiązywali nie liczące się do ogólnej klasyfikacji zadanie „Lot na Marsa”. W dniach konkursowych zawodnicy rozwiązywali zadania: „Banknoty”, „Sumy Fibonacciego”, „Kości” oraz „Szablon”, 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:

- **LOT — próbne** — Lot na Marsa

	LOT — próbne	
	liczba zawodników	czyli
100 pkt.	43	12,9%
75–99 pkt.	19	5,7%
50–74 pkt.	13	3,9%
1–49 pkt.	129	38,9%
0 pkt.	118	35,5%
brak rozwiązania	10	3,0%

- **BAN** — Banknoty

	BAN	
	liczba zawodników	czyli
100 pkt.	55	16,6%
75–99 pkt.	33	9,9%
50–74 pkt.	57	17,2%
1–49 pkt.	157	43,7%
0 pkt.	20	6,0%
brak rozwiązania	10	3,0%

- **SUM** — Sumy Fibonacciego

	SUM	
	liczba zawodników	czyli
100 pkt.	28	8,4%
75–99 pkt.	15	4,5%
50–74 pkt.	95	28,6%
1–49 pkt.	137	41,3%
0 pkt.	44	13,3%
brak rozwiązania	13	3,9%

- **KOS** — Kości

	KOS	
	liczba zawodników	czyli
100 pkt.	3	0,9%
75–99 pkt.	10	3,0%
50–74 pkt.	11	3,3%
1–49 pkt.	161	48,5%
0 pkt.	112	33,7%
brak rozwiązania	35	10,6%

- **SZA** — Szablon

	SZA	
	liczba zawodników	czyli
100 pkt.	4	1,2%
75–99 pkt.	5	1,5%
50–74 pkt.	45	13,6%
1–49 pkt.	180	54,2%
0 pkt.	77	23,2%
brak rozwiązania	21	6,3%

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

SUMA	liczba zawodników	czyli
400 pkt.	0	0,0%
300–399 pkt.	5	1,5%
200–299 pkt.	44	13,3%
1–199 pkt.	275	82,8%
0 pkt.	8	2,4%

Wszystkim zawodnikom przesłano informację o uzyskanych wynikach, a na stronie Olimpiady dostępne były testy, według których sprawdzano rozwiązania.

ZAWODY III STOPNIA

Zawody III stopnia odbyły się w ośrodku firmy Combidata Poland S.A. w Sopocie w dniach od 5 do 9 kwietnia 2005 r.

Do zawodów III stopnia zakwalifikowano 72 najlepszych uczestników zawodów II stopnia, którzy uzyskali wynik nie mniejszy niż 178 pkt.

Zawodnicy uczęszczali do szkół w następujących województwach:

małopolskie	12 uczniów
mazowieckie	12
pomorskie	11
kujawsko-pomorskie	8
śląskie	7
dolnośląskie	5
wielkopolskie	5
podkarpackie	4
zachodniopomorskie	3
lubelskie	2
podlaskie	1 uczeń
świętokrzyskie	1
warmińsko-mazurskie	1

Niżej wymienione szkoły miały w finale więcej niż jednego zawodnika:

V LO im. A. Witkowskiego	Kraków	11 uczniów
III LO im. Marynarki Wojennej RP	Gdynia	8
XIV LO im. St. Staszica	Warszawa	8
VIII LO im. A. Mickiewicza	Poznań	4
VI LO im. J. i J. Śniadeckich	Bydgoszcz	2
VI LO im. W. Sierpińskiego	Gdynia	2
I LO im. St. Staszica	Lublin	2
IV LO im. M. Kopernika	Rzeszów	2
XIII LO	Szczecin	2
LXVII LO	Warszawa	2
III LO im. A. Mickiewicza	Wrocław	2

5 kwietnia odbyła się sesja próbna, na której zawodnicy rozwiązywali nie liczące się do ogólnej klasyfikacji zadanie: „Dziuple”. W dniach konkursowych zawodnicy rozwiązywali zadania: „Akcja komandosów”, „Dwa przyjęcia”, „Dwuszereg”, „Autobus”, „Lustrzana pułapka” i „Prawoskrętny wielbłąd”, 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:

- **DZI — próbne** — Dziuple

	DZI — próbne	
	liczba zawodników	czyli
100 pkt.	7	9,7%
75–99 pkt.	4	5,6%
50–74 pkt.	7	9,7%
1–49 pkt.	20	27,8%
0 pkt.	30	41,6%
brak rozwiązania	4	5,6%

- **AKC** — Akcja komandosów

	AKC	
	liczba zawodników	czyli
100 pkt.	4	5,6%
75–99 pkt.	1	1,4%
50–74 pkt.	2	2,8%
1–49 pkt.	1	1,4%
0 pkt.	62	86,0%
brak rozwiązania	2	2,8%

- **DWA** — Dwa przyjęcia

	DWA	
	liczba zawodników	czyli
100 pkt.	1	1,4%
75–99 pkt.	0	0,0%
50–74 pkt.	0	0,0%
1–49 pkt.	35	48,6%
0 pkt.	23	31,9%
brak rozwiązania	13	18,1%

- **DWU** — Dwuszereg

	DWU	
	liczba zawodników	czyli
100 pkt.	22	30,6%
75–99 pkt.	1	1,4%
50–74 pkt.	1	1,4%
1–49 pkt.	19	26,3%
0 pkt.	27	37,5%
brak rozwiązania	2	2,8%

- **AUT** — Autobus

	AUT	
	liczba zawodników	czyli
100 pkt.	27	37,5%
75–99 pkt.	2	2,8%
50–74 pkt.	12	16,6%
1–49 pkt.	18	25,0%
0 pkt.	9	12,5%
brak rozwiązania	4	5,6%

- **LUS** — Lustrzana pułapka

	LUS	
	liczba zawodników	czyli
100 pkt.	1	1,4%
75–99 pkt.	0	0,0%
50–74 pkt.	0	0,0%
1–49 pkt.	0	0,0%
0 pkt.	38	52,8%
brak rozwiązania	33	45,8%

- **PRA** — Prawoskrętny wielbłąd

	PRA	
	liczba zawodników	czyli
100 pkt.	3	4,2%
75–99 pkt.	3	4,2%
50–74 pkt.	2	2,7%
1–49 pkt.	19	26,4%
0 pkt.	35	48,6%
brak rozwiązania	10	13,9%

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

SUMA	liczba zawodników	czyli
600 pkt.	0	0,0%
450–599 pkt.	1	1,4%
300–449 pkt.	4	5,6%
150–299 pkt.	16	22,2%
1–149 pkt.	45	62,5%
0 pkt.	6	8,3%

W dniu 9 kwietnia 2005 roku, w Sali Posiedzeń Urzędu Miasta w Sopocie, ogłoszono wyniki finału XII Olimpiady Informatycznej 2004/2005 i rozdano nagrody ufundowane przez: PROKOM Software S.A., Wydawnictwa Naukowo-Techniczne i Olimpiadę Informatyczną.

Poniżej zestawiono listę wszystkich laureatów i finalistów:

(1) **Filip Wolski**, III LO im. Marynarki Wojennej RP w Gdyni — laureat I miejsca, złoty medal, 500 pkt. (puchar — Olimpiada Informatyczna; notebook — PROKOM; roczny abonament na książki — WNT)

(2) **Adam Gawarkiewicz**, X LO im. prof. S. Banacha w Toruniu — laureat I miejsca, złoty medal, 388 pkt. (notebook — PROKOM)

(3) **Daniel Czajka**, LO im. KEN w Stalowej Woli — laureat I miejsca, złoty medal, 352 pkt. (notebook — PROKOM)

(4) **Tomasz Kulczyński**, VI LO im. J. i J. Śniadeckich w Bydgoszczy — laureat II miejsca, srebrny medal, 317 pkt. (aparat cyfrowy — PROKOM)

(5) **Jakub Łącki**, III LO im. Marynarki Wojennej RP w Gdyni — laureat II miejsca, srebrny medal, 305 pkt. (aparat cyfrowy — PROKOM)

(6) **Adam Radziwończyk-Syta**, LXVII LO w Warszawie — laureat II miejsca, srebrny medal, 293 pkt. (aparat cyfrowy — PROKOM)

(7) **Piotr Świgoń**, I LO im. Jana Kasprowicza w Inowrocławiu — laureat II miejsca, srebrny medal, 286 pkt. (aparat cyfrowy — PROKOM)

(8) **Mateusz Kwiatek**, V LO im. A. Witkowskiego w Krakowie — laureat II miejsca, srebrny medal, 283 pkt. (aparat cyfrowy — PROKOM)

(9) **Marcin Skotniczny**, V LO im. A. Witkowskiego w Krakowie — laureat II miejsca, 277 pkt. (aparat cyfrowy — PROKOM)

(10) **Paweł Zaborski**, V LO im. A. Witkowskiego w Krakowie — laureat III miejsca, brązowy medal, 220 pkt. (aparat cyfrowy — PROKOM)

(11) **Andrzej Grzywacz**, Prywatne LO im. M. Wańkowicza w Katowicach — laureat III miejsca, brązowy medal, 218 pkt. (aparat cyfrowy — PROKOM)

(12) **Piotr Szmigiel**, V LO im. A. Witkowskiego w Krakowie — laureat III miejsca, brązowy medal, 214 pkt. (aparat cyfrowy — PROKOM)

(13) **Tomasz Łakota**, VIII LO im. M. Skłodowskiej-Curie w Katowicach — laureat III miejsca, brązowy medal, 210 pkt. (aparat cyfrowy — PROKOM)

(14) **Adam Blokus**, III LO im. Marynarki Wojennej RP w Gdyni — laureat III miejsca, brązowy medal, 207 pkt. (aparat cyfrowy — PROKOM)

(15) **Kamil Herba**, XIII LO w Szczecinie — laureat III miejsca, brązowy medal, 205 pkt. (aparat cyfrowy — PROKOM)

(16) **Artur Koniński**, II LO w Koninie — laureat III miejsca, brązowy medal, 200 pkt. (drukarka atramentowa — PROKOM)

(17) **Łukasz Wołochowski**, Gimnazjum nr 24 w Gdyni — laureat III miejsca, brązowy medal, 194 pkt. (drukarka atramentowa — PROKOM)

(18) **Radosław Kożuch**, V LO im. A. Witkowskiego w Krakowie — laureat III miejsca, brązowy medal, 185 pkt. (drukarka atramentowa — PROKOM)

(19) **Sebastian Kochman**, VIII LO im. A. Mickiewicza w Poznaniu — laureat III miejsca, brązowy medal, 163 pkt. (drukarka atramentowa — PROKOM)

(20) **Rafał Pytko**, V LO im. A. Witkowskiego w Krakowie — laureat III miejsca, brązowy medal, 163 pkt. (drukarka atramentowa — PROKOM)

(21) **Michał Pilipczuk**, XIV LO im. St. Staszica w Warszawie — laureat III miejsca, brązowy medal, 150 pkt. (drukarka atramentowa — PROKOM)

(22) **Wojciech Tyczyński**, VI LO im. J. i J. Śniadeckich w Bydgoszczy — finalista z wyróżnieniem, 129 pkt. (drukarka atramentowa — PROKOM)

(23) **Bartłomiej Wołowiec**, V Liceum Ogólnokształcące w Bielsku-Białej — finalista z wyróżnieniem, 128 pkt. (drukarka atramentowa — PROKOM)

(24) **Marcin Mikołajczak**, VIII LO im. A. Mickiewicza w Poznaniu — finalista z wyróżnieniem, 125 pkt. (drukarka atramentowa — PROKOM)

(25) **Krzysztof Pawłowski**, XIV LO im. St. Staszica w Warszawie — finalista z wyróżnieniem, 125 pkt. (drukarka atramentowa — PROKOM)

(26) **Krzysztof Templin**, XIV LO im. Polonii Belgijskiej we Wrocławiu — finalista z wyróżnieniem, 125 pkt. (drukarka atramentowa — PROKOM)

(27) **Bartosz Gęza**, III LO im. Marynarki Wojennej RP w Gdyni — finalista z wyróżnieniem, 121 pkt. (drukarka atramentowa — PROKOM)

(28) **Piotr Jastrzębski**, III LO im. Marynarki Wojennej RP w Gdyni — finalista z wyróżnieniem, 120 pkt. (pamięć Pen Drive — Olimpiada Informatyczna)

(29) **Szymon Wrzeszcz**, II LO im. Jana III Sobieskiego w Grudziądzu — finalista z wyróżnieniem, 118 pkt. (pamięć Pen Drive — Olimpiada Informatyczna)

(30) **Norbert Potocki**, XIV LO im. St. Staszica w Warszawie — finalista z wyróżnieniem, 118 pkt. (pamięć Pen Drive — Olimpiada Informatyczna)

(31) **Adrian Galewski**, Zespół Szkół Technicznych w Wodzisławiu Śl. — finalista z wyróżnieniem, 114 pkt. (pamięć Pen Drive — Olimpiada Informatyczna)

(32) **Oskar Strączkowski**, VI LO im. W. Sierpińskiego w Gdyni — finalista z wyróżnieniem, 113 pkt. (pamięć Pen Drive — Olimpiada Informatyczna)

(33) **Bolesław Kulbabiński**, Gimnazjum im. Św. Jadwigi Królowej w Kielcach — finalista z wyróżnieniem, 113 pkt. (pamięć Pen Drive — Olimpiada Informatyczna)

Lista pozostałych finalistów w kolejności alfabetycznej:

- **Kamil Adamczyk**, I LO im. St. Staszica w Lublinie
- **Kamil Anikiej**, XIV LO im. St. Staszica w Warszawie

- **Jacek Caban**, III LO we Wrocławiu
- **Grzegorz Chilkiewicz**, I LO im. S. Żeromskiego w Goleniowie
- **Sebastian Chojniak**, I Liceum Ogólnokształcące w Piszu
- **Adam Choma**, III LO im. Marynarki Wojennej RP w Gdyni
- **Bartosz Dąbrowski**, XIV LO im. St. Staszica w Warszawie
- **Mateusz Gajczewski**, III LO im. Stefana Batorego w Chorzowie
- **Piotr Gurgul**, V LO im. A. Witkowskiego w Krakowie
- **Miłosz Kordecki**, II LO im. Piastów Śląskich we Wrocławiu
- **Przemysław Kosiak**, I Liceum Ogólnokształcące w Bydgoszczy
- **Grzegorz Kossowski**, XIV LO im. St. Staszica w Warszawie
- **Krzysztof Kotuła**, I LO im. St. Staszica w Lublinie
- **Kamil Kraszewski**, V LO im. A. Witkowskiego w Krakowie
- **Krzysztof Krygiel**, IV LO im. Tadeusza Kościuszki w Toruniu
- **Karol Kurach**, XIV LO im. St. Staszica w Warszawie
- **Wojciech Leja**, LO WSIiZ w Rzeszowie
- **Dariusz Leniowski**, IV LO w Rzeszowie
- **Krzysztof Magiera**, II LO im. Jana III Sobieskiego w Krakowie
- **Jakub Muszyński**, V LO im. Jana III Sobieskiego w Białymstoku
- **Robert Nasiadek**, XIV LO im. St. Staszica w Warszawie
- **Jakub Neumann**, V LO im. A. Witkowskiego w Krakowie
- **Kamil Nowosad**, I Liceum Ogólnokształcące w Gliwicach
- **Hubert Orlik-Grzesik**, V LO im. A. Witkowskiego w Krakowie
- **Grzegorz Paszt**, XIII LO w Szczecinie
- **Maciej Pawlisz**, III LO im. Marynarki Wojennej RP w Gdyni
- **Kamil Serwus**, VIII LO im. A. Mickiewicza w Poznaniu
- **Filip Skalski**, 67 LO przy wydziale UW w Warszawie
- **Karol Sobczak**, LO im. K. I. Gałczyńskiego w Otwocku
- **Juliusz Sompolski**, Gimnazjum i Liceum Akademickie w Toruniu

- **Jakub Staszak**, VIII LO im. A. Mickiewicza w Poznaniu
- **Bartosz Szreder**, VI LO im. W. Sierpińskiego w Gdyni
- **Marek Szykuła**, Prywatne Salezjańskie LO we Wrocławiu
- **Wojciech Śmietanka**, III LO im. Marynarki Wojennej RP w Gdyni
- **Łukasz Wiatrak**, V LO im. A. Witkowskiego w Krakowie
- **Przemysław Witek**, I LO im. K. Miarki w Żorach
- **Michał Wróbel**, III LO im. Adama Mickiewicza we Wrocławiu
- **Bartosz Zaborowski**, XVIII LO im. J. Zamoyskiego w Warszawie
- **Grzegorz Ziemiański**, IV LO im. M. Kopernika w Rzeszowie

Wszyscy laureaci i finaliści otrzymali książki ufundowane przez Wydawnictwa Naukowo-Techniczne.

Ogłoszono komunikat o powołaniu reprezentacji Polski na:

- Międzynarodową Olimpiadę Informatyczną IOI'2005 — Nowy Sącz, Polska, 18–25 sierpnia 2005
  - I reprezentacja:
    (1) Filip Wolski
    (2) Adam Gawarkiewicz
    (3) Daniel Czajka
    (4) Tomasz Kulczycki

    rezerwowi:

    (5) Jakub Łącki
    (6) Adam Radziwończyk-Syta
  - II reprezentacja (poza konkursem):
    (1) Piotr Świgoń
    (2) Marcin Skotniczny
    (3) Andrzej Grzywocz
    (4) Piotr Szmigiel

    rezerwowi:

    (5) Kamil Herba
    (6) Artur Koniński
- Olimpiadę Informatyczną Krajów Europy Środkowej CEOI'2005 — Węgry, 28 lipca – 4 sierpnia 2005
  (1) Filip Wolski

(2) Adam Gawarkiewicz
(3) Daniel Czajka
(4) Tomasz Kulczycki

rezerwowi:

(5) Jakub Łącki
(6) Adam Radziwończyk-Syta

- Bałtycką Olimpiadę Informatyczną — Litwa, 5–9 maja 2005

(1) Filip Wolski
(2) Adam Gawarkiewicz
(3) Tomasz Kulczyński
(4) Piotr Świgoń
(5) Marcin Skotniczny
(6) Andrzej Grzywocz

rezerwowi:

(7) Piotr Szmigiel
(8) Kamil Herba

- Obóz czesko-polsko-słowacki: Słowacja, 4–10 lipca 2005:
  - członkowie pierwszej 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 ponadgimnazjalnych.

Sekretariat wystawił łącznie 21 zaświadczeń o uzyskaniu tytułu laureata, 12 o uzyskaniu tytułu finalisty z wyróżnieniem i 39 zaświadczeń o uzyskaniu tytułu finalisty XII Olimpiady Informatycznej.

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

- Wiesława Amietszajew (II LO w Koninie)
  - Artur Koniński — laureat III miejsca
- Mariusz Blank (Lucent Technologies w Bydgoszczy)
  - Przemysław Kosiak — finalista

- Przemysław Broniek (V LO im. A. Witkowskiego w Krakowie)
  - Mateusz Kwiatek — laureat II miejsca
  - Paweł Zaborski — laureat III miejsca
- Roman Chojniak (TP Edukacja i Wypoczynek w Warszawie)
  - Sebastian Chojniak — finalista
- Czesław Drozdowski (XIII LO w Szczecinie)
  - Grzegorz Paszt — finalista
- Jacek Dymel (II LO im. Jana III Sobieskiego w Krakowie)
  - Krzysztof Magiera — finalista
- Andrzej Dyrek (V LO im. A. Witkowskiego w Krakowie)
  - Mateusz Kwiatek — laureat II miejsca
  - Marcin Skotniczny — laureat II miejsca
  - Rafał Pytko — laureat III miejsca
  - Radosław Kożuch — laureat III miejsca
  - Paweł Zaborski — laureat III miejsca
  - Łukasz Wiatrak — finalista
  - Kamil Kraszewski — finalista
  - Hubert Orlik-Grzesik — finalista
- Mirosława Firszt (IV LO w Toruniu)
  - Krzysztof Krygiel — finalista
- Andrzej Gajczewski (Gimnazjum nr 14 w Rudzie Śląskiej)
  - Mateusz Gajczewski — finalista
- Maciej Gielnik (I LO im. S. Żeromskiego w Goleniowie)
  - Grzegorz Chilkiewicz — finalista
- Alina Gościniak (VIII LO im. A. Mickiewicza w Poznaniu)
  - Kamil Serwus — finalista
  - Marcin Mikołajczak — finalista z wyróżnieniem
- Adam Herman (I LO im. K. Miarki w Żorach)
  - Przemysław Witek — finalista
- Andrzej Jackowski (V LO im. Jana III Sobieskiego w Białymstoku)

  - Jakub Muszyński — finalista
- Kamil Kloch (V LO im. A. Witkowskiego w Krakowie)
  - Piotr Szmigiel — laureat III miejsca
  - Piotr Gurgul — finalista
  - Jakub Neumann - finalista
- Jarosław Kruszyński (I LO im. J. Kasprowicza w Inowrocławiu)
  - Piotr Świgoń — laureat II miejsca
- Rafał Kulbabiński (Agencja Reklamowa Studio 99 w Kielcach)
  - Bolesław Kulbabiński — finalista z wyróżnieniem
- Anna Beata Kwiatkowska (Gimnazjum i Liceum Akademickie w Toruniu)
  - Juliusz Sompolski — finalista
- Stanisław Łakota (Katowice)
  - Tomasz Łakota — laureat III miejsca
- Mirosław Maciejczyk (XXVIII LO im. J. Zamoyskiego w Warszawie)
  - Bartosz Zaborowski — finalista
- Dawid Matla (XIV LO im. Polonii Belgijskiej we Wrocławiu)
  - Krzysztof Templin — finalista z wyróżnieniem
- Andrzej Michalczyk (LO im. K. I. Gałczyńskiego w Otwocku)
  - Karol Sobczak — finalista
- Bartosz Nowierski (Politechnika Poznańska w Poznaniu)
  - Sebastian Kochman — laureat III miejsca
- Irena Olender (IV LO w Rzeszowie)
  - Dariusz Leninowski — finalista
- Grzegorz Owsiany (WSIiZ w Rzeszowie)
  - Wojciech Leja — finalista
- Paweł Parys (Uniwersytet Warszawski)
  - Adam Radziwończyk-Syta — laureat II miejsca
  - Filip Skalski — finalista
- Ryszard Popardowski (Zespół Szkół Technicznych w Wodzisławiu Śląskim)

   - Adrian Galewski — finalista z wyróżnieniem
- Izabela Potocka (Główny Inspektorat Pracy w Warszawie)
   - Norbert Potocki — finalista z wyróżnieniem
- Włodzimierz Raczek (V LO w Bielsku-Białej)
   - Bartłomiej Wołowiec — finalista z wyróżnieniem
- Jadwiga Rogucka (Gimnazjum i Liceum Akademickie w Toruniu)
   - Juliusz Sompolski — finalista
- Dorota Roman-Jurdzińska (II L O im. Piastów Śląskich we Wrocławiu)
   - Miłosz Korecki — finalista
- Wojciech Roszczyński (VIII LO im. A. Mickiewicza w Poznaniu)
   - Jakub Staszak — finalista
- Bożena Rutkowska (Gimnazjum im. Jana Pawła II w Nurze)
   - Grzegorz Kossowski — finalista
- Ryszard Szubartowski (III LO im. Marynarki Wojennej RP w Gdyni)
   - Filip Wolski — laureat I miejsca
   - Adam Blokus — laureat II miejsca
   - Jakub Łącki — laureat II miejsca
   - Andrzej Grzywacz — laureat III miejsca
   - Łukasz Wołochowski — laureat III miejsca
   - Bartosz Gęza — finalista z wyróżnieniem
   - Piotr Jastrzębski — finalista z wyróżnieniem
   - Oskar Strączkowski — finalista z wyróżnieniem
   - Adam Choma — finalista
   - Maciej Pawlisz — finalista
   - Bartosz Szreder — finalista
   - Wojciech Śmietanka — finalista
- Michał Szuman (XIII LO w Szczecinie)
   - Kamil Herba — laureat III miejsca
- Szymon Wąsik (Politechnika Poznańska)
   - Sebastian Kochman — laureat III miejsca
- Marcin Wrzeszcz (Instytut Informatyki Uniwersytetu Wrocławskiego)

  - Krzysztof Templin — finalista z wyróżnieniem

- Joanna Śmigielska (XIV LO im. St. Staszica w Warszawie)
  - Kamil Anikiej — finalista
  - Karol Kurach — finalista
  - Michał Pilipczuk — laureat III miejsca
- Iwona Waszkiewicz (VI LO im. J. i J. Śniadeckich w Bydgoszczy)
  - Tomasz Kulczyński — laureat II miejsca
  - Wojciech Tyczyński — finalista z wyróżnieniem
- Bolesław Wojdyło (Gimnazjum i Liceum Akademickie w Toruniu)
  - Juliusz Sompolski — finalista
- Marek Wojtan (I LO im. St. Staszica w Lublinie)
  - Krzysztof Kotuła — finalista
  - Kamil Adamczyk — finalista
- Krzysztof Ziemiański (Uniwersytet Warszawski)
  - Grzegorz Ziemiański — finalista

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

Warszawa, 24 czerwca 2005 roku

Regulamin Olimpiady Informatycznej

§1 WSTĘP

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

§2 CELE OLIMPIADY

(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, zwany dalej Komitetem.

(2) Olimpiada jest trójstopniowa.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

§4 KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

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

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

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

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

(5) Komitet powołuje i rozwiązuje komitety okręgowe Olimpiady.

(6) Komitet:

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

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

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

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

(9) Do organizacji zawodów II stopnia w miejscowościach, których nie obejmuje żaden komitet okręgowy, Komitet powołuje komisję zawodów co najmniej 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 dla każdej edycji Olimpiady na pierwszym posiedzeniu Komitetu w nowym roku szkolnym.

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

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

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

(16) 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.

(17) 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.

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

§5 KOMITETY OKRĘGOWE

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

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

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

§6 PRZEBIEG OLIMPIADY

(1) Komitet rozsyła do szkół wymienionych w § 3.3 oraz kuratoriów oświaty i koordynatorów edukacji informatycznej treści zadań I stopnia wraz z Zasadami.

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

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

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

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

§7 UPRAWNIENIA I NAGRODY

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

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

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

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

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

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

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

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

§8 FINANSOWANIE OLIMPIADY

Komitet będzie się ubiegał o pozyskanie środków finansowych z budżetu państwa, składając wniosek w tej sprawie do Ministra Edukacji Narodowej i 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 zatwierdza sprawozdanie merytoryczne z przeprowadzonej edycji Olimpiady w ciągu 3 miesięcy po zakończeniu zawodów III stopnia i przedstawia je Organizatorowi i Ministerstwu Edukacji Narodowej i Sportu.

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

Warszawa, 3 września 2004 r.

Zasady organizacji zawodów w roku szkolnym 2004/2005

Podstawowym aktem prawnym dotyczącym Olimpiady jest Regulamin Olimpiady Informatycznej, którego pełny tekst znajduje się w kuratoriach. Poniższe zasady są uzupełnieniem tego Regulaminu, zawierającym szczegółowe postanowienia Komitetu Głównego Olimpiady Informatycznej o jej organizacji w roku szkolnym 2004/2005.

§1 WSTĘP

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

§2 ORGANIZACJA OLIMPIADY

(1) Olimpiadę przeprowadza Komitet Główny Olimpiady Informatycznej

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

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

(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 —

60 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 — 25.10–22.11.2004 r.

  ogłoszenie wyników:

  - w witrynie Olimpiady — 17.12.2004 r.,
  - pocztą — 29.12.2004 r.
- zawody II stopnia — 8–10.02.2005 r.

  ogłoszenie wyników:

  - w witrynie Olimpiady — 18.02.2005 r.
  - pocztą — 25.02.2005 r.
- zawody III stopnia — 5–9.04.2005 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 22 listopada 2004 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 podane w witrynie.
- Pocztą, przesyłką poleconą, na adres:

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

w nieprzekraczalnym terminie nadania do 22 listopada 2004 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 jest konieczne 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) Rozwiązania zespołowe, niesamodzielne, niezgodne z „Zasadami organizacji zawodów” lub takie, co do których nie można ustalić autorstwa, nie będą oceniane.

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

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

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

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

(7) Programy w *C*/*C++* będą kompilowane w systemie Linux za pomocą kompilatora GCC v. 3.3.
Programy w Pascalu będą kompilowane w systemie Linux za pomocą kompilatora FreePascal v. 1.0.10. Wybór polecenia kompilacji zależy od podanego rozszerzenia pliku w następujący sposób (np. dla zadania *abc*):

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

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

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

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

(10) Uczestnik korzystający z poczty zwykłej przysyła:

- Dyskietkę lub CD-ROM w standardzie dla komputerów PC, zawierające:
  - spis zawartości dyskietki oraz dane osobowe zawodnika w pliku nazwanym `SPIS.TXT`,
  - do każdego rozwiązanego zadania — program źródłowy.

  Dyskietka nie powinna zawierać żadnych podkatalogów.
- Wypełniony dokument zgłoszeniowy (dostępny w witrynie internetowej Olimpiady).

Gorąco prosimy o podanie adresu elektronicznego. Podanie adresu jest niezbędne do wzięcia udziału w procedurze reklamacyjnej opisanej w punktach 14, 15 i 16.

(11) Uczestnik korzystający z witryny Olimpiady postępuje zgodnie z instrukcjami umieszczonymi w witrynie.

(12) W witrynie Olimpiady 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.

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

(14) Od dnia 6 grudnia 2004 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.

(15) Do dnia 10 grudnia 2004 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.

(16) Reklamacje złożone po 10 grudnia 2004 r. nie będą rozpatrywane.

§4 UPRAWNIENIA I NAGRODY

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

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

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

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

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

(6) Komitet Główny ustala skład reprezentacji Polski na XVII Międzynarodową Olimpiadę Informatyczną w 2005 roku na podstawie wyników zawodów III stopnia i regulaminu tej Olimpiady Międzynarodowej.

(7) Nauczyciel, który przygotował laureata lub finalistę Olimpiady Informatycznej, otrzymuje nagrodę przyznawaną przez Komitet Główny.

(8) Wyznaczeni przez Komitet Główny reprezentanci Polski na olimpiady międzynarodowe oraz finaliści, którzy nie są w ostatniej programowo klasie swojej szkoły, zostaną zaproszeni do nieodpłatnego udziału w VI Obozie Naukowo-Treningowym im. Antoniego Kreczmara, który odbędzie się w czasie wakacji 2005 r.

(9) 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 zakwalifikował się 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

opracowania zadań

Piotr Stańczyk — Treść zadania, Opracowanie
Rafał Rusin, Jakub Radoszewski — Program

Bankomat

Bajtocki Bank Bitowy (w skrócie BBB) ma największą w Bajtocji sieć bankomatów. Klienci BBB wypłacają pieniądze z bankomatów na podstawie karty bankomatowej i 4-cyfrowego kodu PIN. Niedawno, w celu zwiększenia bezpieczeństwa swoich klientów, BBB zainstalował przy każdym bankomacie kamerę. Kamery przesyłają rejestrowany obraz do BBB drogą radiową. Niestety, sygnał wysyłany przez kamery jest przechwytywany przez szajkę złodziejaszków komputerowych. Złodziejaszkowie starają się odkryć 4-cyfrowe kody PIN klientów BBB, którym następnie kradną karty bankomatowe. Wiedząc o tym, klienci BBB, wprowadzając PIN i przesuwając palec nad klawiaturą, starają się wykonywać nadmiarowe ruchy. Kamera nie jest w stanie wychwycić naciskania klawiszy, rejestruje jedynie przesunięcia palca. Tak więc, jednoznaczne wyznaczenie wprowadzanego kodu PIN jest zazwyczaj niemożliwe. Przykładowo, klient przesuwający swój palec najpierw nad klawiszem 1, a potem 5, mógł wprowadzić następujące kody PIN: 1111, 1115, 1155, 1555 lub 5555. Zdesperowani złodziejaszkowie gromadzą nagrania z kamer, licząc na to, że być może na podstawie wielu nagrań tego samego klienta będą w stanie wyznaczyć wprowadzany przez niego PIN, lub choćby ograniczyć liczbę możliwych kodów. Zebrawszy sekwencje wprowadzane przez pewnego bogatego klienta BBB, złożyli Ci propozycję „nie do odrzucenia”.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis zarejestrowanych sekwencji ruchów palca, jakie klient banku wykonywał wprowadzając swój PIN,*
- *wyznaczy liczbę różnych kodów PIN, jakie może mieć klient (czyli liczbę tych 4-cyfrowych kodów PIN, które mogły być wprowadzone do bankomatu przy wykonywaniu zadanych sekwencji ruchów palca),*
- *wypisze znalezione rozwiązanie na standardowe wyjście.*

Wejście

Pierwszy wiersz wejścia zawiera jedną dodatnią liczbę całkowitą n równą liczbie zarejestrowanych scen wprowadzania kodu PIN przez klienta, $1 \leqslant n \leqslant 1000$. W kolejnych n wierszach znajdują się opisy tych scen, po jednej w wierszu. Opis każdej sceny składa się z dwóch dodatnich liczb całkowitych oddzielonych pojedynczym odstępem. Pierwsza z tych liczb, t, to długość sekwencji ruchów, $1 \leqslant t \leqslant 10\,000$. Druga z tych liczb to t-cyfrowa liczba, której kolejne cyfry reprezentują kolejne klawisze, nad którymi klient przesuwał palec. Łączna długość wszystkich sekwencji nie przekracza $1\,000\,000$.

Wyjście

W pierwszym i jedynym wierszu wyjścia powinna znaleźć się jedna dodatnia liczba całkowita równa liczbie możliwych kodów PIN klienta.

Przykład

Dla danych wejściowych:

```
2
3 123
3 234
```

poprawnym wynikiem jest:

```
5
```

Rozwiązanie

Wprowadzenie

W treści zadania zwraca uwagę stosunkowo małe ograniczenie na długość kodu PIN (jest ono stałe — równe 4). W wielu urządzeniach, z których korzystamy w życiu codziennym (choćby w telefonach komórkowych), kody PIN są właśnie czterocyfrowe. Mają więc tylko 10 000 możliwych wartości, co pozwala na ich sekwencyjną weryfikację.

Zauważmy także, że na obrazie zarejestrowanym przez kamerę widać jedynie przesuwanie palca nad klawisze, nie widać natomiast, czy klient naciska klawisz i ile razy. Tak więc scena przedstawiająca przesuwanie palca kolejno nad klawisze 1, 2, 0 i 3 może oznaczać zarówno, że klient wpisał PIN 2233, jak i 1203 (oraz wiele innych).

Oznaczenia

Niech $s_1, s_2, \ldots, s_n$ będą sekwencjami wprowadzania kodu przez jednego klienta zarejestrowanymi przez kamerę. Oznaczmy przez w dowolny czterocyfrowy kod PIN, a przez $P(w)$ — jego *postać bez powtórzeń*, czyli sekwencję uzyskaną z kodu w przez usunięcie sąsiednich powtórzeń cyfr (przykładowo $P(1211) = 121$, $P(4333) = 43$, $P(1873) = 1873$).

Aby znaleźć wszystkie kody PIN, które mogą być kodem obserwowanego klienta, musimy dla każdej zarejestrowanej sekwencji s i każdego potencjalnego kodu PIN w sprawdzić, czy $P(w)$ jest podciągiem s. Szybkie wykonywanie tego sprawdzenia jest więc kluczowe dla efektywności algorytmu. Okazuje się, że można je wykonać w czasie liniowo zależnym od długości testowanego kodu PIN (a w naszym przypadku wynosi ona tylko 4).

Procedurę sprawdzenia, czy dla słowa a (potencjalnego kodu) jego postać bez powtórzeń $P(a)$ jest podciągiem słowa $b = b_0 b_1 \ldots b_{m-1}$ (zarejestrowanej sekwencji), rozpoczniemy od przetworzenia słowa b. Zdefiniujmy tablicę $B[x][y]$, dla $x \in \{0 \ldots m-1\}$ oraz $y \in \{0 \ldots 9\}$, w następujący sposób:

$$B[x][y] = \begin{cases} x & \text{dla } y = b_x \text{ oraz } x \leqslant m-1 \\ -1 & \text{dla } y \neq b_x \text{ oraz } x = m-1 \\ B[x+1][y] & \text{dla } y \neq b_x \text{ oraz } x \leqslant m-2 \end{cases}$$

Zawartość tablicy B można obliczyć w czasie $O(10 \cdot m)$ korzystając bezpośrednio z powyższej definicji. Z kolei mając już tablicę B można łatwo przeprowadzić test, o którym była mowa wcześniej.

```
procedure wB(a)
  pos = 0;
  for(int k = 0; k ⩽ 3 && pos ≠ −1; k++)
    pos = B[pos][a[k]];
  return (pos ≠ −1);
```

Sposób działania powyższego algorytmu jest prosty. Wystarczy zauważyć, że $B[x][y]$ to najmniejsza pozycja $z \geqslant x$ w słowie b, na której występuje cyfra y (lub -1, gdy cyfra y nie występuje na pozycjach $z \geqslant y$). Poszukiwania rozpoczynamy od cyfry $a[0]$ i $pos = 0$, znajdując za pomocą tablicy B w czasie stałym pierwsze wystąpienie $a[0]$ w b. Potem poszukujemy $a[1]$ począwszy od pozycji, na której znaleźliśmy $a[0]$. Analogicznie postępujemy dla $a[2]$ i $a[3]$. Dodatkowo, w przypadku serii jednakowych cyfr w słowie a, nie zmieniamy wartości pos, gdyż $B[pos][a[k]] = pos$ dla $a[k] = b_{pos}$. W ten sposób poszukujemy w rzeczywistości wystąpienia $P(a)$, a nie słowa a.

Ostatecznie procedura zwraca wartość *true* wtedy i tylko wtedy, gdy $P(a)$ występuje w b jako podciąg, a więc słowo a jest kodem PIN, którego wprowadzanie mogło zostać zarejestrowane, jako sekwencja b.

Rozwiązanie wzorcowe

Rozwiązanie wzorcowe rozpoczyna się od wczytania wszystkich sekwencji $s_1, s_2, \ldots, s_n$ i stworzenia dla nich tablic $B_1, B_2, \ldots, B_n$ zgodnie ze schematem opisanym wcześniej. Po zakończeniu tych przygotowań przeglądane są kolejno wszystkie kody PIN ze zbioru $\{0000, \ldots, 9999\}$. Dla każdego z nich sprawdzamy, czy procedura `wB` wykonana dla tablicy B_i pozwoli odnaleźć jego postać bez powtórzeń w sekwencji s_i. Końcowy wynik obliczamy zliczając kody występujące we wszystkich wczytanych sekwencjach.

Budowa jednej tablicy B_i odbywa się w czasie liniowym ze względu na jej wielkość, zatem sumaryczny czas potrzebny na skonstruowanie wszystkich tablic wynosi $O(10N_s) = O(N_s)$, gdzie $N_s = \sum_{i=1}^{n} |s_i|$. Wykonanie pojedynczego testu wymaga czasu stałego, zatem skonfrontowanie jednego kodu PIN ze wszystkimi sekwencjami $s_1, s_2, \ldots, s_n$ zajmuje czas $O(n)$. Takie sprawdzenie musimy wykonać dla każdego możliwego kodu PIN — jest ich $r = 10\,000$. Cały algorytm działa więc w czasie $O(N_s + n \cdot r)$.

Opisane rozwiązanie ma złożoność pamięciową $O(N_s)$, ponieważ wszystkie tablice B_i są konstruowane na początku i przechowywane w pamięci. Mieści się to w ograniczeniach podanych dla zadania. Można jednak nieco zmienić algorytm istotnie redukując jego zapotrzebowanie na pamięć. W tym celu wystarczy rozważać sekwencje $s_1, s_2, \ldots, s_n$ kolejno, dla każdej z nich budować tablicę B i konfrontować wszystkie możliwe kody PIN z tą sekwencją. Do zapisania wyników testów jest wówczas potrzebna dodatkowa tablica o rozmiarze $O(r)$, zawierająca dla każdego kodu liczbę sekwencji, które mogą oznaczać jego wprowadzenie. W ten sposób złożoność pamięciowa algorytmu zostaje zredukowana do $O(r+m)$, gdzie m jest ograniczeniem długości jednej sekwencji, przy zachowaniu bez zmian złożoności czasowej.

Inne rozwiązania

Zadanie można rozwiązać wykonując zwykłe wyszukiwanie kodu PIN (a dokładniej, jego postaci bez powtórzeń) w sekwencji w czasie liniowo zależnym od sumy długości kodu i sekwencji. Daje to w rezultacie program działający w czasie $O(N_s \cdot r)$.

Testy

Zadanie było sprawdzane na 15 testach.

Pierwsze trzy testy (*ban1.in* – *ban3.in*) to małe testy poprawnościowe. Test *ban4.in* to duży test poprawnościowy o specyficznej strukturze. Okazał się on najtrudniejszy — na nim najczęściej programy zawodników zwracały złe wyniki. Reszta testów (*ban5.in* – *ban15.in*) została wygenerowana w sposób pseudo-losowy.

Poniżej znajduje się krótki opis każdego z testów, zawierający liczbę zarejestrowanych scen wprowadzania kodu PIN n oraz sumaryczną długość sekwencji reprezentujących zarejestrowane sceny N_s.

Nazwa	**n**	**N$_s$**
ban1.in	2	6
ban2.in	1	1
ban3.in	5	195
ban4.in	100	15 125
ban5.in	100	807 101
ban6.in	100	973 642
ban7.in	1 000	973 718
ban8.in	1 000	974 747

Nazwa	**n**	**N$_s$**
ban9.in	1 000	974 369
ban10.in	1 000	974 127
ban11.in	500	987 664
ban12.in	500	987 696
ban13.in	1 000	974 364
ban14.in	1 000	974 459
ban15.in	1 000	974 731

Piotr Stańczyk Treść zadania, Opracowanie
Arkadiusz Paterek Program

Punkty

Dany jest zbiór punktów na płaszczyźnie o współrzędnych całkowitych, który będziemy nazywać **wzorem***, oraz zestaw innych zbiorów punktów na płaszczyźnie (również o współrzędnych całkowitych). Interesuje nas, które z podanych zestawów są podobne do wzoru, tzn. można je tak przekształcić za pomocą obrotów, przesunięć, symetrii osiowej i jednokładności, aby były identyczne ze wzorem. Przykładowo: zbiór punktów* $\{(0,0),(2,0),(2,1)\}$ *jest podobny do zbioru* $\{(6,1),(6,5),(4,5)\}$*, ale nie do zbioru* $\{(4,0),(6,0),(5,-1)\}$.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opisy: wzoru oraz zestawu badanych zbiorów punktów,*
- *wyznaczy, które z badanych zbiorów punktów są podobne do wzoru,*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu standardowego wejścia zapisana jest jedna liczba całkowita k $(1 \leqslant k \leqslant 25\,000)$ *— liczba punktów tworzących wzór. W kolejnych* k *wierszach zapisane są pary liczb całkowitych pooddzielanych pojedynczymi odstępami. W* $i+1$*-ym wierszu są współrzędne* i*-tego punktu należącego do wzoru:* x_i *i* y_i $(-20\,000 \leqslant x_i, y_i \leqslant 20\,000)$*. Punkty tworzące wzór są (parami) różne.*

W kolejnym wierszu zapisana jest liczba zbiorów do zbadania n $(1 \leqslant n \leqslant 20)$*. Dalej następuje* n *opisów zbiorów punktów. Opis każdego zbioru rozpoczyna się od wiersza zawierającego jedną liczbę całkowitą* l *— liczbę punktów w danym zbiorze* $(1 \leqslant l \leqslant 25\,000)$*. Punkty te są opisane w kolejnych wierszach, po jednym w wierszu. Opis punktu to dwie liczby całkowite oddzielone pojedynczym odstępem oznaczające jego współrzędne* x *i* y $(-20\,000 \leqslant x, y \leqslant 20\,000)$*. Punkty tworzące jeden zbiór są parami różne.*

Wyjście

Twój program powinien wypisać na standardowe wyjście n *wierszy — po jednym dla każdego badanego zbioru punktów. Wiersz* i*-ty powinien zawierać słowo* `TAK`*, gdy* i*-ty z podanych zbiorów punktów jest podobny do podanego wzoru, lub słowo* `NIE` *w przeciwnym przypadku.*

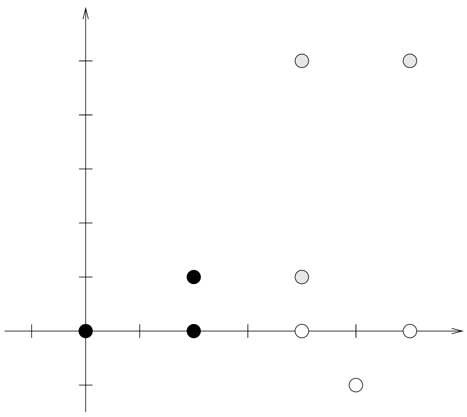
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
TAK
NIE
```



Rozwiązanie

Analiza problemu

Na wstępie zauważmy, że aby zbiory były podobne, muszą zawierać po tyle samo punktów. Załóżmy więc, że mamy dwa równoliczne zbiory punktów S_1 i S_2. Chcąc sprawdzić, czy są one do siebie podobne, możemy każdy z nich przekształcać za pomocą operacji:

- symetrii osiowej,
- przesunięcia,
- jednokładności (o skali większej od 0),
- obrotu.

Jeżeli w wyniku otrzymamy dwa identyczne zbiory, to mamy twierdzącą odpowiedź na pytanie.

W rozwiązaniu wzorcowym korzystamy z faktu, iż złożenie dowolnego ciągu wymienionych wyżej przekształceń można sprowadzić do złożenia najwyżej czterech operacji — symetrii osiowej, przesunięcia, jednokładności i obrotu. W algorytmie staramy się wyznaczyć te operacje — jeśli nam się to uda, to oznacza, że testowane zbiory punktów są podobne. Co więcej, poszukiwane operacje można wyznaczać niezależnie od siebie. Jest to możliwe dzięki określeniu pewnych niezmienników, które muszą być zachowane przez operacje pozostałe do wykonania.

Zauważmy, że zadanie istotnie upraszcza się, jeżeli znajdziemy parę punktów $P_1 \in S_1$ oraz $P_2 \in S_2$, o których wiemy, że w wyniku podobieństwa P_1 musi przejść na P_2. Wówczas możemy każdy ze zbiorów przesunąć tak, by P_1 i P_2 znalazły się w środku układu współrzędnych. Potem wystarczy sprawdzić, czy za pomocą symetrii osiowej (według osi OX lub OY), jednokładności (o środku $(0,0)$) i obrotu (również o środku $(0,0)$) można zbiory sprowadzić do tej samej postaci.

Kluczowym dla rozwiązania zadania jest spostrzeżenie, że rolę takich „punktów zaczepienia" mogą odegrać środki ciężkości zbiorów.

Definicja 1 Dla zbioru punktów $S = \{(x_1,y_1),(x_2,y_2),\ldots(x_k,y_k)\}$ *środek ciężkości* to punkt

$$H(S) = (s_x,s_y) = (\frac{x_1+x_2+\ldots+x_k}{k}, \frac{y_1+y_2+\ldots+y_k}{k})$$

Środek ciężkości ma dwie istotne dla nas własności:

- przy wszystkich rozważanych operacjach środek ciężkości zbioru punktów przed przekształceniem przechodzi na środek ciężkości zbioru po przekształceniu, czyli $R(H(S)) = H(R(S))$, gdzie R oznacza dowolną z wymienionych wcześniej operacji zastosowaną do punktu lub zbioru punktów;
- zbiory punktów S_1 i S_2 (równoliczne) są do siebie podobne wtedy i tylko wtedy, gdy zbiory $S_1 \cup \{H(S_1)\}$ i $S_2 \cup \{H(S_2)\}$ (czyli S_1 i S_2 rozszerzone o swoje środki ciężkości) są do siebie podobne.

Pierwszą z powyższych własności można łatwo sprawdzić analizując kolejno każdą z dopuszczalnych operacji. Druga własność wynika z pierwszej. W ciągu operacji $R_1,R_2,\ldots,R_r$ przekształcających zbiór S_1 w S_2, środek ciężkości $H(S_1)$ przechodzi na $H(S_2)$. Stąd ten sam ciąg operacji przeprowadza zbiór $S_1 \cup \{H(S_1)\}$ w $S_2 \cup \{H(S_2)\}$. Z drugiej strony, jeśli $Q_1,Q_2,\ldots,Q_t$ jest ciągiem przekształcającym zbiór $S_1 \cup \{H(S_1)\}$ w $S_2 \cup \{H(S_2)\}$, to musi on (ponownie z pierwszej własności) przekształcić punkt $H(S_1)$ na $H(S_2)$. W takim razie ciąg $Q_1,Q_2,\ldots,Q_t$ zastosowany do pozostałych punktów zbioru, czyli S_1, przeprowadza je na S_2.

Chcąc wykorzystać własności środka ciężkości, na początku każdy ze zbiorów punktów rozszerzymy o jego środek ciężkości. Następnie sprawdzimy podobieństwo zbiorów przekształcając je przez odpowiednie przesunięcie (sprowadzając środki ciężkości do punktu $(0,0)$) i testując, czy stosując jednokładność i obrót można sprowadzić zbiory do tej samej postaci. Analogiczny test przeprowadzamy po przekształceniu jednego ze zbiorów przez symetrię osiową.

Symetria osiowa

Badając, czy do wykazania podobieństwa zbiorów należy zastosować symetrię osiową, rozważymy dwa przypadki. Oddzielnie sprawdzimy podobieństwo zbiorów S_1 i S_2 oraz $O(S_1)$ i S_2, gdzie O jest symetrią według osi OY. Zbiory S_1 i S_2 są podobne tylko wówczas, gdy jeden z testów zakończy się wynikiem pozytywnym. Symetrię osiową wykonujemy zamieniając wszystkie współrzędne x punktów zbioru S_1 na wartości przeciwne.

Dalszą część rozważań prowadzimy oddzielnie dla par zbiorów S_1 i S_2 oraz $O(S_1)$ i S_2.

Przesunięcie

Wykonujemy przesunięcie każdego ze zbiorów odpowiednio o wektor $(-a_x, -a_y)$ oraz $(-b_x, -b_y)$, gdzie $H(S_1) = (a_x, a_y)$ oraz $H(S_2) = (b_x, b_y)$. Po tych operacjach zarówno $H(S_1)$, jak i $H(S_2)$ znajdują się w punkcie $(0,0)$.

Jednokładność

Zdefiniujmy *promień* zbioru jako maksymalną odległość pomiędzy środkiem ciężkości zbioru i jednym z punktów zbioru. Oznaczmy promienie zbiorów S_1 i S_2 odpowiednio przez d_1 i d_2. Starając się sprowadzić zbiory do tej samej postaci należy przekształcić je przez jednokładność w ten sposób, by ich promienie były jednakowe. Aby uzyskać promień d, zbiór S_1 należy przeskalować ze współczynnikiem $\frac{d}{d_1}$, a zbiór S_2 ze współczynnikiem $\frac{d}{d_2}$. Wybór d jest dowolny (ale oczywiście $d \neq 0$).

Obrót

Ostatnią fazą jest sprawdzenie, czy dwa odpowiednio przeskalowane zbiory można sprowadzić do tej samej postaci za pomocą obrotu o środku w punkcie $(0,0)$. Problem ten, choć z natury geometryczny, przedstawimy w języku algorytmów tekstowych i rozwiążemy za pomocą algorytmu sprawdzania *cyklicznej równoważności słów*. Dwa słowa są równoważne cyklicznie, jeśli po przestawieniu pewnej liczby początkowych liter pierwszego słowa na jego koniec otrzymujemy drugie słowo. Przykładowo, słowa *aaababa* i *ababaaa* są równoważne cyklicznie, natomiast słowa *aabaaba* i *ababaaa* nie są. Istnieje prosty, choć nieoczywisty, algorytm sprawdzania równoważności cyklicznej dwóch słów w czasie liniowo zależnym od ich długości (patrz na przykład [14]).

Chcąc zastosować wspomniany algorytm, należy przedstawić zbiory punktów w postaci słów w ten sposób, aby słowa reprezentujące zbiory były równoważne cyklicznie wtedy i tylko wtedy, gdy zbiory można sprowadzić do tej samej postaci za pomocą obrotu o środku w punkcie $(0,0)$. Aby utworzyć taką reprezentację, posortujmy punkty zbioru według współrzędnych kątowych (punkty o jednakowych kątach uporządkujmy według odległości od punktu $(0,0)$). Następnie utwórzmy sekwencję liczb całkowitych: $c_1^2, -e_1^2, c_2^2, -e_2^2, \ldots, c_k^2, -e_k^2$, gdzie c_i oznacza odległość i–tego (według porządku kątowego) punktu zbioru od początku układu współrzędnych, natomiast e_i jest odległością pomiędzy i–tym, a $(i+1)$–szym punktem (e_k jest odległością pomiędzy ostatnim i pierwszym punktem). Sekwencję tę nazwiemy *obwódką*.

Zauważmy, że w obwódce punkty zbioru są opisane poprzez wartości, które nie ulegają zmianie podczas obrotu wokół punktu $(0,0)$. W związku z tym reprezentacja zbioru i jego obrazów poprzez takie obroty różni się jedynie punktem początkowym, od którego rozpoczynamy zapis sekwencji. To pozwala nam sformułować następujący lemat.

Lemat 1 *Dwa zbiory punktów można sprowadzić do tej samej postaci za pomocą obrotu o środku* $(0,0)$ *wtedy i tylko wtedy, gdy ich obwódki są słowami cyklicznie równoważnymi.*

W ten sposób ostatni test równoważności zbiorów możemy przeprowadzić w czasie liniowo zależnym od liczby punktów zbioru, stosując wspomniany algorytm testowania cyklicznej równoważności słów.

Rozwiązanie wzorcowe

Rozwiązanie wzorcowe jest realizacją opisanego powyżej schematu. Poszczególne etapy sprawdzania podobieństwa zbiorów są następujące.

1. Sprawdzamy, czy liczba punktów w porównywanych zbiorach jest różna, jeśli tak, to odpowiadamy, że zbiory nie są podobne.

2. Dla każdego ze zbiorów obliczamy jego środek ciężkości — aby zagwarantować, że punkt ten ma współrzędne całkowite (patrz dyskusja poniżej na temat zakresu liczb i dokładności obliczeń) mnożymy współrzędne wszystkich punktów przez liczność zbioru; dodajemy do zbioru jego środek ciężkości, o ile już do niego nie należy (na tym etapie sprawdzamy, czy nie występuje przypadek, w którym jeden zbiór zawiera już swój środek ciężkości, a drugi nie — w takiej sytuacji odpowiadamy, że zbiory nie są podobne).

3. Przesuwamy oba zbiory tak, by ich środki ciężkości znalazły się w punkcie $(0,0)$.

4. Obliczamy promienie zbiorów, po czym przeskalowujemy oba zbiory tak, by miały takie same promienie.

5. Tworzymy obwódki obu zbiorów. W tym celu sortujemy punkty każdego zbioru według współrzędnych kątowych (punkty o jednakowych współrzędnych kątowych — według odległości od punktu $(0,0)$) i obliczamy odpowiednie odległości pomiędzy punktami.

6. Stosujemy algorytm testowania cyklicznej równoważności słów do utworzonych obwódek; jeżeli są sobie równoważne, to odpowiadamy, że zbiory są podobne; w przeciwnym razie odpowiadamy, że nie są podobne.

7. Jeżeli powyższy test wypadł pomyślnie, to kończymy algorytm. W przeciwnym razie przekształcamy jeden ze zbiorów przez symetrię według osi OY i ponawiamy kroki od 2 do 6.

Na czas działania powyższej procedury zasadniczy wpływ ma sortowanie wykonywane w trakcie obliczania obwódki. Wszystkie pozostałe operacje potrafimy wykonać w czasie $O(k)$, zakładając, że do testowania równoważności cyklicznej zastosujemy algorytm liniowy. Stąd złożoność całego programu wykonującego opisany test dla wzoru i n zbiorów punktów wynosi $O(n \cdot k \cdot \log k)$.

Algorytm możemy zaimplementować tak, by wszystkie przeprowadzane obliczenia były całkowitoliczbowe, co gwarantuje ich dokładność. Jednak już pobieżna analiza wykonywanych operacji pokazuje, że współrzędne występujące w obliczeniach mogą być zbyt duże dla arytmetyki 64-bitowej. Początkowo współrzędne punktów należą do przedziału $[-2 \cdot 10^4, 2 \cdot 10^4]$. Podczas obliczania środków ciężkości przemnażamy wszystkie współrzędne przez k, co zwiększa ich zakres do przedziału $[-5 \cdot 10^8, 5 \cdot 10^8]$. Po przeskalowaniu wyrównującym promienie zbiorów zakres współrzędnych można ograniczyć do przedziału $[-25 \cdot 10^{16}, 25 \cdot 10^{16}]$. Kwadraty odległości między punktami występujące w obwódkach są już więc liczbami z przedziału $[0, 10^{36}]$. Wymusza to zaimplementowanie operacji arytmetycznych na liczbach 128-bitowych.

Inne rozwiązania

W zadaniu nie widać sensownej alternatywy dla podstawowej idei algorytmu polegającej na oparciu testu podobieństwa zbiorów na zrównaniu ich środków ciężkości. Mogą natomiast istnieć rozwiązania bez dodatkowej arytmetyki i z mniej efektywną procedurą sprawdzania cyklicznej równoważności słów. Testy dla zadania przechodzi na przykład program reprezentujący zmienne w postaci zmiennoprzecinkowej, w którym po przesunięciu środków ciężkości do punktu $(0,0)$ przeskalowujemy zbiory zmieniając ich promienie tak, by wynosiły 20 000. Co prawda w wykonywane obliczenia wkradają się pewne błędy zaokrągleń, ale są one na tyle małe, że program działa poprawnie na wszystkich przygotowanych testach. Również zastosowanie algorytmu naiwnego — o pesymistycznej złożoności $O(k^2)$ — do sprawdzania cyklicznej równoważności obwódek jest dla testów z zadania wystarczająco szybkie.

Niepoprawne rozwiązania

We wspomnianym wyżej algorytmie naiwnym przerywamy sprawdzanie cyklicznej równoważności obwódek natychmiast, gdy znajdziemy odpowiedź. Przy takim podejściu znaczna część sprawdzeń kończy się po porównaniu kilku elementów. Gdybyśmy zawsze kontynuowali porównywanie słów do końca, to otrzymamy procedurę działającą *zawsze* w czasie $\Theta(k^2)$. Algorytm z taką procedurę działa zbyt długo i nie przechodzi testów przy zadanych ograniczeniach czasowych.

Niepoprawne rozwiązania zadania mogą także wynikać z nieuwzględnienia sytuacji, w których środek ciężkości zbioru punktów należy do tego zbioru. Potencjalnym źródłem błędów są także przypadki, gdy w zbiorze znajduje się kilka punktów o tej samej odległości kątowej względem środka ciężkości.

Testy

Zadanie było sprawdzane na 10 testach. Pięć pierwszych testów to testy poprawnościowe. Pozostałe testy miały na celu sprawdzenie zarówno poprawności rozwiązań zawodników, jak i ich efektywności. Poniżej znajduje się krótki opis każdego z testów, zawierający liczbę punktów tworzących wzór k oraz liczbę zbiorów do zbadania n.

Nazwa	**k**	**n**
pun1.in	5	8
pun2.in	1	12
pun3.in	3	5
pun4.in	3	4
pun5.in	1 301	3

Nazwa	**k**	**n**
pun6.in	6 400	6
pun7.in	9 999	5
pun8.in	16 000	7
pun9.in	10 000	7
pun10.in	25 000	20

Krzysztof Sikora — Treść zadania, Opracowanie
Tomasz Malesiński — Program

OI, Etap I

Samochodziki

Jasio jest trzylatkiem i bardzo lubi bawić się samochodzikami. Jasio ma n różnych samochodzików. Wszystkie samochodziki leżą na wysokiej półce, do której Jasio nie dosięga. W pokoju jest mało miejsca, więc na podłodze może znajdować się jednocześnie co najwyżej k samochodzików.

Jasio bawi się jednym z samochodzików leżących na podłodze. Oprócz Jasia w pokoju cały czas przebywa mama. Gdy Jasio chce się bawić jakimś innym samochodzikiem i jeśli ten samochodzik jest na podłodze, to sięga po niego. Jeśli jednak samochodzik znajduje się na półce, to musi mu go podać mama. Mama podając Jasiowi jeden samochodzik, może przy okazji zabrać z podłogi dowolnie wybrany przez siebie inny samochodzik i odłożyć go na półkę (tak, aby na podłodze nie brakło miejsca).

Mama bardzo dobrze zna swoje dziecko i wie dokładnie, którymi samochodzikami Jasio będzie chciał się bawić. Dysponując tą wiedzą mama chce zminimalizować liczbę przypadków, gdy musi podawać Jasiowi samochodzik z półki. W tym celu musi bardzo rozważnie odkładać samochodziki na półkę.

Zadanie

Napisz program który:

- *wczyta ze standardowego wejścia ciąg kolejnych samochodzików, którymi będzie chciał się bawić Jasio,*
- *obliczy minimalną liczbę przypadków zdejmowania przez mamę samochodzików z półki,*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu standardowego wejścia znajdują się trzy liczby całkowite: n, k, p ($1 \leqslant k \leqslant n \leqslant 100\,000$, $1 \leqslant p \leqslant 500\,000$), pooddzielane pojedynczymi odstępami. Są to kolejno: łączna liczba samochodzików, liczba samochodzików mogących jednocześnie znajdować się na podłodze oraz długość ciągu samochodzików, którymi będzie chciał się bawić Jasio. W kolejnych p wierszach znajduje się po jednej liczbie całkowitej. Są to numery kolejnych samochodzików, którymi będzie chciał się bawić Jasio (samochodziki są ponumerowane od 1 do n).

Wyjście

W pierwszym i jedynym wierszu standardowego wyjścia należy zapisać jedną liczbę całkowitą — minimalną liczbę przypadków podania samochodzika z półki przez mamę.

Przykład

Dla danych wejściowych:

```
3 2 7
1
2
3
1
3
1
2
```

poprawnym wynikiem jest:

```
4
```

Rozwiązanie

Wprowadzenie

Problem przedstawiony w zadaniu jest związany z problemem realizacji pamięci wirtualnej oraz zagadnieniem stronicowania występującymi w systemach operacyjnych ([32], rozdział 9.3). Rozkazy wykonywane przez procesor oraz dane, na których on operuje, muszą znajdować się w pamięci operacyjnej. Jej wielkość jest jednak ograniczona i dlatego część danych jest przechowywana w wolniejszej pamięci zewnętrznej (na przykład na dysku). Dane w pamięci operacyjnej i na dysku podzielone są na porcje zwane stronami. Zawsze, gdy procesor potrzebuje strony, której nie ma aktualnie w pamięci operacyjnej, występuje *błąd braku strony*. System operacyjny musi wówczas ściągnąć potrzebną stronę z dysku do pamięci. Może jednak zdarzyć się, że w pamięci nie ma już wolnego miejsca na nową stronę. Wówczas najpierw trzeba przenieść inną, chwilowo niepotrzebną stronę na dysk. Ponieważ obsługa błędów braku strony jest kosztowna czasowo, zależy nam na zminimalizowaniu liczby ich wystąpień. Podstawową metodą pozwalającą osiągnąć ten cel jest odpowiednia strategia doboru stron, które przenosimy na dysk robiąc miejsce w pamięci operacyjnej.

W zadaniu, Jasio odgrywa rolę procesora, mama zaś spełnia zadania systemu operacyjnego. Odpowiednikiem ograniczonej pamięci operacyjnej jest podłoga, natomiast bardziej pojemną pamięć zewnętrzną reprezentuje półka. Wreszcie samochodziki, którymi bawi się chłopiec, to strony pamięci z potrzebnymi danymi. Podstawowa różnica pomiędzy historyjką przedstawioną w zadaniu, a rzeczywistą sytuacją polega na tym, że system operacyjny nie jest w stanie przewidzieć przyszłych potrzeb procesora tak dobrze, jak mama potrafi przewidzieć zachcianki Jasia. W praktycznych zastosowaniach algorytm *off–line*, który musi mieć na wejściu zadany z góry ciąg odwołań do stron, jest właściwie bezużyteczny. Jedynymi praktycznymi rozwiązaniami są algorytmy typu *on–line* wyznaczające strategię sprowadzania i usuwania stron z pamięci (stronicowania) na podstawie informacji o wykorzystaniu stron w przeszłości. Strategie *off–line* są stosowane tylko do sprawdzenia jakości innych algorytmów, gdyż jak się wkrótce przekonamy, istnieje *optymalny* algorytm stronicowania *off–line*, który powoduje minimalną liczbę błędów braku strony.

Rozwiązanie wzorcowe

Wprowadźmy następujące oznaczenia:

- n — liczba wszystkich samochodzików,
- k — liczba samochodzików mogących znajdować się jednocześnie na podłodze,
- p — długość ciągu samochodzików, którymi będzie chciał się bawić Jasio,
- $S(t)$ — samochodzik, którym Jasio chce się bawić w chwili t ($t = 1, \ldots, p$);
- $P(t)$ — zbiór samochodzików przygotowanych dla Jasia do zabawy w chwili t (w tym celu mama być może musi odłożyć na półkę jakiś samochodzik i zdjąć z półki inny — wykonuje to wszystko pomiędzy chwilą $t-1$ a chwilą t, w dalszej części będziemy pisać, że mama wykonuje te czynności *tuż przed chwilą* t).

Rodzina zbiorów $P(t)$ dla $t = 0, \ldots, n$ dokładnie opisuje czynności wykonywane przez mamę w trakcie zabawy Jasia (ze zbioru $P(t) \setminus P(t-1)$ można wywnioskować, który samochodzik został zabrany, a który podany przez mamę tuż przed chwilą t).

Definicja 1 Rodzinę zbiorów P nazwiemy *strategią*, a mówiąc algorytm P, będziemy mieć na myśli algorytm, który usuwa samochodziki z podłogi zgodnie ze strategią P.

Strategia P jest *poprawna*, jeśli spełnia następujące warunki:

1. $P(0) = \emptyset$,
2. $S(t) \in P(t)$ dla $t = 1, \ldots, p$,
3. $|P(t)| \leqslant k$ dla $t = 1, \ldots, p$.

Strategia P jest *optymalna*, jeśli jest poprawna i nie istnieje inna strategia, w której mama wykonuje mniej operacji zdjęcia samochodzika z półki.

Fakt 1 *Jeśli w pewnej strategii optymalnej P mama zabiera z podłogi na półkę samochodzik s tuż przed chwilą t oraz $|P(t)| < k$, to strategia P', różniąca się tylko tym, że mama nie zabiera tuż przed chwilą t z podłogi samochodzika s, jest nadal optymalna.*

Zatem możemy poszukiwać strategii optymalnej tylko pośród tych, w których mama nie odkłada na półkę samochodzika, dopóki na podłodze jest miejsce na kolejny. Pozostaje ustalić, który samochodzik mama ma zabrać z podłogi, gdy wystąpi taka konieczność. Okazuje się, że wystarczy, jeśli będzie to samochodzik, którym chłopiec najdłużej nie będzie chciał się bawić, spośród tych, które obecnie znajdują się na podłodze.

Twierdzenie 2 *Strategia OPT, która polega na odkładaniu na półkę samochodzika, o który Jasio poprosi ponownie najpóźniej, jest optymalna.*

Dowód Pokażemy, że mając daną dowolną strategię optymalną P, możemy skonstruować strategię OPT powodującą tyle samo błędów „braku samochodzika”, co P, w której zawsze usuwany jest z podłogi samochodzik, o który Jaś poprosi ponownie najpóźniej.

Przypuśćmy, że w strategii P powyższa zasada nie jest przestrzegana i t_1 jest pierwszą chwilą, w której na półkę jest odkładany samochodzik s inny niż ten, którym Jasio najdłużej nie będzie się bawił (oznaczmy go s'). Rozważmy strategię P' równą P z dokładnością do s i s', to znaczy:

1. $P'(t) = P(t)$ dla $t < t_1$;
2. tuż przed chwilą t_1 w strategii P' odkładamy na półkę samochodzik s', a pozostawiamy s (w strategii P odkładamy s i zatrzymujemy s');
3. przez wszystkie kolejne chwile t, dopóki chłopiec nie zechce bawić się s, a mama w strategii P nie decyduje się odstawić na półkę samochodzika s', obie strategie mogą działać identycznie dla samochodzików innych niż s i s', tzn. $P'(t) = P(t) \setminus \{s'\} \cup \{s\}$.

W strategii P przychodzi w końcu chwila (oznaczmy ją t_2), że chłopiec chce się bawić samochodzikiem s. Może także zdarzyć się, że tuż przed pewną chwilą (oznaczmy ją t_3) mama decyduje się odstawić s' na półkę.

Przypadek 1: Przypuśćmy, że $t_3 < t_2$. Wówczas definiujemy strategię P' tak, by tuż przed chwilą t_3 mama odstawiała na półkę samochodzik s. W dalszej części obie strategie są identyczne.

Przypadek 2a: Niech $t_2 < t_3$ i w strategii P mama usuwa z podłogi tuż przed chwilą t_2 samochodzik $s'' \neq s'$, by zrobić miejsce dla s. Wówczas w strategii P' tuż przed chwilą t_2 usuwamy z podłogi s'' i zestawiamy z półki s'. Dalej obie strategie przebiegają identycznie.

Przypadek 2b: Niech $t_2 < t_3$ i w strategii P mama usuwa z podłogi tuż przed chwilą t_2 samochodzik s', by zrobić miejsce dla s. Wówczas w strategii P' nie musimy dokonywać żadnej zamiany tuż przed chwilą t_2 i począwszy od tej chwili mamy sytuację identyczną jak w P. Osiągamy ją jednak wykonując jedną zamianę mniej, co stoi w sprzeczności z założeniem, że P była optymalna. Opisany przypadek nie może więc wystąpić.

Zastanówmy się, co zyskaliśmy konstruując strategię P'. Otrzymaliśmy strategię nadal optymalną, bo wymagającą takiej samej liczby zamian co P, ale zachowującą dłużej zasadę odstawiania na półkę samochodzika, którym chłopiec będzie chciał się bawić najpóźniej. Jeżeli w P' zasada ta jest nadal łamana (w jakimś późniejszym momencie niż w P), to możemy przekształcić tym razem P' w kolejną optymalną strategię odsuwając jeszcze dalej w czasie złamanie zasady. W ten sposób w skończonej liczbie kroków otrzymujemy strategię *OPT* (optymalną, bo wymagającą tyle samo zamian, co wyjściowa strategia P), w której w każdej chwili jest przestrzegana omawiana zasada. To kończy dowód twierdzenia.

■

Implementacja

Kluczowym elementem implementacji jest wybór struktury danych umożliwiającej wydajne sprawdzenie, który samochodzik należy usunąć z podłogi. W rozwiązaniu wzorcowym użyto kopca binarnego. Znajdują się w nim numery samochodzików będących aktualnie na podłodze, uporządkowane malejąco według czasów następnego żądania Jasia (na szczycie znajduje się auto, którym chłopiec najdłużej nie będzie się bawił). Dzięki takiemu wyborowi struktury danych, operacje wyznaczenia samochodzika do odłożenia na półkę oraz dołączenia nowego — podanego Jasiowi — do struktury można wykonać w czasie $O(\log k)$.

Dodatkowo dla każdego samochodzika utrzymywany jest stos zawierający czasy jego żądań uporządkowanych rosnąco (na szczycie stosu znajduje się czas następnego żądania). Pozwala to w czasie stałym wyznaczyć czas kolejnego żądania wykorzystywany podczas operacji na kopcu. Poniższy program przedstawia pseudokod algorytmu wzorcowego:

```
{ zainicjowanie zmiennej zliczającej }
{ liczbę operacji zdjęcia samochodzika z półki }
wynik:=0;
{ zainicjowanie pustego kopca }
HeapInit(h);
{ zainicjowanie stosów dla każdego samochodzika }
for i:=1 to n do
  begin
    { pusty stos }
    StackInit(Pos[i]);
    { element symbolizujący nieskończoność }
    Push(Pos[i], p+1);
  end
for t:=p downto 1 do
  begin
    { budowa stosów czasów żądań samochodzików }
    Push(Pos[S(t)], t);
  end
for t:=1 to p do
  begin
    Pop(Pos[S(t)]);
    if S(t) nie należy do kopca h then
      begin
        { nie ma miejsca na podłodze }
        if Size(h) = k then
          begin
        { usuń numer samochodu, którego Jaś zażąda ponownie najpóźniej }
            HeapDeleteMax(h);
          end
        { dodaj do kopca element S(t) z czasem Top(Pos[S(t)]) }
        HeapInsert(h,S(t));
        wynik:=wynik+1;
      end
    else
      begin
        { zwiększ wartość elementu S(t) na czas Top(Pos[S(t)]) }
        HeapIncreaseKey(h,S(t));
      end
  end
```

Łatwo, można zauważyć, że algorytm działa w czasie $O(n+p\log k)$. Złożoność pamięciowa wynosi $O(p+n+k)$.

Testy

Zadanie testowane było na zestawie 10 danych testowych.

Nazwa	n	k	p	Opis
sam1.in	10	3	20	prosty test poprawnościowy
sam2.in	100	30	500	test losowy
sam3.in	500	200	2 000	test losowy
sam4.in	1 000	200	20 000	test z wolno zmieniającym się zbiorem
sam5.in	5 000	2 000	40 000	test losowy
sam6.in	10 000	3 000	80 000	test losowy
sam7.in	20 000	1 000	100 000	test z wolno zmieniającym się zbiorem
sam8.in	50 000	20 000	300 000	test losowy
sam9.in	100 000	50 000	500 000	test losowy
sam10.in	100 000	80 000	500 000	test losowy

Wojciech Guzicki
Treść zadania, Opracowanie

Paweł Wolff
Program

Skarbonki

Smok Bajtazar ma n skarbonek. Każdą skarbonkę można otworzyć jej kluczem lub rozbić młotkiem. Bajtazar powrzucał klucze do pewnych skarbonek, pamięta przy tym który do której. Bajtazar zamierza kupić samochód i musi dostać się do wszystkich skarbonek. Chce jednak zniszczyć jak najmniej z nich. Pomóż Bajtazarowi ustalić, ile skarbonek musi rozbić.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia liczbę skarbonek i rozmieszczenie odpowiadających im kluczy,*
- *obliczy minimalną liczbę skarbonek, które trzeba rozbić, aby dostać się do wszystkich skarbonek,*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita n ($1 \leqslant n \leqslant 1\,000\,000$) — tyle skarbonek posiada smok. Skarbonki (jak również odpowiadające im klucze) są ponumerowane od 1 do n. Dalej na wejściu mamy n wierszy: w $(i+1)$-szym wierszu zapisana jest jedna liczba całkowita — numer skarbonki, w której znajduje się i-ty klucz.

Wyjście

W pierwszym i jedynym wierszu standardowego wyjścia należy zapisać jedną liczbę całkowitą — minimalną liczbę skarbonek, które trzeba rozbić, aby dostać się do wszystkich skarbonek.

Przykład

Dla danych wejściowych:

```
4
2
1
2
4
```

poprawnym wynikiem jest:

```
2
```

W powyższym przykładzie wystarczy rozbić skarbonki numer 1 i 4.

Rozwiązanie

Interpretacja grafowa

Zadanie ma naturalną interpretację w języku teorii grafów. Utwórzmy graf skierowany G mający n wierzchołków ponumerowanych liczbami od 1 do n. Wierzchołki o numerach i i j są połączone krawędzią (biegnącą od i do j), jeśli klucz do skarbonki i znajduje się w skarbonce j. Zauważmy, że z każdego wierzchołka wychodzi dokładnie jedna krawędź; stąd wynika, że w grafie jest dokładnie n krawędzi. Oczywiście do jednego wierzchołka może wchodzić wiele krawędzi, w takim przypadku w grafie będą także wierzchołki, do których nie wchodzi żadna krawędź. Jeśli do jednego wierzchołka wchodzi wiele krawędzi, to znaczy, że w odpowiedniej skarbonce znajduje się wiele kluczy. Jeśli zaś do jakiegoś wierzchołka nie wchodzi żadna krawędź, to znaczy, że w skarbonce o numerze równym numerowi tego wierzchołka nie ma żadnego klucza.

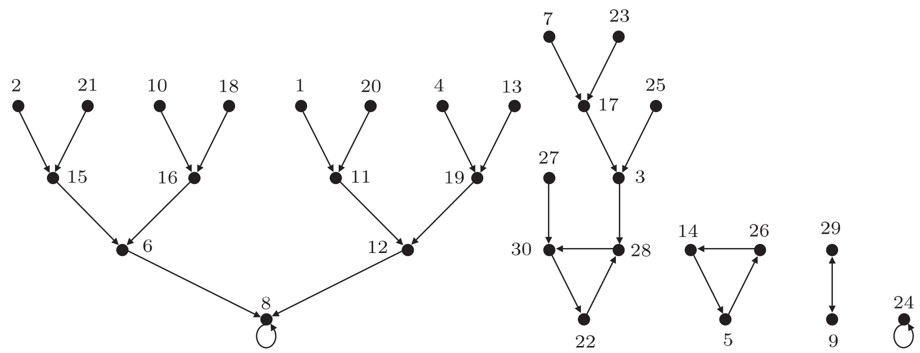
Przykład Popatrzmy na przykład takiego grafu. Przypuśćmy, że $n = 30$ oraz w pliku wejściowym (po liczbie 30) znajdują się następujące liczby:

$$11, 15, 28, 19, 26, 8, 17, 8, 29, 16, 12, 8, 19, 5, 6, 6, 3, 16, 12, 11, 15, 28, 17, 24, 3, 14, 30, 30, 9, 22.$$

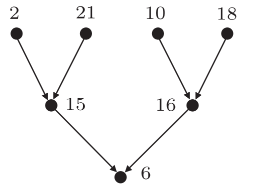
Inaczej mówiąc, w grafie G znajdują się następujące krawędzie:

$1 \to 11$	$2 \to 15$	$3 \to 28$	$4 \to 19$	$5 \to 26$
$6 \to 8$	$7 \to 17$	$8 \to 8$	$9 \to 29$	$10 \to 16$
$11 \to 12$	$12 \to 8$	$13 \to 19$	$14 \to 5$	$15 \to 6$
$16 \to 6$	$17 \to 3$	$18 \to 16$	$19 \to 12$	$20 \to 11$
$21 \to 15$	$22 \to 28$	$23 \to 17$	$24 \to 24$	$25 \to 3$
$26 \to 14$	$27 \to 30$	$28 \to 30$	$29 \to 9$	$30 \to 22$

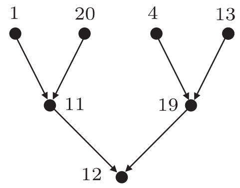
A oto rysunek tego grafu:



Zauważmy, że przedstawiony graf ma 5 składowych (czyli, mówiąc językiem potocznym, składa się z pięciu oddzielnych kawałków) i każda z tych składowych ma postać cyklu (być może długości 1), do którego dochodzą skierowane drzewa (czyli grafy bez cykli). W pierwszej składowej mamy cykl jednoelementowy złożony z wierzchołka o numerze 8, do którego dochodzą dwa drzewa binarne:



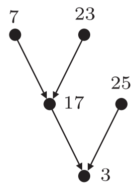
oraz



W drugiej składowej mamy cykl długości 3:

$$22 \to 28 \to 30 \to 22,$$

do którego dochodzą dwa drzewa. Jedno z nich składa się z jednego wierzchołka o numerze 27 i jest dołączone do wierzchołka o numerze 30; drugie zaś ma postać:



i jest dołączone do wierzchołka o numerze 28. Trzecia, czwarta i piąta składowa składają się z samych cykli, odpowiednio o długości 3, 2 i 1:

$$5 \to 26 \to 14 \to 5, \quad 9 \to 29 \to 9 \quad \text{oraz} \quad 24 \to 24.$$

■ *Przykład*

Graf G, w którym, jak w treści zadania, z każdego wierzchołka wychodzi dokładnie jedna krawędź, ma zawsze podobną postać: każda składowa ma jeden cykl (być może jednoelementowy), do którego są dołączone drzewa (być może zero drzew).

Zastanówmy się, dlaczego tak jest? Wybierzmy z grafu dowolny wierzchołek v i startując z niego poruszajmy się idąc wzdłuż krawędzi. Zauważmy, że nasza droga jest wyznaczona jednoznacznie, gdyż z każdego wierzchołka musimy wyjść jedyną możliwą krawędzią. Z drugiej strony, z tego samego powodu, możemy w opisany sposób poruszać się bez końca — zawsze mamy krawędź wychodzącą z wierzchołka. Zapisując kolejno odwiedzone wierzchołki otrzymujemy ciąg:

$$v = v_0 \to v_1 \to v_2 \to \dots$$

Oczywiście któryś wierzchołek musi się w tym ciągu powtórzyć. Niech pierwszym takim wierzchołkiem będzie wierzchołek v_k. Mamy wtedy następujący ciąg, cykliczny od wyrazu v_k:

$$v = v_0 \to v_1 \to v_2 \to \dots \to v_{k-1} \to v_k \to v_{k+1} \to \dots \to v_{k+l-1} \to v_{k+l} = v_k \to v_{k+1} \to \dots$$

Zatem z każdego wierzchołka dochodzimy do pewnego cyklu. Łatwo spostrzec, że cykle te są rozłączne; wynika to ponownie stąd, że z każdego wierzchołka wychodzi dokładnie jedna krawędź. Wreszcie zauważmy (nietrudny dowód zostawimy jako ćwiczenie), że w każdej składowej z każdego wierzchołka dochodzimy do tego samego cyklu.

Oczywiście rozbicie którejkolwiek skarbonki odpowiadającej wierzchołkowi w cyklu pozwala otworzyć wszystkie skarbonki odpowiadające wierzchołkom składowej, w której ten cykl się znajduje. Oznacza to, że zadanie sprowadza się do policzenia składowych grafu.

Zliczanie składowych

Istnieje wiele metod znajdowania liczby składowych grafu. Wspomnimy najpierw o dwóch.

Jedna metoda polega na przeszukiwaniu grafu (w głąb lub wszerz). Wierzchołki, do których można dotrzeć z jednego wierzchołka poruszając się po krawędziach w dowolnym kierunku (także „pod prąd"), tworzą jedną składową.

Innym algorytmem jest tzw. algorytm *find - union*, polegający na sukcesywnym łączeniu w coraz większe zbiory wierzchołków połączonych krawędziami i należących do jednej składowej. Opis tego algorytmu Czytelnik może znaleźć na przykład w [17] (rozdział 22).

Jeszcze inną metodę znajdziemy w algorytmie stosowanym do tzw. sortowania topologicznego. Polega ona na usuwaniu z grafu kolejno wierzchołków, do których nie wchodzi żadna krawędź; w ten sposób każda składowa grafu zostaje zredukowana do cyklu, a cykle możemy już łatwo zliczyć. Opiszemy tę procedurę dokładniej. Przypuśćmy, że mamy dane dwie tablice:

Gdzie, *Ile* : **array**[1..*n*] **of longint**;

Tablica `Gdzie` wskazuje, w której skarbonce znajduje się dany klucz; czyli jeśli `Gdzie[5]=7`, to klucz 5 znajduje się w skarbonce 7, a więc w grafie G mamy krawędź $5 \rightarrow 7$. Tablica `Ile` wskazuje, ile krawędzi wchodzi do danego wierzchołka. Obie tablice wypełniamy początkowo na podstawie danych wczytanych z pliku wejściowego (szczegóły pominiemy).

Teraz przeglądamy kolejno wierzchołki i usuwamy z grafu te, do których nie wchodzi żadna krawędź. Usunięcie wierzchołka zaznaczamy wpisując 0 na odpowiedniej pozycji w tablicy `Gdzie`. A oto procedura usuwania wierzchołków:

```
procedure Usun;
  var
    i,j,k: longint;
  begin
    for i:=1 to n do
      if (Ile[i]=0) and (Gdzie[i]<>0) then
        begin
          j:=i;
          repeat
            k:=Gdzie[j];
            Gdzie[j]:=0;
            j:=k;
            Dec(Ile[j]);
          until (Ile[j]<>0)
        end
  end;
```

Po wykonaniu powyższej procedury w grafie G pozostaną same cykle. Tworzące je wierzchołki rozpoznajemy po tym, że odpowiadająca im wartość w tablicy `Ile` jest niezerowa. Cykle zliczamy za pomocą następującej procedury:

```
procedure ZliczCykle;
  var
    i,j : longint;
  begin
    Liczba:=0;
    for i:=1 to n do
      if Ile[i]<>0 then
        begin
          j:=Gdzie[i];
          while j<>i do
            begin
              Ile[j]:=0;
              j:=Gdzie[j]
            end;
          Inc(Liczba)
        end
  end;
```

Po zakończeniu procedury zmienna globalna `Liczba` zawiera szukaną liczbę cykli.

Zliczanie składowych — inny sposób

Chwila zastanowienia pokazuje, że składowe w grafie G możemy także zliczyć nie usuwając wierzchołków (i nie korzystając z tablicy `Ile`). W tym celu tworzymy tablicę:

$$Numer : \textbf{array}[1..n] \textbf{ of longint}$$

Początkowo wypełniamy ją zerami. Następnie dla wierzchołka o numerze i poruszamy się po grafie zgodnie z kierunkiem krawędzi (jak wskazuje tablica `Gdzie`) i wpisujmy liczbę i w każde wolne (tzn. zawierające 0) miejsce w tablicy `Numer` odpowiadające numerowi odwiedzanego wierzchołka. To postępowanie może zakończyć się dwoma sposobami. Albo natkniemy się na wierzchołek, któremu już przypisaliśmy liczbę i, albo natkniemy się na wierzchołek, któremu była wcześniej przypisana jakaś liczba j różna od i. Pierwszy przypadek oznacza, że na naszej drodze natknęliśmy się na nowy cykl; zwiększamy więc liczbę cykli o 1. Drugi przypadek oznacza, że doszliśmy do miejsca, przez które już przechodziliśmy i z którego w dalszym ciągu doszlibyśmy do cyklu znalezionego wcześniej — takiej drogi nie musimy dalej kontynuować.

Przykład (cd.) Prześledźmy przedstawione postępowanie na przykładzie grafu G opisanego na początku tego rozdziału. Po zainicjowaniu tablicy `Gdzie` rozważamy kolejno wierzchołki $i = 1, 2, \ldots$ Dla $i = 1$ wpisujemy liczbę 1 w następujące miejsca tablicy `Numer`:

$$1, 11, 12, 8.$$

Następnie dla $i = 2$ liczbę 2 wpiszemy w miejsca:

$$2, 15, 6$$

i stwierdzimy, że następny wierzchołek (o numerze 8) był już odwiedzony, a cykl, do którego należy, już policzony. Liczbę 3 wpiszemy następnie w miejsca:

$$3, 28, 30, 22$$

i stwierdzimy, że znaleźliśmy nowy cykl. Liczbę 4 wpiszemy w miejsca 4 i 19 i stwierdzimy, że następny wierzchołek (o numerze 12) był już znaleziony. Liczba 5 doprowadzi nas do nowego cyklu:

$$5, 14, 26.$$

Wierzchołek 6 był już odwiedzony. Z wierzchołka 7 dojdziemy do odwiedzonego wcześniej wierzchołka 3 i tak dalej. ■ *Przykład (cd.)*

A oto algorytm oparty na opisanym pomyśle.

```
procedure Zlicz;
  var
    i,j : longint;
  begin
    Liczba:=0;
    for i:=1 to n do Numer[i]:=0;
    for i:=1 to n do
      begin
        j:=i;
        while Numer[j]=0 do
          begin
            Numer[j]:=i;
            j:=Gdzie[j]
          end;
        if Numer[j]=i then Inc(Liczba)
      end
  end
```

Podobnie jak poprzednio na zakończenie liczba cykli jest zapisana w zmiennej `Liczba`.

Przedstawione rozwiązanie zostało zaimplementowane w programie wzorcowym i działa w czasie $O(n)$.

Testy

Do zadania opracowano 16 testów. Początkowe testy służą sprawdzeniu poprawności rozwiązania w pewnych typowych sytuacjach (cykle jednoelementowe, dwuelementowe, cykle z dołączonymi drzewami, cykle bez dołączonych wierzchołków itp.). Następne testy służą sprawdzeniu szybkości działania programu. W ostatnich testach znaczna część grafu jest wygenerowana w sposób losowy.

Poniżej zamieszczamy szczegółowy opis testów:

Nazwa	n	Opis
ska1.in	32	mały test poprawnościowy: 12 wierzchołków tworzy 3 spójne składowe, reszta stanowi jednoelementowe spójne składowe (aby odpowiedź nie była zbyt małą liczbą, którą można z dużym prawdopodobieństwem zgadnąć na „chybił-trafił")
ska2a.in	1	najmniejszy możliwy test
ska2b.in	11	mały test poprawnościowy: jeden cykl 3 elementowy, 8 cykli jednoelementowych
ska3.in	28	mały test poprawnościowy: drzewo binarne, którego korzeń jest jednoelementowym cyklem i dodatkowo 10 małych cykli
ska4.in	31	mały test poprawnościowy: kilka ścieżek zakończonych pętlami (cyklami jednoelementowymi), jeden cykl dwuelementowy i kilka cykli jednoelementowych
ska5.in	36	mały test poprawnościowy: cykl 5-elementowy z dołączonymi do niego ścieżkami, jedna ścieżka z pętlą na końcu plus kilka cykli jednoelementowych
ska6.in	2 187	średni test: cykl długości 37 z dołączonymi drzewami rozmiaru 50 plus część losowa grafu (stanowiąca odrębne składowe) rozmiaru 300
ska7.in	34 507	średni test: cykl długości 7 z dołączonymi drzewami rozmiaru 3 500 plus część losowa rozmiaru 10 000
ska8.in	455 091	duży test: 13 cykli długości od 1 do 13 z dołączonymi ścieżkami długości 5 000
ska9.in	20 100	średni test: 100 cykli długości od 1 do 100 z dołączonymi ścieżkami długości 1, oprócz jednej, która jest długości 10 001
ska10.in	300 100	duży test: 3 cykle długości 100 000 plus część losowa rozmiaru 100
ska11.in	701 000	duży test: 1 000 cykli długości 700 plus część losowa rozmiaru 1 000
ska12.in	50 000	średni test losowy
ska13.in	200 000	duży test losowy
ska14.in	500 000	duży test losowy
ska15.in	1 000 000	duży test: 990 000 cykli jednoelementowych plus część losowa rozmiaru 10 000

Wojciech Guzicki — Treść zadania, Opracowanie
Marcin Mucha — Program

Skoczki

Skoczek porusza się po nieskończonej szachownicy. Każdy z ruchów, jakie może wykonać skoczek, można opisać parą liczb całkowitych — para (a,b) odpowiada możliwości wykonania ruchu z pola o współrzędnych (x,y) na pole $(x+a,y+b)$ lub $(x-a,y-b)$. Dla każdego skoczka określony jest zestaw takich par, opisujących ruchy jakie może wykonywać skoczek. Zakładamy, że pola, na które może ruszyć się skoczek z pola $(0,0)$ nie leżą wszystkie na jednej prostej.

Powiemy, że dwa skoczki są równoważne, jeżeli dla obu skoczków zbiory pól, do jakich mogą dotrzeć z pola $(0,0)$ (być może w wielu ruchach) są takie same. (Przy czym równoważne skoczki mogą docierać do tych pól w różnych liczbach kroków). Można pokazać, że dla każdego skoczka istnieje równoważny mu skoczek, którego ruchy są opisane za pomocą tylko dwóch par liczb.

Zadanie

Twoje zadanie polega na napisaniu programu, który:

- *wczyta ze standardowego wejścia pary liczb całkowitych opisujące ruchy skoczka,*
- *wyznaczy dwie pary liczb całkowitych opisujące skoczka równoważnego danemu skoczkowi,*
- *wypisze wyznaczone dwie pary liczb na standardowe wyjście.*

Wejście

W pierwszym wierszu standardowego wejścia zapisana jest jedna liczba całkowita n, liczba par liczb opisujących ruchy danego skoczka, $3 \leqslant n \leqslant 100$. W kolejnych n wierszach zapisane są pary liczb opisujące ruchy danego skoczka, po jednej w wierszu. W każdym z tych wierszy zapisane są po dwie liczby całkowite a_i i b_i oddzielone pojedynczym odstępem, $-100 \leqslant a_i,b_i \leqslant 100$. Zakładamy, że $(a_i,b_i) \neq (0,0)$.

Wyjście

W pierwszym wierszu standardowego wyjścia należy wypisać dwie liczby całkowite a i b oddzielone pojedynczym odstępem, $-10\,000 \leqslant a,b \leqslant 10\,000$. W drugim wierszu należy zapisać dwie liczby całkowite c i d oddzielone pojedynczym odstępem, $-10\,000 \leqslant c,d \leqslant 10\,000$. Powinny to być takie liczby całkowite, że skoczek, którego ruchy są opisane parami (a,b) i (c,d) jest równoważny skoczkowi opisanemu w danych wejściowych.

Przykład

Dla danych wejściowych:

```
3
24 28
15 50
12 21
```

poprawne wyjście może mieć np. postać:

```
468 1561
2805 9356
```

lub:

```
3 0
0 1
```

Rozwiązanie

Wprowadzenie – kraty na płaszczyźnie

Problem przedstawiony w zadaniu sprowadza się do zagadnienia znajdowania bazy kraty na płaszczyźnie. Pojęcie kraty jest omówione dokładnie w [11] (zadanie „Tomki”, str. 217 – 222). Tutaj je tylko krótko przypomnimy.

Definicja 1 Niech $\overrightarrow{a}_1, \ldots, \overrightarrow{a}_n$ będą dowolnymi niezerowymi wektorami na płaszczyźnie. Zbiór $L(\overrightarrow{a}_1, \ldots, \overrightarrow{a}_n)$ zdefiniowany w następujący sposób:

$$L(\overrightarrow{a}_1, \ldots, \overrightarrow{a}_n) = \left\{ m_1 \cdot \overrightarrow{a}_1 + \ldots + m_n \cdot \overrightarrow{a}_n : \ m_1, \ldots, m_n \in \mathbb{Z} \right\}$$

nazywamy *kratą* rozpiętą przez wektory $\overrightarrow{a}_1, \ldots, \overrightarrow{a}_n$. Jeśli $\overrightarrow{c} = m_1 \cdot \overrightarrow{a}_1 + \ldots + m_n \cdot \overrightarrow{a}_n$, to wektor $\overrightarrow{c}$ nazywamy *kombinacją liniową* wektorów $\overrightarrow{a}_1, \ldots, \overrightarrow{a}_n$, a liczby $m_1, \ldots, m_n$ nazywamy *współczynnikami* tej kombinacji. Krata rozpięta przez n wektorów składa się zatem ze wszystkich kombinacji liniowych tych n wektorów, przy czym wszystkie współczynniki tych kombinacji są liczbami całkowitymi.

W dalszym ciągu będziemy korzystać z następującego prostego warunku równości dwóch krat. Przypuśćmy, że mamy dwie kraty $L(\overrightarrow{a}_1, \ldots, \overrightarrow{a}_m)$ oraz $L(\overrightarrow{b}_1, \ldots, \overrightarrow{b}_n)$. Wówczas jeśli

$$\overrightarrow{a}_1, \ldots, \overrightarrow{a}_m \in L(\overrightarrow{b}_1, \ldots, \overrightarrow{b}_n) \quad \text{oraz} \quad \overrightarrow{b}_1, \ldots, \overrightarrow{b}_n \in L(\overrightarrow{a}_1, \ldots, \overrightarrow{a}_m),$$

to

$$L(\overrightarrow{a}_1, \ldots, \overrightarrow{a}_m) = L(\overrightarrow{b}_1, \ldots, \overrightarrow{b}_n).$$

Definicja 2 Wektory $\overrightarrow{a}_1, \ldots, \overrightarrow{a}_n$ nazywamy wektorami *liniowo niezależnymi*, jeśli dla dowolnych liczb rzeczywistych $t_1, \ldots, t_n$ wektor

$$t_1 \cdot \overrightarrow{a}_1 + \ldots + t_n \cdot \overrightarrow{a}_n$$

jest wektorem zerowym tylko wówczas, gdy wszystkie liczby $t_1, \ldots, t_n$ są zerami. Wektory nazywamy *liniowo zależnymi*, jeśli nie są liniowo niezależne. Jeśli wektory generujące kratę są liniowo niezależne, to mówimy, że tworzą one *bazę* tej kraty.

Można udowodnić, że na płaszczyźnie każdy zbiór wektorów liniowo niezależnych składa się z co najwyżej dwóch wektorów. Ponadto dwa wektory na płaszczyźnie są liniowo zależne wtedy i tylko wtedy, gdy są równoległe. Z tego wynika, że jeśli wszystkie wektory kraty są równoległe, to baza tej kraty składa się z tylko jednego wektora. Jeśli zaś w kracie istnieją dwa wektory nierównoległe, to baza tej kraty musi składać się z dokładnie dwóch wektorów. Nasze zadanie polega zatem na znalezieniu co najwyżej dwuelementowej bazy kraty rozpiętej przez n wektorów, z których każdy ma obie współrzędne całkowite.

Rozwiązanie pierwsze

Przedstawimy dwa rozwiązania naszego zadania — jako pierwsze opiszemy bardziej ogólne (patrz porównanie rozwiązań w rozdziale **Uwagi końcowe**). Schemat postępowania w obu rozwiązaniach będzie taki sam. Pokażemy jak wejściowy zbiór wektorów $\overrightarrow{a}_1, \overrightarrow{a}_2, \ldots, \overrightarrow{a}_n$ można redukować krok po kroku zmniejszając jego moc, by na końcu otrzymać bazę złożoną z dwóch wektorów (bądź z jednego wektora, gdy zbiór wejściowy składa się z wektorów równoległych). Podstawowym krokiem algorytmu będzie zastąpienie trójelementowego (odpowiednio dwuelementowego) zbioru wektorów generującego kratę przez zbiór dwuelementowy (odpowiednio jednoelementowy) generujący tę samą kratę.

Na początku pokażemy kilka własności kombinacji liniowych wektorów, w których dopuszczamy *współczynniki wymierne* kombinacji. Niech $\overrightarrow{a} = [x_a, y_a]$ i $\overrightarrow{b} = [x_b, y_b]$ będą dwoma niezerowymi wektorami o współrzędnych całkowitych. W zależności od ich wzajemnego położenia zachodzą następujące fakty.

Fakt 1 *Niech $\overrightarrow{a}$ i $\overrightarrow{b}$ będą wektorami równoległymi. Wtedy istnieje liczba wymierna t taka, że $\overrightarrow{b} = t \cdot \overrightarrow{a}$.*

Dowód Przypuśćmy, że $x_a \neq 0$. Przyjmijmy

$$t = \frac{x_b}{x_a}.$$

Ponieważ wektory $\overrightarrow{a}$ i $\overrightarrow{b}$ są równoległe, więc

$$\begin{vmatrix} x_a & x_b \\ y_a & y_b \end{vmatrix} = 0,$$

czyli $x_a \cdot y_b = x_b \cdot y_a$. Stąd wynika, że $y_a = \frac{x_a}{x_b} \cdot y_b$, a więc

$$t \cdot \overrightarrow{a} = \left[x_a \cdot \frac{x_b}{x_a}, y_a \cdot \frac{x_b}{x_a} \right] = [x_b, y_b] = \overrightarrow{b}.$$

Oczywiście liczba t jest wymierna.

■

Fakt 2 *Załóżmy, że wektory $\overrightarrow{a}$ i $\overrightarrow{b}$ nie są równoległe. Wtedy dla dowolnego wektora $\overrightarrow{c}$ o współrzędnych całkowitych istnieją liczby wymierne t i u takie, że*

$$\overrightarrow{c} = t \cdot \overrightarrow{a} + u \cdot \overrightarrow{b}.$$

Dowód Oznaczmy $\overrightarrow{c} = [x_c, y_c]$. Poszukiwane liczby t i u są rozwiązaniem układu równań

$$\begin{cases} x_a \cdot t + x_b \cdot u &= x_c \\ y_a \cdot t + y_b \cdot u &= y_c \end{cases},$$

czyli

$$t = \frac{\begin{vmatrix} x_c & x_b \\ y_c & y_b \end{vmatrix}}{\begin{vmatrix} x_a & x_b \\ y_a & y_b \end{vmatrix}} \quad \text{oraz} \quad u = \frac{\begin{vmatrix} x_a & x_c \\ y_a & y_c \end{vmatrix}}{\begin{vmatrix} x_a & x_b \\ y_a & y_b \end{vmatrix}}.$$

Oczywiście liczby t i u są wymierne, a więc mamy poszukiwane współczynniki wymierne. ■

W przedstawionych faktach pokazaliśmy, jak można zredukować liczbę wektorów „generujących”, gdy dopuszczamy w kombinacjach liniowych współczynniki wymierne. Teraz powróćmy do krat i rozważania kombinacji liniowych o *współczynnikach całkowitych*. Ponownie oddzielnie rozważymy przypadek wektorów równoległych i nierównoległych.

Lemat 3 *Dla wektorów równoległych $\overrightarrow{a}$ i $\overrightarrow{b}$ istnieje wektor $\overrightarrow{c}$ taki, że $L(\overrightarrow{a}, \overrightarrow{b}) = L(\overrightarrow{c})$.*

Dowód W fakcie 1 pokazaliśmy, że istnieje liczba wymierna t taka, że

$$\overrightarrow{b} = t \cdot \overrightarrow{a},$$

zatem przyjmując $t = \frac{p}{q}$ dla $\mathrm{NWD}(p,q) = 1$ mamy

$$q \cdot \overrightarrow{b} \quad = \quad p \cdot \overrightarrow{a}. \tag{1}$$

Szukamy wektora $\overrightarrow{c}$ takiego, że

$$\overrightarrow{a} = m \cdot \overrightarrow{c}, \quad \overrightarrow{b} = n \cdot \overrightarrow{c} \quad \text{oraz}$$

$$\overrightarrow{c} \quad = \quad k \cdot \overrightarrow{a} + l \cdot \overrightarrow{b} \tag{2}$$

dla pewnych liczb całkowitych k, l, m i n. Stąd równość (1) przybiera postać

$$qn \cdot \overrightarrow{c} = pm \cdot \overrightarrow{c},$$

a równość (2) postać

$$\overrightarrow{c} = km \cdot \overrightarrow{c} + ln \cdot \overrightarrow{c}.$$

Przyjmując $m = q$ oraz $n = p$ mamy więc

$$\overrightarrow{c} = (kq + lp) \cdot \overrightarrow{c}.$$

Wystarczy więc wybrać takie liczby k i l, by

$$kq + lp = 1.$$

Jest to możliwe, gdyż liczby p i q są względnie pierwsze, stąd liczby k i l można obliczyć na przykład za pomocą rozszerzonego algorytmu Euklidesa (patrz [17]). ■

Podsumujmy rozwiązanie w przypadku, gdy wektory $\overrightarrow{a}$ i $\overrightarrow{b}$ są równoległe. Wtedy istnieją liczby całkowite p i q, takie że $q \cdot \overrightarrow{b} = p \cdot \overrightarrow{a}$ oraz $\mathrm{NWD}(p,q) = 1$. Wystarczy znaleźć liczby całkowite k i l takie, że

$$kq + lp = 1,$$

a następnie przyjąć $\overrightarrow{c} = k \cdot \overrightarrow{a} + l \cdot \overrightarrow{b}$. Wówczas oczywiście $\overrightarrow{c} \in L(\overrightarrow{a}, \overrightarrow{b})$, a ponadto zachodzą równości:

$$\begin{aligned}
q \cdot \overrightarrow{c} &= q(k \cdot \overrightarrow{a} + l \cdot \overrightarrow{b}) = kq \cdot \overrightarrow{a} + lq \cdot \overrightarrow{b} = (1 - lp) \cdot \overrightarrow{a} + lq \cdot \overrightarrow{b} = \\
&= \overrightarrow{a} - lp \cdot \overrightarrow{a} + lq \cdot \overrightarrow{b} = \overrightarrow{a} + l(q \cdot \overrightarrow{b} - p \cdot \overrightarrow{a}) = \overrightarrow{a}, \\
p \cdot \overrightarrow{c} &= p(k \cdot \overrightarrow{a} + l \cdot \overrightarrow{b}) = kp \cdot \overrightarrow{a} + lp \cdot \overrightarrow{b} = kp \cdot \overrightarrow{a} + (1 - kq) \cdot \overrightarrow{b} = \\
&= kp \cdot \overrightarrow{a} + \overrightarrow{b} - kq \cdot \overrightarrow{b} = \overrightarrow{b} + k(p \cdot \overrightarrow{a} - q \cdot \overrightarrow{b}) = \overrightarrow{b}.
\end{aligned}$$

To dowodzi, że $\overrightarrow{a}, \overrightarrow{b} \in L(\overrightarrow{c})$, czyli $L(\overrightarrow{a}, \overrightarrow{b}) = L(\overrightarrow{c})$. Ostatecznie więc zastąpiliśmy dwa równoległe wektory generujące kratę przez jeden.

Lemat 4 *Załóżmy, że wektory $\overrightarrow{a}$ i $\overrightarrow{b}$ nie są równoległe. Niech $\overrightarrow{c}$ będzie dowolnym wektorem o współrzędnych całkowitych. Pokażemy, że istnieją wektory $\overrightarrow{d}$ i $\overrightarrow{e}$ takie, że*

$$L(\overrightarrow{a}, \overrightarrow{b}, \overrightarrow{c}) = L(\overrightarrow{d}, \overrightarrow{e}).$$

Dowód lematu przeprowadzimy najpierw dla pewnych przypadków szczególnych. Potem rozważymy sytuację ogólną i pokażemy jak można wówczas udowodnić lemat odwołując się do pokazanych wcześniej przypadków.

Na wstępie przypomnijmy, iż z faktu 2 wiemy już, że istnieją liczby wymierne t i u takie, że

$$\overrightarrow{c} = t \cdot \overrightarrow{a} + u \cdot \overrightarrow{b}.$$

Jeśli $u = 0$, to $\overrightarrow{c} = t \cdot \overrightarrow{a}$, więc wektory $\overrightarrow{a}$ i $\overrightarrow{c}$ są równoległe. Z lematu 3 istnieje zatem wektor $\overrightarrow{d}$ taki, że $L(\overrightarrow{a}, \overrightarrow{c}) = L(\overrightarrow{d})$. Wówczas oczywiście

$$L(\overrightarrow{a}, \overrightarrow{b}, \overrightarrow{c}) = L(\overrightarrow{b}, \overrightarrow{d}).$$

Analogiczna sytuacja ma miejsce, gdy $t = 0$.

Przypuśćmy teraz, że obie liczby t i u są różne od zera. Wtedy istnieją liczby całkowite (różne od zera) p, q i r takie, że

$$r \cdot \overrightarrow{c} = p \cdot \overrightarrow{a} + q \cdot \overrightarrow{b}. \tag{3}$$

Możemy oczywiście wybrać liczby p, q i r tak, by nie miały wspólnego dzielnika większego od 1 (choć każde dwie mogą mieć taki wspólny dzielnik).

Dowód (*Przypadek szczególny*) Przypuśćmy najpierw, że liczby p i r są względnie pierwsze:

$$\mathrm{NWD}(p,r) = 1.$$

Pokażemy, że istnieje wtedy wektor $\overrightarrow{\boldsymbol{d}}$ taki, że

$$L(\overrightarrow{\boldsymbol{a}}, \overrightarrow{\boldsymbol{b}}, \overrightarrow{\boldsymbol{c}}) = L(\overrightarrow{\boldsymbol{b}}, \overrightarrow{\boldsymbol{d}}).$$

Poszukiwany wektor $\overrightarrow{\boldsymbol{d}}$ musi być kombinacją liniową wektorów $\overrightarrow{\boldsymbol{a}}$, $\overrightarrow{\boldsymbol{b}}$ i $\overrightarrow{\boldsymbol{c}}$ o współczynnikach całkowitych, spełniającą równości:

$$\overrightarrow{\boldsymbol{a}} = k \cdot \overrightarrow{\boldsymbol{b}} + l \cdot \overrightarrow{\boldsymbol{d}}, \tag{4}$$
$$\overrightarrow{\boldsymbol{c}} = m \cdot \overrightarrow{\boldsymbol{b}} + n \cdot \overrightarrow{\boldsymbol{d}} \tag{5}$$

dla pewnych liczb całkowitych k, l, m i n. To oznacza, iż równość (3) musi mieć postać

$$rm \cdot \overrightarrow{\boldsymbol{b}} + rn \cdot \overrightarrow{\boldsymbol{d}} = kp \cdot \overrightarrow{\boldsymbol{b}} + lp \cdot \overrightarrow{\boldsymbol{d}} + q \cdot \overrightarrow{\boldsymbol{b}}. \tag{6}$$

Ponieważ $\overrightarrow{\boldsymbol{b}}$ i $\overrightarrow{\boldsymbol{d}}$ są niezależne (nierównoległe), więc równość (6) zachodzi wtedy i tylko wtedy, gdy

$$rm - kp = q \quad \text{oraz} \quad rn = lp. \tag{7}$$

Liczby k, l, m i n spełniające warunki (7) możemy dobrać w następujący sposób. Najpierw przyjmujemy $n = p$ oraz $l = r$, a następnie (za pomocą rozszerzonego algorytmu Euklidesa) znajdujemy takie liczby m_0 i k_0, że

$$rm_0 - pk_0 = 1.$$

Wreszcie przyjmujemy $m = m_0 q$ oraz $k = k_0 q$. Wówczas oczywiście $rm - pk = q$, a równości (4) i (5) przyjmują postać

$$\overrightarrow{\boldsymbol{a}} = k_0 q \cdot \overrightarrow{\boldsymbol{b}} + r \cdot \overrightarrow{\boldsymbol{d}}, \tag{8}$$
$$\overrightarrow{\boldsymbol{c}} = m_0 q \cdot \overrightarrow{\boldsymbol{b}} + p \cdot \overrightarrow{\boldsymbol{d}}. \tag{9}$$

Mnożąc obie strony równości (8) przez m_0, równości (9) przez k_0, a następnie odejmując je stronami, otrzymujemy

$$m_0 \cdot \overrightarrow{\boldsymbol{a}} - k_0 \cdot \overrightarrow{\boldsymbol{c}} = (rm_0 - pk_0) \cdot \overrightarrow{\boldsymbol{d}},$$

czyli

$$\overrightarrow{\boldsymbol{d}} = m_0 \cdot \overrightarrow{\boldsymbol{a}} - k_0 \cdot \overrightarrow{\boldsymbol{c}}. \tag{10}$$

Stąd (oraz z równości (8) i (9)) oczywiście wynika, że

$$L(\overrightarrow{\boldsymbol{a}}, \overrightarrow{\boldsymbol{b}}, \overrightarrow{\boldsymbol{c}}) = L(\overrightarrow{\boldsymbol{b}}, \overrightarrow{\boldsymbol{d}}).$$

■

Podsumujmy rozwiązanie w *przypadku szczególnym*. Mamy wektory $\vec{a}$, $\vec{b}$ i $\vec{c}$ oraz liczby całkowite p, q i r spełniające równość (3). Dodatkowo zakładamy, że liczby r i p są względnie pierwsze. Za pomocą rozszerzonego algorytmu Euklidesa znajdujemy liczby m_0 i k_0 takie, że $rm_0 - pk_0 = 1$ i definiujemy wektor

$$\vec{d} = m_0 \cdot \vec{a} - k_0 \cdot \vec{c}.$$

Możemy sprawdzić, że zachodzą równości:

$$\begin{aligned}
k_0q\cdot\vec{b} + r\cdot\vec{d} &= k_0q\cdot\vec{b} + m_0r\cdot\vec{a} - k_0r\cdot\vec{c} = \\
&= k_0q\cdot\vec{b} + m_0r\cdot\vec{a} - k_0p\cdot\vec{a} - k_0q\cdot\vec{b} = \\
&= (m_0r - k_0p)\cdot\vec{a} = \vec{a}, \\
m_0q\cdot\vec{b} + p\cdot\vec{d} &= m_0q\cdot\vec{b} + m_0p\cdot\vec{a} - k_0p\cdot\vec{c} = \\
&= m_0q\cdot\vec{b} + m_0p\cdot\vec{a} - (m_0r - 1)\cdot\vec{c} = \\
&= m_0q\cdot\vec{b} + m_0p\cdot\vec{a} - m_0r\cdot\vec{c} + \vec{c} = \\
&= m_0(p\cdot\vec{a} + q\cdot\vec{b} - r\cdot\vec{c}) + \vec{c} = \vec{c}.
\end{aligned}$$

To dowodzi, że $L(\vec{a}, \vec{b}, \vec{c}) = L(\vec{b}, \vec{d})$.

Dowód (*Sytuacja ogólna*) Rozważmy teraz sytuację, gdy liczby występujące w równości (3) są dowolne, czyli liczba r być może nie jest względnie pierwsza z p.

Najpierw znajdujemy liczby r_1 i r_2 takie, że $r = r_1r_2$, $\mathrm{NWD}(r_1, q) = 1$ oraz $\mathrm{NWD}(r_2, p) = 1$. Na przykład możemy przyjąć, że liczba r_1 jest iloczynem tych czynników pierwszych liczby r, które są dzielnikami p, a r_2 jest iloczynem pozostałych czynników pierwszych r. Wówczas równość (3) ma postać

$$r_2\cdot r_1\cdot\vec{c} \;=\; p\cdot\vec{a} + q\cdot\vec{b}. \tag{11}$$

Zauważmy, że w powyższym wzorze współczynniki przy wektorach $\vec{a}$ i $r_1\cdot\vec{c}$ są względnie pierwsze, stąd dla trójki wektorów $\vec{a}$, $\vec{b}$ i $r_1\cdot\vec{c}$ możemy zastosować postępowanie opisane w *przypadku szczególnym*. Znajdujemy więc liczby m_0 i k_0 takie, że $r_2m_0 - pk_0 = 1$ i definiujemy wektor $\vec{d}$ analogicznie jak we wzorze (10)

$$\vec{d} = m_0\cdot\vec{a} - k_0(r_1\cdot\vec{c}).$$

Wtedy

$$L(\vec{b}, \vec{d}) = L(\vec{a}, \vec{b}, r_1\cdot\vec{c}).$$

Ponieważ rozszerzając zbiory generujące dwóch krat o ten sam wektor nadal otrzymujemy równe kraty, więc mamy (ostatnią równość otrzymujemy dzięki wyeliminowaniu ze zbioru generującego jednego z dwóch wektorów równoległych):

$$L(\vec{b}, \vec{d}, \vec{c}) = L(\vec{a}, \vec{b}, r_1\cdot\vec{c}, \vec{c}) = L(\vec{a}, \vec{b}, \vec{c}).$$

Następnie dla wektora $r_1\cdot\vec{c}$ mamy równość analogiczną do (9)

$$r_1\cdot\vec{c} = m_0q\cdot\vec{b} + p\cdot\vec{d}. \tag{12}$$

Zauważmy, że ponieważ $\mathrm{NWD}(r_1, q) = 1$, także liczby m_0 i r_1 muszą być względnie pierwsze. Gdyby nie były i liczba $s > 1$ byłaby ich wspólnym dzielnikiem, to liczba s byłaby także dzielnikiem p (ze wzoru (12)) i stąd także dzielnikiem $r_2m_0 - pk_0$. Ponieważ ostatnia liczba jest równa 1, więc dochodzimy w ten sposób do sprzeczności. Zatem $\mathrm{NWD}(r_1, m_0) = 1$, czyli także $\mathrm{NWD}(r_1, m_0q) = 1$. To pozwala nam ponownie zastosować rozumowanie analogiczne jak w *przypadku szczególnym* tym razem dla wektorów $\vec{c}$, $\vec{b}$ i $\vec{d}$ powiązanych zależnością (12). W ten sposób znajdujemy wektor $\vec{e}$ taki, że

$$L(\vec{b}, \vec{d}, \vec{c}) = L(\vec{d}, \vec{e}).$$

Ostatecznie więc

$$L(\vec{a}, \vec{b}, \vec{c}) = L(\vec{b}, \vec{d}, \vec{c}) = L(\vec{d}, \vec{e}),$$

co kończy dowód lematu. Wypisanie dokładnych wzorów na współrzędne wektorów $\vec{d}$ i $\vec{e}$ pozostawiamy jako ćwiczenie. ■

W dowodach lematów 3 i 4 jest zawarty opis algorytmu rozwiązania zadania. Rozpoczynamy od zbioru wektorów $\{\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_n\}$ i przyjmujemy $\vec{a} = \vec{a}_1$ oraz $\vec{b} = \vec{a}_2$. Następnie rozważamy kolejno $\vec{c} = \vec{a}_i$, dla $i = 3, 4, \ldots, n$. Wiemy już, że krata $L(\vec{a}, \vec{b}, \vec{c})$ ma bazę złożoną z co najwyżej dwóch wektorów. Znajdujemy je jak w dowodach lematów, zastępujemy nimi $\vec{a}$ i $\vec{b}$, a zbiór wektorów redukujemy do $\{\vec{a}, \vec{b}, \vec{a}_{i+1}, \ldots, \vec{a}_n\}$. Na końcu dostajemy poszukiwaną bazę kraty złożoną z co najwyżej dwóch wektorów. Podstawową wadą tej metody jest to, że współrzędne otrzymanych wektorów bazowych szybko rosną, ale korzystając z rozwiązania zadania „Tomki” (por.[11]) możemy znaleźć za każdym razem równoważną bazę złożoną z wektorów krótkich.

Rozwiązanie wzorcowe

Opiszemy teraz drugie rozwiązanie. Jest ono oparte na następującej obserwacji.

Lemat 5 *Dla dowolnych wektorów $\vec{a}$ i $\vec{b}$ istnieją wektory*

$$\vec{c} = [x_c, 0] \quad \textit{oraz} \quad \vec{d} = [x_d, y_d]$$

takie, że $L(\vec{a}, \vec{b}) = L(\vec{c}, \vec{d})$.

Dowód Jeśli $y_a = 0$ lub $y_b = 0$, to lemat jest oczywisty, rozważmy więc przypadek, gdy $y_a \neq 0$ oraz $y_b \neq 0$. Zdefiniujmy

$$m = \mathrm{NWD}(y_a, y_b), \quad k = \frac{y_a}{m} \quad \text{oraz} \quad l = \frac{y_b}{m}.$$

Wówczas $\mathrm{NWD}(k, l) = 1$ i za pomocą rozszerzonego algorytmu Euklidesa znajdujemy liczby p i q takie, że $kp + lq = 1$. Wtedy $y_ap + y_bq = m$. Stąd łatwo wynika, że wektory

$$\vec{c} = l \cdot \vec{a} - k \cdot \vec{b} \quad \text{oraz} \quad \vec{d} = p \cdot \vec{a} + q \cdot \vec{b}$$

są szukanymi wektorami. Mianowicie

$$\vec{c} = [lx_a - kx_b, 0]$$

oraz $\overrightarrow{c}, \overrightarrow{d} \in L(\overrightarrow{a}, \overrightarrow{b})$. Ponadto

$$\overrightarrow{a} = q \cdot \overrightarrow{c} + k \cdot \overrightarrow{d} \quad \text{oraz} \quad \overrightarrow{b} = -p \cdot \overrightarrow{c} + l \cdot \overrightarrow{d},$$

skąd wynika, że $\overrightarrow{a}, \overrightarrow{b} \in L(\overrightarrow{c}, \overrightarrow{d})$.

Można także łatwo dowieść, że jeśli wektory $\overrightarrow{a}$ i $\overrightarrow{b}$ są równoległe, to otrzymany wektor $\overrightarrow{c}$ jest wektorem zerowym i wtedy $L(\overrightarrow{a}, \overrightarrow{b}) = L(\overrightarrow{d})$. ■

Teraz możemy opisać algorytm oparty na lemacie 5. Rozpoczynamy od zbioru wektorów $\{\overrightarrow{a}_1, \overrightarrow{a}_2, \ldots, \overrightarrow{a}_n\}$. Przyjmujemy $\overrightarrow{a} = \overrightarrow{a}_1$ i rozważamy kolejno pary wektorów $\overrightarrow{a}$, $\overrightarrow{a}_i$ dla $i = 2, 3, \ldots, n$. Każdą taką parę zastępujemy przez dwa wektory, z których jeden wektor (oznaczymy go $\overrightarrow{b}_i$) ma drugą współrzędną zerową (drugi wektor z pary podstawiamy pod $\overrightarrow{a}$). W ten sposób otrzymujemy zbiór wektorów $\{\overrightarrow{a}, \overrightarrow{b}_2, \overrightarrow{b}_3, \ldots, \overrightarrow{b}_n\}$, gdzie tylko $\overrightarrow{a}$ może mieć obie współrzędne niezerowe, a pozostałe wektory mają drugą współrzędną zerową. Teraz zauważmy, że dwa wektory

$$\overrightarrow{b}_2 = [x_2, 0] \quad \text{oraz} \quad \overrightarrow{b}_3 = [x_3, 0]$$

możemy zastąpić jednym wektorem $\overrightarrow{b} = [x, 0]$, gdzie $x = \mathrm{NWD}(x_2, x_3)$. Powtarzając to postępowanie kolejno dla par wektorów $\overrightarrow{b}$ i $\overrightarrow{b}_i$ dla $i = 4, 5, \ldots, n$ na końcu dostajemy poszukiwaną bazę złożoną z co najwyżej dwóch wektorów. Program wzorcowy był oparty na tym algorytmie.

Uwagi końcowe

Zauważmy, że w algorytmie wzorcowym istotne było to, że wszystkie współrzędne wektorów były liczbami całkowitymi. Natomiast pierwszy z przedstawionych algorytmów jest nieco ogólniejszy. Założenie, iż współrzędne wektorów są całkowite, było w nim wykorzystane tylko do wykazania, że istnieją liczby całkowite p, q i r spełniające równości (1) i (3). Wiele innych krat ma te własności i algorytm pierwszy pozwala znaleźć co najwyżej dwuelementową bazę takiej kraty. Natomiast może okazać się, że w takiej kracie nie istnieje żaden wektor postaci $[x, 0]$, a więc nie można zastosować algorytmu wzorcowego.

Testy

Zadanie było testowane za pomocą 10 testów generowanych losowo. Zawsze najpierw była ustalana baza, a następnie były generowane losowo wektory kraty. Dla utrudnienia dbano o to, by wektory bazowe nie znalazły się wśród wektorów generowanych losowo, a także, by nie istniała baza złożona z wektora poziomego i pionowego.

W poniższej tabeli n oznacza liczbę wektorów odpowiadających ruchom skoczka.

Nazwa	**n**	**Opis**
sko1.in	3	test zawierający 2 wektory liniowo zależne
sko2.in	5	test losowy
sko3.in	10	test losowy
sko4.in	20	test losowy
sko5.in	50	test losowy
sko6.in	60	test losowy
sko7.in	70	test losowy
sko8.in	80	test losowy
sko9.in	90	test losowy
sko10.in	100	test losowy

Zawody II stopnia

opracowania zadań

Paweł Parys | **Marcin Pilipczuk**
Treść zadania, Opracowanie | Program

OI, Etap II, dzień próbny, 8–02–2005

Lot na marsa

Bajtazar postanowił polecieć na Marsa, aby zwiedzić istniejące tam stacje badawcze. Wszystkie stacje na Marsie leżą na okręgu. Bajtazar ląduje w jednej z nich, a następnie porusza się za pomocą specjalnego pojazdu, który jest napędzany odpowiednim paliwem. Litr paliwa starcza na metr jazdy. Zapasy paliwa są rozmieszczone w różnych stacjach. Bajtazar może tankować paliwo na stacji, na której w danym momencie się znajduje (zawsze może zatankować cały zapas z danej stacji — pojemność baku jego pojazdu jest nieograniczona). Musi mu to wystarczyć na dojazd do następnej stacji. Bajtazar musi zdecydować, gdzie powinien wylądować, tak żeby mógł zwiedzić wszystkie stacje. Na koniec Bajtazar musi wrócić do stacji, w której wylądował. W czasie podróży Bajtazar musi poruszać się po okręgu, stale w wybranym jednym z dwóch kierunków.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia liczbę stacji na Marsie, odległości między nimi i ilości paliwa dostępne w każdej z nich,*
- *dla każdej stacji sprawdzi, czy Bajtazar może tam wylądować, czyli czy zaczynając tam i jadąc w wybranym przez siebie kierunku, może objechać wszystkie stacje i wrócić do swojej rakiety,*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu standardowego wejścia zapisana jest jedna liczba całkowita n $(3 \leqslant n \leqslant 1\,000\,000)$*. Jest to liczba stacji na Marsie. Stacje są ponumerowane od 1 do* n*. W kolejnych* n *wierszach znajdują się opisy poszczególnych stacji i odległości między nimi. W* $(i+1)$*-szym wierszu znajdują się dwie liczby całkowite:* p_i *oraz* d_i $(p_i \geqslant 0,\ d_i > 0)$*. Pierwsza z nich to ilość paliwa w litrach dostępna na* i*-tej stacji. Druga z nich to odległość w metrach pomiędzy stacją* i *a* $i+1$ *(oczywiście* d_n *to odległość między stacją* n *a 1). Łączna ilość dostępnego paliwa, a także suma wszystkich odległości między stacjami nie przekracza* $2\,000\,000\,000$*.*

Wyjście

Na standardowe wyjście należy wypisać n *wierszy. W* i*-tym wierszu powinno znajdować się słowo* `TAK`*, jeśli Bajtazar może wylądować w stacji numer* i*, lub* `NIE` *w przeciwnym wypadku.*

Przykład

Dla danych wejściowych:

```
5
3 1
1 2
5 2
0 1
5 4
```

poprawnym wynikiem jest:

```
TAK
NIE
TAK
NIE
TAK
```

Rozwiązanie

Wstępne spostrzeżenia

Pierwsze spostrzeżenie wiąże się z rozmiarem danych wejściowych. Liczba stacji rzędu miliona sugeruje, że powinniśmy poszukiwać algorytmu działającego w czasie liniowym $O(n)$, w najgorszym razie w czasie rzędu $O(n \log n)$, gdzie n oznacza liczbę stacji.

Następnie zauważmy, że zadanie można rozwiązać oddzielnie dla dwóch przypadków — przy założeniu, że Bajtazar porusza się zawsze zgodnie z rosnącym porządkiem numerów stacji (z wyjątkiem przejazdu ze stacji n-tej do pierwszej), lub przy założeniu, że kierunek jego podróży jest przeciwny. Poniżej rozważymy pierwszy z tych przypadków i przedstawimy algorytm pozwalający odnaleźć stacje, z których Bajtazar może wówczas rozpocząć podróż. Aby uzyskać końcowe rozwiązanie, wystarczy opisany algorytm lekko zmodyfikować i zastosować ponownie do przypadku, gdy Bajtazar porusza się w przeciwnym kierunku.

Rozwiązanie zadania rozpoczniemy od oczywistego spostrzeżenia, iż jeśli sumaryczna ilość paliwa na wszystkich stacjach jest mniejsza niż sumaryczna długość drogi, jaką ma przebyć Bajtazar, to nie istnieje stacja, z której Bajtazar może rozpocząć podróż. Co ważniejsze, i już nie tak oczywiste, jeżeli sumaryczna ilość paliwa wystarcza na przebycie całej drogi, to taka stacja istnieje. Pokażemy jak ją znaleźć, a następnie na jej podstawie odnajdziemy pozostałe stacje, w których Bajtazar może wylądować, by odbyć swą podróż po Marsie.

Uważny Czytelnik pilnie śledzący zadania olimpijskie zauważy, że w rozwiązaniu zadania wykorzystuje się ten sam fakt, co w zadaniu „Zwiedzanie” *z finału VIII OI [8].*

Rozwiązanie wzorcowe

Oznaczenia

Przyjmujemy dalej następujące oznaczenia:

- ilość paliwa na stacji i-tej oznaczymy przez p_i, a odległość w metrach między stacją i oraz stacją $i+1$ przez d_i (d_n będzie oznaczać odległość pomiędzy stacją n i stacją 1);
- *stan paliwa na stacji* i to ilość paliwa w baku pojazdu Bajtazara w momencie wjeżdżania na stację i (przed zatankowaniem na tej stacji);
- $S[a,b]$ oznacza zmianę stanu baku pojazdu Bajtazara na odcinku pomiędzy stacją a i stacją b; oczywiście zakładamy, że Bajtazar na każdej stacji tankuje całe znajdujące się tam paliwo, stąd

$$S[a,b] = p_a - d_a + p_{a+1} - d_{a+1} + \ldots + p_{b-1} - d_{b-1},$$

  przy czym sumowanie we wzorze przeprowadzamy cyklicznie, tzn. $S[a,b] = S[a,n] + p_n - d_n + S[1,a]$, dla $b < a$; wartość $S[a,a]$ można interpretować jako zero (zmianę stanu baku, gdy Bajtazar nie rusza się ze stacji a) lub zmianę stanu baku po objechaniu wszystkich stacji i powrocie do stacji a — od pewnego momentu nie powinno to powodować niejednoznaczności (patrz uwaga 4);
- stację i nazwiemy *stacją wypadową*, jeśli lądując w niej Bajtazar może objechać wszystkie stacje (w rozważanym przypadku zgodnie z rosnącym porządkiem numerów);

Ponadto we wszystkich dalszych rozważaniach numery stacji będziemy traktować cyklicznie, tzn. numery: $-2,-1,0,1,2,\ldots$ są równoważne odpowiednio numerom: $n-2, n-1, n, n+1, n+2, \ldots$.

Znajdowanie pierwszej stacji wypadowej

Na początek pokażemy dwa proste fakty.

Fakt 1 *Jeśli sumaryczna ilość paliwa na Marsie jest mniejsza niż sumaryczna droga do przebycia, tzn.* $\sum_{i=1}^{n} p_i < \sum_{i=1}^{n} d_i$, *to na Marsie nie istnieje stacja wypadowa.*

Fakt 2 *Stacja a jest stacją wypadową wtedy i tylko wtedy, jeśli dla każdego b zachodzi nierówność* $S[a,b] \geqslant 0$.

Pierwszy fakt jest oczywisty. Podobnie pierwsza część dowodu drugiego faktu. Wystarczy zauważyć, że jeśli a jest stacją, z której Bajtazar rozpoczyna podróż, to $S[a,b]$ oznacza stan baku na stacji b, gdyż bak w chwili początkowej (w momencie wylądowania w stacji a) jest pusty. Stąd jeśli a jest stacją wypadową, to dla każdej stacji b stan baku po dojechaniu do niej z a musi być nieujemny.

Pozostaje wykazać, że jeśli dla każdego b zachodzi $S[a,b] \geqslant 0$, to a jest stacją wypadową. W tym celu rozważmy kolejno $i = a+1, a+2, \ldots, a-1, a$. Nierówność $S[a,a+1] \geqslant 0$ oznacza, że da się dojechać ze stacji a do następnej startując z pustym bakiem. Następnie wiedząc, że da się dojechać ze stacji a do stacji i, zauważamy, że zapas w baku w momencie wyjazdu ze stacji i, czyli $S[a,i]+p_i$, jest dodatni i wystarcza do pokonania drogi d_i, gdyż $S[a,i]+p_i-d_i=S[a,i+1]\geqslant 0$.

W kolejnym fakcie pokażemy, jak stwierdzić, czy na Marsie istnieje stacja wypadowa, i jak ją znaleźć.

Fakt 3 *Jeśli $\sum_{i=1}^{n} p_i \geqslant \sum_{i=1}^{n} d_i$, to istnieje stacja wypadowa.*

Dowód Najpierw, aby uprościć dowód, zmienimy nieco dane wejściowe dbając jednak, by nie wpłynęło to na rozwiązanie zadania. Zauważmy, że możemy wydłużyć drogę na Marsie tak, by dostępna ilość paliwa dokładnie wystarczała na jej pokonanie. Na przykład możemy ustalić $d_n = d_n + \sum_{i=1}^{n} p_i - \sum_{i=1}^{n} d_i$. Jeżeli przy tak zmienionych danych istnieje stacja a, która jest stacją wypadową, to ta sama stacja jest stacją wypadową dla danych oryginalnych. Również w sposób oczywisty stacja, która przed zmianą nie mogła być stacją wypadową — nadal nią nie będzie.

Teraz rozważymy drogę, którą nazwiemy *fikcyjną* — założymy, że pojazd Bajtazara ma na początku w baku zapas paliwa $Z = \sum_{i=1}^{n} d_i$. To oczywiście wystarczy, by odbyć drogę rozpoczynając z dowolnej stacji. W takim razie rozpoczniemy od stacji 1 i będziemy kontynuować podróż przez kolejne stacje zawsze tankując całe znajdujące się tam paliwo. Przez x_i oznaczymy stan baku na stacji i. Niech x_k będzie minimalną spośród wartości $\{x_i \mid 1\leqslant i\leqslant n\}$.

Możemy teraz rozważyć drugą drogę, którą nazwiemy *rzeczywistą*. Rozpoczynamy ją od stacji k z pustym bakiem. Niech y_i oznacza stan baku na stacji i w trakcie drogi rzeczywistej (mamy więc $y_k = 0$).

Zauważmy, że dla każdego i zarówno $x_{i+1}-x_i = p_i - d_i$, jak i $y_{i+1}-y_i = p_i-d_i$. Stąd dla każdego i zachodzi równość $x_i - y_i = x_{i+1}-y_{i+1}$, a więc różnica pomiędzy stanem baku w trakcie podróży fikcyjnej i rzeczywistej na poszczególnych stacjach jest stała.

Na koniec wystarczy przypomnieć, że x_k była minimalną spośród wartości $\{x_i \mid 1\leqslant i\leqslant n\}$, więc $y_k=0$ musi być minimalną spośród wartości $\{y_i \mid 1\leqslant i\leqslant n\}$. Z tego wnioskujemy, że stan baku w trakcie podróży rzeczywistej nigdy nie spada poniżej zera, więc k jest stacją wypadową. ■

Na podstawie dowodu faktu 3 możemy opisać algorytm znajdowania stacji wypadowej. Jeśli

$$\sum_{i=1}^{n} p_i \quad \geqslant \quad \sum_{i=1}^{n} d_i, \tag{1}$$

to obliczamy wartości $x_i = S[1,i]$ dla $1\leqslant i\leqslant n$ i wybieramy $x_k=\min\{x_i \mid 1\leqslant i\leqslant n\}$ znajdując stację wypadową k. Jeśli nierówność (1) nie zachodzi, to odpowiadamy, że stacja wypadowa nie istnieje.

Uwaga 4 *Jeżeli podane wartości $p_1,p_2,\ldots,p_n$ i $d_1,d_2,\ldots,d_n$ spełniają nierówność (1) i dla pewnej stacji a da się z niej dojechać do wszystkich stacji $b\neq a$, to da się także powrócić do rakiety zaparkowanej w stacji a.*

To proste spostrzeżenie powoduje, że w dalszych rozważaniach wykorzystujących fakt 2 nie musimy brać pod uwagę wartości $S[a,a]$ oznaczających zmianę stanu baku po objechaniu wszystkich baz. Od tej chwili przyjmiemy więc $S[a,a]=0$ dla wszystkich stacji a.

Poszukiwanie pozostałych stacji wypadowych

Na początek pokażemy prosty fakt.

Fakt 5 *Jeśli stacja b jest stacją wypadową i dla pewnej stacji a zachodzą nierówności*

$$\begin{array}{rcl} S[a,b] & \geqslant & 0 \text{ oraz} \\ S[i,b] & < & 0 \text{ dla } i = a+1, a+2, \ldots, b-1, \end{array}$$

to stacja a jest stacją wypadową, a stacje $a+1, a+2, \ldots, b-1$ *nie są stacjami wypadowymi.*

Dowód Oznaczmy $A = \{a+1, a+2, \ldots, b-1\}$. Ponieważ dla każdej stacji $x \in A$ zachodzi $S[x,b] < 0$, więc z faktu 2 wynika, iż żadna z tych stacji nie jest stacją wypadową.

Pozostaje wykazać, że stacja a jest stacją wypadową. Zauważmy, że dla każdej stacji $x \in A$ zachodzi równość $S[a,x] + S[x,b] = S[a,b]$. Skoro także $S[x,b] < 0$ i $S[a,b] \geqslant 0$, to otrzymujemy, iż $S[a,x] \geqslant 0$. Następnie rozważmy stacje $x \notin A \cup \{b\}$. Dla każdej z nich zachodzi równość $S[a,x] = S[a,b] + S[b,x]$. Ponieważ $S[a,b] \geqslant 0$ oraz b jest stacją wypadową, czyli dla każdej stacji y zachodzi $S[b,y] \geqslant 0$, stąd otrzymujemy $S[a,x] \geqslant 0$. Pozostaje przypomnieć, że również $S[a,b] \geqslant 0$, i na mocy faktu 2 kończymy dowód. ■

Na podstawie dowodu faktu 5 możemy opisać algorytm znajdowania wszystkich stacji wypadowych, znając jedną stację wypadową k. Wystarczy rozważać kolejno (cyklicznie) stacje w malejącym porządku numerów $i = k-1, k-2, \ldots, k+1$ obliczając wartości $S[i,k]$ do czasu, aż natrafimy na pierwszą stację x, dla której $S[x,k] \geqslant 0$. Wówczas z faktu 5 wnioskujemy, że stacja x jest wypadowa, a stacje o numerach $x+1, x+2, \ldots, k-1$ nie są stacjami wypadowymi. Jeśli $x \neq k$, to powtarzamy rozumowanie rozpoczynając tym razem od stacji wypadowej x. W ten sposób rozpoznajemy wszystkie stacje wypadowe dla podróży w pierwszym kierunku.

Algorytm wzorcowy

Wykorzystując pokazane fakty skonstruujemy procedurę znajdowania wszystkich stacji wypadowych dla Bajtazara. Musimy przy tym pamiętać, by uruchomić ją dwukrotnie, aby wykryć stacje wypadowe dla podróży w obu dopuszczalnych kierunkach.

1: Jeżeli $\sum_{i=1}^{n} p_i < \sum_{i=1}^{n} d_i$, to odpowiadamy n razy „NIE” i kończymy.

2: Poszukujemy stacji wypadowych przy założeniu, że Bajtazar będzie podróżował w kierunku rosnących numerów stacji.

- a: Obliczamy wartości $S[1,i]$ dla $1 \leqslant i \leqslant n$ i znajdujemy stację k_0, dla której $S[1,k_0] = \min\{S[1,i] \mid 1 \leqslant i \leqslant n\}$.
- b: Szukamy pozostałych stacji wypadowych k_i, dla $i = 1, 2, \ldots$
  - i: Sprawdzamy kolejno stacje $k = k_{i-1} - 1, k_{i-1} - 2, \ldots$, aż natrafimy na pierwszą stację k, dla której $S[k, k_{i-1}] \geqslant 0$ (natomiast $S[j, k_{i-1}] < 0$ dla $j = k+1, k+2, \ldots, k_{i-1} - 1$). Oznaczamy znalezioną stację jako k_i.
  - ii: Jeśli $k_i = k_0$, to kończymy pętlę (b).

c: Stacje $k_0, k_1, \ldots, k_{i-1}$ oznaczamy jako stacje wypadowe.

3: Poszukujemy stacji wypadowych przy założeniu, że Bajtazar będzie podróżował w kierunku malejących numerów stacji. W tym celu wykonujemy krok 2 odwracając wszędzie kierunek przeglądania stacji.

4: Dla każdej stacji wypadowej wypisujemy odpowiedź „TAK", dla pozostałych wypisujemy odpowiedź „NIE".

W przedstawionym algorytmie kilkakrotnie przeglądamy tablice $p[1..n]$ i $d[1..n]$ oraz wyliczamy kolejne wartości $S[1,i]$ dla $i=1,2,\ldots$ oraz $S[i,k]$ dla $i=k-1,k-2,\ldots$. Wszystkie te obliczenia wymagają wykonania $O(n)$ operacji. Zauważmy także, że wartości występujące w obliczeniach nie wykraczają poza zakres 32-bitowych liczb całkowitych ze znakiem. Ostatecznie algorytm ma więc złożoność czasową i pamięciową $O(n)$.

Inne rozwiązanie

Przedstawimy także nieco wolniejsze rozwiązanie o złożoności $O(n \log n)$. Podobnie, jak w rozwiązaniu wzorcowym oddzielnie rozważymy dwa dopuszczalne kierunki podróży.

Zakładając, że będziemy poruszać się zgodnie z rosnącym porządkiem numerów stacji, definiujemy tak samo jak poprzednio wartości $S[a,b]$ i sprawdzamy warunek ze wzoru (1). Następnie dla każdej stacji będziemy testować, czy może być stacją wypadową, opierając się, jak poprzednio, na warunku z faktu 2. W związku z tym, dla każdej stacji $a=1,2,\ldots,n$ musimy znać wartości $S[a,i]$ dla $i=a+1,a+2,\ldots,a$. Jeżeli minimalna spośród nich jest nieujemna, to będziemy wiedzieli, że stacja x może być stacją wypadową.

Dla stacji a obliczane wartości $S[a,i]$ podzielimy na dwie grupy.

$$\begin{aligned}\overline{X}_a &= \{S[a,i] \mid 1 < i \leqslant a\};\\ \overline{Y}_a &= \{S[a,i] \mid a < i \leqslant 1\}.\end{aligned}$$

Aby stacja a była stacją wypadową musi zachodzić nierówność

$$\min\{\min(\overline{X_a}), \min(\overline{Y_a})\} \geqslant 0 \tag{2}$$

Jej sprawdzenie wymaga śledzenia zawartości zbiorów $\overline{X}_a$ i $\overline{Y}_a$, co jest o tyle kłopotliwe, że zbiory te zmieniają się istotnie dla kolejnych stacji a. Zastąpimy je więc zbiorami

$$\begin{aligned}X_a &= \{S[1,i] \mid 1 < i \leqslant a\},\\ Y_a &= \{S[1,i] \mid a < i \leqslant 1\},\end{aligned}$$

które są związane z poprzednimi następującymi zależnościami

$$\begin{aligned}\overline{X}_a &= \{S[a,i] \mid 1 < i \leqslant a\} &= \{S[a,1]+S[1,i] \mid 1 < i \leqslant a\} &= S[a,1]+X_a\\ Y_a &= \{S[1,i] \mid a < i \leqslant 1\} &= \{S[1,a]+S[a,i] \mid a < i \leqslant 1\} &= S[1,a]+\overline{Y}_a,\end{aligned}$$

gdzie dla zbioru $A = \{a_1, a_2, \ldots, a_m\}$ i liczby d przez $d+A$ oznaczymy zbiór $\{d+a_1, d+a_2, \ldots, d+a_m\}$. Wartości zbiorów X_a i Y_a można łatwo aktualizować dla kolejnych stacji a, ponieważ

$$X_{a+1} = X_a \cup S[1,a+1]\}, \quad (3)$$
$$Y_{a+1} = Y_a \setminus \{S[1,a+1]\}. \quad (4)$$

Ostatecznie warunek (2) pozwalający sprawdzić, czy stacja a jest stacją wypadową, możemy zastąpić warunkiem

$$\min\{\min(X_a + S[a,1]), \min(Y_a - S[1,a])\} \geqslant 0.$$

Cała procedura wyznaczania stacji wypadowych ma następującą postać.

1: Utwórz zbiory $X = \{0\}$ oraz $Y = \{S[1,i] \mid 1 < i \leqslant n\}$;

2: Rozważaj kolejno stacje $a = 1, 2, 3, \ldots, n$:

  a: oblicz $M = \min\{\min(X + S[a,1]), \min(Y - S[1,a])\}$;

  b: jeśli $M < 0$, to odpowiedz, że a nie jest stacją wypadową i przejdź do sprawdzania kolejnej stacji;

  c: oblicz zbiory X i Y dla kolejnej stacji zgodnie ze wzorami (3) i (4).

Zauważmy, że dla kolejnych wartości zbioru X minima możemy wyznaczać w czasie stałym, gdyż do zbioru tego w każdej iteracji pętli 2 jest dorzucany tylko jeden element. W przypadku zbioru Y, z którego w kolejnych iteracjach ujmujemy po jednym elemencie, należy zastosować strukturę, która umożliwi wyznaczanie minimów w czasie logarytmicznym (na przykład kopiec), lub wstępnie posortować zbiór.

Ostatecznie wyliczenie wszystkich minimów może być wykonane w czasie $O(n \log n)$ i w pamięci $O(n)$.

Rozwiązania mniej efektywne i niepoprawne

Kolejne poprawne, aczkolwiek tym razem znacznie wolniejsze od wzorcowego, rozwiązanie polega na „naiwnym” sprawdzeniu każdej stacji. W tym celu symulujemy podróż Bajtazara z tej stacji w obu kierunkach i sprawdzamy, czy da się obejść wszystkie stacje i powrócić do wyjściowej. Algorytm ten działa w czasie $O(n^2)$ i wymaga pamięci $O(n)$.

Wśród rozwiązań zawodników mogą również pojawić się różne błędne heurystyki. Na przykład decydowanie o tym, czy stacja jest wypadową, na podstawie możliwości dojścia do kilku stacji pośrednich. Programy takie nie powinny przejść przez zaproponowane testy.

Testy

Rozwiązania zawodników były sprawdzane na 11 testach. W każdym, poza *lot1b.in*, opisana jest sytuacja, gdy istnieje stacja wypadowa, tzn. na Marsie jest wystarczająco wiele paliwa, by objechać wszystkie stacje. Testy *lot1a.in* i *lot1b.in* zostały połączone w jeden zestaw.
W przedstawionej poniżej tabeli n oznacza liczbę stacji na Marsie, natomiast #TAK liczbę stacji wypadowych.

Nazwa	n	#TAK	Opis
lot1a.in	20	5	prosty test poprawnościowy
lot1b.in	100 000	0	spory test, w którym w stacjach nie ma dość paliwa, by obejść wszystkie stacje
lot2.in	1 001	9	test, w którym występują małe wartości p_i oraz d_i, a w czasie podróży Bajtazar ma zawsze w baku niewiele paliwa — algorytmy działające w czasie kwadratowym powinny zaliczyć ten test
lot3.in	15 000	5 919	test, w którym w stacjach jest bardzo dużo paliwa; oprócz dwóch najszybszych algorytmów, także zoptymalizowany algorytm kwadratowy ma szansę go zaliczyć
lot4.in	50 053	50 001	test, w którym prawie każda stacja jest dopuszczalna
lot5.in	306 000	67 416	test, w którym droga jest podzielona na małe odcinki po 100 wierzchołków i w stacjach pomiędzy odcinkami jest dużo paliwa — wystarczy wyjść poza odcinek, by objechać wszystkie stacje
lot6.in	901 800	71 498	test analogiczny do poprzedniego, ale złożony z 900 odcinków długości 1000
lot7.in	100 000	1	test, w którym z każdej stacji da sie objechać prawie wszystkie, ale istnieje tylko jedna wypadowa
lot8.in	1 000 000	492 891	test, w którym zapasy paliwa są duże
lot9.in	1 000 000	18	test, w którym zapasy paliwa ledwo wystarczają, by objechać wszystkie stacje
lot10.in	1 000 000	19 484	test, w którym zapasy paliwa są nieco większe niż w poprzednim, ale nadal jest go niewiele

Bartosz Walczak **Jakub Radoszewski**
Treść zadania, Opracowanie Program

OI, Etap II, dzień 1, 9–02–2005

Banknoty

Bajtocki Bank Bitowy (w skrócie BBB) ma największą w Bajtocji sieć bankomatów. BBB postanowił usprawnić swoje bankomaty i zwrócił się do Ciebie o pomoc. Środkiem płatniczym w Bajtocji są banknoty o nominałach $b_1, b_2, \ldots, b_n$. *BBB postanowił, że bankomaty powinny wypłacać żądaną kwotę w jak najmniejszej łącznej liczbie banknotów.*

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis zapasu banknotów, które posiada bankomat, oraz kwotę do wypłacenia,*
- *obliczy minimalną łączną liczbę banknotów, za pomocą jakiej bankomat może wypłacić żądaną kwotę, oraz znajdzie pewien sposób jej wypłacenia,*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu wejścia znajduje się liczba nominałów n, $1 \leqslant n \leqslant 200$. *Drugi wiersz zawiera* n *liczb całkowitych* $b_1, b_2, \ldots, b_n$, $1 \leqslant b_1 < b_2 < \ldots < b_n \leqslant 20\,000$, *pooddzielanych pojedynczymi odstępami. Trzeci wiersz zawiera* n *liczb całkowitych* $c_1, c_2, \ldots, c_n$, $1 \leqslant c_i \leqslant 20\,000$, *pooddzielanych pojedynczymi odstępami;* c_i *jest liczbą banknotów o nominale* b_i *znajdujących się w bankomacie. W ostatnim, czwartym wierszu wejścia znajduje się jedna liczba całkowita* k *— kwota, którą bankomat ma wypłacić,* $1 \leqslant k \leqslant 20\,000$. *Możesz założyć, dla danych testowych, że kwotę* k *można wypłacić za pomocą dostępnych banknotów.*

Wyjście

Pierwszy wiersz wyjścia powinien zawierać jedną dodatnią liczbę całkowitą równą minimalnej łącznej liczbie banknotów, za pomocą których bankomat może wypłacić kwotę k. *Drugi wiersz wyjścia powinien zawierać* n *liczb całkowitych, oddzielonych pojedynczymi odstępami i oznaczających liczby sztuk poszczególnych banknotów użytych do wypłacenia kwoty* k. *W przypadku, gdy istnieje więcej niż jedno rozwiązanie, program powinien wypisać którekolwiek.*

Przykład

Dla danych wejściowych:
```
3
2 3 5
2 2 1
10
```
poprawnym wynikiem jest:
```
3
1 1 1
```

Rozwiązanie

Podstawowy algorytm dynamiczny

Zadanie należy do grupy problemów *wydawania reszty*, które często rozwiązuje się metodą *programowania dynamicznego*. Metoda ta polega na stopniowym konstruowaniu coraz większych częściowych rozwiązań, aż do uzyskania rozwiązania całego problemu.

W naszym zadaniu będziemy konstruować rozwiązania kolejno, dla coraz większych zbiorów nominałów $\{b_1\}, \{b_1, b_2\}, \ldots, \{b_1, \ldots, b_n\}$. Najpierw obliczymy, jak optymalnie wypłacić kwoty z przedziału $[0, k]$ korzystając tylko z banknotów o nominale b_1. Potem ponownie obliczymy optymalne rozwiązania dla tych kwot, ale korzystając już z banknotów o nominałach $\{b_1, b_2\}$. Następnie powtórzymy obliczenia dla nominałów $\{b_1, b_2, b_3\}$ itd.

Rozważmy sytuację, w której bierzemy pod uwagę tylko banknoty o nominałach $b_1, \ldots, b_i$ dla pewnego $0 \leqslant i \leqslant n$, i wprowadźmy następujące oznaczenia:

- Przez $R_{i,j}$ oznaczymy minimalną liczbę banknotów potrzebną do wypłacenia kwoty j ($0 \leqslant j \leqslant k$), za pomocą banknotów $\{b_1, \ldots, b_i\}$. Jeżeli kwoty j nie można wypłacić za pomocą banknotów $\{b_1, \ldots, b_i\}$, przyjmujemy $R_{i,j} = \infty$.

- Jeśli kwotę j można wypłacić za pomocą rozważanych banknotów, czyli $i \geqslant 1$ oraz $R_{i,j} \neq \infty$, to przez $W_{i,j}$ oznaczymy liczbę banknotów o nominale b_i, które są użyte do wypłacenia kwoty j za pomocą $R_{i,j}$ banknotów ($0 \leqslant W_{i,j} \leqslant c_i$).

Podstawową własnością rozważanego problemu, dzięki której do jego rozwiązania możemy zastosować metodę programowania dynamicznego, jest własność *optymalnej podstruktury*. Pozwala ona konstruować rozwiązanie optymalne problemu, opierając się *tylko* na rozwiązaniach optymalnych podproblemów. W naszym przypadku zasada ta mówi, że jeżeli $R_{i,j}$ (dla $i \geqslant 1$ oraz $R_{i,j} \neq \infty$) jest minimalną liczbą banknotów $b_1, \ldots, b_i$ potrzebną do wypłacenia kwoty j, to $R_{i-1,j'}$, gdzie $j' = j - b_i \cdot W_{i,j}$, jest minimalną liczbą banknotów $b_1, \ldots, b_{i-1}$ potrzebną do wypłacenia kwoty j'. Stąd mamy następujący wzór pozwalający wyznaczać wartości $R_{i,j}$ dla $i \geqslant 1$:

$$R_{i,j} \quad = \quad \min\{R_{i-1,j'} + w : 0 \leqslant w \leqslant c_i \wedge 0 \leqslant j' = j - w \cdot b_i\} \qquad (1)$$

Przy tym jeżeli $R_{i,j} \neq \infty$, to $W_{i,j}$ jest tą wartością w, dla której występuje minimum. Dla $i = 0$ mamy oczywiście:

$$R_{0,j} = \begin{cases} 0 & \text{gdy } j = 0 \\ \infty & \text{gdy } j \geqslant 1 \end{cases}$$

Stosując powyższe wzory, możemy w czasie $\Theta(n \cdot k^2)$ obliczyć wszystkie wartości $R_{i,j}$ — dla każdej z $\Theta(n \cdot k)$ wartości $R_{i,j}$ obliczamy minimum (1) w czasie $\Theta(k)$. W ten sposób mamy algorytm znajdujący poszukiwaną wartość $R_{n,k}$. Jednak dla ograniczeń na n i k podanych w treści zadania działa on zbyt długo na dużych testach.

Ulepszony algorytm dynamiczny

Zauważmy, że do obliczenia wartości $R_{i,j}$ ze wzoru (1) wykorzystujemy tylko te wartości $R_{i-1,j'}$, w których $j' = j - w \cdot b_i$ dla pewnego w. Zatem dla liczby $r \in [0, b_i)$ wartości $R_{i,r}, R_{i,r+b_i}, \ldots, R_{i,r+l \cdot b_i}$ obliczamy tylko na podstawie wartości $R_{i-1,r}, R_{i-1,r+b_i}, \ldots, R_{i-1,r+l \cdot b_i}$. To spostrzeżenie pozwoli nam lepiej zorganizować i przyśpieszyć obliczanie minimów według wzoru (1).

Rozważmy ustalone $i \in [1, n]$ oraz $r \in [0, b_i)$ i niech kwota do wydania będzie postaci $j = r + l \cdot b_i$ dla pewnego l. Wtedy ze wzoru (1), po przedstawieniu liczby wydawanych banknotów o nominale b_i w postaci $w = l - l'$, otrzymujemy:

$$R_{i,j} = \min\{R_{i-1,r+l' \cdot b_i} + (l - l') : l - c_i \leqslant l' \leqslant l \wedge 0 \leqslant l'\}$$

Wprowadzając dla l, takich że $0 \leqslant r + l \cdot b_i \leqslant k$, pomocnicze oznaczenie:

$$M_l = R_{i-1,r+l \cdot b_i} - l$$

możemy wzór (1) zapisać następująco:

$$R_{i,j} \quad = \quad \min\{M_{l'} : l - c_i \leqslant l' \leqslant l \wedge 0 \leqslant l'\} + l \tag{2}$$

Stąd obliczanie kolejno wartości $R_{i,r+l \cdot b_i}$ dla $l = 0, 1, \ldots$ sprowadza się do wyliczania minimów zbiorów $S_l = \{M_{l'} : \max\{l - c_i, 0\} \leqslant l' \leqslant l\}$ dla kolejnych wartości l.

Zauważmy, że jeżeli $\max\{l - c_i, 0\} \leqslant l'' < l' \leqslant l$ oraz $M_{l''} \geqslant M_{l'}$, to wartość $M_{l''}$ nie ma wpływu na wartości $R_{i,r+l \cdot b_i}$, gdyż jest zawsze przysłaniana przez wartość $M_{l'}$. Stąd ze zbioru S_l wystarczy pamiętać tylko takie elementy $Q_l = (M_{l_1}, \ldots, M_{l_m})$, że $\max\{l - c_i, 0\} \leqslant l_1 < \ldots < l_m \leqslant l$ oraz $M_{l_1} < \ldots < M_{l_m}$. Wówczas $\min(S_l) = \min(Q_l) = M_{l_1}$. Co więcej, ciąg Q_{l+1} można łatwo wyznaczyć na podstawie ciągu Q_l. W tym celu:

- jeśli $l_1 = l - c_i$, to usuwamy z początku Q_l element M_{l_1};
- usuwamy z końca ciągu Q_l wszystkie wartości większe lub równe M_{l+1};
- wstawiamy na koniec ciągu Q_l element M_{l+1} otrzymując w ten sposób ciąg Q_{l+1}.

Implementacja algorytmu

Zauważmy, że wszystkie operacje wykonywane na ciągach Q_l dotyczą elementów stojących na początku lub na końcu ciągu. Stąd Q_l możemy zaimplementować jako kolejkę o dwóch końcach (ang. *double-ended queue*, w skrócie *deque*).

W poniższym pseudokodzie tablica `R` jest przeznaczona na wartości $R_{i,j}$ dla aktualnie rozważanego i (jest uaktualniana w każdym kroku pętli), a `Q` jest kolejką o dwóch końcach, w której znajdują się indeksy elementów tworzących Q_l: $l_1, \ldots, l_m$. Liczba `k` jest kwotą, którą mamy wydać, a tablice `b` i `c` zawierają dane nominały banknotów i ich liczbę. Operacje `pop_front` i `push_front` oznaczają odpowiednio usunięcie i wstawienie elementu na początek kolejki. Analogicznie, operacje `pop_back` i `push_back` oznaczają odpowiednio usunięcie i wstawienie elementu na koniec kolejki. Funkcja `front` zwraca pierwszy element kolejki.

```
1:  R[0] := 0;
2:  R[1..k] := ∞;
3:  for i := 1 to n do
4:     for r := 0 to b[i] − 1 do
5:        Q.clear;
6:        l := 0;
7:        while r + l · b[i] ⩽ k do
8:           M[l] := R[r + l · b[i]] − l;
9:           while Q.not_empty and M[Q.back] ⩾ M[l] do Q.pop_back;
10:          Q.push_back(l);
11:          R[r + l · b[i]] := M[Q.front] + l;
12:          W[i, r + l · b[i]] := l − Q.front;
13:          if Q.front = l − c[i] then Q.pop_front;
14:          l := l + 1;
```

Dokładne odtworzenie sposobu wypłacenia kwoty k nie stanowi problemu:

```
j := k;
for i := n downto 1 do
   wypłać W[i,j] banknotów o nominale b[i];
   j := j − W[i, j] · b[i];
```

Aby oszacować złożoność pierwszej procedury przyjrzyjmy się bliżej zewnętrznej pętli `while` (wiersze 7–14). Jeśli pominiemy wewnętrzną pętlę `while` (wiersz 9), to złożoność pozostałych operacji jest stała. Natomiast sumaryczną złożoność wszystkich iteracji wewnętrznej pętli `while` (wykonywanych we wszystkich zewnętrznych pętlach `while`) możemy oszacować zauważając, że dla ustalonych i i r każde l jest raz wstawiane do kolejki, więc co najwyżej raz może zostać z niej usunięte. Dlatego całkowita złożoność zewnętrznej pętli `while` dla nominału b_i wynosi $\Theta(k/b_i)$. Tym samym złożoność czasowa algorytmu wynosi:

$$\Theta(k) + \sum_{i=1}^{n} b_i \cdot \Theta(k/b_i) = \Theta(n \cdot k).$$

Złożoność pamięciowa algorytmu jest także równa $\Theta(n \cdot k)$. Gdybyśmy jednak chcieli obliczyć tylko optymalną liczbę banknotów potrzebnych do wypłacenia kwoty, to moglibyśmy zmodyfikować algorytm tak, by działał w pamięci $\Theta(n+k)$ (nie byłaby potrzebna tablica `W`).

Przedstawione rozwiązanie zostało zaimplementowane w `ban.cpp` (z STL), `ban0.cpp` (bez STL) i `ban1.pas`.

Testy

Rozwiązania zawodników były oceniane na zestawie 18 testów. Testy 1–4 to proste testy poprawnościowe (w szczególności, testy 1–2 pozwalają sprawdzić pewne przypadki brzegowe), testy 5–7 to niewielkie testy losowe, testy 8–15 to duże losowe testy wydajnościowe, natomiast testy 16–18 to specyficzne testy wydajnościowe.

Nazwa	n	k	Opis
ban1.in	5	28	
ban2.in	1	104	
ban3.in	3	5 000	
ban4.in	10	500	
ban5.in	46	1 000	
ban6.in	40	2 000	
ban7.in	181	6 982	
ban8.in	113	10 034	
ban9.in	127	16 933	dokładnie jeden nieparzysty nominał, nieparzysta kwota
ban10.in	21	19 998	małe nominały, duża kwota
ban11.in	170	19 989	gęste rozmieszczenie nominałów na małym przedziale
ban12.in	190	19 123	
ban13.in	130	19 999	kwota daje resztę 49 z dzielenia przez 50, dokładnie jeden z nominałów daje resztę 1 (pozostałe 0), więc musi zostać użyty 49-krotnie
ban14.in	185	18 888	
ban15.in	175	20 000	banknoty wyraźnie podzielone na grupy: o małych i dużych nominałach, przy czym żadnego z dużych nie można wziąć do rozkładu wynikowego
ban16.in	15	19 999	test z maksymalną liczbą banknotów o nominałach będących potęgami 2
ban17.in	34	19 999	
ban18.in	20	19 999	

Marcin Kubica — Treść zadania, Opracowanie
Marcin Michalski — Program

OI, Etap II, dzień 1, 9–02–2005

Sumy Fibonacciego

Liczby Fibonacciego to ciąg liczb całkowitych zdefiniowany następująco: $\mathit{Fib}_0 = 1$, $\mathit{Fib}_1 = 1$, $\mathit{Fib}_i = \mathit{Fib}_{i-2} + \mathit{Fib}_{i-1}$ *(dla* $i \geqslant 2$*). Oto kilka pierwszych wyrazów tego ciągu:* $(1, 1, 2, 3, 5, 8, \ldots)$.

Wielki informatyk Bajtazar konstruuje niezwykły komputer. Liczby w tym komputerze są reprezentowane **w układzie Fibonacciego**. *Liczby w takim układzie są reprezentowane jako ciągi zer i/lub jedynek (bitów), ciąg* $(b_1, b_2, \ldots, b_n)$ *reprezentuje liczbę* $b_1 \cdot \mathit{Fib}_1 + b_2 \cdot \mathit{Fib}_2 + \cdots + b_n \cdot \mathit{Fib}_n$. *(Zwróć uwagę, że nie korzystamy z* Fib_0*.) Taka reprezentacja liczb nie jest niestety jednoznaczna, tzn. tę samą liczbę można reprezentować na wiele sposobów. Na przykład, liczbę 42 można reprezentować jako:* $(0, 0, 0, 0, 1, 0, 0, 1)$, $(0, 0, 0, 0, 1, 1, 1, 0)$ *lub* $(1, 1, 0, 1, 0, 1, 1)$. *Dlatego też Bajtazar ograniczył się wyłącznie do reprezentacji spełniających następujące warunki:*

- *jeżeli* $n > 1$, *to* $b_n = 1$, *czyli reprezentacja liczby nie zawiera wiodących zer,*
- *jeżeli* $b_i = 1$, *to* $b_{i+1} = 0$ *(dla* $i = 1, \ldots, n-1$*), czyli reprezentacja liczby nie zawiera dwóch (lub więcej) jedynek obok siebie.*

Konstrukcja komputera okazała się trudniejsza, niż Bajtazar myślał. Ma on problemy z zaimplementowaniem dodawania. Pomóż mu!

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia reprezentacje dwóch dodatnich liczb całkowitych,*
- *obliczy i wypisze reprezentację ich sumy na standardowe wyjście.*

Wejście

Na wejściu znajdują się reprezentacje Fibonacciego (spełniające podane powyżej warunki) dwóch dodatnich liczb całkowitych x *i* y *— jedna w pierwszym, a druga w drugim wierszu. Każda z tych reprezentacji jest zapisana jako ciąg nieujemnych liczb całkowitych, pooddzielanych pojedynczymi odstępami. Pierwsza liczba w wierszu to długość reprezentacji* n, $1 \leqslant n \leqslant 1\,000\,000$. *Po niej następuje* n *zer i/lub jedynek.*

Wyjście

W pierwszym i jedynym wierszu wyjścia Twój program powinien wypisać reprezentację Fibonacciego (spełniającą podane powyżej warunki) sumy $x + y$. *Tak jak to opisano dla wejścia, reprezentacja powinna mieć postać ciągu nieujemnych liczb całkowitych, pooddzielanych pojedynczymi odstępami. Pierwsza liczba w wierszu to długość reprezentacji* n, $1 \leqslant n \leqslant 1\,000\,000$. *Po niej następuje* n *zer i/lub jedynek.*

Przykład

Dla danych wejściowych:

```
4 0 1 0 1
5 0 1 0 0 1
```

poprawnym wynikiem jest:

```
6 1 0 1 0 0 1
```

Rozwiązanie

System Zeckendorfa

Opisany w zadaniu układ Fibonacciego jest znany jako system Zeckendorfa ([28], p. 6.6, [30], ćw. 1.2.8-34). System ten stanowi raczej ciekawostkę matematyczną i nie jest wykorzystywany w praktyce. Wynika to stąd, że implementacje operacji arytmetycznych są skomplikowane i mniej efektywne niż w przypadku klasycznego systemu dwójkowego.

Przekonajmy się na początek, że jest to jednoznaczny i poprawny system pozycyjny, tzn. każdą liczbę naturalną można w nim zapisać w dokładnie jeden sposób. Przez $b_k b_{k-1} \ldots b_1{}_{Fib}$ oznaczmy $\sum_{i=1}^{k} b_i \cdot Fib_i$ i niech x będzie dowolną liczbą naturalną. Możemy ją przekształcić do systemu Zeckendorfa w następujący sposób:

1. Jeśli $x \leqslant 1$, to reprezentacja x składa się tylko z jednej cyfry $x_{Fib} = x$.
2. Jeśli $x > 1$, to niech Fib_k będzie największą liczbą Fibonacciego nieprzekraczającą x. Wówczas reprezentacja x ma k cyfr, a najstarsza cyfra to $b_k = \lfloor x/Fib_k \rfloor$. Pozostałe cyfry $b_{k-1}, \ldots, b_1$ to reprezentacja liczby $x - b_k \cdot Fib_k$ uzupełniona do długości $k-1$ wiodącymi zerami.

Sprawdźmy, czy przedstawiony algorytm jest poprawny, to znaczy, czy dla każdej liczby naturalnej daje wynik i czy wynik ten spełnia warunki zadania.

Lemat 1 *Dla dowolnej liczby naturalnej x powyższy algorytm generuje taki ciąg k cyfr 0 i 1, że:*

(a) $b_k b_{k-1} \ldots b_1{}_{Fib} = x$,

(b) $b_k = 1$,

(c) jeżeli $b_i = 1$, *to* $b_{i+1} = 0$ *(dla* $i = 1, \ldots, k-1$*).*

Dowód Reprezentacja wygenerowana przez algorytm oczywiście spełnia warunek (a). Dowód dla pozostałych warunków przeprowadzimy przez indukcję względem x.

1. Dla $x \leqslant 2$ można łatwo sprawdzić, że lemat jest prawdziwy.
2. Załóżmy, że $x > 2$ i lemat jest spełniony dla wartości mniejszych niż x.

   Wiemy, że Fib_k jest największą liczbą Fibonacciego nie przekraczającą x, czyli

$$Fib_k \leqslant x < Fib_{k+1} = Fib_{k-1} + Fib_k$$

i dalej

$$x - Fib_k < Fib_{k-1}. \tag{1}$$

Stąd widzimy, iż $b_k \leqslant 1$ (bo $Fib_{k-1} < Fib_k$), a z algorytmu dodatkowo wynika, iż $b_k \geqslant 1$. Mamy więc warunek (b).

Z nierówności (1) mamy także, że $b_{k-1} = 0$, a reprezentacja $x - Fib_k$ składa się z co najwyżej $k-2$ cyfr. Z założenia indukcyjnego możemy teraz wywnioskować, że ciąg cyfr $b_k \dots b_1$ spełnia warunki (b) – (c).

Dodatkowo zauważamy, że wszystkie wygenerowane cyfry to zera i jedynki.

Na mocy zasady indukcji kończymy dowód. ■

Pokażemy teraz, że reprezentacja w systemie Zeckendorfa jest jednoznaczna.

Lemat 2 (Zeckendorf) *Dla każdej dodatniej liczby całkowitej x istnieje dokładnie jeden ciąg $b_k \dots b_1$ cyfr 0 i/lub 1 spełniający warunki (b) – (c) oraz taki, że $b_k \dots b_1{}_{Fib} = x$.*

Dowód Dowód będzie przebiegał nie wprost. Załóżmy, że istnieje dodatnia liczba całkowita, która ma dwie różne reprezentacje spełniające warunki (b) – (c) i niech x będzie najmniejszą z takich liczb. Niech $b_k \dots b_1$ i $b'_l \dots b'_1$ będą dwiema różnymi reprezentacjami x spełniającymi warunki (b) – (c).

Zauważmy, że k musi być różne od l, w przeciwnym przypadku usuwając b_k i b_l (i ewentualnie wiodące zera) uzyskalibyśmy dwie różne reprezentacje liczby $x - Fib_k$ mniejszej niż x. Przyjmijmy, że $k > l$.

Spróbujmy oszacować wartość x od góry i od dołu wiedząc, że ma reprezentację k-cyfrową i l-cyfrową. Skoro $b_k = 1$, to $x \geqslant Fib_k$. Największa liczba mająca reprezentację długości l spełniającą warunki (b) – (c), to $101010\dots10_{Fib}$ (dla l parzystego) lub $101010\dots01_{Fib}$ (dla l nieparzystego). Stąd, dla l parzystego mamy

$$\begin{aligned}
x &< x+1 \leqslant \\
&\leqslant 101010\dots1010_{Fib}+1 = \\
&= 101010\dots1011_{Fib} = \\
&= 101010\dots1100_{Fib} = \\
&\vdots \\
&= 101100\dots0000_{Fib} = \\
&= 110000\dots0000_{Fib} = \\
&= 1000000\dots0000_{Fib} = \\
&= Fib_{l+1} \leqslant Fib_k \leqslant x
\end{aligned}$$

i podobnie dla l nieparzystego mamy

$$\begin{aligned}
x &< x+1 \leqslant \\
&\leqslant 101010\dots101_{Fib}+1 = \\
&= 101010\dots110_{Fib} =
\end{aligned}$$

$$\begin{aligned} &\vdots \\ = \quad & 101100\ldots0000_{Fib} = \\ = \quad & 110000\ldots0000_{Fib} = \\ = \quad & 1000000\ldots0000_{Fib} = \\ = \quad & Fib_{l+1} \leqslant Fib_k \leqslant x \end{aligned}$$

W ten sposób doszliśmy do sprzeczności: $x < x$. Tak więc założenie, że x może mieć dwie różne reprezentacje spełniające warunki (b) – (c) jest fałszywe, co kończy dowód. ■

Prosty algorytm dodawania

Najprostszy sposób dodawania liczb w systemie Zeckendorfa polega na dodawaniu do pierwszej liczby kolejno liczb Fibonacciego odpowiadających jedynkom w reprezentacji drugiej liczby. Aby do liczby x dodać liczbę Fib_i, zwiększamy i-tą cyfrę reprezentacji x o 1. W wyniku możemy dostać:

- reprezentację spełniającą warunki (b) – (c),
- reprezentację zawierającą dwie lub trzy sąsiadujące ze sobą jedynki,
- reprezentację zawierającą dwójkę, która może sąsiadować tylko z zerami.

Eliminacja dwójek (1) Jeżeli wynik zawiera dwójkę, to możemy ją wyeliminować stosując następujące przekształcenia począwszy od najbardziej znaczących cyfr:

$$x0200y_{Fib} = x1001y_{Fib} \tag{2}$$
$$x0201y_{Fib} = x1002y_{Fib} \tag{3}$$
$$x02_{Fib} = x10_{Fib} \tag{4}$$

Zauważmy, że w przekształceniu 3 dwójka nie znika, lecz przesuwa się w prawo. Jednak przeglądając ciąg w podanym kierunku wyeliminujemy ją (lub przesuniemy dalej w prawo) w kolejnym kroku. Warto przy tym zauważyć, że nowopowstała dwójka jest również otoczona zerami lub jest ostatnią cyfrą sumy, gdyż jedynka przed operacją również była otoczona zerami. Cały proces może wymagać czasu liniowego. Na koniec otrzymujemy reprezentację, która nie zawiera dwójek, ale może zawierać sąsiadujące ze sobą jedynki.

Eliminacja sąsiednich jedynek (2) Sąsiadujące ze sobą jedynki możemy wyeliminować przekształcając reprezentację począwszy od najmniej znaczących cyfr i stosując następujące tożsamości:

$$x0\underbrace{11\ldots11}_{2l\text{ cyfr}}y_{Fib} = x\underbrace{10\ldots10}_{2l\text{ cyfr}}0y_{Fib} \tag{5}$$
$$x0\underbrace{11\ldots11}_{2l+1\text{ cyfr}}y_{Fib} = x\underbrace{10\ldots10}_{2l\text{ cyfr}}01y_{Fib} \tag{6}$$

Całe przekształcenie może wymagać czasu liniowego.

Łączny koszt dodania liczby Fib_i do liczby x jest więc liniowy, stąd opisany algorytm dodawania dwóch liczb działa w czasie kwadratowym.

Efektywny algorytm dodawania

Efektywny algorytm dodawania w systemie Zeckendorfa jest bardziej skomplikowany. Najpierw dodajemy odpowiadające sobie cyfry sumowanych liczb. W rezultacie uzyskujemy reprezentację, w której:

- mogą występować cyfry 0, 1 i 2,
- jedynki mogą sąsiadować ze sobą,
- dwójki mogą sąsiadować tylko z zerami.

Ciąg ten musimy tak przekształcić, by spełniał warunki (b) – (c).

Eliminacja sąsiednich jedynek (3) W poprzednim punkcie opisaliśmy procedurę eliminacji sąsiadujących ze sobą jedynek (Eliminacja 2). Możemy ją rozszerzyć wykorzystując również następujące tożsamości:

$$x0210y_{Fib} = x1100y_{Fib} \tag{7}$$
$$x0211y_{Fib} = x1101y_{Fib} \tag{8}$$

Operacja 7 jest konieczna, gdyż w trakcie operacji 5 i 6 skrajnie lewa nowopowstała jedynka może być koło dwójki, tworząc układ 210. Natomiast w operacji 7 powstają dwie jedynki obok siebie, które mogą utworzyć układ 211. Taki układ jest eliminowany przez operację 8, która może znów wygenerować układ 211, jednak przesunięty dalej o dwa miejsca.

W ten sposób wyeliminujemy sąsiadujące ze sobą jedynki również z reprezentacji zawierającej dwójki sąsiadujące początkowo tylko z zerami. Po eliminacji dostajemy ciąg, który nie zawiera sąsiadujących jedynek, ale nadal może zawierać dwójki — sąsiadujące tylko z zerami. Procedura wymaga jednorazowego przejrzenia ciągu począwszy od najmniej znaczących cyfr i działa w czasie liniowym.

W następnym kroku przekształcimy reprezentację tak, by nie zawierała dwójek. Efektem ubocznym przekształcenia może być ponowne pojawienie się w reprezentacji sąsiadujących ze sobą jedynek. Możemy jednak sobie na to pozwolić, bo potrafimy je wyeliminować w czasie liniowym (Eliminacja 2).

Eliminacja dwójek (4) Przekształcenie reprezentacji zaczynamy od wyeliminowania sekwencji cyfr 201 i 200. Zauważmy, iż skoro wyeliminowaliśmy sekwencje jedynek, sekwencją 201 jest w rzeczywistości sekwencją 2010 lub występuje na końcu liczby.

Przeglądamy ciąg począwszy od najbardziej znaczących cyfr i stosujemy następujące dwie tożsamości:

$$x201y_{Fib} = x112y_{Fib} \tag{9}$$
$$x200y_{Fib} = x111y_{Fib} \tag{10}$$

Tę fazę algorytmu możemy wykonać w czasie liniowym, a po jej zakończeniu dostajemy ciąg, w którym:

- mogą występować sąsiadujące jedynki,

- po lewej stronie każdej dwójki musi być zero lub ciąg przynajmniej dwóch jedynek poprzedzony zerem;
- trzy najmniej znaczące cyfry reprezentacji są różne od 201.

Kolejny krok polega na takim przekształceniu reprezentacji, żeby dwie najmniej znaczące cyfry były różne od dwójki. W tym celu wystarczy do ciągów niespełniających tego warunku zastosować najpierw tożsamość:

$$x20_{Fib} = x12_{Fib} \tag{11}$$

a następnie:

$$x0\underbrace{11\ldots11}_{2l \text{ cyfr}}2_{Fib} = x\underbrace{10\ldots10}_{2l+2 \text{ cyfr}}{}_{Fib} \tag{12}$$

$$x0\underbrace{11\ldots11}_{2l+1 \text{ cyfr}}2_{Fib} = x\underbrace{10\ldots10}_{2l+2 \text{ cyfr}}1_{Fib} \tag{13}$$

Krok ten wymaga czasu najwyżej liniowego.

Zastanówmy się jakie może być sąsiedztwo dwójek w powstałej reprezentacji. Na lewo od dwójki wciąż może być zero albo sekwencja $01\ldots1$. Zaś na prawo od dwójki:

- może powstać jedynka (w wyniku operacji 12 i 13), jednak wówczas zawsze otrzymamy sekwencję 210,
- nie mogą znajdować się dwa zera, gdyż zostały wyeliminowane w wyniku operacji 10 i nie mogły powstać w wyniku późniejszych operacji,
- może znajdować się sekwencja 011, ale nie 010 — w wyniku operacji 9 i 10 wyeliminowano wszystkie wcześniej istniejące sekwencje 201, lecz mogły powstać sekwencje 2011 (np. z sekwencji 20200 lub 20201); późniejsze operacje nie mogły wprowadzić sekwencji 2010, gdyż do tego potrzebna by była sekwencja 200, a wszystkie takie sekwencje zostały wyeliminowane przez operacje 10.

Wobec tego w wyniku dostajemy reprezentację, w której:

- na lewo od każdej dwójki może występować zero albo $01\ldots1$,
- na prawo od każdej dwójki może wystąpić sekwencja 02 albo 011, albo 10.

Z takiego ciągu możemy ostatecznie wyeliminować dwójki przeglądając cyfry od najmniej znaczącej i stosując następujące tożsamości:

$$x2011y_{Fib} = x2100y_{Fib} \tag{14}$$

$$x021y_{Fib} = x110y_{Fib} \tag{15}$$

$$x121y_{Fib} = x210y_{Fib} \tag{16}$$

W wyniku otrzymujemy reprezentację, w której nie występują dwójki, lecz mogą występować sąsiadujące jedynki. Stosujemy do niej Eliminację 2 i otrzymujemy postać spełniającą warunki (b) – (c).

Sumaryczny czas wykonania wszystkich przekształceń jest liniowo zależny od długości reprezentacji. Implementacja opisanego algorytmu znajduje się w pliku `suma.pas` na płycie CD załączonej do książki.

Testy

Rozwiązania zawodników były sprawdzane na 10 testach. Cztery spośród testów, to testy poprawnościowe, w których sumowane składniki mają do 101 cyfr. Pozostałe sześć testów, oprócz poprawności sprawdzało również wydajność rozwiązań. Zostały one tak dobrane, żeby nieefektywne algorytmy dodawania wymagały przynajmniej czasu kwadratowo zależnego od długości reprezentacji. Rozmiar danych w tych testach zwiększa się stopniowo, tak aby za ich pomocą dało się odróżnić rozwiązania o różnych złożonościach czasowych. Wszystkie testy można znaleźć na płycie CD.
W poniższej tabeli l_1 oraz l_2 oznaczają długość liczb, które należy dodać.

Nazwa	$\mathbf{l_1}$	$\mathbf{l_2}$	**Opis**
sum1.in	1	5	mały test
sum2.in	1	1	test sprawdzający poprawność stanów „końcowych”
sum3.in	50	51	kolejny prosty test sprawdzający poprawność
sum4.in	101	101	test, w którym po zsumowaniu dostajemy naprzemienny ciąg 2 i 0
sum5.in	1 001	1 001	duży test losowy
sum6.in	3 750	3 750	„złośliwy” test losowy
sum7.in	50 005	50 005	test „złośliwy”— na zmianę występują grupy 1 i 2
sum8.in	250 005	250 005	test analogiczny do poprzedniego, ale większy
sum9.in	500 005	500 005	test, w którym po zsumowaniu najpierw występują grupy 1, a później 2
sum10.in	975 005	975 005	największy test

Wojciech Rytter — Treść zadania, Opracowanie
Jakub Radoszewski — Opracowanie, Program
Marek Cygan — Program

OI, Etap II, dzień 2, 10–02–2005

Szablon

Bajtazar chce umieścić na swoim domu długi napis. W tym celu najpierw musi wykonać odpowiedni szablon z wyciętymi literkami. Następnie przykłada taki szablon we właściwe miejsce do ściany i maluje po nim farbą, w wyniku czego na ścianie pojawiają się literki znajdujące się na szablonie. Gdy szablon jest przyłożony do ściany, to malujemy od razu wszystkie znajdujące się na nim literki (nie można tylko części). Dopuszczamy natomiast możliwość, że któraś litera na ścianie zostanie narysowana wielokrotnie, w wyniku różnych przyłożeń szablonu. Literki na szablonie znajdują się koło siebie (nie ma tam przerw). Oczywiście można wykonać szablon zawierający cały napis. Bajtazar chce jednak zminimalizować koszty i w związku z tym wykonać szablon tak krótki, jak to tylko możliwe.

Zadanie

Napisz program który:

- *wczyta ze standardowego wejścia napis, który Bajtazar chce umieścić na domu,*
- *obliczy minimalną długość potrzebnego do tego szablonu,*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym i jedynym wierszu standardowego wejścia znajduje się jedno słowo. Jest to napis, który Bajtazar chce umieścić na domu. Napis składa się z nie więcej niż 500 000, oraz nie mniej niż 1 małej litery alfabetu angielskiego.

Wyjście

W pierwszym i jedynym wierszu standardowego wyjścia należy zapisać jedną liczbę całkowitą — minimalną liczbę liter na szablonie.

Przykład

Dla danych wejściowych:
`ababbababbabababbabababbababbaba`
poprawnym wynikiem jest:
`8`

```
                                                 a b a b b a b a
                                      a b a b b a b a
                        a b a b b a b a
          a b a b b a b a
a b a b b a b a

a b a b b a b a b b a b a b a b b a b a b a b b a b a b b a b a
```

Rysunek pokazuje, że szablon ababbaba może służyć do namalowania napisu z przykładu.

Rozwiązanie

Problem występujący w zadaniu jest związany z zagadnieniem wyszukiwania wzorca w tekście. Napis, który Bajtazar chce umieścić na domu, pełni rolę *tekstu* a szablon wykorzystywany do malowania to *wzorzec*. W zadaniu zależy nam na znalezieniu wzorca występującego odpowiednio gęsto w tekście (by szablon pokrył cały napis). Dodatkowo procedura wyszukiwania wzorca musi być odpowiednio efektywna, z racji dopuszczalnego rozmiaru danych.

Najprostsze rozwiązanie — złożoność czasowa $O(n^3)$

Niech x będzie napisem Bajtazara i niech $n = |x|$ będzie długością słowa x. Istnieje wiele algorytmów rozwiązania zadania o różnych złożonościach czasowych. Na początek omówimy najprostszy (a zarazem najwolniejszy) z nich.

Algorytm ten polega na sprawdzeniu wszystkich podsłów słowa x i wybraniu najkrótszego z nich, będącego szablonem dla x. Wszystkich podsłów słowa x jest $O(n^2)$. „Naiwne" sprawdzanie dla każdej pozycji w słowie x, czy występuje na niej słowo y, wymaga czasu $O(|y| \cdot n)$. Potem wystarczy sprawdzić, czy pomiędzy znalezionymi wystąpieniami nie ma luki większej niż długość słowa y. W ten sposób dostajemy algorytm działający w czasie $O(n^4)$.

W miejsce algorytmu naiwnego możemy zastosować szybszy algorytm wyszukiwania wzorca w tekście. Na przykład działający w czasie liniowo zależnym od sumy długości wzorca i tekstu (w tym przypadku w czasie $O(n)$) algorytm Knutha-Morrisa-Pratta, patrz rozdział 5 w [14]. Stosując taki algorytm uzyskujemy złożoność $O(n^3)$.

Szybszy algorytm — złożoność czasowa $O(n^2)$

Definicja 1 Dla słowa x powiemy, że y jest jego *prefiksem*, jeśli $y = x[1..i]$ dla pewnego $1 \leqslant i \leqslant n$. Powiemy, że y jest *sufiksem* słowa x, jeśli $y = x[i..n]$ dla pewnego $1 \leqslant i \leqslant n$. Prefiks (analogicznie sufiks) nazwiemy *właściwym*, jeśli jego długość jest mniejsza niż długość x.

Na potrzeby zadania wprowadzimy także pojęcie *prefikso-sufiksu x* — słowa, które jest jednocześnie prefiksem i sufiksem x.

Podstawowa prosta obserwacja, na której oprzemy rozwiązanie zadania, jest następująca.

Obserwacja 1 *Szablon napisu musi być jego prefikso-sufiksem.*

Oznaczmy przez $\mathcal{S}$ zbiór prefikso-sufiksów napisu x, które mogą potencjalnie być szablonami dla x. Aby go dokładnie opisać wykorzystamy pojęcie tablicy sufiksowej.

Definicja 2 Tablicę $P[1..n]$ nazwiemy *tablicą sufiksową* słowa x, jeśli

$$P[i] = \max\{ j \mid 0 \leqslant j < i \text{ oraz } x[1..j] = x[i-j+1..i]\} \quad \text{dla } 1 \leqslant i \leqslant n,$$

co oznacza, że $x[1..P[i]]$ jest najdłuższym właściwym prefikso-sufiksem słowa $x[1..i]$.

Istnieją algorytmy wyliczania tablicy sufiksowej w czasie liniowo zależnym od długości słowa x. Czytelnik może się z nimi zapoznać na przykład w [14]. My natomiast możemy już sformułować lemat dokładnie charakteryzujący zbiór $\mathcal{S}$.

Lemat 2

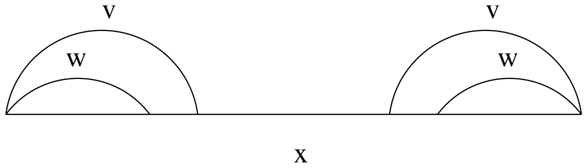
$$\mathcal{S} = \Big\{x[1..n],\ x[1..P[n]],\ x[1..P[P[n]]],\ \ldots x[1..P^k[n]]\Big\}$$

gdzie $P^k[n]$ *oznacza* $\underbrace{P[P[\cdots P[n]\cdots]]}_{k \text{ razy}}$, *a* k *jest dobrane tak, by* $P^k[n] > 0$ *oraz* $P^{k+1}[n] = 0$.

Dowód Zauważmy, że:

(i) jeżeli słowa v i w są prefikso-sufiksami słowa x oraz $|w| \leqslant |v|$, to słowo w jest prefikso-sufiksem słowa v (patrz rysunek);

(ii) jeżeli słowo w jest prefikso-sufiksem słowa v i słowo v jest prefikso-sufiksem słowa x, to w jest prefikso-sufiksem x (także patrz rysunek).



Prawdziwość lematu wykażemy przez indukcję względem długości słowa x. Dla $|x| = 1$ lub $P[n] = 0$ zbiór $\mathcal{S}$ zawiera tylko słowo x i lemat jest oczywiście prawdziwy.

Dla $|x| > 1$ i $P[n] > 0$ niech $y = x[1..P[n]]$ będzie najdłuższym właściwym prefikso-sufiksem x. Zauważmy, że dla zbioru $\mathcal{S}'$ utworzonego dla y analogicznie jak $\mathcal{S}$ dla x, zachodzi równość

$$\mathcal{S} = \mathcal{S}' \cup \{x\}.$$

Ponadto dla słowa y zachodzi założenie indukcyjne mówiące, że zbiór $\mathcal{S}'$ to wszystkie prefikso-sufiksy tego słowa.

Aby wykazać, że wszystkie elementy zbioru $\mathcal{S}$ są prefikso-sufiksami x zauważmy, że x jest w sposób oczywisty swoim prefikso-sufiksem. Także y jest prefikso-sufiksem x i z założenia indukcyjnego wiemy, że wszystkie elementy $\mathcal{S}'$ są prefikso-sufiksami słowa y, a więc z własności (ii) są także prefikso-sufiksami słowa x.

By pokazać, że każdy prefikso-sufiks x należy do zbioru $\mathcal{S}$, najpierw zauważmy, że $x \in \mathcal{S}$. Pozostałe prefikso-sufiksy słowa x są nie dłuższe niż y, więc z własności (i) widzimy, że są prefikso-sufiksami y i z założenia indukcyjnego należą do zbioru $\mathcal{S}' \subset \mathcal{S}$.

■

Teraz możemy zapisać algorytm poszukiwania szablonu, który sprawdzi każde słowo $y \in \mathcal{S}$, począwszy od najkrótszych. Sprawdzenia dokonujemy, jak poprzednio, za pomocą liniowego algorytmu wyszukiwania wzorca w tekście. Otrzymujemy algorytm działający w czasie $O(n^2)$, gdyż moc zbioru $\mathcal{S}$ jest ograniczona przez n (chociaż zbiór ten jest zazwyczaj znacznie mniejszy).

Algorytm o złożoności czasowej $O(n \log n)$

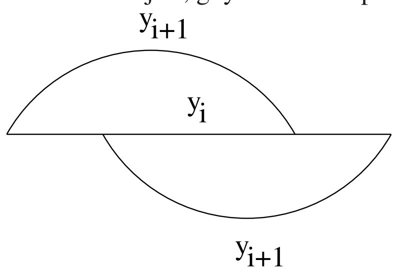
Okazuje się, że liczbę testowanych szablonów możemy istotnie ograniczyć dzięki następującemu spostrzeżeniu:

Obserwacja 3 *Niech $y_1, y_2, \ldots, y_k$ będą elementami zbioru $\mathcal{S}$ wypisanymi w takiej kolejności, że $|y_1| > |y_2| > \cdots > |y_k|$. Wówczas, jeśli dla pewnego $1 \leqslant i < k$ zachodzi nierówność*

$$\frac{|y_i|}{2} \leqslant |y_{i+1}|, \tag{1}$$

to słowo y_i nie jest najkrótszym szablonem dla napisu x.

Dowód Niech y_i oraz y_{i+1} będą elementami $\mathcal{S}$ spełniającymi powyższą nierówność. Jeśli y_i nie jest szablonem dla x, to oczywiście teza jest prawdziwa. Jeśli natomiast y_i jest szablonem dla x, to krótsze słowo y_{i+1} również nim jest, gdyż w całości pokrywa y_i, a tym samym x.



■

W ten sposób poszukując najkrótszego szablonu dla napisu x możemy rozważać tylko niektóre elementy zbioru $\mathcal{S}$. Zaczynamy od najkrótszego $y = y_k$ i jeśli nie jest on szablonem, to jako kolejnego kandydata wybieramy najkrótsze słowo z $\mathcal{S}$ o długości większej niż $2|y|$. W najgorszym razie musimy więc sprawdzić $\log n$ słów y.

Czas działania takiego algorytmu jest rzędu $O(n \log n)$ i program napisany na podstawie tego algorytmu przechodzi wszystkie testy dla zadania.

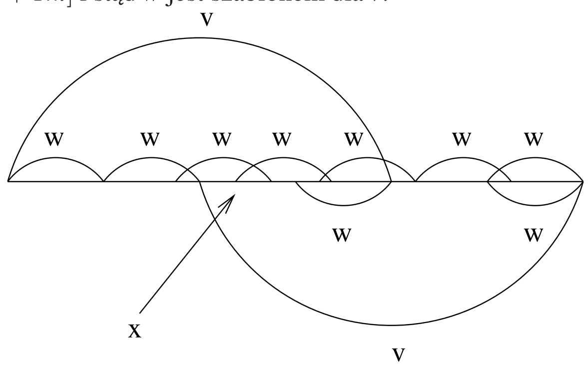
Algorytm o złożoności czasowej $O(n)$

Okazuje się, że istnieje jeszcze szybszy algorytm — pozwalający rozwiązać zadanie w czasie *liniowo* zależnym od rozmiaru danych. Polega on na wyliczaniu kolejno dla $i = 1, 2, \ldots, n$ wartości $Szablon[i]$ — długości najkrótszego szablonu dla słowa $x[1 \ldots i]$.

W algorytmie istotnie wykorzystujemy własność pokazaną w następującym lemacie:

Lemat 4 *Jeżeli słowo v jest prefikso-sufiksem słowa x i słowo w jest szablonem dla słowa x oraz $|w| \leqslant |v|$, to słowo w jest także szablonem dla słowa v.*

Dowód Niech $v = x[1..i]$. Weźmy tylko te wystąpienia szablonu w w napisie x, które w całości mieszczą się w słowie v. Zauważmy, że muszą one pokrywać słowo $v[1..i-|w|]$, bo wystąpienia w wystające poza v muszą zaczynać się na pozycjach większych niż $i - |w| + 1$. Ponadto, ponieważ w jest także prefikso-sufiksem v (z własności (i) w dowodzie lematu 2), więc $w = v[i - |w| + 1..i]$ i stąd w jest szablonem dla v.



■

W kolejnym lemacie prezentujemy główną ideę pozwalającą rozwiązać problem w czasie liniowym.

Lemat 5 *Jeżeli najkrótszy szablon słowa $v = x[1..P[n]]$ nie jest szablonem słowa $x = x[1..n]$, to słowo x ma tylko jeden szablon i jest nim całe słowo x.*

Dowód Przeprowadzimy dowód nie wprost. Oznaczmy przez s najkrótszy szablon słowa v.

Załóżmy nie wprost, że s nie jest szablonem słowa x i istnieje szablon s' słowa x, dla którego $s' \neq x$.

Rozważmy następujące przypadki, w zależności od długości słowa s':

1. Jeżeli $|s'| > |v|$, to mamy natychmiastową sprzeczność, gdyż s' — jako szablon x — jest jego prefikso-sufiksem i nie może być dłuższy od v — najdłuższego prefikso-sufiksu słowa x.

2. Jeżeli $|s| < |s'| \leqslant |v|$, to z lematu 4 (zastosowanego dla słów s', v i x) s' jest szablonem v, a więc jego prefikso-sufiksem. Ponownie wykorzystując lemat 4, tym razem dla słów s, s' i v, mamy, że słowo s jest szablonem słowa s'. Stąd jeżeli słowo x może być pokryte słowem s' jako szablonem, to może być także pokryte słowem s jako szablonem, co jest sprzeczne z założeniami lematu.

3. Jeżeli $|s'| < |s|$, to z lematu 4 mamy, że s' jest szablonem v, a to stanowi sprzeczność z założeniem, że s jest najkrótszym szablonem v.

■

Algorytm

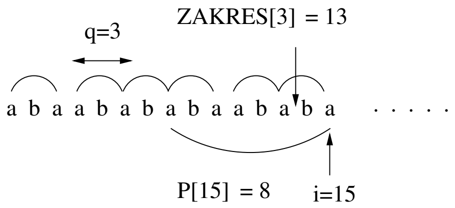
Dla każdej pozycji i w słowie x obliczymy najkrótszy szablon słowa $x[1..i]$. W tym celu sprawdzimy najkrótszy szablon słowa $x[1\ldots P[i]]$. Jeśli jest to szablon słowa $x[1..i]$, to jest to także najkrótszy szablon $x[1..i]$. W przeciwnym razie najkrótszym szablonem $x[1..i]$ jest $x[1..i]$.

Aby przyśpieszyć sprawdzanie powyższego warunku, w dodatkowej tablicy $Zakres[1\ldots n]$ na pozycji j będziemy przechowywać informacje o tym, jaki prefiks słowa x można pokryć słowem $x[1\ldots j]$ jako szablonem, czyli $Zakres[j] = k$ oznacza, iż słowo $x[1\ldots j]$ jest szablonem dla słowa $x[1\ldots k]$.

Przyjmijmy początkowo $Zakres[i] = i$, $Szablon[i] = i$, dla $0 \leqslant i \leqslant n$.

```
procedure Algorytm liniowy
 begin
   for i:=2 to n do
     s:=Szablon[P[i]];
     if P[i]> 0 and Zakres[s]⩾i−s then { warunek (1) }
       begin
         Szablon[i]:=s;
         Zakres[s]:=i;
       end
   return Szablon[n];
 end
```

Przykład



Iteracja algorytmu dla $i = 15$ oraz słowa $x = abaabababaababa\ldots$. Tuż przed rozpoczęciem tej iteracji mamy $P[i] = 8$, $q = Szablon[8] = 3$, $Zakres[3] = 13$. Rozważamy więc maksymalny prefikso-sufiks x, czyli $x[1..8]$ — da się go pokryć szablonem $x[1..Szablon[8]]$. Tym szablonem (długości 3) da się także pokryć lukę $x[Zakres[3]+1..i]$. Stąd po zakończeniu iteracji mamy $Szablon[15] = 3$, $Zakres[3] = 15$.

Dowód poprawności rozwiązania

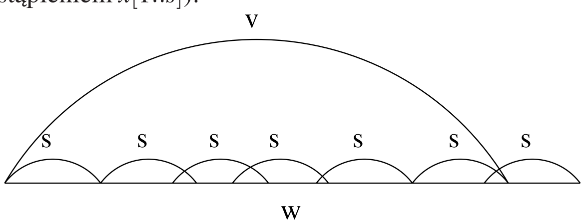
Pokażemy przez indukcję względem i, że warunek (1) instrukcji **if** w i-tej iteracji pętli **for** jest równoważny następującemu warunkowi:

$$\text{najkrótszy szablon słowa x[1\ldots P[i]] jest szablonem słowa x[1\ldots i].} \qquad (2)$$

Dla $i = 1$ żaden z warunków nie zachodzi. Załóżmy zatem, że $i > 1$.

Po pierwsze zauważmy, że z założenia indukcyjnego wynika, że dla wszystkich wartości $q < i$, jeśli $Zakres[q] = k$, to $x[1..q]$ jest rzeczywiście szablonem $x[1..k]$, bo tylko wtedy, gdy zachodzi warunek (1) jest nadawana nowa wartość tabeli *Zakres*. Stąd warunek (1) implikuje warunek (2).

Przyjmijmy, że $Szablon[P[i]] = s$. Wystarczy pokazać, że jeżeli słowo $x[1..s]$ jest szablonem $x[1..i]$, to zachodzi warunek (1), czyli $Zakres[s] \geqslant i - s$. W tym celu zauważmy, że istnieje prefiks właściwy $x[1..j]$ słowa $x[1..i]$, o długości większej lub równej $i - s$ (właściwy, czyli $j < i$), którego szablonem jest $x[1..s]$ (wystarczy wziąć $x[1..i]$ bez końcówki pokrytej tylko ostatnim wystąpieniem $x[1..s]$):



Gdyby dla $x[1..j]$ istniał szablon $x[1..s']$ krótszy niż $x[1..s]$, to stosując lemat 4 dla $x[1..s']$, $x[1..s]$ (który jako szablon jest prefikso-sufiksem $x[1..j]$) oraz $x[1..j]$ mielibyśmy, że $x[1..s']$ jest szablonem $x[1..s]$, a więc i szablonem słowa $x[1\ldots P[i]]$. To z kolei stałoby w sprzeczności z tym, że $x[1..s]$ jest najkrótszym szablonem $x[1\ldots P[i]]$.

Stąd $x[1..s]$ jest najkrótszym szablonem słowa $x[1..j]$, a więc w j-tej iteracji pętli ustaliliśmy $Zakres[s] = j$, a potem mogliśmy tę wartość już tylko zwiększyć. Czyli przy założeniu, że zachodzi warunek (2), warunek (1) także jest spełniony, co należało pokazać.

Rozwiązania wzorcowe

Na dysku dołączonym do książeczki w plikach `sza.pas` i `sza1.cpp` jest zaimplementowany algorytm działający w czasie $O(n \log n)$. Implementacja rozwiązania działającego w czasie $O(n)$ znajduje się w plikach `sza2.pas` i `sza3.cpp`.

Testy

Rozwiązania zawodników były sprawdzane na 10 testach.

Nazwa	n	k	l	Opis
sza1.in	42	4	5	aab/aaab
sza2.in	61	4	9	aabcc
sza3.in	11 168	623	3 723	aab
sza4.in	12 611	318	3 153	aaabb
sza5.in	22 816	385	3 805	yyyyx
sza6.in	437 214	2 496	19 875	aabc
sza7.in	420 008	35 002	140 003	ab
sza8.in	495 019	5 505	99 005	aaaabcdcb
sza9.in	495 020	906	99 006	aaaaabcd...zz...b
sza10.in	494 211	123 555	247 105	a

W powyższym opisie n oznacza długość napisu, k to liczba prefikso-sufiksów, a l to długość szablonu. Słowa podane w opisach to fragmenty napisów, które najczęściej w nich występują. Testy były konstruowane poprzez wielokrotne powtarzanie tych fragmentów, sporadycznie rozdzielanych innymi, krótkimi słowami.

Aby rozróżnić poszczególne rozwiązania testy musiały zawierać dużą liczbę prefikso-sufiksów, co spowodowało, że mają one w większości bardzo regularną strukturę (losowy test z bardzo dużym prawdopodobieństwem ma tylko jeden szablon, będący całym napisem). Ponadto w większości testów użyta jest mała liczba różnych liter, gdyż zwiększa to liczbę prefikso-sufiksów.

Łukasz Kowalik
Treść zadania, Opracowanie

Marcin Stefaniak, Piotr Stańczyk
Program

OI, Etap II, dzień 2, 10–02–2005

Kości

Gra w kości jest grą dwuosobową, w pełni losową. W ostatnim czasie zdobywa ona w Bajtocji rosnącą popularność. W stolicy tego kraju istnieje nawet specjalny klub dla jej wielbicieli. Bywalcy klubu przyjemnie spędzają czas rozmawiając ze sobą i od czasu do czasu rozgrywają partyjkę swojej ulubionej gry z losowo napotkanym graczem. Osoby, które wygrają najwięcej rozgrywek danego dnia, zyskują miano **szczęściarza**. *Zdarza się, że wieczór w klubie upływa w spokojnej atmosferze i rozgrywanych jest niewiele partii. Wtedy nawet jedna wygrana może wystarczyć, aby zostać szczęściarzem.*

Pewnego razu ten zaszczytny tytuł wywalczył sobie straszny pechowiec Bajtazar. Był on tym tak zaskoczony, że całkowicie zapomniał, ile partii wygrał. Zastanawia się teraz, jak wielkie było jego szczęście i czy może pech go wreszcie opuścił. Wie kto z kim i ile partii grał tego wieczora. Nie wie jednak, jakie były wyniki. Bajtazar chce wiedzieć, jaka była najmniejsza liczba wygranych partii, która mogła dać tytuł szczęściarza. Pomóż zaspokoić ciekawość Bajtazara!

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia dla każdej rozegranej partii parę uczestniczących w niej graczy,*
- *znajdzie najmniejszą liczbę k, taką że istnieje układ wyników rozgrywek, w którym każdy gracz wygrywa co najwyżej k partii,*
- *wypisze na wyjście liczbę k i wyniki wszystkich partii w znalezionym układzie.*

Wejście

W pierwszym wierszu wejścia znajduje się para liczb całkowitych n i m oddzielonych pojedynczym odstępem, $1 \leqslant n \leqslant 10\,000$, $0 \leqslant m \leqslant 10\,000$; n oznacza liczbę graczy, a m liczbę rozgrywek. Gracze są ponumerowani od 1 do n. W kolejnych m wierszach znajdują się pary numerów graczy, oddzielone pojedynczym odstępem, opisujące ciąg rozgrywek. Ta sama para może pojawiać się wiele razy w podanym ciągu.

Wyjście

Pierwszy wiersz wyjścia powinien zawierać znalezioną liczbę k. Dla każdej pary numerów graczy a, b podanej w i-tym wierszu wejścia, w i-tym wierszu wyjścia powinna pojawić się liczba 1 gdy gracz o numerze a wygrywa z graczem o numerze b w znalezionym układzie wyników rozgrywek, lub 0 w przeciwnym przypadku.

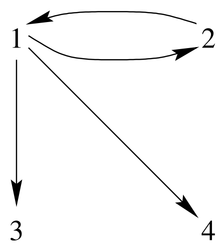
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
1
0
0
0
1
```



Rozwiązanie

Z bajtockiego na nasze

Jak to zwykle bywa w przypadku zadań olimpijskich, dobrze jest zacząć rozwiązanie zadania od przetłumaczenia jego treści z bajtockiej anegdoty na problem abstrakcyjny, wyrażony za pomocą prostych, precyzyjnie zdefiniowanych pojęć.

Już na pierwszy rzut oka widać, że graczy oraz rozgrywki można przedstawić w postaci swego rodzaju grafu, w którym wierzchołki oznaczają graczy, a krawędzie reprezentują rozgrywki. W odróżnieniu od zwykłego grafu, mamy tu jednak do czynienia z sytuacją, gdy dwa wierzchołki mogą być połączone wieloma krawędziami (bo ci sami gracze mogli grać ze sobą wiele razy). Taki graf nazywamy *multigrafem*.

W zadaniu musimy określić zwycięzców poszczególnych rozgrywek. Jak zapisać w multigrafie, że w pewnej rozgrywce gracz u wygrał z graczem v? Dość naturalnym jest przyjęcie umowy, że krawędź odpowiadająca rozgrywce, w której wygrał gracz u, będzie skierowana w kierunku gracza v (równie naturalne jest przyjęcie umowy dokładnie przeciwnej).

Definicja 1 Multigraf, w którym wszystkie krawędzie mają określony kierunek nazywamy *multigrafem zorientowanym* lub inaczej *orientacją* multigrafu. Liczbę krawędzi wychodzących z wierzchołka u będziemy nazywać *stopniem wyjściowym* u i oznaczać przez $\deg^+(u)$. Największy ze stopni wyjściowych wierzchołków multigrafu zorientowanego nazwiemy *stopniem wyjściowym multigrafu*. Orientację o stopniu wyjściowym nieprzekraczającym d będziemy nazywać *d-orientacją*.

Teraz już możemy wyrazić problem z zadania w języku grafów: *dla zadanego multigrafu poszukujemy orientacji, której stopień wyjściowy jest najmniejszy z możliwych*.

Rozwiązanie wzorcowe

Pomysł, na którym opiera się rozwiązanie wzorcowe jest bardzo prosty.

Algorytm 1 (poszukiwanie optymalnej orientacji)

1. Rozpoczynamy od znalezienia dowolnej orientacji (np. każdej krawędzi multigrafu nadajemy kierunek losowo). Taka orientacja oczywiście nie musi być optymalna. Nic to, w końcu nie od razu Kraków zbudowano.

2. Przystępujemy do *poprawiania* aktualnie posiadanej orientacji.

   (a) Znajdujemy wierzchołek v o największym stopniu wyjściowym — nazwiemy go *wierzchołkiem maksymalnym*, a jego stopień oznaczymy przez D.

   (b) Szukamy w multigrafie takiego wierzchołka w, do którego prowadzi ścieżka z wierzchołka v (możemy się na niej poruszać tylko zgodnie z kierunkiem krawędzi) i który ma stopień wyjściowy nie większy niż $D-2$. Do poszukiwania możemy użyć algorytmu przeszukiwania w głąb (patrz np. [17]).

   (c) Jeśli taki wierzchołek w nie istnieje, to kończymy algorytm. W przeciwnym razie przechodzimy do kolejnego punktu

   (d) Odwracamy kierunki wszystkich krawędzi na znalezionej ścieżce z v do w. Zauważmy, że w ten sposób:

      - zmniejszyliśmy stopień wyjściowy wierzchołka v o 1 (jest teraz równy $D-1$),
      - zwiększyliśmy stopień wyjściowy wierzchołka w o 1 (może być teraz równy najwyżej $D-1$),
      - nie zmieniliśmy stopni wyjściowych pozostałych wierzchołków.

      A więc jesteśmy odrobinę bliżej celu: albo udało nam się zmniejszyć stopień wyjściowy multigrafu, albo chociaż zmniejszyliśmy liczbę wierzchołków o stopniu wyjściowym D.

   (e) Powracamy do punktu 2(a).

Widzimy, że algorytm kończy się, gdy wszystkie wierzchołki, do których można dojść z wierzchołka maksymalnego v mają stopnie wyjściowe równe $\deg^+(v)$ lub $\deg^+(v)-1$. Otrzymaną wówczas orientację multigrafu uważamy za optymalną.

Poprawność rozwiązania wzorcowego

Jeśli nie jesteście przekonani, że orientacja otrzymana po zakończeniu algorytmu jest orientacją o najmniejszym możliwym stopniu wyjściowym, to macie rację — to nie jest oczywiste. Widzimy co prawda, że orientacji tej nie można już poprawić naszą metodą, ale może istnieje lepsze rozwiązanie, które znajduje się zupełnie inaczej? Okazuje się, że jednak nie istnieje, a wynika to z poniższego lematu.

Lemat 1 *Niech H będzie orientacją multigrafu G oraz niech v będzie wierzchołkiem H o największym stopniu wyjściowym. Jeśli wszystkie wierzchołki, do których można dojść z v w multigrafie zorientowanym H mają stopnie wyjściowe większe od $\deg^+(v)-2$, to stopień wyjściowy każdej orientacji multigrafu G jest równy co najmniej $\deg^+(v)$.*

Dowód Niech W będzie zbiorem tych wierzchołków, do których można dojść w multigrafie H z wierzchołka v. Załóżmy, że wszystkie wierzchołki W mają stopnie wyjściowe większe od $\deg^+(v)-2$.

Zauważmy, że każda krawędź grafu H wychodząca z wierzchołka należącego do zbioru W wchodzi do wierzchołka, który także należy do zbioru W (tzn. jeśli $u \in W$ oraz (u,w) jest krawędzią w H, to $w \in W$). Dodatkowo z każdego wierzchołka ze zbioru W wychodzi co najmniej $\deg^+(v)-1$ krawędzi, a z wierzchołka v (również należącego do zbioru W) wychodzi $\deg^+(v)$ krawędzi. Stąd liczba krawędzi łączących wierzchołki ze zbioru W jest równa co najmniej $(\deg^+(v)-1)\cdot|W|+1$.

Przypuśćmy, że dla multigrafu G istnieje orientacja J, której stopień wyjściowy nie przekracza $\deg^+(v)-1$. Policzmy ponownie krawędzie przebiegające pomiędzy wierzchołkami zbioru W. Tym razem otrzymujemy, że jest ich najwyżej $(\deg^+(v)-1)\cdot|W|$, gdyż z każdego spośród $|W|$ wierzchołków wychodzi najwyżej $(\deg^+(v)-1)$ krawędzi (na dodatek w orientacji J nie wszystkie krawędzie wychodzące z wierzchołków zbioru W muszą prowadzić do W, ale to tylko oznacza, że nasze szacowanie może być zawyżone). W ten sposób doszliśmy do wniosku sprzecznego z otrzymanym wcześniej ograniczeniem, stąd założenie o stopniu wyjściowym orientacji J było błędne i musi on wynosić co najmniej $\deg^+(v)$. ■

Efektywność rozwiązania wzorcowego

Jak dotąd przekonaliśmy się, że jeśli nasz algorytm zakończy się, to zwróci poprawny wynik. Teraz czas na uzasadnienie, że algorytm rzeczywiście zakończy się i to w miarę szybko. Niech $H_0, H_1, \ldots$ będą kolejnymi orientacjami grafu G wygenerowanymi po kolejnych poprawach i niech $D_0, D_1, \ldots$ oznaczają stopnie wyjściowe tych orientacji. Przez $\deg_i^+(w)$ oznaczymy stopień wyjściowy wierzchołka w w orientacji H_i. Zauważmy, że prawdziwe jest następujące spostrzeżenie:

Lemat 2 *Dla dowolnego wierzchołka w multigrafu G, jeśli w jest maksymalny w pewnej orientacji H_j, to dla każdej orientacji H_i, gdzie $j < i$, zachodzi* $\deg_i^+(w) \geqslant D_i - 1$.

Dowód Udowodnimy przez indukcję względem i, że wszystkie wierzchołki, które były maksymalne w orientacji H_j dla $j < i$, w orientacji H_i mają stopień wyjściowy równy $D_i - 1$ lub D_i. Oczywiście dla $i = 0$ lemat jest spełniony, bo nie istnieją wierzchołki, które byłyby maksymalne we wcześniejszych fazach.

Rozważmy teraz $i > 0$ oraz wierzchołek w, który był kiedyś maksymalny, i niech $j < i$ będzie maksymalnym indeksem orientacji, w której w był maksymalny.

Przypadek 1. Jeżeli $j = i-1$, to w poprzedniej fazie algorytmu w był wierzchołkiem maksymalnym i albo poprawiając orientację H_j obniżyliśmy jego stopień wyjściowy o 1 (przy czym stopień wyjściowy orientacji mógł zmaleć lub pozostać niezmieniony), albo obniżyliśmy stopień wyjściowy innego wierzchołka, a $\deg_{i-1}^+(w) = \deg_i^+(w)$ (podobnie $D_{i-1} = D_i$). W obu przypadkach zachodzi $deg_i^+(w) \geqslant D_i - 1$.

Przypadek 2. Jeżeli $j < i-1$, to z założenia indukcyjnego i sposobu, w jaki zdefiniowaliśmy j, mamy $\deg_{i-1}^+(w) = D_{i-1} - 1$. Stąd w czasie poprawiania orientacji H_{i-1} stopień wyjściowy wierzchołka w nie uległ zmianie i $\deg_i^+(w) = D_i$ (gdy obniżyliśmy stopień wyjściowy orientacji przechodząc z H_{i-1} do H_i) lub $deg_i^+(w) = D_i - 1$ (gdy poprawa nie zmniejszyła stopnia wyjściowego orientacji). ■

Z lematu wynika, że jeśli wierzchołek w stał się maksymalny w pewnej orientacji H_j, to od tego czasu *jego stopień wyjściowy nigdy nie rośnie*. W procesie poprawiania orientacji H_i dla $i \geqslant j$ wzrasta bowiem tylko stopień wierzchołka o stopniu równym najwyżej $D_i - 2$. A *ile razy może maleć* stopień wyjściowy w? Na pewno mniej niż wynosi jego stopień w multigrafie G. Tymczasem podczas każdej poprawy orientacji maleje stopień wyjściowy jednego wierzchołka maksymalnego. Stąd początkową orientację możemy poprawiać najwyżej $\sum_v \deg(v)$ razy.

Powyższą sumę możemy łatwo policzyć. Wystarczy zauważyć, że każdą krawędź multigrafu G liczymy w niej dwa razy[1], a więc $\sum_v \deg(v) = 2m$, jeśli przez m oznaczymy liczbę krawędzi multigrafu G. Stąd wiemy, że algorytm wykonuje $O(m)$ razy poprawę orientacji.

Pozostaje już tylko zastanowić się, ile czasu wymaga jedna poprawa. Algorytm wyszukiwania ścieżki (przeszukiwanie w głąb) zajmuje czas $O(m+n)$, gdzie n jest liczbą wierzchołków grafu. W podobnym czasie możemy znaleźć wierzchołek maksymalny obliczając stopnie wyjściowe wszystkich wierzchołków przy zadanej orientacji.

Choć nie wpłynie to znacząco na sumaryczną złożoność algorytmu, możemy wierzchołki maksymalne znajdować efektywniej. W tym celu liczymy raz, na początku, stopnie wyjściowe wszystkich wierzchołków i tworzymy tablicę d, w której na pozycji i mamy listę wierzchołków stopnia i. Dzięki temu wyszukiwanie wierzchołka największego stopnia zajmuje czas stały — po prostu wybieramy pierwszy element z odpowiedniej listy. Musimy tylko pamiętać, aby po każdej poprawie uaktualniać wartość największego stopnia (gdy lista wierzchołków dotychczas maksymalnych stanie się pusta) oraz żeby przenieść do odpowiednich list wierzchołki, których stopnie wyjściowe uległy zmianie.

Podsumowując, cały algorytm działa w czasie $O(m \cdot (m+n))$, czyli kwadratowo zależnym od liczby krawędzi multigrafu.

Rozwiązanie z przyszłością

Problem można rozwiązać również w inny sposób — dostrzegając w nim specjalny przypadek zagadnienia *maksymalnego przepływu w sieciach*. Problem przepływu w sieciach ma bardzo naturalne sformułowanie i znajduje szereg różnorodnych zastosowań.

Dane dla problemu to graf skierowany $G = (V, E)$, w którym z każdą krawędzią (u, v) związana jest liczba $c(u, v)$ nazywana *przepustowością krawędzi*. Przyjmujemy, że $c(u, v) = 0$, gdy $(u, v) \notin E$. Dwa wierzchołki grafu są wyróżnione: pierwszy oznaczamy przez s i nazywamy *źródłem*, a drugi przez t i nazywamy *ujściem*.

Definicja 2 *Przepływem* w grafie G nazywamy funkcję $f : V \times V \to \mathbf{R}$ spełniającą następujące warunki:

- dla wszystkich $u, v \in V$ zachodzi $f(u, v) \leqslant c(u, v)$,
- dla wszystkich $u, v \in V$ zachodzi $f(u, v) = -f(v, u)$,
- dla wszystkich $u \in V \setminus \{s, t\}$ zachodzi $\sum_{v \in V} f(u, v) = 0$.

Wartością przepływu nazywamy liczbę $|f| = \sum_{v \in V} f(s, v)$.

[1] Fakt ten nosi nazwę *lematu o uściskach dłoni*, patrz [26]

Nieformalnie, możemy myśleć o naszym grafie jak o systemie kanałów, w których płynie woda. Pierwszy warunek mówi, że przez krawędź nie może przepłynąć więcej wody niż wynosi jej przepustowość. Warunki drugi i trzeci mówią, że do każdego wierzchołka (oprócz s i t) wpływa tyle samo wody co z niego wypływa. Innymi słowy, woda wypływa ze źródła i spływa do ujścia, a w pozostałych miejscach nie może jej ani przybyć, ani ubyć. Zgodnie z naszą interpretacją wartość przepływu to ilość wody płynącej w sieci kanałów (czyli wypływającej ze źródła).

Zadanie polega na znalezieniu przepływu o maksymalnej wartości. Szczegółowy opis problemu można znaleźć w książce [17]. Znajduje się tam również szereg bardzo interesujących algorytmów dla tego zagadnienia. Najprostszy z nich, algorytm Forda-Fulkersona, działa w czasie $O(|E| \cdot |f^*|)$, gdzie f^* jest maksymalnym przepływem, przy założeniu, że wszystkie przepustowości krawędzi są liczbami całkowitymi. Dodatkowo, dla grafu o całkowitych przepustowościach algorytm znajduje przepływ całkowitoliczbowy, w którym dla dowolnych wierzchołków u, v liczba $f(u, v)$ jest całkowita.

Pokażemy teraz jak można *zredukować* problem znajdowania d-orientacji multigrafu do znajdowania maksymalnego przepływu w grafie. Redukcja problemu A do problemu B polega na takim przedstawieniu danych dla problemu A, by można było do niego zastosować algorytm rozwiązywania problemu B i z otrzymanego wyniku prosto i szybko uzyskać wynik dla problemu A. Nasza redukcja będzie więc polegać na przedstawieniu multigrafu w postaci sieci w ten sposób, by ze znalezionego optymalnego przepływu w tej sieci można było odtworzyć d-orientację multigrafu.

Rozważmy multigraf $M = (V_M, E_M)$ i utwórzmy sieć $G_d = (V, E)$. Zdefiniujmy

$$V = V_M \cup \{s, t\} \cup \{v_{xy} : E_M \text{ zawiera choć jedną krawędź } xy\},$$

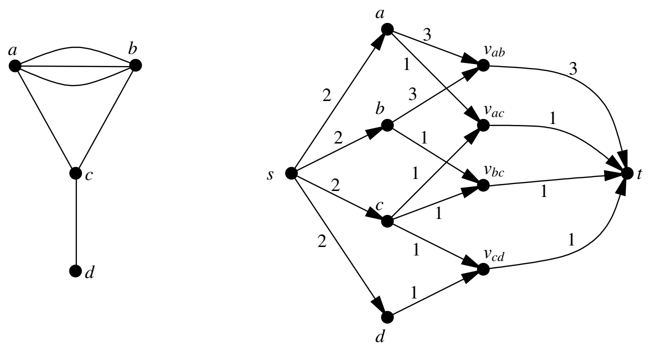
czyli V zawiera wierzchołki zbioru V_M, dwa nowe wierzchołki s (źródło) i t (ujście) oraz po jednym wierzchołku dla każdej pary wierzchołków x, y połączonych w M choć jedną krawędzią. Teraz czas na opisanie zbioru krawędzi E. W zbiorze tym umieszczamy krawędź (s, x) o przepustowości d dla każdego wierzchołka $x \in V_M$. Następnie dorzucamy krawędzie: (x, v_{xy}), (y, v_{xy}), oraz (v_{xy}, t) dla każdej pary wierzchołków x, y, połączonych choć jedną krawędzią w M — wszystkim tym krawędziom przypisujemy przepustowość równą liczbie krawędzi łączących wierzchołki x i y w multigrafie M.

Na rysunku jest przedstawiony przykład multigrafu i sieci utworzonej na jego podstawie (dla $d = 2$).

Zauważmy, że prawdziwy jest następujący lemat.

Lemat 3 *Multigraf M ma d-orientację wtedy i tylko wtedy, gdy maksymalny przepływ w sieci G_d wynosi $|E_M|$.*

Dowód Załóżmy najpierw, że multigraf M ma d-orientację H. Zauważmy, że maksymalny przepływ w sieci G_d nie może przekraczać $|E_M|$, gdyż taka jest suma przepustowości krawędzi wchodzących do ujścia t. Teraz pokażemy, jak można skonstruować przepływ w G_d o wartości $|E_M|$. Zaczynamy od przepływu zerowego f (przypisujemy $f(u, v) = 0$ wszystkim krawędziom G_d). Potem dla każdej krawędzi (x, y) skierowanej w orientacji H od x do y powiększamy o 1 wartości przepływu na krawędziach: (s, x), (x, v_{xy}) oraz (v_{xy}, t). Zauważmy, że taka modyfikacja nie narusza warunków poprawności przepływu z definicji 2 (w szczególności nie przekraczamy przepustowości krawędzi (s, x)) i jednocześnie powoduje



Multigraf (z lewej) i sieć przepływowa służąca do wyznaczania jego 2-orientacji (z prawej — liczby oznaczają przepustowości krawędzi).

wzrost wartości przepływu o 1. Po przeprowadzeniu jej dla wszystkich krawędzi multigrafu M otrzymujemy przepływ o wartości $|E_M|$.

Teraz przyjmijmy, że w grafie G_d istnieje przepływ całkowitoliczbowy f o wartości $|E_M|$. Na jego podstawie skonstruujemy d-orientację multigrafu M. W tym celu rozważmy dowolną parę wierzchołków x, y, które są połączone w M, i oznaczmy przez k liczbę krawędzi łączących te wierzchołki. Skoro przepływ ma wartość $|E_M|$, to oczywiście wszystkie krawędzie wchodzące do ujścia t muszą być „pełne" i stąd $f(v_{xy},t) = k$. W takim razie z drugiego i trzeciego warunku z definicji 2 zastosowanych dla wierzchołka v_{xy} wynika, że $f(x,v_{xy}) + f(y,v_{xy}) = k$. Wybierzmy dowolne $f(x,v_{xy})$ spośród k krawędzi łączących x i y i zorientujmy je jako wychodzące z x, a pozostałe $f(y,v_{xy})$ krawędzi jako wychodzące z y. W ten sposób określimy kierunek wszystkich krawędzi M, a stopień wyjściowy żadnego wierzchołka nie będzie przekraczał d, co wynika stąd, iż do każdego wierzchołka $x \in V_M$ wpływa najwyżej d jednostek z wierzchołka s (definiując G_d określiliśmy $f(s,v) = d$), a więc najwyżej tyle jednostek może wypływać i $f(x,v_{xy}) \leqslant d$. ■

Z lematu 3 dowiadujemy się, jak dla pewnej liczby d sprawdzić, czy multigraf M ma d-orientację, a jeśli ją ma, to jak ją znaleźć.

Algorytm 2 (znajdowanie d-orientacji)

1. Budujemy graf G_d.
2. Za pomocą algorytm Forda-Fulkersona znajdujemy maksymalny przepływ f w sieci G_d.
3. Jeśli $|f| = |E_M|$, to w sposób podany w dowodzie lematu konstruujemy z przepływu f d-orientację. W przeciwnym razie stwierdzamy, że multigraf M nie ma d-orientacji.

Całość działa w czasie $O(|E| \cdot |E_M|) = O((|E_M| + |V_M|) \cdot |E_M|)$. Pozostaje jeszcze odszukanie optymalnej (minimalnej) wartości d, dla której istnieje d-orientacja multigrafu M. Zauważmy, że liczba d musi należeć do przedziału $[1, |E_M|]$, więc zastosujemy wyszukiwanie binarne w tym przedziale.

Algorytm 3 (znajdowanie optymalnej orientacji)

1. Definiujemy przedział poszukiwań $[l, p] = [1, |E_M|]$.
2. Podstawiamy $d = \lfloor (l+p)/2 \rfloor$ i uruchamiamy Algorytm 2.
3. Jeśli algorytm 2 zwraca d-orientację multigrafu, to za przedział poszukiwań podstawiamy $[l, p] = [l, \lfloor (l+p)/2 \rfloor]$. W przeciwnym razie musimy szukać w przedziale $[l, p] = [\lfloor (l+p)/2 \rfloor + 1, p]$.
4. Jeśli poszukiwanie zostało zawężone do przedziału zawierającego jedną wartość, to kończymy algorytm. W przeciwnym razie wracamy do punktu 2.

Widzimy, że wystarczy uruchomić algorytm 2 najwyżej $\log_2(|E_M|)$ razy, by znaleźć optymalną wartość d. To daje nam końcowy algorytm o złożoności $O((|E_M| + |V_M|) \cdot |E_M| \cdot \log |E_M|)$.
Zadanie (ciekawe i nietrudne): Spróbuj, Drogi Czytelniku, sformułować algorytm znajdowania optymalnej d-orientacji, który wywoła algorytm Forda-Fulkersona jedynie $O(\log d^*)$ razy, gdzie d^* jest poszukiwanym maksymalnym stopniem wyjściowym orientacji.

Stosując metodę przepływów uzyskaliśmy w efekcie algorytm o nieco gorszej złożoności niż rozwiązanie wzorcowe. Jest to jednak podejście z przyszłością. Jeśli bowiem zamiast algorytmu Forda-Fulkersona zastosujemy bardziej zaawansowany algorytm[2], to można pokazać, że operacja znajdowania maksymalnego przepływu zajmie w tym szczególnym przypadku jedynie czas $O(|E_M|^{3/2})$.

Testy

Wygenerowano 26 testów, wszystkie z użyciem generatora liczb pseudolosowych. W każdym teście kolejność partii i graczy została pseudolosowo wymieszana. Większość testów została zgrupowana po dwa testy. Multigrafy zdefiniowane w testach można podzielić na cztery kategorie:

(1) graf dwudzielny, niesymetryczny: jedna część (wielkości rzędu $\sqrt{n}$) mała i gęsta, druga część duża i rzadka,

(2) jedna długa ścieżka zakończona gęstym kłębkiem,

(3) wielokrotnie powtórzony układ trzech wierzchołków, z których jeden jest połączony potrójnymi krawędziami z dwoma pozostałymi,

(4) graf jednorodnie losowy.

W poniższej tabeli liczby n i m oznaczają odpowiednio liczbę zawodników i rozgrywek (wierzchołków — V_M i krawędzi V_M multigrafu), natomiast d oznacza minimalną liczbę wygranych gwarantującą tytuł szczęściarza.

[2] Mowa tu o algorytmie Dinica, opisanym w książce [20]

Nazwa	n	m	d	Opis
kos1a.in	5	8	2	rodzaj (1)
kos1b.in	90	180	2	rodzaj (3)
kos2a.in	100	100	2	rodzaj (1)
kos2b.in	300	600	2	rodzaj (3)
kos3a.in	100	1 000	12	rodzaj (1)
kos3b.in	498	996	2	rodzaj (3)
kos4a.in	1 000	1 000	3	rodzaj (1)
kos4b.in	699	1 398	2	rodzaj (3)
kos5a.in	1 000	10 000	12	rodzaj (1)
kos5b.in	999	1 998	2	rodzaj (3)
kos6a.in	10 000	10 000	3	rodzaj (1)
kos6b.in	1 500	3 000	2	rodzaj (3)
kos7a.in	10	30	5	rodzaj (2)
kos7b.in	100	300	4	rodzaj (4)
kos8a.in	100	200	11	rodzaj (2)
kos8b.in	100	1 000	10	rodzaj (4)
kos9a.in	100	1 000	91	rodzaj (2)
kos9b.in	50	10 000	200	rodzaj (4)
kos10a.in	1 000	1 000	2	rodzaj (2)
kos10b.in	100	3 000	30	rodzaj (4)
kos11a.in	1 000	10 000	91	rodzaj (2)
kos11b.in	1 000	8 000	9	rodzaj (4)
kos12a.in	10 000	10 000	2	rodzaj (2)
kos12b.in	4 000	10 000	3	rodzaj (4)
kos13.in	10 000	9 999	1	duże grafy rzadkie
kos14.in	10 000	10 000	1	duże grafy rzadkie

Zawody III stopnia

opracowania zadań

Jakub Radoszewski — Treść zadania, Opracowanie
Marek Cygan — Program

OI, Etap III, dzień próbny, 5–04–2005

Dziuple

W Bajtocji rosną dwa bardzo wysokie pionowe drzewa, a w każdym z nich są wydrążone jedna pod drugą dziuple dla ptaków. Pewnego dnia w dziuplach postanowiło zamieszkać n bardzo szybkich ptaszków. Niektóre ptaszki znają się wzajemnie, więc po wprowadzeniu się chciałyby mieć możliwość odwiedzania się nawzajem w swoich dziuplach. Ptaszki latają bardzo szybko i zawsze po liniach prostych. Chcąc uniknąć niebezpieczeństwa zderzenia postanowiły rozlokować się w dziuplach w taki sposób, żeby:

- *każde dwa ptaszki, które chciałyby się odwiedzać, mieszkały na różnych drzewach oraz*
- *dla dowolnych dwóch par znajomych ptaszków, odcinki łączące dziuple znajomych ptaszków nie przecinały się (co najwyżej mogą mieć wspólny koniec).*

Dodatkowo, ptaszki chcą mieszkać jak najniżej. Na każdym z drzew zajmują więc pewną liczbę dolnych dziupli. W każdym z drzew jest więcej dziupli niż wszystkich ptaszków.

Jak wiadomo, ptaszki mają niewielkie rozumki. Dlatego też poprosiły Cię — znanego ornitologa — o pomoc w sprawdzeniu, na ile różnych sposobów mogą rozlokować się w dziuplach.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis relacji znajomości wśród ptaszków,*
- *obliczy, na ile różnych sposobów można rozmieścić ptaszki w dziuplach spełniając podane powyżej wymagania,*
- *wypisze wynik na standardowe wyjście.*

Wejście

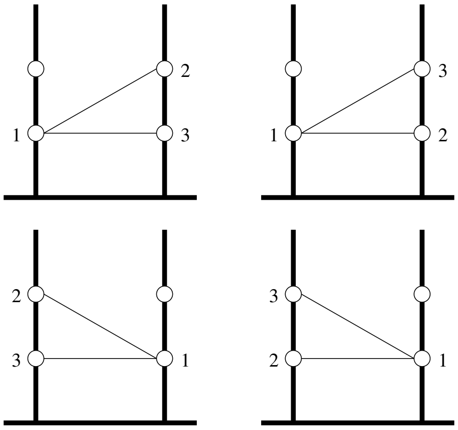
W pierwszym wierszu wejścia zapisano trzy liczby całkowite n, m oraz k, oznaczające odpowiednio: liczbę ptaszków, liczbę różnych par ptaszków znających się nawzajem oraz liczbę której należy użyć przy podawaniu wyniku (por. p. Wyjście), $2 \leqslant n \leqslant 1\,000\,000$, $1 \leqslant m \leqslant 10\,000\,000$, $2 \leqslant k \leqslant 2\,000\,000$. Ptaszki są ponumerowane od 1 do n. W kolejnych m wierszach podane są pary znających się nawzajem ptaszków, po jednej parze w wierszu. W $(i+1)$-szym wierszu są zapisane dwie liczby całkowite a_i i b_i oddzielone pojedynczym odstępem, $1 \leqslant a_i, b_i \leqslant n$, $a_i \neq b_i$. Są to numery znajomych ptaszków. Każda (nieuporządkowana) para znajomych ptaszków jest podana dokładnie raz.

Wyjście

Niech r będzie liczbą różnych rozmieszczeń ptaszków w dziuplach, spełniających podane warunki. Twój program powinien wypisać w pierwszym wierszu wyjścia jedną liczbę całkowitą: resztę z dzielenia r przez k. Jeżeli nie istnieje szukane rozmieszczenie ptaszków, to poprawnym wynikiem jest 0.

Przykład

Dla danych wejściowych:
```
3 2 10
1 2
1 3
```
poprawnym wynikiem jest:
```
4
```



Rozwiązanie

Grafy pladzielne

Rozpoczniemy od sformułowania problemu z zadania w języku teorii grafów. W tym celu zbudujemy *graf znajomości*, w którym wierzchołki to ptaszki. Wierzchołki połączymy w grafie krawędzią (nieskierowaną), jeśli odpowiadające im ptaszki znają się wzajemnie. Naszym zadaniem jest rozmieszczenie grafu na dwóch drzewach — ponieważ słowo drzewo ma swoje znaczenie w języku grafów, więc by nie mylić drzew prawdziwych z drzewami-grafami przyjmiemy od tej pory, że drzewa z dziuplami, w których chcą osiedlić się ptaszki to *lipy*.

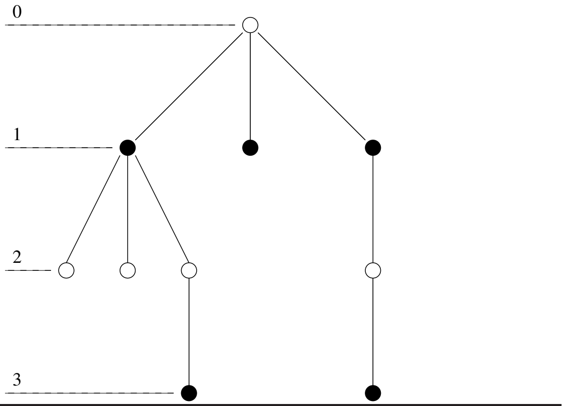
Definicja 1 Łatwo zauważyć, że nie każdy graf da się rozmieścić na dwóch lipach. Aby było to możliwe, graf musi być:

- dwudzielny — wierzchołki (ptaszki) trzeba podzielić na dwa zbiory (lipy) tak, by nie było krawędzi łączących wierzchołki należące do tego samego zbioru (tzn. ptaszki mieszkające na tej samej lipie nie mogą się znać i odwiedzać);
- w pewien sposób planarny — graf trzeba narysować w ten sposób, że wierzchołki z każdego zbioru (lipy) są rozmieszczone na prostych równoległych, krawędzie są narysowane jako odcinki i w takiej reprezentacji żadne dwie krawędzie nie przecinają się poza wierzchołkami.

Na potrzeby opisu rozwiązania graf spełniający powyższe warunki nazwiemy *grafem pladzielnym*.

Spróbujmy scharakteryzować grafy pladzielne dokładniej. Nietrudno zauważyć, że graf pladzielny nie może zawierać cykli — w przeciwnym wypadku nie dałoby się go narysować bez przecinających się krawędzi. Dlatego musi być *lasem* (w sensie grafowym — czyli grafem acyklicznym), a więc każda jego spójna składowa musi być *drzewem* (czyli acyklicznym grafem spójnym).

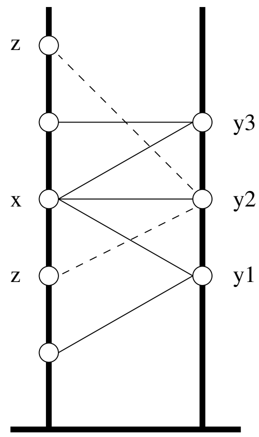
Sprawdźmy, czy każde drzewo może być składową spójności grafu pladzielnego. Każde drzewo jest grafem dwudzielnym. Wystarczy wybrać dowolny wierzchołek w grafie i nazwać go *korzeniem* (w ten sposób *ukorzeniamy* drzewo), a następnie pokolorować na biało wszystkie wierzchołki, których odległość od korzenia jest parzysta, i na czarno pozostałe (takie kolorowanie jest przedstawione na rysunku 1). W ten sposób dzielimy wierzchołki drzewa na dwie grupy takie, że wewnątrz żadnej grupy nie ma krawędzi, co dowodzi dwudzielności tego grafu[1].



Rys. 1: Dwukolorowanie drzewa pozwalające wykazać, że drzewo jest dwudzielne.

Niestety, nie każde drzewo jest grafem pladzielnym. Zauważmy, że problem stwarza sytuacja, gdy jeden wierzchołek x ma trzech sąsiadów y_1, y_2 i y_3, z których każdy ma stopień (czyli liczbę sąsiadów) co najmniej dwa (patrz rysunek 2). Rozlokowując pladzielnie takie wierzchołki na dwóch lipach musimy umieścić x na jednej lipie, a y_1, y_2, y_3 na drugiej — przypuśćmy, że rozmieścimy je w takim właśnie porządku: y_1 najniżej, y_2 pośrodku, a y_3 najwyżej. Niech z będzie sąsiadem y_2 różnym od x. Wierzchołek z musimy umieścić na tej samej lipie co x — i tu mamy problem. Jeśli umieścimy go poniżej x, to krawędź (y_2, z) będzie przecinać się z (y_1, x). Jeśli umieścimy go powyżej x, to wtedy krawędź (y_2, z) będzie przecinać się z (y_3, x). Można sprawdzić, że zmiana porządku y_1, y_2, y_3 nic tu nie pomoże.

[1] Więcej informacji na temat własności drzew i lasów można znaleźć na przykład w książce [26].



Rys. 2: Próby rozmieszczenia drzewa niepladzielnego na dwóch lipach.

W ten sposób doszliśmy do wniosku, że aby drzewo było pladzielne, jego wierzchołki stopnia większego niż jeden (nazywane *wierzchołkami wewnętrznymi*) muszą tworzyć ścieżkę — nazwiemy ją *osią* tego drzewa.

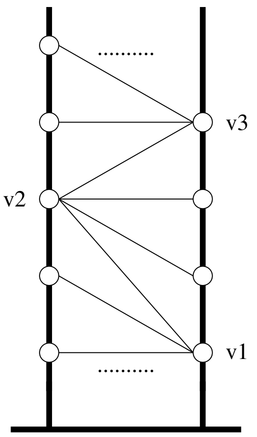
Lemat 1 *Drzewa, w których wierzchołki wewnętrzne tworzą jedną ścieżkę są jedynymi spójnymi grafami pladzielnymi.*

Dowód Pokazaliśmy już, że inne spójne grafy nie są pladzielne. Pozostaje pokazać, że opisane drzewo da się rozmieścić na dwóch lipach. Niech $(v_1, v_2, \ldots, v_t)$ będzie osią drzewa. Pozostałe wierzchołki drzewa — o stopniu jeden (nazywane *liśćmi*) — można podzielić na zbiory sąsiadów wierzchołków z osi: $S(v_1), S(v_2), \ldots, S(v_t)$. Widzimy, że drzewo można rozmieścić planarnie na dwóch lipach. Wystarczy:

- wierzchołki osi o nieparzystych indeksach $(v_1, v_3, \ldots)$ umieścić kolejno od dołu na jednej lipie;
- wierzchołki osi o parzystych indeksach $(v_2, v_4, \ldots)$ umieścić kolejno od dołu na drugiej lipie;
- sąsiadów v_i, czyli $S(v_i)$, umieścić na przeciwnej lipie niż v_i pomiędzy v_{i-1} oraz v_{i+1}.

■

W ten sposób stworzyliśmy pełną charakteryzację składowych grafu pladzielnego.



Rys. 3: Rozmieszczenie drzewa pladzielnego na dwóch lipach.

Zliczanie rozmieszczeń ptaszków w dziuplach

Wiemy już jak rozpoznać graf pladzielny i jak rozmieścić go na lipach przynajmniej na jeden sposób. Przystąpmy teraz do policzenia wszystkich możliwości.

Podzielmy nasz graf na spójne składowe — wiemy już, że są to drzewa pladzielne. Teraz osobno rozpatrzymy *duże* spójne składowe mające więcej niż jeden wierzchołek i co najmniej jedną krawędź (powiedzmy, że jest to p pierwszych składowych: T_1, T_2,..., T_p) i *małe* składowe — wierzchołki izolowane (powiedzmy, że jest to s kolejnych składowych X_1, X_2,..., X_s).

Rozmieszczenie dużej składowej

Rozpoczniemy od wyznaczenia liczby możliwych rozmieszczeń składowej T_i na dwóch lipach. Niech $(v_1, v_2, \ldots, v_t)$ będzie osią drzewa, a $S_1, S_2, \ldots, S_t$, zbiorami sąsiadów stopnia jeden kolejnych wierzchołków z osi. Zauważmy, że:

- jeśli $t = 1$ to oś drzewa można rozmieścić na lipach na dwa sposoby: $(\{v_1\}, \emptyset)$ oraz $(\emptyset, \{v_1\})$;
- także dla $t = 2$ istnieją dwa sposoby rozmieszczenia osi na lipach: $(\{v_1\}, \{v_2\})$ oraz $(\{v_2\}, \{v_1\})$;
- dla $t > 2$ są już cztery sposoby rozmieszczenia osi na lipach: $(\{v_1, v_3, \ldots\}, \{v_2, v_4 \ldots\})$ i $(\{v_2, v_4, \ldots\}, \{v_1, v_3, \ldots\})$ oraz $(\{v_t, v_{t-2}, \ldots\}, \{v_{t-1}, v_{t-3} \ldots\})$ i $(\{v_{t-1}, v_{t-3}, \ldots\}, \{v_t, v_{t-2}, \ldots\})$, gdzie w zbiorach wypisujemy wierzchołki począwszy od najniżej umieszczonych na lipie.

Po rozmieszczeniu wierzchołków osi, wierzchołki sąsiadujące z v_j możemy dowolnie rozmieścić pomiędzy v_{j-1} oraz v_{j+1}. Czyli możemy je rozlokować na $|S_j|!$ sposobów.

Stąd jedną dużą składową możemy rozmieścić na

$$L(T_i) \quad = \quad c_t \cdot |S_1|! \cdot |S_2|! \cdot \cdots \cdot |S_t|! \tag{1}$$

sposobów, gdzie $c_t = 2$ dla $t \leqslant 2$ i $c_t = 4$ dla $t > 2$.

Wzajemne rozmieszczenie dużych składowych

Dla każdej pary składowych T_i, T_j ($1 \leqslant i < j \leqslant p$) są możliwe tylko dwa ich wzajemne położenia na lipach. Albo składowa T_i leży (cała!) poniżej składowej T_j, albo składowa T_j leży (także cała) poniżej składowej T_i. Jakiekolwiek zaburzenie tego porządku powoduje, że krawędzie z tych składowych przecinają się i rozmieszczenie przestaje być pladzielne. Stąd istnieje

$$p! \tag{2}$$

możliwości wzajemnego rozmieszczenia dużych składowych.

Rozmieszczenie małych składowych

Małe składowe możemy rozmieszczać dowolnie (ptaszki nie mające znajomych, nie będą nikomu przeszkadzać). Jeśli na pierwszej lipie mamy już x wierzchołków, a na drugiej y, to składową X_i możemy umieścić na $x+1+y+1$ sposobów. Tak więc jeśli przez w oznaczymy liczbę wierzchołków we wszystkich dużych składowych, to pierwszą małą składową X_1 możemy umieścić na $w+2$ sposobów, składową X_2 na $w+3$ sposobów,... Ogólnie, składową X_i możemy umieścić na $(w+2+(i-1))$ sposobów. Stąd liczba rozmieszczeń małych składowych w każdym rozmieszczeniu dużych składowych wynosi

$$W \quad = \quad (w+2) \cdot (w+3) \cdot \cdots \cdot (w+2+s-1) \tag{3}$$

Ze wzorów (1), (2) oraz (3) widzimy, że liczba wszystkich rozmieszczeń ptaszków na lipach wynosi:

$$r \quad = \quad L(T_1) \cdot L(T_2) \cdot \cdots \cdot L(T_p) \cdot p! \cdot W$$

i może być bardzo duża. Dlatego w zadaniu wystarczy podać tylko resztę z dzielenia r przez k (tak dużej liczby ptaszki i tak z pewnością nie mogłyby ogarnąć swoimi niewielkimi rozumkami), stąd wszystkie obliczenia można wykonywać na liczbach całkowitych w standardowej reprezentacji.

Rozwiązanie wzorcowe

Schemat algorytmu rozwiązania wzorcowego jest następujący.

1. Wczytujemy wartości m i n, a następnie sprawdzamy, czy $m < n$. Jeśli nie, to graf z pewnością nie jest acykliczny, czyli nie może być także pladzielny. Nie musimy więc wczytywać krawędzi grafu — możemy od razu wypisać odpowiedź 0 i skończyć.

2. Jeśli $m < n$, to wczytujemy opis krawędzi grafu. Następnie sprawdzamy, czy jest to graf pladzielny:

   (a) czy jest acykliczny (tzn. nie zawiera cyklu) oraz

   (b) czy każdy wierzchołek ma co najwyżej dwóch sąsiadów o stopniu większym niż jeden.

3. Jeśli wykryjemy, że graf nie jest pladzielny, to wypisujemy odpowiedź 0 i kończymy.

4. Dzielimy graf na spójne składowe (właściwie można to zrobić przy okazji sprawdzania acykliczności w punkcie 2(a)).

5. Następnie obliczamy stopniowo wynik:

   (a) dla dużych składowych T obliczamy wartości $L(T)$ (patrz wzór (1)) i wymnażamy je;

   (b) domnażamy przez liczbę permutacji dużych składowych (patrz wzór (2));

   (c) domnażamy przez liczbę położeń małych składowych wśród dużych (patrz wzór (3)).

Sprawdzanie acykliczności grafu i znajdowanie spójnych składowych (kroki 2(a) i 4) wykonujemy za pomocą algorytmu przeszukiwania w głąb (DFS, patrz na przykład [17]), który działa w czasie $O(n+m)$, co w naszym przypadku jest równe $O(n)$. Oddzielnie obliczamy stopnie wierzchołków grafu i sprawdzamy drugi warunek pladzielności dla każdej składowej. Całość nadal działa w czasie $O(n)$ i wymaga pamięci $O(n+m) = O(n)$.

Wszystkie obliczenia w kroku 5 są wykonywane modulo k na liczbach całkowitych 64-bitowych (typ Int64 w Pascalu i long long w C/C++), aby uniknąć przekroczeń zakresu typów całkowitych.

Opisane rozwiązanie znajduje się na dysku dołączonym do książeczki, zaimplementowane w języku Pascal (`dzi.pas`) i C++ (`dzi1.cpp`).

Alternatywna metoda szukania spójnych składowych

Sprawdzanie acykliczności i wyznaczanie spójnych składowych, które w rozwiązaniu wzorcowym były wykonywane za pomocą algorytmu DFS, można zrobić przy pomocy struktury danych dla zbiorów rozłącznych, tzw. „Find & Union" (patrz na przykład [17]).

1. Początkowo z każdego wierzchołka v grafu tworzymy zbiór jednoelementowy S_v (w algorytmie w jednym zbiorze będziemy umieszczać wierzchołki, o których już wiemy, że są połączone w grafie i należą do jednej składowej spójności).

2. Następnie przeglądamy kolejno wszystkie krawędzie grafu i dla każdej krawędzi (u,v):

   (a) szukamy zbioru U, do którego należy wierzchołek u;

   (b) szukamy zbioru V, do którego należy wierzchołek v;

   (c) jeśli $U = V$, to widzimy, że w grafie jest cykl (bo już wcześniej znaleźliśmy połączenie pomiędzy u i v) i kończymy odpowiadając, że graf nie jest pladzielny;

   (d) w przeciwnym razie łączymy zbiory U i V w jeden (wiemy już, że ich wierzchołki należą do tej samej składowej).

3. Na zakończenie mamy zbiory $V_1, V_2, \ldots$ zawierające wierzchołki poszczególnych składowych grafu.

4. W trakcie przeglądania krawędzi możemy przy okazji obliczać stopnie wierzchołków grafu; wówczas po wyznaczeniu składowych możemy sprawdzić pladzielność drzew-składowych.

Pozostaje jeszcze opisać strukturę danych dla zbiorów wierzchołków, która umożliwi sprawne wykonywanie wszystkich powyższych operacji. Ale tutaj już odeślemy Czytelnika do literatury (patrz na przykład [12] lub [17]). Opisana tam implementacja pozwala wykonać przedstawiony algorytm w czasie $O(n \cdot \log^* n)$. Funkcja $\log^* n$ to tak zwany logarytm iterowany, który definiujemy jako

$$\log^* n = \min\{i \geqslant 0 : \log^{(i)} n \leqslant 1\}$$

gdzie $\log^{(i)}$ to i-krotne złożenie funkcji logarytm dwójkowy ze sobą. Logarytm iterowany jest funkcją *bardzo, bardzo* wolno rosnącą:

$$\begin{aligned}
\log^* 2 &= 1 \\
\log^* 4 &= 2 \\
\log^* 16 &= 3 \\
\log^* 65536 &= 4 \\
\log^*(2^{65536}) &= 5
\end{aligned}$$

więc dla danych n z rozważanego zakresu $log^*(n)$ nie przekracza 5 i rozwiązanie działa praktycznie w czasie liniowo zależnym od n.

Rozwiązanie to znajduje się na dysku dołączonym do książeczki, zaimplementowane w języku Pascal (`dzi2.pas`) i C++ (`dzi3.cpp`).

Rozwiązania niepoprawne

Błędy najczęściej występujące w rozwiązaniach zawodników, to:

- używanie procedur rekurencyjnych o dużej liczbie parametrów i zmiennych lokalnych (zwłaszcza algorytmu DFS) — dla niektórych danych (grafów o długich ścieżkach) zagłębienie rekursji było tak duże, że programy przekraczały założone limity pamięci; aby uniknąć takich błędów należało zaprogramować rekurencję na własnym stosie;
- wczytywanie za każdym razem całego grafu do pamięci, co w przypadku $m \geqslant 1\,000\,000$ było zbyt kosztowne pamięciowo (i czasowo);
- pomijanie pewnych możliwości rozmieszczenia ptaszków w dziuplach.

Testy

Zadanie było testowane na 10 grupach testów. Grupowanie testów przeciwdziałało uzyskiwaniu punktów przez programy zawsze wypisujące 0.

Nazwa	n	m	Opis
dzi1a.in	4	2	test poprawnościowy
dzi1b.in	3	2	test poprawnościowy
dzi1c.in	4	3	test poprawnościowy
dzi2a.in	7	1	test z wierzchołkami izolowanymi
dzi2b.in	16	8	test z wierzchołkami izolowanymi
dzi3a.in	50 342	50 325	graf zawiera cykle
dzi3b.in	8	5	test poprawnościowy
dzi4a.in	35	31	graf acykliczny, niepladzielny
dzi4b.in	12	11	graf acykliczny, niepladzielny
dzi4c.in	14	11	test poprawnościowy
dzi5.in	100 001	74 695	specyficzny test
dzi6a.in	904 000	5 400 000	wczytywanie wszystkich krawędzi zbędne
dzi6b.in	13	12	test poprawnościowy
dzi7.in	10 040	10 030	test losowy
dzi8a.in	500 012	495 949	duży test losowy
dzi8b.in	700 000	699 995	dwa drzewa o głębokości 350 000
dzi9a.in	1 000 000	990 677	duży test losowy
dzi9b.in	1 000 000	999 995	dwa drzewa o głębokości 500 000
dzi10a.in	1 000 000	999 998	drzewo o głębokości 1 000 000
dzi10b.in	1 000 000	997 659	duży test losowy

Wojciech Rytter
Treść zadania, Opracowanie

Arkadiusz Paterek
Program

OI, Etap III, dzień 1, 6–04–2005

Dwuszereg

W dwuszeregu stoi $2n$ żołnierzy. Trzeba przestawić żołnierzy tak, aby w tym samym szeregu nie było dwóch żołnierzy tego samego wzrostu — wówczas powiemy, że żołnierze są ustawieni poprawnie.

Pojedyncza operacja polega na zamianie miejscami dwóch żołnierzy, którzy są na tej samej pozycji w obu szeregach. Twoim zadaniem jest policzenie minimalnej liczby zamian, jakie trzeba wykonać, aby żołnierze byli ustawieni poprawnie.

Przykład

Na rysunku mamy dwuszereg złożony z 18 żołnierzy. Strzałkami zaznaczono 3 operacje zamiany, po wykonaniu których żołnierze są ustawieni poprawnie.

2	5	5	2	7	4	7	3	9
↕		↕		↕				
1	6	8	4	6	3	9	1	8

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia liczbę i wzrost żołnierzy, tak jak są ustawieni na początku,*
- *wyznaczy minimalną liczbę zamian miejscami (żołnierzy stojących na tej samej pozycji w obu szeregach) potrzebnych do poprawnego ustawienia żołnierzy,*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu wejścia znajduje się jedna liczba całkowita n, $1 \leqslant n \leqslant 50\,000$. W każdym z dwóch szeregów stoi n żołnierzy. W każdym z dwóch kolejnych wierszy znajduje się po n dodatnich liczb całkowitych pooddzielanych pojedynczymi odstępami. W drugim wierszu znajdują się liczby $x_1, x_2, \ldots, x_n$, $1 \leqslant x_i \leqslant 100\,000$; x_i to wzrost i-go żołnierza w pierwszym szeregu. W trzecim wierszu znajdują się liczby $y_1, y_2, \ldots, y_n$, $1 \leqslant y_i \leqslant 100\,000$; y_i to wzrost i-go żołnierza w drugim szeregu.

Możesz założyć, że dla danych testowych zawsze możliwe jest poprawne ustawienie żołnierzy.

Wyjście

W pierwszym i jedynym wierszu wyjścia powinna znaleźć się jedna nieujemna liczba całkowita — minimalna liczba zamian jakie należy wykonać, aby żołnierze byli poprawnie ustawieni.

Przykład

Dla danych wejściowych:
```
9
2 5 5 2 7 4 7 3 9
1 6 8 4 6 3 9 1 8
```

poprawnym wynikiem jest:
```
3
```

Rozwiązanie

Wstępne uwagi i oznaczenia

Przyjmiemy następujące oznaczenia.

- Wysokości żołnierzy z pierwszego szeregu są zapisane w tablicy $\mathcal{S}_1[1..n]$, a wysokości żołnierzy z drugiego szeregu — w tablicy $\mathcal{S}_2[1..n]$.
- Pozycje $\mathcal{S}_1[i]$ i $\mathcal{S}_2[i]$ nazwiemy *kolumną i*.
- Operacja $zamiana(i)$ powoduje zamianę miejscami żołnierzy stojących w i-tej kolumnie: $\mathcal{S}_1[i] \leftrightarrow \mathcal{S}_2[i]$.
- Dwie różne kolumny i, j nazwiemy *zależnymi*, gdy $\{\ \mathcal{S}_1[i], \mathcal{S}_2[i]\ \} \cap \{\ \mathcal{S}_1[j], \mathcal{S}_2[j]\ \} \neq \emptyset$.

Zauważmy także, iż aby istniało rozwiązanie zadania (a taką gwarancję mamy w treści), nie może być *trzech żołnierzy jednakowego wzrostu*.

Interpretacja grafowa

Definicja 1 *„Graf” zależności* jest to graf $G = (V,E)$ o zbiorze wierzchołków $V = \{1,2,\ldots,n\}$ i zbiorze krawędzi E łączących wierzchołki odpowiadające kolumnom zależnym. W przypadku kolumn i, j ($i \neq j$), dla których $\{\ \mathcal{S}_1[i], \mathcal{S}_2[i]\ \} = \{\ \mathcal{S}_1[j], \mathcal{S}_2[j]\ \}$, wierzchołki i oraz j są połączone dwiema krawędziami (z tego powodu słowo graf w niniejszej definicji zostało ujęte w cudzysłów).

Możemy zauważyć, że graf zależności ma bardzo prostą strukturę.

Lemat 1 *Graf zależności G składa się z rozłącznych cykli i ścieżek.*

Dowód Wystarczy zauważyć, że z wierzchołka odpowiadającego kolumnie i mogą wychodzić krawędzie tylko do kolumn, w których stoi żołnierz o wzroście $\mathcal{S}_1[i]$ lub $\mathcal{S}_2[i]$. Ponieważ jest najwyżej dwóch żołnierzy o wzroście $\mathcal{S}_1[i]$, z tego jeden w kolumnie i — podobnie dla $\mathcal{S}_2[i]$ — więc z wierzchołka i mogą wychodzić najwyżej dwie krawędzie. Stąd w grafie zależności każdy wierzchołek ma co najwyżej stopień dwa, czyli graf musi być złożony z rozłącznych cykli i ścieżek. ∎

Zmodyfikujmy graf zależności nadając każdej krawędzi zwrot w ten sposób, by powstały skierowane cykle i skierowane ścieżki (patrz rysunek). Przypiszmy także krawędziom grafu etykiety, które nazwiemy *kolorami*. Dla krawędzi $e = i \to j$ określmy:

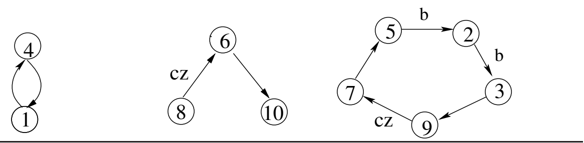
$$kolor(e) = \begin{cases} czarny & \text{gdy } \mathcal{S}_1[i] = \mathcal{S}_2[j] \\ biały & \text{gdy } \mathcal{S}_2[i] = \mathcal{S}_1[j] \end{cases}$$

W przypadku cyklu dwuelementowego (i, j), dla którego powstaje niejednoznaczność przy określaniu kolorów zgodnie z powyższym wzorem, umówmy się, że gdy $\mathcal{S}_2[i] = \mathcal{S}_1[j]$ (i tym samym $\mathcal{S}_1[i] = \mathcal{S}_2[j]$), to obu krawędziom cyklu przypisujemy kolor biały.

W pozostałych przypadkach będziemy mówić, że *krawędź nie ma koloru*.

Przykład.

numer kolumny	1	2	3	4	5	6	7	8	9	10
pierwszy szereg	2	6	5	2	7	4	7	3	9	4
drugi szereg	1	5	8	1	6	3	9	10	8	11



Rys. 1: Konfiguracja początkowa i odpowiadający jej skierowany graf zależności z etykietami krawędzi.

■ *Przykład*

Porządkowanie ustawienia

Rozważmy teraz jedną składową grafu zależności i zajmijmy się doprowadzeniem do poprawnego ustawienia żołnierzy z tej składowej (możemy to robić niezależne od „porządkowania” pozostałych składowych). Po pierwsze zauważmy, że prawdziwy jest następujący lemat:

Lemat 2 *Ustawienie żołnierzy z kolumn jednej składowej grafu zależności jest poprawne wtedy i tylko wtedy, gdy wszystkie krawędzie tej składowej mają taki sam kolor.*

Dowód Po pierwsze zauważmy, że przy poprawnym ustawieniu wszystkie krawędzie rozważanej składowej muszą mieć określony kolor — w przeciwnym razie występuje błąd ustawienia ($\mathcal{S}_1[i] = \mathcal{S}_1[j]$ lub $\mathcal{S}_2[i] = \mathcal{S}_2[j]$).

Rozważmy teraz przypadek, gdy składowa ma przynajmniej trzy wierzchołki. Wystarczy wówczas zauważyć, że jeśli krawędzie $i \to j$ oraz $j \to k$ są jej kolejnymi krawędziami i obie mają określony kolor, to musi to być ten sam kolor. W przeciwnym razie jedna wartość wzrostu żołnierza musiałaby wystąpić w danych przynajmniej trzykrotnie ($\mathcal{S}_1[i] = \mathcal{S}_2[j] = \mathcal{S}_1[k]$ lub $\mathcal{S}_2[i] = \mathcal{S}_1[j] = \mathcal{S}_2[k]$).

Pozostaje rozważyć przypadek, gdy składowa ma dwa wierzchołki. W takiej sytuacji albo składowa (ścieżka) ma dokładnie jedną krawędź, albo jest to cykl i wówczas również widać, że obie jego krawędzie muszą mieć ten sam kolor. ■

Z lematu wynika, że dla każdej składowej grafu zależności istnieją dokładnie dwa poprawne ustawienia. Pozostaje rozstrzygnąć, do którego z nich możemy doprowadzić wykonując mniejszą liczbę zamian. Dla składowej $\mathcal{X}$ i koloru x oznaczmy przez $LZ(\mathcal{X},x)$ minimalną liczbę zamian, które trzeba wykonać dla kolumn tworzących tę składową, by wszystkie jej krawędzie miały kolor x. Zakładając, że potrafimy znajdować kolejne wierzchołki składowej w czasie stałym, mamy następujący lemat.

Lemat 3 *Dla każdej składowej grafu zależności $\mathcal{X}$ i każdego koloru x wartość $LZ(\mathcal{X},x)$ można obliczyć w czasie liniowo zależnym od liczby wierzchołków składowej $\mathcal{X}$.*

Dowód W niniejszym dowodzie będziemy mówili, że krawędź $i \rightarrow j$ *zaczyna się w pierwszym szeregu*, gdy jest czarna lub $\mathcal{S}_1[i] = \mathcal{S}_1[j]$. W przeciwnym razie powiemy, że krawędź *zaczyna się w drugim szeregu*. Analogicznie powiemy, że krawędź $i \rightarrow j$ *kończy się w pierwszym szeregu*, gdy jest to krawędź biała lub $\mathcal{S}_1[i] = \mathcal{S}_1[j]$. W przeciwnym razie powiemy, że krawędź ta *kończy się w drugim szeregu*. Łatwo zauważyć, że składowa jest biała, gdy wszystkie jej krawędzie zaczynają się w drugim szeregu i kończą w pierwszym. Natomiast składowa jest czarna, gdy wszystkie jej krawędzie zaczynają się w pierwszym szeregu i kończą w drugim.

Rozważmy najpierw składową-ścieżkę $\mathcal{X} = (i_1, i_2, \ldots, i_k)$ i niech x będzie kolorem *białym*. Jeśli $\mathcal{S}_1[i_1] = \mathcal{S}_2[i_2]$ lub $\mathcal{S}_1[i_1] = \mathcal{S}_1[i_2]$ (czyli krawędź $i_1 \rightarrow i_2$ zaczyna się w pierwszym szeregu), to dokonujemy zamiany w kolumnie i_1. Następnie w kolejnych kolumnach i_j dla $j = 2, 3, \ldots, k$ dokonujemy zamiany tylko wówczas, gdy krawędź $i_{j-1} \rightarrow i_j$ kończy się w drugim szeregu. Łatwo zauważyć, że po wykonaniu wszystkich zamian uzyskamy składową w kolorze białym. Co więcej, zauważmy, że wszystkie dokonane zmiany **były konieczne**. Operacja $zamiana(i_1)$ była wykonywana tylko wówczas, gdy pierwsza krawędź zaczynała się w pierwszej kolumnie. Pozostałe zamiany wykonywaliśmy tylko wówczas, gdy rozważana krawędź $(i_{j-1} \rightarrow i_j)$ kończyła się w drugim szeregu. W ten sposób przeglądając jednokrotnie składową $\mathcal{X}$ wyznaczyliśmy wartość $LZ(\mathcal{X}, \textit{biały})$. Analogicznie (lub jednocześnie) możemy wyznaczyć wartość $LZ(\mathcal{X}, \textit{czarny})$.

Niech teraz $\mathcal{X} = (i_1, i_2, \ldots, i_k)$ będzie składową-cyklem i ponownie przyjmijmy najpierw, że x to kolor biały. Znajdujemy minimalne j, dla którego krawędź $(i_{j-1} \rightarrow i_j)$ kończy się w drugim szeregu (jeśli krawędź $i_k \rightarrow i_1$ kończy się w drugim szeregu, to przyjmujemy $j = 1$). Zauważmy, że znaleziona krawędź musi być czarna lub niepokolorowana. Dokonujemy zamiany w kolumnie i_j i poszukujemy kolejnej krawędzi cyklu kończącej się w drugim szeregu. Dokonujemy dla niej analogicznej zamiany jak poprzednio i kontynuujemy proces, aż wszystkie krawędzie będą kończyć się w pierwszym szeregu (tym samym wszystkie krawędzie cyklu będą zaczynać się w drugim szeregu, a więc wszystkie będą białe). Podobnie jak w przypadku składowej-ścieżki łatwo zauważyć, że wszystkie wykonane zamiany były konieczne, by osiągnąć białe pokolorowanie składowej. Także podobnie jak poprzednio, analogicznie możemy wyznaczyć wartość $LZ(\mathcal{X}, \textit{czarny})$.

■

Przykład c.d.

Dla naszego przykładu mamy dwie składowe-cykle: $X_1 = \{1,4\}$, $X_3 = \{2,3,9,7,5\}$ oraz jedną ścieżkę: $X_2 = \{8,6,10\}$. Początkowo krawędzie $8 \rightarrow 6$ i $9 \rightarrow 7$ mają kolor czarny, natomiast krawędzie $2 \rightarrow 3$ i $5 \rightarrow 2$ mają kolor biały.

Aby składowej X_1 nadać białe etykiety, wystarczy wykonać operację zamiany na pozycji 1 albo na pozycji 4. Zgodnie z wyjątkiem opisanym w definicji etykietowania, tej składowej nie da się pokolorować inaczej.

Aby składowej X_3 nadać białe etykiety, wystarczy wykonać zamiany na pozycjach: 7 i 9. By doprowadzić do pokolorowania składowej na czarno, trzeba wykonać zamiany w kolumnach 2, 3 i 5.

Pozostaje jeszcze składowa X_2. Pokolorowanie jej na biało uzyskamy dokonując zamian w kolumnach 6 i 8. Aby pokolorować ją na czarno, wystarczy zamienić miejscami żołnierzy w kolumnie 10.

Stąd optymalny sposób uzyskania poprawnego ustawienia wymaga 4 zamian.

numer kolumny	1	2	3	4	5	6	7	8	9	10
pierwszy szereg	2	5	8	1	6	4	7	3	9	11
drugi szereg	1	6	5	2	7	3	9	10	8	4
				*			*		*	*

Rys. 2: Ustawienie, w którym składowe X_1 i X_3 są pokolorowane na biało, a składowa X_2 na czarno. Gwiazdkami zaznaczono kolumny, w których należy wykonać zamiany, by uzyskać jednobarwne kolorowanie każdej składowej. Jest to optymalny sposób uporządkowania dwuszeregu.

■ *Przykład c.d.*

Algorytm

Podsumowując opisane rozważania możemy zapisać algorytm dla zadania.

```
procedure Algorytm
  begin
    LZ:=0;
    forall składowa X do
      LZ:=LZ+ min{LZ(X,czarny),LZ(X,biały)};
    return LZ;
  end
```

Algorytm można zaimplementować tak, by działał w czasie $O(n)$. Zakładamy, że dane (wysokości żołnierzy) są na tyle małymi liczbami całkowitymi, że można je posortować w czasie liniowym (np. za pomocą algorytmu *radixsort*, jego opis można znaleźć w [12], [14] lub [17]). Wtedy możemy zastąpić je ich numerami w porządku, czyli kolejnymi liczbami naturalnymi. Pozwala to wyznaczyć graf zależności w czasie liniowym. Pozostałe operacje, czyli wyznaczenie składowych i etykiet ich krawędzi, wymagają już tylko jednokrotnego odwiedzenia każdego wierzchołka grafu.

Ciąg dalszy

Oprócz wyznaczenia poprawnego ustawienia osiąganego przez minimalną liczbę zamian, możemy także łatwo wyznaczyć liczbę poprawnych ustawień żołnierzy — wynosi ona 2^r, gdzie r jest liczbą składowych grafu zależności, włącznie ze składowymi jednoelementowymi (ale tylko tymi, dla których w odpowiadających im kolumnach stoją dwaj żołnierze różnego wzrostu).

Możemy także zmodyfikować zadanie przyjmując, że żołnierze stoją w trzech, a nie w dwóch szeregach. Jedna zamiana jest teraz jakąkolwiek wymianą trzech żołnierzy stojących w tej samej kolumnie. Opisany algorytm w tej sytuacji nie działa. Pozostawiamy Czytelnikowi zastanowienie się nad rozwiązaniem tak zmienionego zadania. Czy jest równie proste jak poprzednie?

Testy

Rozwiązanie było testowane na zestawie 15 testów. Opis poszczególnych testów znajduje się poniżej, *n* oznacza w nich liczbę żołnierzy stojących w szeregu.

Nazwa	**n**	**Opis**
dwu1.in	100	krótkie cykle
dwu2.in	200	średnie ścieżki
dwu3.in	500	średnie cykle
dwu4.in	1 000	długie ścieżki
dwu5.in	2 000	długie cykle
dwu6.in	5 000	krótkie ścieżki
dwu7.in	50 000	krótkie cykle, krótkie ścieżki
dwu8.in	50 000	krótkie cykle, średnie ścieżki
dwu9.in	50 000	krótkie cykle, długie ścieżki
dwu10.in	50 000	średnie cykle, krótkie ścieżki
dwu11.in	50 000	średnie cykle, średnie ścieżki
dwu12.in	50 000	średnie cykle, długie ścieżki
dwu13.in	50 000	długie cykle, krótkie ścieżki
dwu14.in	50 000	długie cykle, średnie ścieżki
dwu15.in	50 000	długie cykle, długie ścieżki

Wojciech Guzicki — Treść zadania, Opracowanie
Arkadiusz Paterek — Program

OI, Etap III, dzień 1, 6–04–2005

Dwa przyjęcia

Król Bajtazar postanowił urządzić dwa wielkie przyjęcia, na które chce zaprosić wszystkich mieszkańców Bajtocji, przy czym każdego na dokładnie jedno z tych przyjęć. Król z własnego doświadczenia wie, że osoba dobrze bawi się na przyjęciu, gdy jest na nim parzysta liczba jej znajomych. Zażądał więc od Ciebie, abyś tak podzielił mieszkańców kraju pomiędzy dwa przyjęcia, żeby jak najwięcej osób miało na swoim przyjęciu parzystą liczbę znajomych. Możliwy jest również taki podział, w którym na jedno z przyjęć nikt nie przyjdzie. Relacja znajomości jest symetryczna, tzn. jeśli osoba A zna osobę B, to osoba B zna osobę A.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia liczbę mieszkańców Bajtocji i opis ich znajomości,*
- *podzieli mieszkańców pomiędzy dwa przyjęcia tak, aby liczba osób, które mają na swoim przyjęciu parzystą liczbę znajomych, była jak największa,*
- *wypisze na standardowe wyjście listę osób, które powinny przyjść na pierwsze przyjęcie.*

Wejście

W pierwszym wierszu standardowego wejścia zapisana jest jedna liczba całkowita n $(1 \leqslant n \leqslant 200)$ — jest to liczba mieszkańców Bajtocji. Mieszkańcy są ponumerowani od 1 do n. W kolejnych n wierszach znajdują się opisy znajomości kolejnych osób. Na początku $(i+1)$-szego wiersza występuje liczba całkowita l_i $(0 \leqslant l_i \leqslant n-1)$ — liczba znajomych i-tego mieszkańca. Dalej w tym samym wierszu zapisanych jest l_i parami różnych numerów znajomych mieszkańca i. Zakładamy, że żaden mieszkaniec nie jest swoim własnym znajomym, a zatem znajomości wypisane są dwukrotnie: jeśli A i B się znają, to B występuje na liście znajomych A oraz A na liście znajomych B.

Wyjście

W pierwszym wierszu standardowego wejścia Twój program powinien wypisać jedną liczbę całkowitą M — liczbę osób, które mają przyjść na pierwsze z przyjęć. W drugim wierszu należy wymienić M numerów tych właśnie osób. Pozostałe osoby pójdą na drugie przyjęcie.

W tym zadaniu możliwych jest wiele poprawnych wyników — Twój program powinien wypisać dowolny z nich.

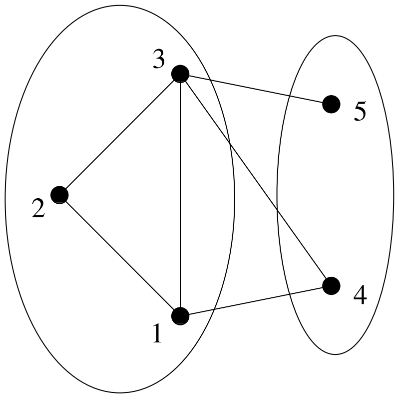
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
3
1 2 3
```



W powyższym przykładzie wszystkie osoby będą miały na przyjęciu parzystą liczbę znajomych.

Rozwiązanie

Problem w języku grafów

Sformułujemy zadanie w języku teorii grafów. Dany jest zbiór n osób — będą to wierzchołki naszego grafu G. Wierzchołki v i w są połączone krawędzią, jeśli osoby v i w znają się. Mamy zatem graf nieskierowany bez krawędzi wielokrotnych i pętli (tzn. krawędzi łączących wierzchołek z samym sobą). Dla danego grafu G symbolem $V(G)$ oznaczamy zbiór wierzchołków tego grafu, a symbolem $E(G)$ oznaczamy zbiór krawędzi. Krawędź łączącą wierzchołki v i w oznaczamy symbolem vw (lub wv). Dla danego wierzchołka v symbolem $S(v)$ oznaczamy zbiór sąsiadów v:

$$S(v) = \{w \in V(G) : \ vw \in E(G)\}.$$

Wreszcie dla dowolnego zbioru A symbolem $|A|$ oznaczamy liczbę elementów zbioru A.

Naszym zadaniem jest podzielić zbiór wierzchołków grafu G na dwa rozłączne zbiory A i B, takie że $A \cup B = V(G)$ oraz:

(1) dla każdego wierzchołka $v \in A$ zbiór $S(v) \cap A$ ma parzystą liczbę elementów;

(2) dla każdego wierzchołka $v \in B$ zbiór $S(v) \cap B$ ma parzystą liczbę elementów.

Podział taki nazwiemy *dobrym podziałem*.

Każdy może dobrze się bawić

Pokażemy, że dla *każdych* danych istnieje rozwiązanie, w którym *każdy* mieszkaniec Bajtocji będzie zadowolony: na przyjęciu, w którym uczestniczy, będzie miał parzystą liczbę znajomych.

Twierdzenie 1 *Dla każdego grafu G istnieje dobry podział zbioru wierzchołków $V(G)$.*

Dowód Pokażemy jak wyznaczyć zbiory A_G i B_G stanowiące dobry podział $V(G)$.

Jeśli każdy wierzchołek grafu G ma parzystą liczbę sąsiadów (w szczególności, jeśli graf G ma tylko jeden wierzchołek), to możemy przyjąć $A_G = V(G)$ oraz $B_G = \emptyset$ i widzimy, że jest do dobry podział.

Przypuśćmy zatem, że graf G ma więcej niż jeden wierzchołek i co najmniej jeden z nich ma nieparzystą liczbę sąsiadów. Pokażemy przez indukcję względem liczby wierzchołków, że wierzchołki grafu G można dobrze podzielić.

Załóżmy, że każdy graf mający mniej wierzchołków niż graf G można dobrze podzielić. Niech $v \in V(G)$ będzie wierzchołkiem grafu G mającym nieparzystą liczbę sąsiadów i niech $S = S(v)$ będzie zbiorem tych sąsiadów.

Definiujemy teraz nowy graf H. Przyjmujemy $V(H) = V(G) \setminus \{v\}$, a następnie definiujemy krawędzie grafu H:

- jeśli $u, w \in S$, to $uw \in E(H) \Leftrightarrow uw \notin E(G)$;
- jeśli $u \notin S$ lub $w \notin S$, to $uw \in E(H) \Leftrightarrow uw \in E(G)$.

Z założenia indukcyjnego wynika, że zbiór wierzchołków grafu H można dobrze podzielić — niech A_H i B_H będą zbiorami tworzącymi taki podział. Zbiór S, który ma nieparzystą liczbę elementów, musi mieć parzystą liczbę elementów wspólnych z jednym z tych zbiorów i nieparzystą liczbę elementów wspólnych z drugim. Zdefiniujmy:

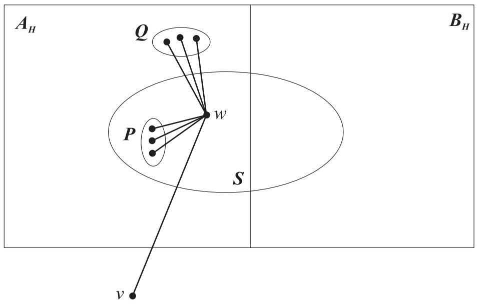
- jeśli liczba $|S \cap A_H|$ jest parzysta, to $A_G = A_H \cup \{v\}$ oraz $B_G = B_H$,
- jeśli liczba $|S \cap B_H|$ jest parzysta, to $A_G = A_H$ oraz $B_G = B_H \cup \{v\}$.

Udowodnimy teraz, że zbiory A_G i B_G tworzą dobry podział dla grafu G. Dla ustalenia uwagi przyjmijmy, że to zbiór $S \cap A_H$ ma parzystą liczbę elementów, w przeciwnym przypadku rozumowanie będzie przebiegać analogicznie. Niech w będzie wierzchołkiem grafu G. Aby sprawdzić, że w grafie G ma on parzystą liczbę sąsiadów w zbiorze, do którego należy, rozważymy oddzielnie trzy przypadki.

Przypadek 1 Niech $w = v$. Wierzchołek v oczywiście należy do zbioru $A_G = A_H \cup \{v\}$ i ma w tym zbiorze parzystą liczbę sąsiadów: są nimi elementy zbioru $S \cap A_H$.

Przypadek 2 Niech $w \neq v$ oraz $w \notin S$. Zauważmy, że wierzchołek w w obu grafach G i H ma tych samych sąsiadów. Ponadto jeśli należał do zbioru A_H, to należy do A_G, a jeśli należał do B_H, to należy do B_G. Ponieważ w grafie H ma parzystą liczbę sąsiadów w zbiorze, do którego należy, więc także ma tę własność w grafie G.

Przypadek 3 Pozostaje sprawdzić warunki (1) i (2) dla wierzchołków $w \in S$. Załóżmy najpierw, że $w \in A_H$. Niech $P = (S \cap S(w)) \cap A_H$ oraz $Q = (S(w) \setminus S) \cap A_H$ i oznaczmy $p = |P|$ oraz $q = |Q|$. Mamy więc sytuację, którą możemy zilustrować następującym rysunkiem:

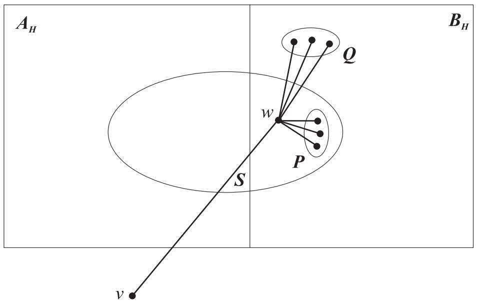


Sąsiadami wierzchołka w w grafie H w części A_H są wierzchołki należące do zbioru $Q \cup \big((S \cap A_H) \setminus (P \cup \{w\})\big)$ — jest ich $q + |S \cap A_H| - p - 1$. Z drugiej strony, ponieważ A_H, B_H jest dobrym podziałem grafu H, więc wiemy, że ich liczba jest parzysta. Ponieważ również liczba $|S \cap A_H|$ jest parzysta, więc także liczba $q - p - 1$ jest parzysta. Zatem liczba

$$p + q + 1 = (q - p - 1) + (2p + 2)$$

jest parzysta. Ale wierzchołek w ma w zbiorze $A_G = A_H \cup \{v\}$ właśnie $p + q + 1$ sąsiadów. To dowodzi, że jest spełniony warunek (1).

Załóżmy teraz, że $w \in B_H$ i zdefiniujmy odpowiednio $P = (S \cap S(w)) \cap B_H$, $Q = (S(w) \setminus S) \cap B_H$, $p = |P|$ oraz $q = |Q|$. Sytuację ilustruje teraz rysunek:



Sąsiadami wierzchołka w w grafie H w części B_H są wierzchołki należące do zbioru $Q \cup \big((S \cap B_H) \setminus (P \cup \{w\})\big)$. Jest ich $q + |S \cap B_H| - p - 1$. Ponieważ A_H, B_H jest dobrym podziałem grafu H, więc musi to być liczba parzysta. Ale liczba $|S \cap B_H|$ jest nieparzysta, więc liczba $q - p$ jest parzysta. Stąd wynika, że także liczba $p + q$ jest parzysta. Ale wierzchołek w ma w zbiorze B_G (równym w tym przypadku zbiorowi B_H) właśnie $p + q$ sąsiadów, co dowodzi, że jest spełniony warunek (2).

W ten sposób na mocy zasady indukcji uwodniliśmy prawdziwość twierdzenia. ■

Program wzorcowy

W dowodzie twierdzenia jest zawarty opis algorytmu. Możemy go podsumować następująco:

1. Najpierw sprawdzamy, czy każdy wierzchołek grafu G ma parzystą liczbę sąsiadów. Jeśli tak, to jako wynik zwracamy podział zbioru $V(G)$ na zbiory: $A = V(G)$ i $B = \emptyset$.

2. Jeśli w grafie G istnieje wierzchołek v stopnia nieparzystego, to konstruujemy graf H, a następnie:

   (a) wywołujemy rekurencyjnie procedurę dla grafu H — w wyniku dostajemy podział zbioru $V(G) \setminus \{v\}$ na zbiory A i B;

   (b) dołączamy v do tego zbioru, w którym ma on parzystą liczbę sąsiadów;

   (c) zwracamy zbiory A i B, jako dobry podział $V(G)$.

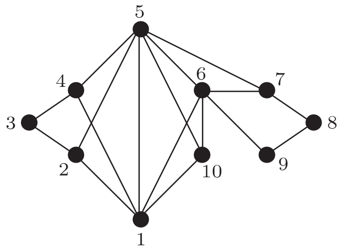
W programie wzorcowym został zaimplementowany powyższy algorytm. Jego czas działania dla grafu o n wierzchołkach i m krawędziach możemy zapisać następującym wzorem rekurencyjnym:

$$T(n) \;=\; c \cdot m \;+\; T(n-1),$$

gdzie c jest pewną stałą. Po wyliczeniu funkcji T z powyższego równania dostajemy, iż $T(n)$ jest rzędu $O(m \cdot n)$. Ponieważ wiemy, że liczba krawędzi w grafie nie przekracza n^2, więc ostatecznie widzimy, że algorytm działa w czasie $O(n^3)$.

Spory przykład

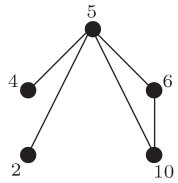
Prześledźmy teraz działanie algorytmu na przykładzie. Rozważmy następujący graf G:



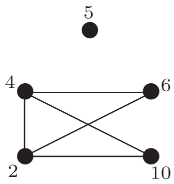
Zauważamy, że wierzchołek 1 ma pięciu sąsiadów:

$$S = S(1) = \{2,4,5,6,10\}.$$

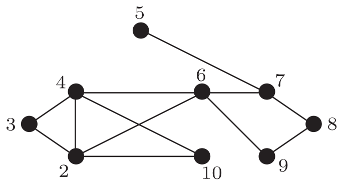
Tworzymy graf H, wyrzucając z grafu G wierzchołek 1. Popatrzmy najpierw na wierzchołki należące do zbioru S i na krawędzie łączące te wierzchołki:



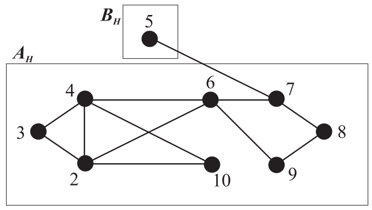
Konstruując graf H usuwamy powyższe krawędzie i dodajemy krawędzie łącząc wierzchołki, które w grafie G połączone nie były:



Graf H wygląda zatem następująco:

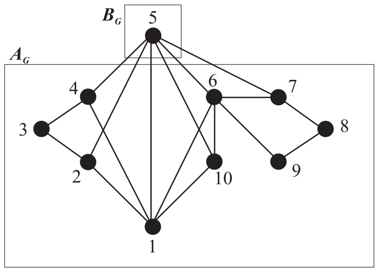


Wierzchołki grafu H można podzielić na zbiory A_H i B_H w następujący sposób (nie ważne skąd wzięliśmy ten podział, powiedzmy, że taki wynik otrzymaliśmy rozważając przykład „rekurencyjnie"):



Ponieważ wierzchołek 1 ma w zbiorze A_H czterech sąsiadów, a w zbiorze B_H jednego, więc dołączamy go do zbioru A_H. W ten sposób otrzymujemy podział zbioru $V(G)$ na zbiory A_G i B_G:

$$A_G = \{1,2,3,4,6,7,8,9,10\}, \quad B_G = \{5\}.$$



Testy

W testach, za pomocą których były weryfikowane programy zawodników, grafy zostały zbudowane w sposób losowy. W każdym teście ustalana była liczba wierzchołków grafu, a następnie dla każdej pary wierzchołków krawędź łącząca je była dołączana do grafu z ustalonym prawdopodobieństwem. Liczby wierzchołków i prawdopodobieństwa były następujące:

Nazwa	**n**	**p**
dwa 1.in	10	0,5%
dwa 2.in	20	0,75%
dwa 3.in	30	0,8%
dwa 4.in	50	0,7%
dwa 5.in	50	0,8%
dwa 6.in	50	0,9%
dwa 7.in	100	0,85%
dwa 8.in	100	0,95%
dwa 9.in	150	0,6%
dwa10.in	150	0,99%

Nazwa	**n**	**p**
dwa11.in	200	0,05%
dwa12.in	200	0,1%
dwa13.in	200	0,5%
dwa14.in	200	0,6%
dwa15.in	200	0,7%
dwa16.in	200	0,8%
dwa17.in	200	0,85%
dwa18.in	200	0,9%
dwa19.in	200	0,95%
dwa20.in	200	0,99%

Szymon Acedański
Treść zadania, Opracowanie

Marek Cygan
Opracowanie, Program

OI, Etap III, dzień 1, 6–04–2005

Akcja komandosów

Na Pustyni Błędowskiej odbywa się w tym roku Bardzo Interesująca i Widowiskowa Akcja Komandosów (BIWAK). Podstawowym elementem BIWAK-u ma być neutralizacja bomby, która znajduje się gdzieś na pustyni, jednak nie wiadomo dokładnie gdzie.

Pierwsza część akcji to desant z powietrza. Z helikoptera krążącego nad pustynią, wyskakują pojedynczo, w ustalonej kolejności komandosi. Gdy któryś z komandosów wyląduje w jakimś miejscu, okopuje się i już z tego miejsca się nie rusza. Dopiero potem może wyskoczyć kolejny komandos.

Dla każdego komandosa określona jest pewna **odległość rażenia***. Jeśli komandos przebywa w tej odległości (lub mniejszej) od bomby, to w przypadku jej ewentualnej eksplozji zginie. Dowództwo chce zminimalizować liczbę komandosów biorących udział w akcji, ale chce mieć też pewność, że w przypadku wybuchu bomby, przynajmniej jeden z komandosów przeżyje.*

Na potrzeby zadania przyjmujemy, że Pustynia Błędowska jest płaszczyzną, a komandosów, którzy się okopali, utożsamiamy z punktami. Mamy dany ciąg kolejno mogących wyskoczyć komandosów. Żaden z nich nie może opuścić swojej kolejki, tzn. jeśli i-ty komandos wyskakuje z samolotu, to wszyscy poprzedni wyskoczyli już wcześniej. Dla każdego z komandosów znamy jego odległość rażenia oraz współrzędne punktu, w którym wyląduje, o ile w ogóle wyskoczy.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opisy komandosów,*
- *wyznaczy minimalną liczbę komandosów, którzy muszą wyskoczyć,*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu standardowego wejścia zapisana jest jedna liczba całkowita n ($2 \leqslant n \leqslant 2\,000$) — liczba komandosów. W kolejnych n wierszach opisani są komandosi — po jednym w wierszu. Opis każdego komandosa składa się z trzech liczb całkowitych: x, y i r ($-1\,000 \leqslant x,y \leqslant 1\,000$, $1 \leqslant r \leqslant 5\,000$). Punkt (x,y) to miejsce, gdzie wyląduje komandos, a r to jego odległość rażenia. Jeśli komandos znajdzie się w odległości r lub mniejszej od bomby, to w przypadku jej wybuchu zginie.

Wyjście

W pierwszym i jedynym wierszu standardowego wyjścia Twój program powinien zapisać jedną liczbę całkowitą — minimalną liczbę komandosów, którzy muszą wyskoczyć, aby zapewnić, *że co najmniej jeden z nich przeżyje. Jeśli nawet po zrzuceniu wszystkich komandosów, nie można mieć pewności, że któryś z nich przeżyje, to należy wypisać słowo NIE.*

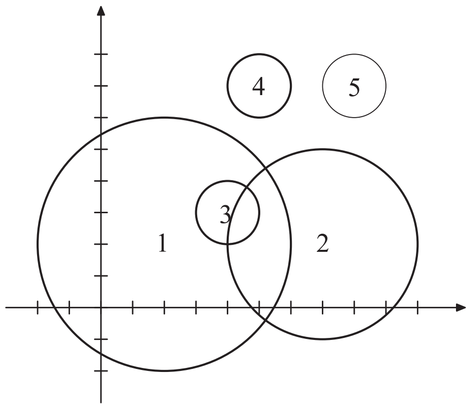
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
4
```



Uwaga

To zadanie można rozwiązać używając typów zmiennopozycyjnych:

- *w Pascalu:* `double` *lub* `extended`,
- *w C oraz C++:* `double` *lub* `long double`.

Użycie typów `float` *lub* `real` *może spowodować błędy w obliczeniach związane z niedokładnością operacji zmiennopozycyjnych.*

Rozwiązanie

Analiza problemu

Pole rażenia każdego komandosa to koło. Bomba umieszczona w punkcie X zabije tych komandosów, których pola rażenia zawierają punkt X. Jeśli więc część wspólna pól rażenia zrzuconych komandosów jest pusta, to dowódcy mogą być pewni, że żadna bomba nie zabije wszystkich żołnierzy.

Ograniczenia na wielkość danych wejściowych podane w zadaniu mogą sugerować, iż złożoność czasowa oczekiwanych rozwiązań powinna być rzędu $O(n^2)$. Taki limit wydaje się być dość łatwy do osiągnięcia — można wymyślić sporo rozwiązań działających w takim lub podobnym czasie. Rozwiązanie wzorcowe zostało oparte na jednym z takich pomysłów zasługującym na szczególną uwagę z racji prostoty i łatwości w implementacji.

Rozwiązanie wzorcowe

Podstawowy problem, który musimy rozwiązać, to rozpoznanie dla zadanego zbioru kół $\mathcal{K} = \{K_1, K_2, \ldots, K_i\}$, czy część wspólna tych kół

$$S \quad = \quad K_1 \cap K_2 \cap \ldots \cap K_i$$

jest pusta. Oznaczmy przez X skrajnie prawy (o największej współrzędnej x) punkt zbioru S, tzn. zdefiniujmy

$$X \in S \text{ oraz dla każdego } Z \in S \text{ zachodzi } Z.x \leqslant X.x,$$

gdzie przez $V.x$ oznaczamy współrzędną x punktu V.

Obserwacja 1 *Punkt X ma następujące własności:*

(a) *jest wyznaczony jednoznacznie;*

(b) *leży na brzegu pewnego koła* $K \in \mathcal{K}$.

(c) *dla* $i > 1$ *jest skrajnie prawym punktem części wspólnej dwóch różnych kół* $K, K' \in \mathcal{K}$.

Dowód Własność (a) wynika z faktu, iż zbiór S jest wypukły i jego brzeg nie zawiera prostych fragmentów. Własność (b) jest także oczywista, bo gdyby nie zachodziła, to znaleźlibyśmy w zbiorze S punkt na prawo od X.

Własność (c) również można łatwo uzasadnić. Niech K będzie kołem, na którego brzegu leży punkt X. Ponieważ X jest skrajnie prawym punktem zbioru S, więc nie możemy przesunąć się na prawo idąc po obwodzie koła K i nie wychodząc poza zbiór S. To oznacza, iż:

- jesteśmy w skrajnie prawym punkcie koła K (jesteśmy więc w skrajnie prawym punkcie przecięcia koła K z dowolnym innym kołem ze zbioru $\mathcal{K}$);
- jesteśmy w punkcie przecięcia koła K z innym kołem K'.

Tak więc w obu przypadkach mamy własność (c). ■

Zdefiniujmy zbiór

$$P \quad = \quad \{Y : Y \text{ jest skrajnie prawym punktem przecięcia dwóch różnych kół } K, K' \in \mathcal{K}\}$$

i sformułujmy jego ważną własność, która stanie się podstawą algorytmu rozwiązania zadania.

Lemat 2 *Jeśli zbiór S jest niepusty, to punkt X należy do zbioru P i jest jego skrajnie lewym punktem.*

Dowód Niech $S \neq \emptyset$. Z poprzedniej obserwacji (punkt (c)) wiemy, że $X \in P$. Niech teraz $K, K' \in \mathcal{K}$ będą takimi kołami, że skrajnie prawy punkt zbioru $K \cap K'$ ma współrzędną x mniejszą niż $X.x$. To oczywiście oznacza, że wszystkie punkty zbioru $K \cap K'$ mają współrzędne x mniejsze od $X.x$, a tym samym nie mogą należeć do zbioru S. Dochodzimy do sprzeczności z założeniem, iż S ma część wspólną z K i K'. ■

W ten sposób sprawdzanie, czy zbiór S (dość niewygodny w obliczeniach) jest pusty, sprowadziliśmy do sprawdzenia, czy należy do niego jeden konkretny punkt — skrajnie lewy punkt zbioru P. Co ważne, zbiór P jest znacznie łatwiejszy do opisania i do wyliczenia niż zbiór S, a przy tym można go łatwo aktualizować po dodaniu nowego koła do zbioru $\mathcal{K}$.

Algorytm

Niech $K_1, K_2, \ldots, K_n$ będą kolejnymi kołami opisanymi w danych. Ponadto przez *prawyPunkt*(K, K') oznaczmy funkcję zwracającą jako wartość skrajnie prawy punkt zbioru $K \cap K'$ — jeśli koła nie mają wspólnych punktów, to przyjmiemy, że funkcja zwraca wartość *NULL*.

```
procedure AkcjaKomandosów
  begin
    X:=prawyPunkt(K1,K2);
    if X=NULL then return 2;
    for i:=3 to n do
      for j:=1 to i−1 do
        Y = prawyPunkt(Ki, Kj);
        if Y=NULL then return i;
        if Y.x⩽X.x then X:=Y;
      for j:=1 to i do
        if X ∉ Kj return i;
    return NIE;
  end
```

Pozostaje jeszcze wyjaśnić, w jaki sposób znaleźć skrajnie prawy punkt części wspólnej dwóch kół. Wystarczy zauważyć, że jest to albo skrajnie prawy punkt któregoś z kół, albo któryś z punktów przecięcia okręgów będących brzegami zadanych kół. Potrafimy więc go znaleźć w czasie stałym. Stąd cały algorytm działa w czasie $O(n^2)$ i wymaga pamięci $O(n)$.

Rozwiązania zawodników

Większość rozwiązań nadesłanych do oceny opierała się na niepoprawnych algorytmach. W wielu programach sprawdzano jedynie, czy w danym momencie każda para kół ma niepuste przecięcie. Warunek ten — jakkolwiek jest warunkiem koniecznym na istnienie niepustego przecięcia wszystkich kół — nie jest dostateczny. Można łatwo narysować już trzy koła, z których każde dwa przecinają się, lecz część wspólna wszystkich trzech jest pusta. Rozwiązania bazujące na tej błędnej idei były oceniane na zero punktów.

Testy

Rozwiązania sprawdzane były przy użyciu 22 wygenerowanych losowo zestawów danych testowych. Testy *akc5a.in* i *akc5b.in*, jak również *akc20a.in* i *akc20b.in* były zgrupowane. Testy można podzielić na kilka grup:

- *akc1.in* – *akc5.in* — testy poprawnościowe
- *akc6.in* – *akc9.in* — testy wydajnościowe pozwalające odrzucić programy działające w czasie $O(n^4)$ i dłuższym
- *akc10.in* – *akc20.in* — testy wydajnościowe pozwalające odrzucić programy działające w czasie $O(n^3)$

Nazwa	n
akc1.in	23
akc2.in	43
akc3.in	58
akc4.in	70
akc5a.in	70
akc5b.in	100
akc6.in	150
akc7.in	176
akc8.in	250
akc9.in	360
akc10.in	500

Nazwa	n
akc11.in	788
akc12.in	943
akc13.in	1 008
akc14.in	1 000
akc15.in	1 363
akc16.in	1 503
akc17.in	1 703
akc18.in	2 000
akc19.in	2 000
akc20a.in	2 000
akc20b.in	2 000

Paweł Parys — Treść zadania, Opracowanie
Karol Cwalina — Program

OI, Etap III, dzień 2, 7–04–2005

Prawoskrętny wielbłąd

Bajtocja składa się z N oaz leżących na pustyni, przy czym żadne trzy oazy nie leżą na jednej linii prostej. Bajtazar mieszka w jednej z nich, a w każdej z pozostałych ma po jednym znajomym. Bajtazar chce odwiedzić jak najwięcej swoich znajomych. Zamierza pojechać na swoim wielbłądzie. Wielbłąd ten porusza się niestety w dosyć ograniczony sposób:

- *po wyjściu z oazy wielbłąd porusza się po linii prostej, aż dotrze do innej oazy;*
- *wielbłąd skręca tylko w oazach i tylko w prawo, o kąt z przedziału $[0°;180°]$ (wielbłąd może tylko raz obrócić się w oazie, tzn. nie jest dozwolony np. obrót o $200°$ w wyniku dwóch kolejnych obrotów o $100°$);*
- *wielbłąd nie chce chodzić po własnych śladach.*

Pomóż Bajtazarowi tak ustalić trasę podróży, aby odwiedził jak najwięcej znajomych. Powinna ona zaczynać i kończyć się w oazie, w której mieszka Bajtazar. Musi składać się z odcinków łączących kolejno odwiedzane oazy. Trasa ta nie może dwa razy przechodzić przez ten sam punkt, z wyjątkiem oazy Bajtazara, gdzie wielbłąd pojawia się dwukrotnie: na początku i na końcu trasy.

Początkowo wielbłąd Bajtazara jest już ustawiony w kierunku konkretnej oazy i musi wyruszyć w tym kierunku. Ustawienie wielbłąda po powrocie z podróży nie ma znaczenia.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia współrzędne oaz oraz ustawienie wielbłąda,*
- *obliczy maksymalną liczbę znajomych, których Bajtazar może odwiedzić zgodnie z powyższymi regułami,*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu wejścia zapisana jest jedna liczba całkowita N $(3 \leqslant N \leqslant 1\,000)$ — liczba oaz w Bajtocji. Oazy są ponumerowane od 1 do N. Bajtazar mieszka w oazie nr 1, a jego wielbłąd stoi zwrócony w stronę oazy nr 2. W kolejnych N wierszach umieszczone są współrzędne oaz. W $(i+1)$-szym wierszu znajdują się dwie liczby całkowite x_i, y_i — pozioma i pionowa współrzędna i-tej oazy — oddzielone pojedynczym odstępem. Wszystkie współrzędne są z zakresu od $-16\,000$ do $16\,000$.

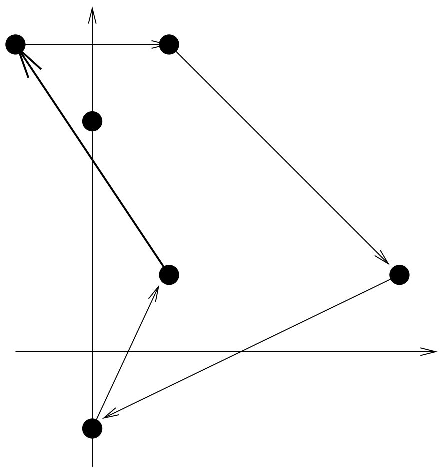
Wyjście

W pierwszym i jedynym wierszu wyjścia Twój program powinien wypisać jedną liczbę całkowitą — największą liczbę znajomych, których może odwiedzić Bajtazar.

Przykład

Dla danych wejściowych:

```
6
1 1
-1 4
0 -1
4 1
0 3
1 4
```



poprawnym wynikiem jest:

```
4
```

Rozwiązanie

Uwagi wstępne

Przedstawimy rozwiązanie zadania oparte na metodzie dynamicznej. Dla każdej pary oaz a, b będziemy liczyć, ilu znajomych może odwiedzić Bajtazar na drodze z początkowej oazy 1 do oazy b, przy założeniu, że do oazy b dotarł bezpośrednio z oazy a.

Zapiszmy oazy w układzie współrzędnych kątowych. Przyjmiemy, że środek układu współrzędnych jest położony w miejscu oazy 1 (od współrzędnych wszystkich oaz odejmujemy współrzędne oazy 1). Następnie ustalimy, że oaza 2 jest pierwsza w porządku kątowym, a pozostałym nadamy numery zgodnie z ruchem wskazówek zegara. Zauważmy, że na każdej poprawnej trasie Bajtazara oazy występują (niekoniecznie wszystkie) zgodnie z opisanym wyżej porządkiem.

Zdefiniujmy tablicę dwuwymiarową, w której będziemy zapisywać obliczone wyniki częściowe. Dla $1 \leqslant a < b \leqslant N$ niech $ile[a,b]$ oznacza największą możliwą liczbę znajomych, których może odwiedzić Bajtazar na drodze z oazy 1 do oazy b, przy założeniu, że do oazy b przybył bezpośrednio z oazy a. Rozwiązaniem zadania jest maksymalna z wartości $\{ile[a,b] : 1 < a < b \leqslant N\}$, ponieważ z każdej oazy b, do której Bajtazar doszedł niebezpośrednio z oazy początkowej, można powrócić do oazy początkowej zgodnie z zasadami podróżowania na prawoskrętnym wielbłądzie.

Obliczanie wartości ile[a,b]

Prawdziwe są następujące zależności:

(i) $ile[1,2] = 1$

(ii) $ile[1,c] = -\infty$ dla $c > 2$

(iii) $ile[b,c] = 1 + \max\{ile[a,b] : 1 \leqslant a < b$ oraz (a,b,c) tworzą zakręt w prawo$\}$

Aby wykorzystać powyższe zależności do obliczenia wartości tablicy *ile*, musimy potrafić szybko określać, czy droga pomiędzy trzema oazami tworzy zakręt w prawo.

Definicja 1 Powiemy, że wektor $VEC(a,b)$ jest *mniejszy* od $VEC(b,c)$ (zapiszemy to $VEC(a,b) \leqslant VEC(b,c)$), gdy droga z a do b i dalej do c skręca w prawo (kąt zorientowany pomiędzy odcinkami bc i ab ma wartość z przedziału $(0,180)$).

Dla trzech różnych oaz a, b i c zdefiniujemy także warunek $wPrawo(a,b,c)$ oznaczający, że prawoskrętny wielbłąd może z oazy a przejść bezpośrednio do oazy b i dalej (także bezpośrednio) do oazy c, a następnie powrócić (niekoniecznie bezpośrednio) do oazy 1:

$$VEC(a,b) \quad \leqslant \quad VEC(b,c) \tag{1}$$

oraz dla $a > 1$

$$VEC(a,b) \leqslant VEC(b,1) \quad \text{i} \quad VEC(b,c) \leqslant VEC(c,1) \tag{2}$$

Warunek (2) gwarantuje, że wielbłąd nie przecina trasy, po której już przeszedł (żaden z odcinków nie przecina półprostej z 1 do 2, czyli oaza 1 jest cały czas na prawo od trasy).

Aby określić wzajemne położenie dwóch wektorów należy obliczyć ich iloczyn wektorowy. Niech $\mathbf{v} = (x_1, y_1)$ oraz $\mathbf{u} = (x_2, y_2)$ będą dwoma wektorami — ich iloczyn wektorowy wynosi $\mathbf{v} \times \mathbf{u} = x_1 \cdot y_2 - x_2 \cdot y_1$. Jeśli jest on dodatni, to drugi wektor $\mathbf{u}$ jest skierowany w prawo od pierwszego; jeśli iloczyn jest ujemy, to wektor $\mathbf{u}$ jest skierowany na lewo od wektora $\mathbf{v}$; jeśli iloczyn jest zerem, to wektory są równoległe.

Algorytm

Teraz możemy zapisać algorytm obliczania wartości $ile[a,b]$. Trzy zagnieżdżone pętle sprawiają, że działa on w czasie $O(n^3)$.

```
zainicjalizuj wszystkie ile[a,b]= −∞;
ile[1,2]:=1;
for b:=2 to n do
  for c:=b+1 to n do
    for a:=1 to b−1 do
      if wPrawo(a,b,c) then ile[b,c]:=max(ile[b,c], 1+ile[a,b]);
return max{ile[a,b] : 1 < a < b ⩽ n};
```

Szybszy algorytm

Zauważmy, że w przedstawionej procedurze dla ustalonej oazy b musimy sprawdzić $\Theta(n^2)$ trójek (a,b,c), by znaleźć maksimum. Możemy jednak spróbować usprawnić przeglądanie oaz rozważając oazy c w takiej kolejności, by zbiór dopuszczalnych wartości a wzrastał.

W tym celu dla ustalonego b sortujemy oazy $1,2,\ldots,b-1$ (oazy a w algorytmie) w kolejności współrzędnych kątowych obliczonych względem oazy b — oznaczmy posortowane oazy $a_1,a_2,\ldots,a_{b-1}$. Podobnie sortujemy oazy o numerach $b+1,b+2,\ldots,n$ (oazy c w algorytmie) i podobnie posortowane oazy oznaczamy $c_{b+1},c_{b+2},\ldots,c_n$. Zaważmy, że dla każdej oazy c_i zbiór oaz a, które musimy dla niej sprawdzić, to początkowy fragment ciągu $a_1,a_2,\ldots,a_{b-1}$ i dodatkowo zbiory te rosną wraz z wzrostem indeksu i, czyli mamy następujące zależności.

Dla oazy c_i istnieje liczba $l_i \geqslant 0$ taka, że

$$\{a_j \,:\, wPrawo(a_j,b,c_i)\} \;=\; \{a_j \,:\, 1 \leqslant j \leqslant l_i\}.$$

Dodatkowo dla dowolnej oazy $a \in \{a_1,a_2,\ldots,a_{b-1}\}$ oraz liczb $b < j < k \leqslant n$ zachodzi implikacja

$$\text{jeśli } wPrawo(a,b,c_j) \text{ to } wPrawo(a,b,c_k),$$

stąd także $l_j < l_k$ oraz

$$ile[b,c_j] \leqslant ile[b,c_k].$$

Po zastosowaniu powyższych spostrzeżeń mamy następujący algorytm.

```
zainicjalizuj wszystkie ile[a,b]= −∞;
ile[1,2]:=1;
for b:=2 to n do
   (a_1,...,a_{b−1}):= oazy 1...b−1  posortowane kątowo względem  b;
   (c_{b+1},...,c_n):= oazy b+1...n  posortowane kątowo względem  b;
   a:=a_1;
   for c:=c_{b+1} to c_n do
      while wPrawo(a,b,c) do
         ile[b,c]:=max(ile[b,c], 1+ile[a,b]);
         a:=next(a);
      ile[b,c+1]:=ile[b,c];
return max{ile[a,b] : 1 < a < b ⩽ n};
```

Po wprowadzonych zmianach algorytm dla jednej wartości b działa w czasie $O(n) + czas\_sortowania = O(n\log n)$. Stąd całość działa w czasie $O(n^2\log n) + czas\_sortowania$ (wstępne sortowanie potrzebne do poprawnego, wstępnego ponumerowania oaz) i wymaga $O(n^2)$ pamięci.

Dalsze ulepszenia

Rozwiązanie można jeszcze nieco przyśpieszyć, ograniczając dla każdego b zbiory przeglądanych a i c do takich, że $VEC(a,b) \leqslant VEC(b,1)$ i $VEC(b,c) \leqslant VEC(c,1)$. Pozwala to sortować nieco mniejsze zbiory i jednocześnie upraszcza obliczanie funkcji *wPrawo* sprowadzając je do sprawdzenia warunku $VEC(a,b) \leqslant VEC(b,c)$.

Kolejne ulepszenie jest związane z efektywniejszym wykonywaniem porównań dla kątów. W każdej wersji rozwiązania pojawia się sortowanie punktów według współrzędnych kątowych, a tym samym konieczność wykonywania licznych porównań kątów pomiędzy wektorami, realizowanych poprzez operacje iloczynu skalarnego i wektorowego. Wykonywanie takich porównań siłą rzeczy jest trochę wolniejsze od zwykłego porównywania liczb — możemy je jednak zastąpić porównywaniem liczb. W tym celu dla wektorów zdefiniujemy pewną funkcję rosnącą wraz z kątem sortowania, obliczymy jej wartości dla wszystkich rozważanych wektorów i będziemy sortować wektory według jej wartości.

Przykładem funkcji odpowiedniej dla kąta nachylenia względem wektora $[0,1]$ (tzn. funkcji, którą możemy zastosować, gdy oaza 2 ma współrzędne $(0,1)$) jest:

$$f([x,y]) = \begin{cases} \frac{x}{x+y} & \text{gdy } x>0,\ y>0 \\ 2-\frac{x}{x-y} & \text{gdy } x>0,\ y<0 \\ 2+\frac{x}{x+y} & \text{gdy } x<0,\ y<0 \\ 4-\frac{x}{x-y} & \text{gdy } x<0,\ y>0 \end{cases}$$

Wartości powyższej funkcji są liczbami wymiernymi i ich reprezentacje w komputerze mogą być obarczone błędami zaokrągleń. Aby zmniejszyć błędy możemy wszystkie wartości przemnożyć przez odpowiednio dużą liczbę naturalną i zaokrąglić do wartości całkowitych (za mnożnik przyjmujemy wartość większą niż $(4 \cdot \mathit{MaxWspółrzędna})^2$).

Przedstawiona modyfikacja przyśpiesza działanie algorytmu około czterokrotnie. Jako program wzorcowy został zaimplementowany powyższy algorytm z wszystkimi usprawnieniami.

Uwagi końcowe

Zadanie jest dość trudne. Co prawda nie są w rozwiązaniu potrzebne żadne skomplikowane algorytmy, ale nagromadzenie prostych algorytmów i różnych przypadków jest na tyle duże, że kompletna implementacja nie jest sprawą prostą.

Testy

Zadanie było testowane na 11 danych testowych.

Nazwa	n	wynik	Opis
pra1.in	11	6	mały prosty test
pra2.in	15	6	mały test poprawnościowy
pra3.in	19	7	mały test poprawnościowy
pra4.in	40	9	losowy test poprawnościowy
pra5.in	106	52	test ze względnie dużą odpowiedzią
pra6.in	200	31	test losowy
pra7.in	500	43	test losowy
pra8.in	500	23	test losowy
pra9.in	743	316	test ze stosunkowo dużą odpowiedzią
pra10.in	997	31	test losowy
pra11.in	1000	61	test losowy

Wojciech Rytter — Treść zadania, Opracowanie
Marek Cygan — Program

OI, Etap III, dzień 2, 7–04–2005

Autobus

Ulice Bajtogrodu tworzą szachownicę — prowadzą z północy na południe lub ze wschodu na zachód. Ponadto każda ulica prowadzi na przestrzał przez całe miasto — każda ulica biegnąca z północy na południe krzyżuje się z każdą ulicą biegnącą ze wschodu na zachód i vice versa. Ulice prowadzące z północy na południe są ponumerowane od 1 do n, w kolejności z zachodu na wschód. Ulice prowadzące ze wschodu na zachód są ponumerowane od 1 do m, w kolejności z południa na północ. Każde skrzyżowanie i-tej ulicy biegnącej z północy na południe i j-tej ulicy biegnącej ze wschodu na zachód oznaczamy parą liczb (i,j) (dla $1 \leqslant i \leqslant n, 1 \leqslant j \leqslant m$).

Po ulicach Bajtogrodu kursuje autobus. Zaczyna on trasę przy skrzyżowaniu $(1,1)$, a kończy przy skrzyżowaniu (n,m). Ponadto autobus może jechać ulicami tylko w kierunku wschodnim i/lub północnym.

Przy pewnych skrzyżowaniach oczekują na autobus pasażerowie. Kierowca autobusu chce tak wybrać trasę przejazdu autobusu, aby zabrać jak najwięcej pasażerów. (Zakładamy, że bez względu na wybór trasy i tak wszyscy pasażerowie zmieszczą się w autobusie.)

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis siatki ulic oraz liczbę pasażerów czekających przy poszczególnych skrzyżowaniach,*
- *obliczy, ilu maksymalnie pasażerów może zabrać autobus,*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu pliku wejściowego zapisane są trzy dodatnie liczby całkowite n, m i k — odpowiednio: liczba ulic biegnących z północy na południe, liczba ulic biegnących ze wschodu na zachód i liczba skrzyżowań, przy których pasażerowie czekają na autobus ($1 \leqslant n \leqslant 10^9$, $1 \leqslant m \leqslant 10^9$, $1 \leqslant k \leqslant 10^5$).

Kolejne k wierszy opisuje rozmieszczenie pasażerów czekających na autobus, w jednym wierszu opisani są pasażerowie czekający na jednym ze skrzyżowań. W wierszu $(i+1)$-szym znajdują się trzy dodatnie liczby całkowite x_i, y_i i p_i, oddzielone pojedynczymi odstępami, $1 \leqslant x_i \leqslant n$, $1 \leqslant y_i \leqslant m$, $1 \leqslant p_i \leqslant 10^6$. Taka trójka liczb oznacza, że przy skrzyżowaniu (x_i, y_i) oczekuje p_i pasażerów. Każde skrzyżowanie pojawia się w danych wejściowych co najwyżej raz. Łączna liczba oczekujących pasażerów nie przekracza 1 000 000 000.

Wyjście

Twój program powinien na wyjściu wypisać jeden wiersz zawierający jedną liczbę całkowitą — maksymalną liczbę pasażerów, których może zabrać autobus.

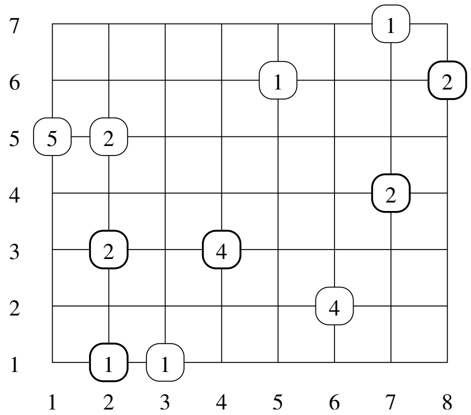
Przykład

Dla danych wejściowych:

```
8 7 11
4 3 4
6 2 4
2 3 2
5 6 1
2 5 2
1 5 5
2 1 1
3 1 1
7 7 1
7 4 2
8 6 2
```

poprawnym wynikiem jest:

```
11
```



W tym przypadku Twój program powinien podać wynik 11. Na rysunku zaznaczono skrzyżowania, przez które przejedzie autobus zabierając 11 pasażerów.

Rozwiązanie

Najbardziej naturalną techniką algorytmiczną, którą można zastosować do rozwiązania zadania, jest *programowanie dynamiczne*. Metoda ta polega na wyznaczeniu dość licznego *zbioru podproblemów*, które następnie rozwiązujemy w *odpowiedniej kolejności*, tak by w trakcie rozwiązywania kolejnego z nich móc wykorzystać uzyskane wcześniej wyniki.

W naszej sytuacji podproblemem będzie wyznaczenie maksymalnej liczby pasażerów, których autobus może dowieźć do konkretnego przystanku. Niech $\mathcal{X}$ oznacza zbiór przystanków (punktów) i niech wartością $W[v]$ dla punktu $v \in \mathcal{X}$ będzie maksymalna liczba pasażerów, których może zabrać autobus dojeżdżając do punktu v (włącznie z pasażerami stojącymi na przystanku v). Oznaczymy również porządek, w jakim autobus może dojeżdżać do przystanków, tzn. $u \lhd v$, dla $u, v \in \mathcal{X}$, jeśli $u \neq v$ i autobus może dojechać z przystanku u do przystanku v zgodnie z zasadami podanymi w zadaniu.

Niech $\{(x_i, y_i, p_i) : 1 \leqslant i \leqslant k\}$ będzie zbiorem danych wejściowych, gdzie x_i jest numerem kolumny, y_i jest numerem wiersza, p_i jest wartością przypisaną do i-tego punktu (liczbą pasażerów na i-tym przystanku). Załóżmy, że punkt (n, m) (skrajnie północno-wschodnie skrzyżowanie) też należy do zbioru wejściowego. Jeśli nie, to uzupełnimy dane o trójkę $(n, m, 0)$ (zwiększając także odpowiednio wartość k), co nie zmieni rozwiązania.

Pierwszy algorytm

Teraz wystarczy przeglądać podproblemy w dowolnym porządku zgodnym z porządkiem $\lhd$ i mamy pierwszy algorytm (zauważmy, że takim porządkiem może być na przykład przeglądanie punktów kolumnami od lewej do prawej, a w każdej kolumnie wierszami od najniższego do najwyższego; równie dobra jest kolejność od najniższego do najwyższego wiersza, a w każdym wierszu od lewej do prawej). Zakładamy, że początkowo wartością $W[v]$ jest liczba pasażerów czekających na przystanku v.

```
1: procedure Algorytm „brutalny”
2:   begin
3:     forall v ∈ X w jakimkolwiek porządku zgodnym z ◁
4:       do
5:         W[v]:=W[v]+max{W[w]: w ◁ v };
6:     return W[(n,m)];
7:   end
```

Przedstawiony algorytm działa w czasie $O(k^2)$ (k iteracji pętli (3), a znalezienie każdego maksimum w wierszu (5) wymaga czasu $O(k)$). Pokażemy teraz, że możliwe jest przyśpieszenie jego działania.

Algorytm sprytniejszy

Zdefiniujmy dwa dodatkowe porządki na zbiorze punktów (x_i, y_i).

Definicja 1 Porządek *wierszowo-kolumnowy*:

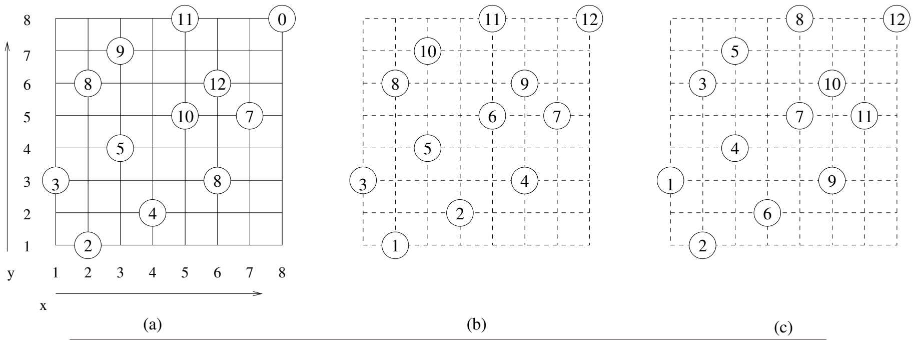
$$(a,b) \overset{w}{\prec} (c,d) \equiv [(b<d) \text{ lub } (b=d \text{ i } a<c)]$$

oraz porządek *kolumnowo-wierszowy*:

$$(a,b) \overset{k}{\prec} (c,d) \equiv [(a<c) \text{ lub } (a=c \text{ i } b<d)]$$

Przykład. Rozważmy dane: (2,1,2), (4,2,4), (1,3,3), (6,3,8), (3,4,5), (5,5,10), (7,5,7), (2,6,8), (6,6,12), (3,7,9), (5,8,11), (8,8,0).

Na rysunku 1 przedstawione są liczby pasażerów czekających na przystankach oraz numeracja wierszowo-kolumnowa i kolumnowo-wierszowa przystanków.



Rys. 1: (a) Dane wejściowe. (b) Porządek wierszowo-kolumnowy przystanków. (c) Porządek kolumnowo-wierszowy przystanków.

Redukcja rozmiaru współrzędnych

Teraz możemy nieco uprościć dane wejściowe. Niech $k_1 < k_2 < \ldots < k_r$ będą współrzędnymi kolumn, w których znajduje się przynajmniej jeden punkt. Podobnie niech $w_1 < w_2 < \ldots < w_s$ będą numerami wierszy, w których znajduje się przynajmniej jeden punkt. Zastąpmy teraz punkt (x_i, y_i) znajdujący się w kolumnie k_j i w wierszu w_l przez punkt (j, l). Zauważmy, że punkty po transformacji pozostają w takim samym porządku względem relacji $\lhd$, relacji $\overset{k}{\prec}$ oraz relacji $\overset{w}{\prec}$. Mamy natomiast gwarancje, że wszystkie współrzędne mieszczą się w przedziale $[1, k]$.

Algorytm

Uporządkujmy dane punkty (x_i, y_i) zgodnie z porządkiem wierszowo-kolumnowym, tzn.

$$(x_j, y_j) \overset{w}{\prec} (x_l, y_l), \text{ dla } j < l.$$

Następnie przyjmijmy, że $(i_1, i_2, \ldots i_k)$ to numery punktów wypisane w porządku kolumnowo-wierszowym, tzn.

$$(x_{i_j}, y_{i_j}) \overset{k}{\prec} (x_{i_l}, y_{i_l}), \text{ dla } j < l.$$

Oba porządki możemy wyznaczyć w czasie liniowym korzystając z algorytmu *radix-sort* (możemy go zastosować, bo po redukcji wartości, według których sortujemy, są niewielkie).

Załóżmy, że początkowo mamy tablicę zawierającą liczby pasażerów na kolejnych przystankach według porządku wierszowo-kolumnowego — oznaczmy ją tym razem $P = [p_1, p_2, \ldots p_k]$. Niech natomiast W będzie, jak poprzednio, tablicą, w której będziemy zapisywać wyliczane kolejno wyniki.

```
1: procedure Algorytm sprytniejszy
2:   begin
3:     Wypełnij tablicę W zerami;
4:     Oblicz porządek kolumnowo-wierszowy punktów N=[i_1, i_2, ... i_k];
5:     for j:=1 to k do
6:       W[N[j]]=P[N[j]]+max{W[t] : t < N[j]};
7:     return W[k]
8:   end
```

Przykład (c.d.) W naszym przykładzie mamy początkowo:

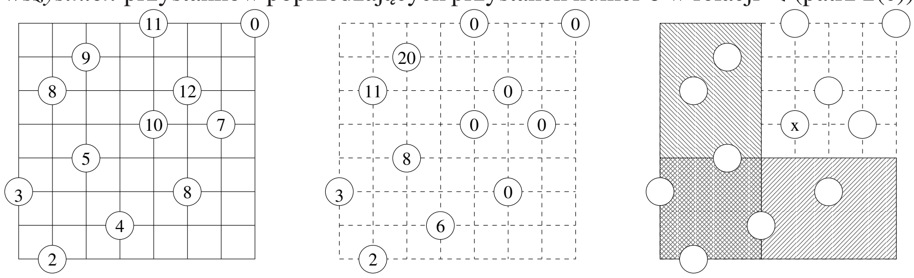
$$P = [2,4,3,8,5,10,7,8,12,9,11,0], \text{ oraz } N = [3,1,8,5,10,2,6,11,4,9,7,12].$$

Załóżmy, że wykonaliśmy już sześć iteracji pętli (4). Wówczas zawartość tablicy W jest następująca (patrz też rysunek 2(b)):

$$W = [2,6,3,0,8,0,0,11,0,20,0,0].$$

Dla sześciu pierwszych punktów w porządku kolumnowo-wierszowym jest to już poprawny wynik, a pozostałe wartości są zerami. Podczas obliczania kolejnej wartości $W[N[7]] = W[6]$ wybieramy maksimum ze zbioru wartości $\{W[1], \ldots, W[5]\}$. Wśród nich są dobrze policzone wartości $\{W[1], W[2], W[3], W[5]\}$ oraz nieistotna (zerowa) wartość $\{W[4]\}$.

Co najważniejsze, dobrze wyliczone wartości $\{W[1], W[2], W[3], W[5]\}$, to są wartości dla *wszystkich* przystanków poprzedzających przystanek numer 6 w relacji $\lhd$ (patrz 2(c)).



(a) (b) (c)

Rys. 2: (a) Dane wejściowe — tablica P. (b) Stan tablicy W po sześciu iteracjach pętli (4). (c) Zaznaczony obszar, w którym znajdują się przystanki wcześniejsze od x według porządku wierszowo-kolumnowego (4 dolne wiersze), przystanki wcześniejsze od x według porządku kolumnowo-wierszowego (4 kolumny z lewej) oraz przystanki wcześniejsze według relacji $\lhd$ (część wspólna powyższych obszarów).

Złożoność i poprawność algorytmu

Algorytm nazwaliśmy sprytniejszym, ponieważ teraz maksimum obliczamy dla łatwiejszego obliczeniowo zbioru — początkowego segmentu tablicy W. Operację $\max\{W[t] : t < N[j]\}$

można wykonać w czasie logarytmicznym, jeśli umieścimy elementy tablicy W w liściach regularnego *kopca*, patrz [14]. W każdym wewnętrznym węźle v kopca jest zawarta maksymalna wartość liścia z poddrzewa o korzeniu v.

Zamiast kopca można zastosować także zbalansowane drzewa uporządkowane (na przykład drzewo AVL), ale kopiec jest znacznie prostszy w implementacji. Można go zapisać w tablicy, a operacje przechodzenia po drzewie realizować za pomocą prostych operacji arytmetycznych.

Czas działania algorytmu „sprytnego” wynosi $O(k \log k)$.

Poprawność algorytmu wynika stąd, że obliczamy wartości w pewnej kolejności zgodnej z porządkiem $\lhd$, a jeśli licząc wartość dla pewnego punktu v, sięgamy po wartość pewnego punktu w (licząc max), to albo $w \lhd v$ (wtedy wartość $W[w]$ jest już dobrze policzona), albo $W[w] = 0$ (co nie ma wpływu na wynik). Mówiąc w sposób *nieprzyjemnie formalny* relacja częściowego porządku $\lhd$ jest częścią wspólną liniowego (pełnego) porządku wierszowo-kolumnowego i kolumnowo-wierszowego (patrz także raz jeszcze rysunek 2(c)).

Testy

Rozwiązania zawodników były sprawdzane na 10 testach. Zostały one w większości wygenerowane losowo.
W poniższej tabeli liczby n, m to rozmiary siatki ulic, natomiast k to liczba przystanków.

Nazwa	n	m	k
aut1.in	50	50	100
aut2.in	500	500	1000
aut3.in	5 000	5 000	2000
aut4.in	50 000	50 000	2000
aut5.in	300 000	100 000	10 000
aut6.in	500 000	500 000	30 000
aut7.in	1 000 000	100 000	50 000
aut8.in	10 000 000	20 000 000	70 000
aut9.in	50 000 000	400 000 000	90 000
aut10.in	1 000 000 000	1 000 000 000	100 000

Zbigniew Czech
Treść zadania

Marcin Pilipczuk
Opracowanie, Program

OI, Etap III, dzień 2, 7–04–2005

Lustrzana pułapka

Lustrzana pułapka to prostopadłościan zbudowany z luster, których powierzchnie odbijające są skierowane do wnętrza prostopadłościanu. Dokładnie w środku prostopadłościanu zawieszony jest miniaturowy laser o rozmiarach punktu. Zadanie polega na takim skierowaniu lasera, aby jego promień przebył jak najdłuższą drogę i trafił w laser. Przy czym przez długość drogi rozumiemy sumę odległości, jaką przebył promień lasera w każdym z trzech kierunków równoległych do krawędzi luster (czyli mierzymy tzw. metryką miejską).

Wymiary lustrzanej pułapki są parzystymi liczbami całkowitymi. Krawędzie oraz wierzchołki, w których lustra stykają się ze sobą, nie odbijają promieni lasera. Wewnątrz pułapki określamy układ współrzędnych: osie układu są równoległe do krawędzi pułapki, a laser znajduje się w początku układu współrzędnych. Laser można skierować na dowolny punkt wewnątrz pułapki o współrzędnych całkowitych, włączając w to punkty na powierzchni luster (z wyjątkiem samego lasera, czyli punktu $(0,0,0)$).

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia wymiary lustrzanej pułapki,*
- *wyznaczy kierunek takiego ustawienia lasera, że promień wystrzelony przez laser:*
  - *będzie odbijał się od luster, choć niekoniecznie od wszystkich,*
  - *nie trafi w krawędź ani w wierzchołek pułapki,*
  - *trafi ponownie do lasera, choć być może z innego kierunku,*
  - *przebędzie możliwie najdłuższą drogę (w sensie wyżej zdefiniowanej odległości).*
- *wypisze wynik na standardowe wyjście.*

Wejście

Pojedynczy test składa się z wielu lustrzanych pułapek do przeanalizowania. W pierwszym wierszu wejścia podana jest jedna liczba całkowita $1 \leqslant K \leqslant 1000$, oznaczająca liczbę pułapek do rozpatrzenia. W wierszach $2 \ldots K+1$ opisane są pułapki, po jednej w wierszu. Opis pułapki składa się z trzech liczb $5 \leqslant x, y, z \leqslant 1000$, oddzielonych pojedynczymi odstępami. Lustrzana pułapka ma wymiary $2x \times 2y \times 2z$.

Wyjście

Twój program powinien wypisać dokładnie K wierszy. W wierszu o numerze i powinno się znaleźć rozwiązanie dla pułapki o numerze i: trzy liczby całkowite k_x, k_y, k_z, oddzielone *pojedynczymi odstępami,* $-x \leqslant k_x \leqslant x$*,* $-y \leqslant k_y \leqslant y$*,* $-z \leqslant k_z \leqslant z$*,* $(k_x, k_y, k_z) \neq (0,0,0)$*. Liczby te oznaczają, że w i-tej pułapce laser powinien być skierowany na punkt o współrzędnych* (k_x, k_y, k_z)*.*

Jeżeli istnieje wiele poprawnych wyników, Twój program powinien wypisać dowolny z nich.

Przykład

Dla danych wejściowych:

```
2
5 6 7
5 6 6
```

poprawnym wynikiem jest:

```
4 5 6
4 6 5
```

Rozwiązanie

Wstęp

Streszczenie

Rozwiązanie wzorcowe składa się z dwóch części.

Najpierw zajmiemy się sprawdzeniem, czy promień lasera skierowany w określony punkt powróci do środka pułapki nie trafiając wcześniej w żadną krawędź. Pokażemy najpierw algorytm działający w czasie $O(xyz)$ dla pułapki o rozmiarach $2x \times 2y \times 2z$. Następnie ulepszymy rozwiązanie przyśpieszając algorytm do czasu $O(\log(xyz))$. Algorytm dodatkowo będzie zwracał długość drogi, którą przebył promień.

Mając powyższy algorytm możemy spróbować przetestować wszystkie ustawienia lasera i sprawdzić, które z nich gwarantuje najdłuższą podróż promienia przed powrotem do punktu wyjścia. Takie rozwiązanie, działające w czasie $O(xyz \cdot \log(xyz))$, jest jednak zbyt wolne przy ograniczeniach podanych w zadaniu. Dlatego też w drugiej części rozwiązania zastanowimy się nad wyeliminowaniem niektórych ustawień. Pokażemy, że przeglądając je w odpowiedniej kolejności już po krótkim czasie będziemy mogli zakończyć poszukiwania, uznawszy, że lepszego rozwiązania nie znajdziemy.

Oznaczenia

Wprowadźmy następujące pomocnicze oznaczenia:

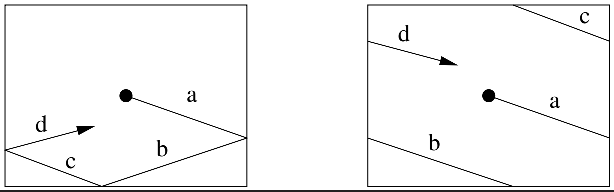
- $ord(n) = max\{k : 2^k \mid n\}$, czyli jest to maksymalna potęga dwójki, przez którą można podzielić liczbę n;
- punkt o nieujemnych współrzędnych całkowitych (a,b,c) nazwiemy *punktem pierwotnym* wtedy i tylko wtedy, gdy $NWD(a,b,c) = 1$.

Uproszczenia

Zauważmy, że zamiast rozważać promień lasera r odbijający się od ściany pułapki, możemy rozważać „promień zawinięty” $w(r)$ o stałym kierunku, który w momencie uderzenia w ścianę wychodzi dalej z odpowiedniego punktu na równoległej ścianie (patrz przykład dla przypadku dwuwymiarowego na rysunku 1). Choć promień $w(r)$ biegnie inną drogą niż oryginalny, to pod pewnymi względami jest mu równoważny:

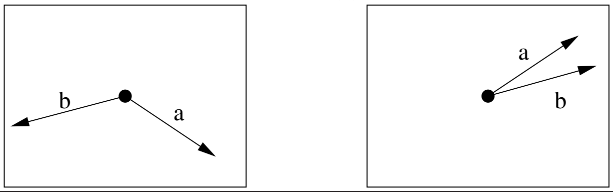
- promień r jest w chwili t w punkcie o współrzędnych całkowitych wtedy i tylko wtedy, gdy promień $w(r)$ jest w chwili t w punkcie o współrzędnych całkowitych;
- promień r trafia w krawędź w chwili t wtedy i tylko wtedy, gdy promień $w(r)$ trafia w krawędź w chwili t;
- promień r wraca do środka pułapki w chwili t po przebyciu drogi d wtedy i tylko wtedy, gdy promień $w(r)$ wraca do środka pułapki w chwili t po przebyciu drogi d.

Prawdziwość tych własności możemy łatwo uzasadnić, gdy tylko zauważymy, że punkty $P(t)$ i $P'(t)$ w których znajdują się odpowiednio promień r i promień $w(r)$ w chwili t, mają odpowiednie współrzędne równe z dokładnością do znaku. To sprawia, że możemy rozważać promień $w(r)$ zamiast promienia r.



Rys. 1: Oryginalny promień r oraz „promień zawinięty” $w(r)$. Dla czytelności odpowiadające sobie fragmenty drogi promienia r i $w(r)$ oznaczono takimi samymi etykietami.

Następnie zauważmy, że możemy ograniczyć się do rozważania promieni wystrzelonych z lasera skierowanego w punkty o współrzędnych nieujemnych (patrz przykład dla przypadku dwuwymiarowego na rysunku 2). Dla każdego innego promienia istnieje równoważny mu (w takim samym sensie jak równoważne są r i $w(r)$) skierowany właśnie w takim kierunku.



Rys. 2: Dwa dowolne promienie oraz równoważne im promienie skierowane na punkty o współrzędnych nieujemnych.

Wobec powyższego przyjmijmy następujące założenia:

- lewy-dolny róg pułapki znajduje się w punkcie $(-x,-y,-z)$, prawy-górny róg jest w punkcie (x,y,z), a laser jest umieszczony w punkcie $(0,0,0)$;
- punkty, których kolejne współrzędne różnią się o całkowitą wielokrotność liczb $2x$, $2y$ i $2z$, będziemy utożsamiać (tzn. punkt (a,b,c), dla $a \in (2ix-x, 2ix+x]$,

$b \in (2jy - y, 2jy + y]$ oraz $c \in (2kz - z, 2kz + z]$ dla pewnych całkowitych i, j oraz k, oznacza punkt o współrzędnych $(a - 2ix, b - 2jy, c - 2kz)$), a operacje arytmetyczne na kolejnych współrzędnych będziemy przeprowadzać cyklicznie odpowiednio w przedziałach: $(-x, x]$, $(-y, y]$ oraz $(-z, z]$;

- powiemy, że laser *strzela* w (a, b, c) jeśli uruchamiamy go po skierowaniu na ten właśnie punkt;
- rozważając laser strzelający w punkt (a, b, c) będziemy mówić, że jego promień pokonuje odległość od środka pułapki do punktu (a, b, c) (czyli według warunków zadania odległość $a + b + c$) w jednym *kroku*;
- powyższy promień będzie po t krokach (dla dowolnego t, także niecałkowitego i/lub niedodatniego) w punkcie o współrzędnych (ta, tb, tc);

Na drodze do algorytmu wzorcowego

Na początku zanotujmy kilka spostrzeżeń. Z przyczyn, które staną się jasne już za chwilę, spostrzeżenia te będą dotyczyły zarówno sytuacji w przestrzeni trójwymiarowej, jak i na płaszczyźnie — w przestrzeni dwuwymiarowej.

Lemat 1 *Rozważmy strzał w pułapce trójwymiarowej w punkt pierwotny (a, b, c). Punkt, w którym jest promień po t krokach, ma współrzędne całkowite wtedy i tylko wtedy, gdy t jest liczbą całkowitą.*

Dla pułapki dwuwymiarowej zachodzi analogiczna własność.

Dowód Po czasie t rozważany promień jest w punkcie o współrzędnych (ta, tb, tc). Oczywiście jeśli liczba t jest całkowita, to wszystkie współrzędne tego punktu też są całkowite. Załóżmy teraz, że liczby ta, tb i tc są całkowite. To oznacza, że liczba t musi być wymierna, czyli można ja przedstawić w postaci nieskracalnego ułamka $t = \frac{p}{q}$ i $NWD(p, q) = 1$. Ale stąd wynika, że liczba q dzieli liczbę a, a także liczby b i c. Widzimy więc, że $q = 1$, gdyż punkt (a, b, c) jest pierwotny i $NWD(a, b, c) = 1$.

Dowód można praktycznie bez zmian zastosować do przypadku dwuwymiarowego. ■

Wiosek 1

1. Jeśli po strzale w punkt (a, b, c) promień wróci do środka pułapki po przebyciu drogi d, to po strzale w punkt pierwotny $(\frac{a}{NWD(a,b,c)}, \frac{b}{NWD(a,b,c)}, \frac{c}{NWD(a,b,c)})$ promień wróci do środka pułapki po przebyciu tej samej drogi d.
2. Laser skierowany w punkt pierwotny (a, b, c) może powrócić do środka pułapki tylko po całkowitej liczbie kroków, czyli po przebyciu drogi $t(a + b + c)$ dla pewnej liczby całkowitej t.
3. Jeżeli po strzale w punkt (a, b, c) promień nie trafi po drodze w krawędź, to zawsze powróci do środka pułapki po wykonaniu nie więcej niż $2x \cdot 2y \cdot 2z$ kroków — tyle jest wszystkich punktów w pułapce.

Teraz pozostaje już tylko zbadać, czy po strzale w punkt pierwotny (a,b,c) promień nie ugrzęźnie na krawędzi. W tym celu dla promienia r oznaczmy przez $P_Z(r)$ jego rzut na ścianę pułapki prostopadłą do osi OZ. Zauważmy, że promień r trafia w jedną z krawędzi pułapki równoległych do osi OZ wtedy i tylko wtedy, gdy jego rzut $P_Z(r)$ trafia w jeden z wierzchołków ściany, na którą został zrzutowany. Stąd mamy kolejne spostrzeżenie, które pozwoli nam odszukać moment uderzenia promienia w krawędź, o ile ono nastąpi.

Wiosek 2

1. Promień r po strzale w punkt pierwotny (a,b,c) trafi w krawędź równoległą do osi OZ wtedy i tylko wtedy, gdy promień $P_Z(r)$ po strzale na płaszczyźnie XOY w punkt (a,b) trafi w punkt (x,y).

2. Promień $P_Z(r)$ biegnie taką samą drogą jak promień r' wystrzelony na płaszczyźnie XOY w punkt pierwotny $\left(\frac{a}{NWD(a,b)}, \frac{b}{NWD(a,b)}\right)$.

3. Promień r' może trafić w wierzchołek ściany tylko po całkowitej liczbie kroków.

Dokonane dotychczas spostrzeżenia pozwalają nam sformułować pierwszy algorytm rozwiązania zadania.

```
procedure Luss1;
  begin
    Tmax=0;
    for a=0 to x do
      for b=0 to y do
        for c=0 to z do
          if NWD(a,b,c)=1 then
            t=(a+b+c)·sprawdźStrzał(a,b,c);
            if t > Tmax then
              Tmax=t;
              (aOpt,bOpt,cOpt)=(a,b,c);
    if Tmax>0 then return (aOpt,bOpt,cOpt)
    else return NULL;
  end
```

```
procedure sprawdźStrzał(a,b,c)
  begin
    a1=a/NWD(a,b); b1=b/NWD(a,b);
    for t=1 to 2x·2y do
      if (ta1,tb1) = (x,y) then return 0;
    a1=a/NWD(a,c); c1=c/NWD(a,c);
    for t=1 to 2x·2z do
      if (ta1,tc1) = (x,z) then return 0;
    b1=b/NWD(b,c); c1=c/NWD(b,c);
    for t=1 to 2y·2z do
      if (tb1,tc1) = (y,z) then return 0;
    for t=1 to 2x·2y·2z do
      if (ta,tb,tc) = (0,0,0) then return t;
  end
```

Algorytm ten dla każdego punktu pierwotnego wykonuje test wymagający czasu xyz. Nie dokonując głębszej analizy liczności zbioru punktów pierwotnych dostajemy stąd ograniczenie złożoności czasowej algorytmu $O((xyz)^2)$. Teoretycznie nie jest więc to szybki algorytm — w praktyce także mieści się w limicie czasowym tylko na jednym z przygotowanych testów.

Przyśpieszenie

Zamiast śledzić bieg promienia i sprawdzać krok po kroku, czy nie powrócił on do środka pułapki, postaramy się wyliczyć moment trafienia promienia we wskazany punkt. Rozważmy strzał w punkt (a,b,c) — oczywiście pierwotny — i niech T oznacza liczbę kroków, po których tak wystrzelony promień po raz pierwszy powróci do środka pułapki, o ile nie zatrzyma go wcześniej żadna krawędź. Z lematu 1 wiemy, że T jest liczbą całkowitą. Podamy wzór, który pozwoli ją wyliczyć.

Lemat 2
$$T = NWW\left(\frac{NWW(2x,a)}{a}, \frac{NWW(2y,b)}{b}, \frac{NWW(2z,c)}{c}\right)$$
(jeśli któraś z liczb a, b lub c jest zerem, to przyjmiemy, że odpowiedni z argumentów NWW jest równy 1*).*

Dowód Liczbę T zdefiniowaliśmy jako liczbę kroków, po których promień wraca do środka pułapki, więc
$$(Ta,Tb,Tc) = (0,0,0),$$
czyli
$$Ta \equiv 0 \ mod \ 2x, \ \ Tb \equiv 0 \ mod \ 2y, \ \ Tc \equiv 0 \ mod \ 2z,$$
skąd
$$2x \mid Ta, \ \ 2y \mid Tb, \ \ 2z \mid Tc.$$
Mamy więc
$$NWW(2x,a) \mid Ta, \ \ NWW(2y,b) \mid Tb, \ \ NWW(2z,c) \mid Tc$$

i dalej

$$\frac{NWW(2x,a)}{a} \mid T, \quad \frac{NWW(2y,b)}{b} \mid T, \quad \frac{NWW(2z,c)}{c} \mid T,$$

czyli

$$NWW\Big(\frac{NWW(2x,a)}{a}, \quad \frac{NWW(2y,b)}{b}, \quad \frac{NWW(2z,c)}{c}\Big) \mid T.$$

Teraz wystarczy sprawdzić, że po T' krokach, gdzie

$$T' = NWW\Big(\frac{NWW(2x,a)}{a}, \frac{NWW(2y,b)}{b}, \frac{NWW(2z,c)}{c}\Big)$$

promień znajdzie się w środku pułapki (zakładając, że wcześniej nie trafi w żadną krawędź). Zauważmy, że

$$\frac{NWW(2x,a)}{a} \mid T', \ \text{ więc } \ NWW(2x,a) \mid aT', \ \text{ stąd } \ 2x \mid aT'.$$

Po analogicznym sprawdzeniu pozostałych współrzędnych widzimy, że

$$(T'a, T'b, T'c) \ = \ (0,0,0).$$

■

Poszukiwanie drogi przebytej przez promień z wykorzystaniem wyniku powyższego lematu przebiega w czasie $O(\log(xyz))$ — wystarczy do wyliczenia wartości T zastosować algorytm Euklidesa. Pamiętajmy jednak, że jest to tylko droga hipotetyczna, nie wiemy bowiem, czy promień nie trafi w krawędź przed powrotem do środka pułapki. Spróbujmy więc teraz podać wzór pozwalający rozstrzygnąć, czy promień trafi w krawędź.

Lemat 3 *Promień wystrzelony w punkt (a,b,c) trafi w krawędź równoległą do osi OZ wtedy i tylko wtedy, gdy $ord(a) - ord(b) = ord(x) - ord(y)$. Analogiczne implikacje zachodzą dla pozostałych krawędzi biegnących w pozostałych kierunkach.*

Dowód Rozważmy krawędź równoległą do osi OZ oraz rzut promienia lasera na współrzędne X i Y. Promień trafi w krawędź wtedy i tylko wtedy, gdy

$$\text{istnieje liczba rzeczywista } t \in (0,T): \ (at, bt) = (x,y) \tag{1}$$

Zauważmy, że powyższy warunek jest równoważny następującemu:

$$\text{istnieje liczba rzeczywista } t: \ (at, bt) = (x,y) \tag{2}$$

Oczywiście wystarczy uzasadnić implikację tylko w jedną stronę. Powiedzmy, że zachodzi warunek (2) i t jest liczbą, która go spełnia. Przypomnijmy, że promień wystrzelony w punkt (a,b,c), gdyby nie krawędzie, dotarłby po T krokach do środka pułapki, czyli

$$(Ta, Tb, Tc) = (0,0,0).$$

Stąd widzimy, że dla dowolnej liczby całkowitej i:

$$(at - i \cdot aT, bt - i \cdot bT) \ = \ (x,y).$$

Przyjmując $i = \lfloor t/T \rfloor$ otrzymujemy warunek (1). Warunek (2) możemy przekształcić dalej do równoważnej postaci:

$$\text{istnieją liczby naturalne } k,l \ : \ at = (2k+1)x \text{ oraz } bt = (2l+1)y. \tag{3}$$

Możemy teraz przystąpić do dowodu równoważności z lematu. Załóżmy najpierw, że promień po strzale w punkt (a,b,c) trafi w krawędź. Wymnażając równości (3), otrzymujemy:

$$at \cdot (2l+1)y \quad = \quad (2k+1)x \cdot bt,$$

więc dzieląc równość stronami przez t i nakładając funkcję ord dostajemy:

$$ord(ay(2l+1)) \quad = \quad ord(bx(2k+1))$$

i dalej

$$ord(a) + ord(y) \quad = \quad ord(b) + ord(x)$$

Aby udowodnić implikację w drugą stronę przyjmijmy, że $ord(a) - ord(b) = ord(x) - ord(y)$ i wybierzmy

$$t = 2^{(ord(x)-ord(a))} \cdot NWW\left(\frac{x}{2^{ord(x)}}, \frac{y}{2^{ord(y)}}\right).$$

Wówczas:

$$\frac{at}{x} = \frac{a}{2^{ord(a)}} \frac{NWW\left(\frac{x}{2^{ord(x)}}, \frac{y}{2^{ord(y)}}\right)2^{ord(x)}}{x},$$

$$\frac{bt}{y} = \frac{b}{2^{ord(b)}} \frac{NWW\left(\frac{x}{2^{ord(x)}}, \frac{y}{2^{ord(y)}}\right)2^{ord(y)}}{y}.$$

Pokażemy, że tak określone at/x jest nieparzystą liczbą całkowitą. Liczba $\frac{a}{2^{ord(a)}}$ jest oczywiście całkowita i nieparzysta. Spójrzmy teraz na drugi ułamek. Jest on liczbą całkowitą, ponieważ $x = \frac{x}{2^{ord(x)}} \cdot 2^{ord(x)}$ oraz $\frac{x}{2^{ord(x)}} \mid NWW\left(\frac{x}{2^{ord(x)}}, \frac{y}{2^{ord(y)}}\right)$. Pozostaje wykazać, że jest to liczba nieparzysta. Ale to jest oczywiste, ponieważ $2^{ord(x)} \mid x$, a $NWW\left(\frac{x}{2^{ord(x)}}, \frac{y}{2^{ord(y)}}\right)$ jest liczbą nieparzystą.

Analogiczny dowód pozwala wykazać, że także bt/y jest nieparzystą liczbą całkowitą. W ten sposób pokazaliśmy, że punkty (at, bt) i (x,y) są równoważne, a więc promień po t krokach uderzy w krawędź. Równoważność warunków (1) oraz (2) pozwala nam przy tym stwierdzić, że uderzy w krawędź przed powrotem do środka pułapki. ■

Wiosek 3 Można sprawdzić w czasie $O(\log(xyz))$, czy promień po strzale w punkt (a,b,c) trafi w krawędź, zanim wróci do środka pułapki.

Po zastosowaniu opisanych ulepszeń mamy następującą procedurę wyliczania liczby kroków potrzebnych do powrotu do środka pułapki dla strzału w punkt pierwotny (a,b,c).

```
procedure sprawdźStrzał1(a,b,c)
  begin
    if ord(a)-ord(b)=ord(x)-ord(y) then return 0;
    if ord(a)-ord(c)=ord(x)-ord(z) then return 0;
    if ord(b)-ord(c)=ord(y)-ord(z) then return 0;
    return NWW(NWW(2x,a)/a, NWW(2y,b)/b, NWW(2z,c)/c);
  end
```

Procedura *sprawdźStrzał1* działa w czasie $O(\log(xyz))$. Zastosowanie jej w algorytmie `luss1` pozwala nam znaleźć najdłuższą drogę promienia w czasie $O(xyz\cdot\log(xyz))$. Niestety, taki algorytm (nazwijmy go `luss2`) wciąż działa zbyt wolno i przechodzi również tylko pierwszy test.

Odcinanie

Ostatnie usprawnienie algorytmu będzie oparte na dwóch spostrzeżeniach, które wskażą w jakiej kolejności warto przeglądać punkty pierwotne i kiedy można zakończyć poszukiwania.

Rozważmy strzał w punkt (a,b,c) w pułapce o rozmiarach $2x\times 2y\times 2z$. Poprzednio stosowaliśmy ograniczenie $2x\cdot 2y\cdot 2z$ na liczbę kroków T, po których promień wraca do środka pułapki. Istnieje jednak lepsze oszacowanie.

Wiosek 4

$$T\leqslant 2NWW(x,y,z)$$

Dowód Oczywiście $\frac{NWW(2x,a)}{a}\mid 2x$, więc

$$T=NWW\left(\frac{NWW(2x,a)}{a},\frac{NWW(2y,b)}{b},\frac{NWW(2z,c)}{c}\right)\leqslant NWW(2x,2y,2z)=2NWW(x,y,z)$$

■

Takie oszacowanie pozwala nam sformułować warunek, który pozwala wyeliminować niektóre punkty pierwotne.

Wiosek 5 Jeśli po strzale w punkt (a,b,c) promień wraca do środka pułapki w $T=2NWW(x,y,z)$ krokach, to lepszy wynik (dłuższą drogę promienia) możemy uzyskać tylko dla punktów (a',b',c'), takich że $a'+b'+c'>a+b+c$.

Powyższe dwa wnioski sugerują, że punkty pierwotne należy analizować począwszy od największych wartości sumy ich współrzędnych — tak też robimy w poniższym algorytmie.

```
procedure Lus;
  begin
    D=0;
    Tmax=2NWW(x,y,z);
    for a+b+c=x+y+z to 1 do
      if NWD(a,b,c)=1 then
         t=sprawdźStrzał1(a,b,c);
         if t=Tmax then return (a,b,c);
         if t(a+b+c) > D then
            D = t(a+b+c);
            (aOpt,bOpt,cOpt)=(a,b,c);
    if D > 0 then return (aOpt,bOpt,cOpt)
    else return NULL;
  end
```

Tak ulepszony algorytm przechodzi już wszystkie testy dla zadania. Pozostaje jednak bardzo istotna kwestia. Czy szybkie odnajdowanie właściwego punktu w narożniku pułapki (tam są punkty, dla których suma współrzędnych jest duża) jest dziełem przypadku, wynika z niedoskonałości testów, czy też tak po prostu musi być.

Tak musi być

Czytelnik usatysfakcjonowany posiadaniem szybkiego algorytmu może w tym miejscu zakończyć lekturę. Jeśli jednak interesuje go, dlaczego algorytm działa szybko, to trzeba jeszcze przebrnąć przez poniższy dowód na istnienie szczęścia.

Udowodnimy, że zawsze istnieje strzał, dla którego $T = T_{max}$, w odległości nie większej niż 141 (w metryce miejskiej) od narożnika pułapki. Jest to szacowanie dość grube: w praktyce sprawdzanie kończy się znacznie szybciej — przetestowanie wszystkich przypadków pokazało, że nie istnieje pułapka, w której trzeba strzelić w punkt odległy od narożnika (x,y,z) o więcej niż 7 (nadal w metryce miejskiej). Uzasadnienie rozpoczniemy od wykazania pewnych własności grup liczb.

Definicja 1 Dla liczb całkowitych n i m powiemy, że są one *względnie pierwsze co do 2*, jeśli $NWD(n,m)$ jest potęgą dwójki.

Lemat 4 *Niech* $48 \leqslant x \leqslant 1000$*. W każdym z wypisanych niżej czterech ciągów znajdują się dwie kolejne liczby względnie pierwsze co do 2 z liczbą x.*

(a) $x-1-4i$ *dla* $i = 0,1,\dots,10$*;*

(b) $x-2-4i$ *dla* $i = 0,1,\dots,9$*;*

(c) $x-3-4i$ *dla* $i = 0,1,\dots,11$*;*

(d) $x-4$*,* $x-8$.

Dowód Dla pierwszych trzech przypadków, pokażemy lemat przez doprowadzenie do sprzeczności. Założymy, że dla każdej pary liczb $x-c-4i$, $x-c-4(i+1)$ z ciągu (a), (b) lub (c) istnieje nieparzysta liczba $d>1$, taka że

$$d \mid NWD(x, x-c-4i) \quad \text{lub} \quad d \mid NWD(x, x-c-4(i+1). \tag{4}$$

W dowodzie będziemy korzystać z następującego, oczywistego spostrzeżenia:

$$\text{jeśli } d = NWD(x, x-a), \text{ to } d \mid a, \tag{5}$$

więc dla $d>1$ i liczby pierwszej a mamy wtedy $a \mid x$.

Pokażmy teraz sprzeczność pomiędzy założeniami lematu i założeniem (4) dla ciągów (a), (b) i (c).

(a) Dla pierwszej pary ciągu: $x-1$ i $x-5$, mamy $NWD(x,x-1)=1$, więc z założenia (4) wynika, że $NWD(x,x-5)>1$, a to z własności (5) oznacza iż $5 \mid x$. Następnie rozważmy parę: $x-13, x-17$. Ponownie z założenia (4) i własności (5) otrzymujemy, że $13 \mid x$ lub $17 \mid x$. Po rozważeniu pary: $x-37$, $x-41$, mamy $37 \mid x$ lub $41 \mid x$. Podsumowując, widzimy, że $x \geqslant 5 \cdot 13 \cdot 37 > 1000$, co jest sprzeczne z założeniami lematu.

(b) W tym przypadku rozważamy kolejno pary elementów ciągu: $(x-2, x-6)$, $(x-10, x-14)$, $(x-22, x-26)$ oraz $(x-34, x-38)$. Otrzymujemy, że $3 \mid x$, następnie, że $5 \mid x$ lub $7 \mid x$, potem, że $11 \mid x$ lub $13 \mid x$ i na koniec, że $17 \mid x$ lub $19 \mid x$. Podsumowując widzimy, że $x \geqslant 3 \cdot 5 \cdot 11 \cdot 17 > 1000$, więc doszliśmy do sprzeczności.

(c) Z tego ciągu rozważamy pary:$(x-7, x-11)$, $(x-19, x-23)$, $(x-43, x-47)$. Pozwala to nam zauważyć, że $7 \mid x$ lub $11 \mid x$, następnie $19 \mid x$ lub $23 \mid x$ i na koniec $43 \mid x$ lub $47 \mid x$. Stąd widzimy, że $x \geqslant 7 \cdot 19 \cdot 43 > 1000$, więc także doszliśmy do sprzeczności.

Zauważmy jeszcze, że liczby $x-4$ i $x-8$ są względnie pierwsze z co do 2 z liczbą x wprost z własności (5). To kończy dowód lematu. ■

Wiosek 6 Jeśli $x>4$ oraz $c \in \{0,1,2,3\}$, to w zbiorze $\{-x, -x+1, \ldots, -1, 1, 2, \ldots, x\}$ istnieją dwie różne liczby a i b spełniające warunki:

- $a = c+4i$ oraz $b = c+4j$ dla pewnych całkowitych i, j;
- a i b są względnie pierwsze co do 2 z liczbą x;
- $a = b + 2^k$, dla pewnego $k \geqslant 2$;
- $|a-x| \leqslant 47$ oraz $|b-x| \leqslant 47$.

Dowód Dla $4 < x < 48$ dowód został przeprowadzony przez sprawdzenie wszystkich przypadków (za pomocą odpowiedniego programu). Dla $x \geqslant 48$ możemy zastosować wprost lemat 4. ■

Teraz pokażemy, że nawet w dość silnie ograniczonym zbiorze punktów znajdziemy zawsze punkt pierwotny. W poniższym lemacie będziemy dla punktu $p=(a,b,c)$ używali oznaczenia $NWD(p) = NWD(a,b,c)$.

Lemat 5 *Niech $a,b,c \leqslant 1000$ i niech przynajmniej jedna z tych liczb będzie nieparzysta. Rozważmy zbiory $A = \{a, a-d_a\}$, $B = \{b, b-d_b\}$, $C = \{c, c-d_c\}$, gdzie liczby d_a, d_b, d_c są potęgami dwójki większymi niż 1, takimi że $A \subset (-x,x]$, $B \subset (-y,y]$ oraz $C \subset (-z,z]$. Zdefiniujmy zbiór ośmiu punktów $S = A \times B \times C$. Wśród nich istnieje przynajmniej jeden punkt pierwotny.*

Dowód Załóżmy, że lemat nie jest prawdziwy. Pokażemy, że doprowadza to do sprzeczności.

Bez utraty ogólności możemy przyjąć, że to a jest liczbą nieparzystą. Wówczas także $a - d_a$ jest nieparzysta. Stąd widzimy, że każdy punkt $s \in S$ ma jedną współrzędną nieparzystą i stąd liczba $NWD(s)$ także jest nieparzysta.

Teraz rozważmy dwa różne punkty $s' = (a', b', c')$ i $s'' = (a'', b'', c'')$ ze zbioru S. Pokażemy, że liczby $NWD(s')$ i $NWD(s'')$ są względnie pierwsze. Gdyby istniała liczba $d > 1$, taka że $d \mid NWD(s')$ i $d \mid NWD(s'')$, to mielibyśmy także: $d \mid a' - a''$, $d \mid b' - b''$, $d \mid c' - c''$. Przynajmniej jedna z tych różnic jest niezerowa i jest potęgą dwójki — nie jest możliwe, by dzieliła ją nieparzysta liczba $d > 1$. Stąd mamy, że dla różnych punktów $s', s'' \in S$ zachodzi $NWD(NWD(s'), NWD(s'')) = 1$.

Oznaczmy $D = \{NWD(s) : s \in S\}$. Zbiór D składa sie z ośmiu liczb nieparzystych, które są parami względnie pierwsze. Jeśli nie ma wśród nich jedynek (czyli w S nie ma punktów pierwotnych), to

$$\prod_{d \in D} d \geqslant 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 > 10^6.$$

Zobaczmy teraz, co wynika z powyższego dla liczb a i $a - d_a$. Liczba a występuje jako pierwsza współrzędna czterech punktów ze zbioru S, stąd dzieli ją połowa liczb ze zbioru D. Podobnie jest dla liczby $a - d_a$. Ponieważ liczby a i $a - d_a$ są względnie pierwsze (przyjęliśmy, że a jest nieparzyste, a d_a jest potęgą dwójki), więc element zbioru D nie może być jednocześnie dzielnikiem liczby a i $a - d_a$. Stąd

$$\prod_{d \in D} d \mid a(a - d_a),$$

więc $a(a - d_a) > 10^6$, co jest sprzeczne za założeniami lematu. ∎

Teraz pozostaje nam wykorzystać udowodnione własności, by wykazać, że blisko narożnika pułapki istnieje punkt pierwotny, w który możemy strzelić uzyskując promień powracający do środka pułapki po T_{max} krokach. Po znalezieniu takiego punktu $s = (a, b, c)$ nie będziemy musieli już rozważać punktów $s' = (a', b', c')$, gdzie $a + b + c \geqslant a' + b' + c'$, bo droga ich promienia na pewno nie będzie dłuższa od $T_{max}(a + b + c)$. Ponieważ $(a + b + c)$ jest wartością stosunkowo dużą, pozwoli to nam istotnie zawęzić obszar poszukiwań.

Twierdzenie 6 *Istnieje punkt pierwotny (a,b,c), taki że $(a,b,c) \in [x-47,x] \times [y-47,y] \times [z-47,z]$ (czyli $a+b+c-(x+y+z) \leqslant 141$) i strzał w ten punkt nie trafia w krawędź pułapki.*

Dowód Dla ustalenia uwagi przyjmijmy, że $ord(x) \geqslant ord(y) \geqslant ord(z)$. Korzystając trzykrotnie z wniosku 6 wybierzmy liczby $a, a - d_a \in [-x,x]$, $b, b - d_b \in [-y,y]$ oraz $c, c - d_c \in [-z,z]$ takie że:

(i) d_a, d_b i d_c są potęgami dwójki nie mniejszymi niż 4;

(ii) $ord(a) = 0$, $ord(b) = 1$ oraz $ord(c) \geqslant 2$;

(iii) liczby a i $a - d_a$ są względnie pierwsze co do 2 z liczbą x, liczby b i $b - d_b$ są względnie pierwsze co do 2 z liczbą y, a liczby c i $c - d_c$ są względnie pierwsze co do 2 z liczbą z;

(iv) $x - a \leqslant 47$ i $x - (a - d_a) \leqslant 47$ oraz $y - b \leqslant 47$ i $y - (b - d_b) \leqslant 47$ oraz $z - c \leqslant 47$ i $z - (c - d_c) \leqslant 47$.

Definiując zbiory $A = \{a, a - d_a\}$, $B = \{b, b - d_b\}$ oraz $C = \{c, c - d_c\}$ z lematu 5 wiemy, że w zbiorze $A \times B \times C$ istnieje punkt pierwotny — oznaczmy go $p = (a', b', c')$. Wówczas liczba $a - d_a$ jest nieparzysta, zachodzi też równość $b - d_b = 2 \; mod \; 4$, a liczba $c - d_c$ jest podzielna przez 4. Dalej mamy: $ord(a) = ord(a - d_a) = 0$, $ord(b) = ord(b - d_b) = 1$, $ord(c) >= 2$, $ord(c - d_c) >= 2$, wobec tego $ord(a') < ord(b') < ord(c')$, czyli

$$\begin{aligned} ord(a') - ord(b') &< 0 \leqslant ord(x) - ord(y), \\ ord(a') - ord(c') &< 0 \leqslant ord(x) - ord(z), \\ ord(b') - ord(c') &< 0 \leqslant ord(y) - ord(z). \end{aligned}$$

Z lematu 3 widzimy więc, że po strzale w punkt (a', b', c') promień nie trafi w żadną krawędź.

Pozostaje wykazać, że promień przebędzie maksymalną drogę przed powrotem do środka pułapki. Zauważmy, że

$$\begin{aligned} T &= NWW\Big(\frac{NWW(2x, a')}{a'}, \frac{NWW(2y, b')}{b'}, \frac{NWW(2z, c')}{c'}\Big) \\ &\geqslant NWW\big(2x, \frac{y}{2^{ord(y)}}, \frac{z}{2^{ord(z)}}\big) = 2NWW(x, y, z) = T_{max}, \end{aligned}$$

ponieważ a' jest liczbą nieparzystą i $ord(x)$ jest największe spośród $ord(x)$, $ord(y)$ i $ord(z)$.

To kończy dowód twierdzenia.

■

Implementacja rozwiązania zadania

Rozwiązania poprawne

Zaimplementowano cztery poprawne rozwiązania zadania.

`lus.c`: rozwiązanie wzorcowe działające zgodnie z algorytmem Lus, czyli polegające na przeglądaniu punktów pierwotnych (każdy punkt sprawdzamy w czasie $O(log(xyz))$) począwszy od narożnika pułapki; algorytm kończy działanie, gdy znajdzie punkt, dla którego powrót do środka pułapki następuje po $T = T_{max}$ krokach. Z twierdzenia 6 wiemy, że algorytm zawsze kończy działanie po przejrzeniu stałej liczby punktów. Tak naprawdę nie ma przykładu, dla którego odległość rozwiązania od rogu jest większa niż 7 (choć w dowodzie twierdzenia pokazaliśmy trochę gorsze oszacowanie), więc program dość szybko znajduje rozwiązania.

luss1.c: w programie przeglądamy wszystkie możliwe punkty dla każdego sprawdzając, czy w kolejnych krokach nie trafiliśmy w krawędź. Algorytm działa w czasie $O((xyz)^2)$.

luss2.c: przeglądamy wszystkie możliwe punkty, dla każdego licząc jak program wzorcowy poprawność i drogę strzału — czas działania wynosi $O(xyz \log(xyz))$.

luss3.c: dla każdego punktu sprawdzamy czas powrotu do środka w czasie $O(xyz)$ jak w programie `luss1.c`. Kończymy działanie po znalezieniu punktu, dla którego czas powrotu jest maksymalny, czyli obliczenia wymagają czasu $O(xyz \cdot S)$, gdzie S jest liczbą sprawdzonych punktów.

Jak wcześniej wspomnieliśmy, oprócz rozwiązania wzorcowego, pozostałe programy przechodziły tylko jeden test otrzymując $1/10$ punktów za rozwiązanie.

Błędne rozwiązania

Błędy pojawiające się w rozwiązaniach tego zadania, to przede wszystkim:

- stosowanie prostych heurystyk wyboru najlepszego punktu, na przykład $(x-1, y-1, z-1)$ dla różnych rozmiarów pułapki x, y, z oraz punktu $(x-1, y, z)$, gdy wymiary x i y są równe; heurystyki te nie dają poprawnych rozwiązań;
- niepoprawne rozpoznawanie, czy promień uderzy w krawędź;
- testowanie tylko kilku (ale nie wystarczająco wielu) punktów w narożniku pułapki (w odległości 6).

Uwagi

Zadanie to okazało się najtrudniejszym zadaniem finału — rozwiązał je poprawnie tylko jeden zawodnik przedstawiając algorytm działający podobnie jak program wzorcowy `lus.c`. Testy, choć jak widać z analizy zadania dopuszczały możliwość uzyskania punktów także za rozwiązania nie do końca optymalne, nie zostały pomyślnie rozwiązane przez inne programy zawodników. Także, co w tego typu zadaniach należy uznać za sukces opracowujących testy, nie udało się zdobyć żadnych punktów za pomocą żadnej heurystyki nie dającej poprawnego rozwiązania.

Algorytm wzorcowy pokonuje testy w niezauważalnym czasie. W trakcie opracowywania zadania zaprogramowano także rozwiązania dające poprawną odpowiedź tylko w niektórych przypadkach (sprawdzanie zbyt małej liczby punktów w sąsiedztwie narożnika) bądź działające wystarczająco szybko tylko w niektórych przypadkach (dla pułapek rozmiaru 100 wystarczy szukać optymalnego punktu wśród odległych od narożnika najwyżej o 6).

Testy

Rozwiązania zawodników były sprawdzane na dziesięciu testach. Dodatkowo przygotowano pięć testów dostarczonych zawodnikom w czasie zawodów.

Nazwa	**Opis**
lus1.in	10 pułapek z zakresu do 100, kilka sześcianów i kilka takich, gdzie odległość rozwiązania od rogu pułapki wynosi 6
lus2.in	wszystkie pułapki o wymiarach z zakresu $[5,21]$
lus3.in	wszystkie pułapki o wymiarach z zakresu $[991,1000]$
lus4.in	seria tysiąca dużych losowych pułapek, w tym pułapki, gdzie odległość rozwiązania od rogu pułapki wynosi 7
lus5.in	396 pułapek, każda ma odległość rozwiązania od rogu pułapki wynoszącą 7
lus6.in	wszystkie sześciany mieszczące się w zakresach plus cztery pułapki o odległości rozwiązania od rogu pułapki wynoszącej 7
lus7.in	seria 995 losowych pułapek o dwóch współrzędnych równych plus cztery pułapki o odległości rozwiązania od rogu pułapki wynoszącej 7 plus jakiś sześcian
lus8.in	wszystkie pułapki, gdzie $x \in \{81,82,\ldots,90\}$, $y \in \{71,72,\ldots,80\}$, $z \in \{41,42,\ldots,50\}$
lus9.in	seria 30 losowych pułapek o wielkości do 100, w tym jedna, dla której rozwiązanie jest w odległości 6 od rogu pułapki
lus10.in	seria 30 pułapek o wielkości do 100, gdzie odległość rozwiązania od rogu pułapki wynosi 6

XVI Międzynarodowa Olimpiada Informatyczna

Ateny, Grecja 2004

Przemysława Kanarek, Tomasz Waleń
Tłumaczenie

IOI, dzień I, 13–09–2004

Artemis

Zadanie

Zeus podarował bogini łowów Artemidzie prostokątny obszar, aby zasadziła na nim las. Lewy bok obszaru leży na dodatniej części osi Y, dolny bok na dodatniej części osi X, a punkt $(0,0)$ jest lewym–dolnym rogiem obszaru. Zeus nakazał Artemidzie sadzić drzewa jedynie w tych punktach obszaru, które mają współrzędne całkowite. Artemida chciała, by las wyglądał naturalnie, więc posadziła drzewa w ten sposób, że żadna linia łącząca dwa różne drzewa nie jest równoległa do osi X lub osi Y.

Pewnego razu Zeus zażądał od Artemidy, by wycięła pewną liczbę drzew zgodnie z następującymi zasadami:

1. *Zeus zażądał wycięcia co najmniej T drzew.*
2. *W miejscu po wyciętych drzewach powstanie stadion piłkarski, stąd Artemida musi wybrać pewien prostokątny obszar i wyciąć wszystkie znajdujące się w nim drzewa (i żadnego innego).*
3. *Boki wybranego prostokąta muszą być równoległe do osi X i osi Y.*
4. *W dwóch przeciwległych rogach wybranego obszaru muszą znajdować się drzewa (one także zostaną wycięte dla Zeusa).*

Artemida kocha drzewa, więc chce spełnić powyższe warunki wycinając jak najmniej drzew. Napisz program, który na podstawie informacji o pozycjach drzew w lesie i żądanej przez Zeusa liczbie drzew T, wybierze prostokątny obszar spełniający powyższe warunki.

Wejście

Dane wejściowe zapisane są w pliku `artemis.in`. *Pierwszy wiersz zawiera jedną liczbę całkowitą N — liczbę drzew w lesie. Drugi wiersz zawiera jedną liczbę całkowitą T — minimalną liczbę drzew, które Artemida musi wyciąć. W kolejnych N wierszach zapisane są pozycje N drzew. Każdy z tych wierszy zawiera dwie liczby całkowite — współrzędne X i Y danego drzewa.*

Wyjście

Wynik należy zapisać w pliku `artemis.out`. *Plik powinien składać się z jednego wiersza zawierającego dwie liczby całkowite I oraz J oddzielone pojedynczą spacją. Oznaczają one, że Artemida powinna wybrać prostokąt wyznaczony przez drzewo I-te oraz J-te (tzn. o współrzędnych zapisanych w $I+2$ oraz $J+2$ wierszu wejścia), czyli taki w którego* *przeciwległych rogach rosną te dwa drzewa. Drzewa mogą być podane w dowolnej kolejności. Jeżeli istnieje wiele rozwiązań spełniających warunki zadania, wypisz dowolne z nich. Wiadomo, że dla każdego zestawu danych wejściowych istnieje przynajmniej jedno rozwiązanie.*

Przykład

WEJŚCIE
```
3
2
1 1
2 3
5 6
```

WYJŚCIE
```
1 2
```

Ograniczenia

Dla wszystkich zestawów danych wejściowych $1 < N \leqslant 20\,000$, $0 \leqslant X, Y \leqslant 64000$ *i* $1 < T \leqslant N$.

Dodatkowo w 50% zestawów $1 < N < 5\,000$.

Przemysława Kanarek, Tomasz Waleń
Tłumaczenie

IOI, dzień I, 13–09–2004

Hermes

Zadanie

Nowoczesna stolica greckich bogów ma przejrzysty układ ulic — dla każdej liczby całkowitej Z istnieje jedna pozioma ulica (o równaniu $y = Z$) i jedna pionowa ulica (o równaniu $x = Z$). Tak więc wszystkie ulice są równoległe do osi X lub osi Y, tworzą pełną kratę i przecinają się jedynie w punktach o współrzędnych całkowitych. Podczas upalnych dni bogowie spędzają czas w kawiarniach znajdujących się na skrzyżowaniach ulic. Boski posłaniec Hermes musi dostarczyć każdemu z bogów przeznaczoną dla niego świetlną wiadomość (nie jest istotne, czy taką wiadomość zobaczą również inni bogowie). Może przy tym biegać tylko po ulicach.

Hermes otrzymał listę adresów kawiarni (czyli ich współrzędnych) i dokładnie w takiej kolejności musi dostarczyć bogom wiadomości. Rozpoczyna pracę w punkcie $(0,0)$. By dostarczyć wiadomość do kawiarni o współrzędnych (X_i,Y_i), wystarczy, by Hermes znalazł się na pewnym skrzyżowaniu o współrzędnej x równej X_i lub współrzędnej y równej Y_i (każdy z bogów ma sokoli wzrok i może dojrzeć przeznaczoną dla niego wiadomość z dowolnej odległości). Hermes kończy swoją wędrówkę w momencie, gdy dostarczy ostatnią wiadomość.

Napisz program, który na podstawie listy punktów (współrzędnych kawiarni) znajdzie długość najkrótszej trasy, którą musi przebyć Hermes, by dostarczyć wszystkie wiadomości.

Wejście

Dane wejściowe są zapisane w pliku `hermes.in`. *W pierwszym wierszu znajduje się jedna liczbą całkowita N — liczba wiadomości, które trzeba dostarczyć. W kolejnych N wierszach są zapisane współrzędne N skrzyżowań, na które należy dostarczyć wiadomości. Skrzyżowania podane są w takiej kolejności, w jakiej wiadomości muszą zostać dostarczone. Każde skrzyżowanie jest opisane przez dwie liczby całkowite — pierwsza z nich to współrzędna x, a druga to współrzędna y.*

Wyjście

Wynik należy zapisać w pliku `hermes.out`. *Plik powinien składać się z jednego wiersza zawierającego jedną liczbę całkowitą — długość najkrótszej trasy pozwalającej Hermesowi dostarczyć wszystkie wiadomości.*

Przykład

WEJŚCIE	**WYJŚCIE**
5 8 3 7 -7 8 1 -2 1 6 -5	11

Ograniczenia

Dla wszystkich zestawów danych wejściowych $1 \leqslant N \leqslant 20\,000$, $-1000 \leqslant X_i, Y_i \leqslant 1\,000$.
Dodatkowo w 50% zestawów $1 \leqslant N \leqslant 80$.

Przemysława Kanarek, Tomasz Waleń
Tłumaczenie

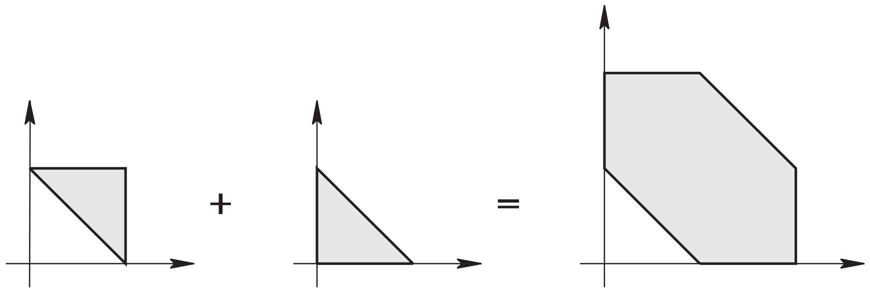
IOI, dzień I, 13–09–2004

Polygon

Zadanie

Wielokąt wypukły ma tę własność, że dla dowolnych dwóch punktów wielokąta X i Y, odcinek je łączący jest zawarty w wielokącie. Wszystkie wielokąty rozważane w tym zadaniu są wypukłe, mają przynajmniej dwa wierzchołki i brzeg należy do wielokąta. Ponadto wszystkie wierzchołki wielokąta są różne i mają całkowite współrzędne. Żadne trzy wierzchołki wielokąta nie są współliniowe. Pisząc poniżej „wielokąt" będziemy mieli na myśli właśnie takie wielokąty.

Dla dwóch wielokątów A i B, suma Minkowskiego A i B składa się z wszystkich punktów w postaci (x_1+x_2, y_1+y_2), gdzie (x_1, y_1) należy do A oraz (x_2, y_2) należy do B. Wiadomo, że suma Minkowskiego wielokątów jest również wielokątem. Na rysunku poniżej przedstawiony jest przykład: dwa trójkąty i ich suma Minkowskiego.



Interesuje nas operacja odwrotna do sumy Minkowskiego. Dla zadanego wielokąta P poszukujemy dwóch wielokątów A i B, takich że:

- *P jest sumą Minkowskiego A i B,*
- *A ma od 2 do 4 różnych wierzchołków, czyli jest odcinkiem (2 wierzchołki), trójkątem (3 wierzchołki) lub czworokątem (4 wierzchołki),*
- *A powinien mieć największą możliwą liczbę wierzchołków, czyli:*
  - *jeśli to możliwe, A powinien być czworokątem,*
  - *wpp. jeśli to możliwe, A powinien być trójkątem,*
  - *wpp. A powinien być odcinkiem.*

Zauważmy, że ani A, ani B nie może być równe P. Oznaczałoby to, że drugi z wielokątów musi być punktem, a takiego nie uważamy w tym zadaniu za wielokąt.

Otrzymasz zbiór plików z danymi wejściowymi. Każdy z nich zawiera opis jednego wielokąta P. Dla każdego pliku wejściowego powinieneś obliczyć wielokąty A i B spełniające podane wyżej warunki i utworzyć plik wyjściowy je zawierający. Wiadomo, że dla zadanych wielokątów P zawsze istnieje rozwiązanie. Jeśli jest wiele poprawnych rozwiązań, powinieneś podać dowolne z nich. Zwróć uwagę, że jako rozwiązanie należy zgłosić pliki z odpowiedziami, a nie program.

Wejście

Otrzymasz 10 zestawów danych wejściowych w plikach o nazwach od `polygon1.in` *do* `polygon10.in`*. Każdy z nich zawiera dane w następującej postaci. Pierwszy wiersz zawiera jedną liczbę całkowitą N — liczbę wierzchołków wielokąta P. W kolejnych N wierszach są opisane wierzchołki P w kolejności przeciwnej do ruchu wskazówek zegara, każdy w jednym wierszu. W wierszu $(I+1)$-szym (dla $I = 1, 2, \ldots, N$) zapisane są współrzędne I-tego wierzchołka — dwie nieujemne liczby całkowite X_I i Y_I oddzielone pojedynczą spacją.*

Wyjście

Jako rozwiązanie powinieneś zgłosić 10 plików wynikowych odpowiadających plikom wejściowym. Każdy z nich powinien zawierać opisy wielokątów A i B.
Pierwszy wiersz pliku wynikowego powinien zawierać napis:
`#FILE polygon` I
gdzie liczba całkowita I $(1 \leqslant I \leqslant 10)$ oznacza numer odpowiedniego pliku wejściowego.

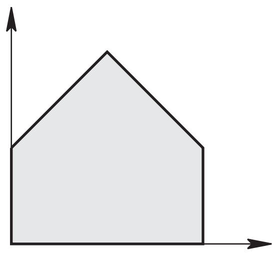
Format pliku wyjściowego jest podobny do formatu danych wejściowych. Drugi wiersz powinien zawierać liczbę całkowitą N_A — liczbę wierzchołków wielokąta A $(2 \leqslant N_A \leqslant 4)$. W kolejnych N_A wierszach powinny znaleźć się opisy wierzchołków A w kolejności przeciwnej do kierunku ruchu wskazówek zegara, po jednym wierzchołku w wierszu. Wiersz $I+2$ (dla $I = 1, 2, \ldots, N_A$) powinien zawierać dwie liczby całkowite X i Y oddzielone pojedynczym odstępem — współrzędne I-tego wierzchołka wielokąta A.

Wiersz $N_A + 3$ powinien zawierać liczbę całkowitą N_B — liczbę wierzchołków wielokąta B $(2 \leqslant N_B)$. W kolejnych N_B wierszach powinny znaleźć się opisy wierzchołków B w kolejności przeciwnej do kierunku ruchu wskazówek zegara, po jednym wierzchołku w wierszu. Wiersz $N_A + J + 3$ (dla $J = 1, 2, \ldots, N_B$) powinien zawierać dwie liczby całkowite X i Y oddzielone pojedynczym odstępem — współrzędne I-tego wierzchołka wielokąta B.

Przykład

WEJŚCIE

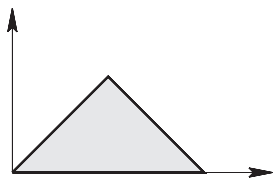
```
5
0 1
0 0
2 0
2 1
1 2
```

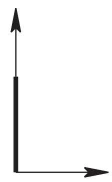


WYJŚCIE

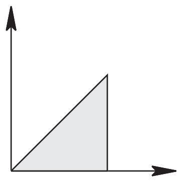
Dla powyższych danych wejściowych każde z podanych niżej rozwiązań jest poprawne (zobacz również rysunki). W obu rozwiązaniach A jest trójkątem i nie istnieje czworokąt spełniający warunki zadania.

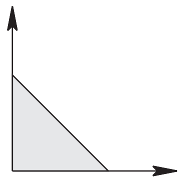
```
#FILE polygon 0
3
0 0
2 0
1 1
2
0 1
0 0
```





```
#FILE polygon 0
3
0 0
1 0
1 1
3
0 1
0 0
1 0
```





Przemysława Kanarek, Tomasz Waleń
Tłumaczenie

IOI, dzień II, 15–09–2004

Empodia

Zadanie

Starożytny matematyk i filozof Pitagoras wierzył, że rzeczywistość ma naturę matematyczną. Współcześni biolodzy badają własności biosekwencji. Biosekwencją nazywamy ciąg M liczb całkowitych, który:

- *zawiera każdą z liczb $0, 1, \ldots, M-1$,*
- *rozpoczyna się liczbą 0, a kończy liczbą $M-1$ oraz*
- *na żadnych dwóch kolejnych pozycjach nie zawiera elementów w postaci $E, E+1$.*

Spójny podciąg złożony z kolejnych elementów biosekwencji nazywamy blokiem.

Blok biosekwencji nazywamy przedziałem otoczonym, jeżeli zawiera wszystkie liczby całkowite o wartościach pomiędzy pierwszym elementem przedziału (musi to być najmniejszy element w bloku), a ostatnim (musi to być największy element w bloku). Pierwszy element i ostatni element przedziału otoczonego muszą być różne. Przedział otoczony nazywamy empodium, jeśli nie zawiera żadnego krótszego przedziału otoczonego.

Rozważmy przykładową biosekwencję $(0,3,5,4,6,2,1,7)$. Jako całość jest ona przedziałem otoczonym. Zawiera jednak inny przedział otoczony $(3,5,4,6)$, więc nie jest empodium. Natomiast przedział otoczony $(3,5,4,6)$ nie zawiera krótszego przedziału otoczonego, więc jest empodium. Ponadto, jest to jedyne empodium w tej biosekwencji.

Twoim zadaniem jest napisanie programu, który dla danej biosekwencji znajdzie wszystkie zawarte w niej empodia (empodia to liczba mnoga od empodium).

Wejście

Dane wejściowe zapisane są w pliku `empodia.in`. *Pierwszy wiersz zawiera jedną liczbę całkowitą M — liczbę elementów w biosekwencji. W kolejnych M wierszach zapisane są liczby całkowite tworzące biosekwencje. Liczby te występują w danych w takiej kolejności jak w biosekwencji i są zapisane po jednej w wierszu.*

Wyjście

Wynik należy zapisać w pliku o nazwie `empodia.out`. *W pierwszym wierszu powinna być zapisana jedna liczba całkowita H — liczba empodiów w wejściowej biosekwencji. W kolejnych H wierszach powinny znaleźć się opisy empodiów podane w rosnącej kolejności występowania ich pozycji początkowych w biosekwencji. Każdy z tych wierszy powinien zawierać dwie liczby* *całkowite A i B (w tej kolejności) oddzielone pojedynczą spacją, gdzie A oznacza pozycję w biosekwencji pierwszego elementu empodium, a B oznacza pozycję w biosekwencji ostatniego elementu empodium.*

Przykład

WEJŚCIE	**WYJŚCIE**
8	1
0	2 5
3	
5	
4	
6	
2	
1	
7	

Ograniczenia

W jednym zestawie danych wejściowych $1\,000\,000 \leqslant M \leqslant 1\,100\,000$. W pozostałych zestawach $1 \leqslant M \leqslant 60\,000$. Dodatkowo dla 50% zestawów $M \leqslant 2\,600$.

Przemysława Kanarek, Tomasz Waleń
Tłumaczenie

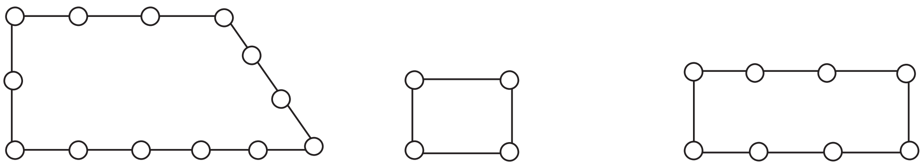
IOI, dzień II, 15–09–2004

Farmer

Zadanie

Farmer ma pola, z których każde jest obsadzone cyprysami. Ma także wąskie pasy ziemi — na każdym z nich rośnie rząd cyprysów. Zarówno na polach jak i pasach ziemi, pomiędzy każdymi dwoma cyprysami rośnie jedno drzewko oliwne. Wszystkie cyprysy, które ma farmer, rosną albo wokół pól, albo na pasach ziemi, a wszystkie drzewka oliwne znajdują się pomiędzy sąsiednimi cyprysami na polach lub pasach ziemi.

Pewnego dnia farmer ciężko zachorował i poczuł, że zbliża się jego koniec. Kilka dni przed śmiercią wezwał najstarszego syna i rzekł mu: „Daję ci Q cyprysów, które możesz dowolnie wybrać, oraz wszystkie drzewka oliwne, które rosną pomiędzy wybranymi przez ciebie cyprysami”. Z każdego pola i każdego pasa ziemi syn może wybrać dowolną kombinację cyprysów. Ponieważ najstarszy syn uwielbia oliwki, zależy mu na wybraniu Q cyprysów w ten sposób, by wraz z nimi odziedziczyć jak najwięcej drzewek oliwnych.



Na polu 1 rośnie 13 cyprysów   Na polu 2 rosną 4 cyprysy   Na polu 3 rośnie 8 cyprysów

Na pasie 1 rosną 4 cyprysy

Na pasie 2 rośnie 8 cyprysów

Na pasie 3 rośnie 6 cyprysów

Rys. 1: Przykładowe rozmieszczenie cyprysów (drzewka oliwne nie są zaznaczone na rysunku).

Dla przykładu przedstawionego na rysunku 1 załóżmy, że syn otrzymał $Q = 17$ cyprysów. Chcąc odziedziczyć jak najwięcej drzewek oliwnych, powinien wybrać cyprysy z pól 1 i 2. W ten sposób będzie miał 17 drzewek oliwnych.

Napisz program, który na podstawie informacji o polach i pasach ziemi oraz liczbie cyprysów darowanych synowi, określi największą możliwą liczbę drzewek oliwnych, które może odziedziczyć syn.

Wejście

Dane wejściowe zapisane są w pliku `farmer.in`. *Pierwszy wiersz zawiera liczbę całkowitą Q oznaczającą liczbę cyprysów, które ma wybrać syn, potem liczbę całkowitą M oznaczającą liczbę pól, na końcu liczbę całkowitą K oznaczającą liczbę pasów ziemi. Drugi wiersz zawiera M liczb całkowitych $N_1, N_2, \ldots N_M$ oznaczających liczby cyprysów na kolejnych polach. Trzeci wiersz zawiera K liczb całkowitych $R_1, R_2, \ldots R_K$ oznaczających liczby cyprysów na kolejnych pasach ziemi.*

Wyjście

Wynik należy zapisać w pliku o nazwie `farmer.out`. *Plik powinien składać się z jednego wiersza zawierającego jedną liczbę całkowitą oznaczającą największa możliwą liczbę drzewek oliwnych, które może odziedziczyć syn.*

Przykład

WEJŚCIE	WYJŚCIE

`17 3 3`
`13 4 8`
`4 8 6`
	`17`

Ograniczenia

Dla wszystkich zestawów danych wejściowych $0 \leqslant Q \leqslant 150\,000$, $0 \leqslant M \leqslant 2\,000$, $0 \leqslant K \leqslant 2\,000$, $3 \leqslant N_1 \leqslant 150, 3 \leqslant N_2 \leqslant 150, \ldots, 3 \leqslant N_M \leqslant 150$, oraz $2 \leqslant R_1 \leqslant 150, 2 \leqslant R_2 \leqslant 150, \ldots, 2 \leqslant R_K \leqslant 150$. Sumaryczna liczba cyprysów na wszystkich polach i pasach ziemi wynosi co najmniej Q. Dodatkowo dla 50% zestawów $Q \leqslant 1\,500$.

Przemysława Kanarek, Tomasz Waleń
Tłumaczenie

IOI, dzień II, 15–09–2004

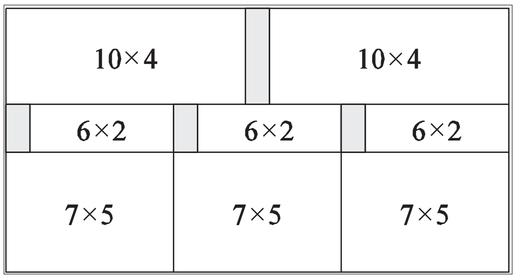
Phidias

Zadanie

Znany grecki rzeźbiarz Fidiasz przygotowuje się do budowy kolejnego marmurowego pomnika. W tym celu potrzebuje pewnej liczby prostokątnych, marmurowych płyt o rozmiarach $W_1 \times H_1, W_2 \times H_2, \ldots, W_N \times H_N$.

Ostatnio rzeźbiarz pozyskał duży, prostokątny marmurowy płat. Zamierza pociąć płat i otrzymać w ten sposób płyty pożądanych rozmiarów. Każdy kawałek marmuru (płat lub odcięte od niego płyty) można rozciąć tylko poziomym lub pionowym cięciem na dwie prostokątne płyty. Ich szerokości i długości zawsze muszą być liczbami całkowitymi. Kawałki marmuru można ciąć tylko w opisanych wyżej sposób i mniejszych kawałków nie da się złączyć w jeden większy. Ponieważ płat jest pokryty wzorkiem, płyty nie mogą być obracane: jeśli Fidiasz wyciął płytę o rozmiarze $A \times B$, *to nie może być ona użyta jako płyta o rozmiarze* $B \times A$, *chyba że* $A = B$. *Fidiasz może wyciąć dowolną liczbę (także zero) płyt każdego pożądanego rozmiaru. Po zakończenie cięć powiemy, że płyta jest zmarnowana, jeśli jej rozmiar nie pokrywa się z rozmiarem żadnej z płyt potrzebnych do budowy pomnika. Fidiasz chciałby wiedzieć, w jaki sposób pociąć otrzymany płat, żeby zmarnować jak najmniej marmuru.*

Jako przykład rozważmy poniższy rysunek. Dany płat ma szerokość równą 21 *i wysokość równą* 11. *Płyty potrzebne do budowy pomnika mają rozmiary* 10×4, 6×2, 7×5 *oraz* 15×10. *Minimalne pole zmarnowanego marmuru wynosi* 10, *a na rysunku pokazano, w jaki sposób pociąć dany płat, żeby zmarnować tylko marmur o polu równym* 10.



Twoje zadanie polega na napisaniu programu, który dla podanego rozmiaru płatu i rozmiarów pożądanych płyt, obliczy minimalne pole marmuru, który musi być zmarnowany.

Wejście

Dane wejściowe są zapisane w pliku phidias.in*. Pierwszy wiersz zawiera dwie liczby całkowite: najpierw* W *— szerokość, a następnie* H *— wysokość płata marmuru. Drugi wiersz zawiera jedną liczbę całkowitą* N *— liczbę różnych rozmiarów pożądanych płyt. W kolejnych* N *wierszach podano rozmiary płyt. Każdy z tych wierszy zawiera dwie liczby całkowite: najpierw szerokość* W_i*, a następnie wysokość* H_i *jednej płyty* $(1 \leqslant i \leqslant N)$*.*

Wyjście

Wynik należy zapisać w pliku o nazwie phidias.out*. Plik powinien zawierać tylko jeden wiersz, z jedną liczbą całkowitą równą minimalnemu polu marmuru, który musi zostać zmarnowany.*

Przykład

WEJŚCIE

```
21 11
4
10 4
6 2
7 5
15 10
```

WYJŚCIE

```
10
```

Ograniczenia

Dla wszystkich zestawów danych wejściowych $1 \leqslant W \leqslant 600$, $1 \leqslant H \leqslant 600$, $0 < N \leqslant 200$, $1 \leqslant W_i \leqslant W$ *oraz* $1 \leqslant H_i \leqslant H$*. Ponadto dla 50% zestawów* $W \leqslant 20$, $H \leqslant 20$ *i* $N \leqslant 5$*.*

XI Bałtycka Olimpiada Informatyczna,

Pasvalys, Lithuania 2005

Przemysława Kanarek, Marcin Michalski
Tłumaczenie

BOI, dzień próbny, 5–05–2005

Karty

Zadanie

Adam lubi łamigłówki liczbowe. Pewnego razu znalazł w szufladzie plik czystych kartek, na każdej z nich po każdej stronie napisał po jednej, wybranej losowo liczbie i wymyślił następującą łamigłówkę: jaka jest najmniejsza wartość pokazanego niżej wyrażenia, którą można uzyskać układając wszystkie kartki w dowolnej kolejności (a każdą z nich dowolną stroną w górę):

$$\square - \square + \square - \square + \square - \square + \ldots - \square$$

Już po chwili Adam znalazł rozwiązanie. Czy Ty także? Napisz program, który rozwiąże opisaną wyżej łamigłówkę.

Wejście

Plik wejściowy nazywa się `CARDS.IN`*. W pierwszym wierszu znajduje się liczba kartek N ($2 \leqslant N \leqslant 100\,000$, N jest liczbą całkowitą, parzystą). W każdym z kolejnych N wierszy znajdują się po dwie liczby całkowite a_i oraz b_i ($-2\,000 \leqslant a_i, b_i \leqslant 2\,000$; $i = 1 \ldots N$). Są to liczby zapisane na dwóch stronach i-tej kartki.*

Wyjście

Plik wynikowy nazywa się `CARDS.OUT`*. W pierwszym i jedynym wierszu powinien zawierać minimalną wartość, którą można uzyskać układając opisane wyżej wyrażenie z wszystkich kartek.*

Przykład

WEJŚCIE

```
6
-8 12
0 5
7 -3
10 -7
-2 7
1 4
```

WYJŚCIE

```
-34
```

WYJAŚNIENIE

Kartki należy ułożyć w wyrażeniu w następującej kolejności: 1, 2, 3, 5, 4, 6.
$(-8) - 5 + (-3) - 7 + (-7) - 4 = -34$

WEJŚCIE	**WYJŚCIE**	**WYJAŚNIENIE**

`10`
`70 70`
`62 73`
`81 65`
`59 77`
`99 40`
`35 88`
`80 57`
`76 67`
`85 57`
`53 96`
	`-155`	*Kartki należy ułożyć w wyrażeniu w następującej kolejności:* $2, 1, 4, 3, 5, 8, 6, 9, 7, 10$.
		$62-70+59-81+40-76+35-85+57-96=-155$

Przemysława Kanarek, Marcin Michalski
Tłumaczenie

BOI, dzień I, 6–05–2005

Baza Wojskowa

Zadanie

Dowódca ściśle tajnej bazy wojskowej szuka miejsca na jej nową lokalizację. Posiada on dokładną (wojskową) mapę topograficzną terenu, przedstawioną jako prostokątna siatka z zaznaczonymi wysokościami każdego punktu terenu. Współrzędne na mapie są wyznaczane przez wiersz i kolumnę w tablicy.

Miejsce na nową bazę musi być prostokątem zawartym całkowicie w zasięgu mapy oraz spełniać określone warunki bezpieczeństwa. Każdy z warunków dotyczy dwóch sąsiadujących prostokątów o jednakowych rozmiarach i nakłada ograniczenia na wysokości punktów w tych prostokątach. Warunek jest zdefiniowany przez:

2	2	2	2	2	2
2	6	6	4	3	2
3	5	8	7	7	4
4	6	8	9	8	6
5	7	8	8	8	7

Digital topographic 5x6 map
Camp of size 3×5 at position (3,2)

- *pozycję, tzn. współrzędne lewego-górnego rogu pierwszego (tzn. lewego lub górnego) prostokąta. Podane współrzędne leżą w zasięgu mapy.*
- *wielkość (szerokość i wysokość) pierwszego (i jednocześnie drugiego, gdyż prostokąty są równe) prostokąta;*
- *typ rozmieszczenia, 0 wskazuje na rozmieszczenie poziome (czyli prostokąty stykają się pionowymi bokami), a 1 na rozmieszczenie pionowe (czyli prostokąty stykają się poziomymi bokami);*
- *typ wysokości, 0 wskazuje, że średnia wysokość pierwszego (czyli lewego lub górnego) prostokąta musi być ściśle ($<$) mniejsza niż drugiego; 1 oznacza odwrotną ($\geqslant$) sytuację.*

Lokalizacja obozu spełnia warunek, gdy spełniony jest warunek wysokości.
Mając daną mapę topograficzną obszaru i warunki, napisz program, który znajdzie najlepsze miejsce pod budowę nowego obozu, tzn. miejsce, które spełnia jak najwięcej warunków. W przypadku wielu rozwiązań wypisz jedno z nich.

2	2	2	2	2	2
2	6	6	4	3	2
3	5	8	7	7	4
4	6	8	9	8	6
5	7	8	8	8	7

Warunek A:
Pozycja - (1,1) *Wielkość - (1,3)*
Typ rozmieszczenia: 1 *Typ wysokości: 0*

Wybrana lokalizacja spełnia warunek A.

2	2	2	2	2	2
2	6	6	4	3	2
3	5	8	7	7	4
4	6	8	9	8	6
5	7	8	8	8	7

Warunek B:
Pozycja - (2,2) *Wielkość - (2,2)*
Typ rozmieszczenia: 0 *Typ wysokości: 0*

Wybrana lokalizacja nie spełnia warunku B.

Wejście

W pierwszym wierszu pliku `CAMP.IN` *znajdują się dwie liczby naturalne R i C $(2 \leqslant R, C \leqslant 1\,000)$. Oznaczają one liczbę wierszy i kolumn na mapie. W każdym z następnych R wierszy znajduje się po C nieujemnych liczb, każda z nich opisuje wysokość punktu na mapie w danym wierszu. W żadnym punkcie wysokość nie przekracza 255.*
W kolejnym wierszu znajdują się 2 liczby naturalne L i W $(1 \leqslant L, W \leqslant 1000; L < R; W < C)$ opisujące wielkość obozu. Następny wiersz zawiera jedną liczbę naturalną H $(1 \leqslant H \leqslant 1000)$ oznaczająca liczbę warunków.
W kolejnych H wierszach opisane są warunki. Każdy opis z nich składa się z sześciu liczb naturalnych: współrzędnych lewego-górnego rogu, rozmiarów pierwszego prostokąta oraz typu rozmieszczenia i wysokości. Prostokąty występujące w każdym z warunków leżą całkowicie w zasięgu obozu.

Wyjście

Jedyny wiersz pliku wyjściowego `CAMP.OUT` *powinien zawierać 2 liczby naturalne — współrzędne lewego-górnego rogu znalezionego miejsca na obóz.*

Przykład

WEJŚCIE

```
5 6
2 2 2 2 2 2
2 6 6 4 3 2
3 5 8 7 7 4
4 6 8 9 8 6
5 7 8 8 8 7
3 5
3
1 1 1 3 1 0
2 2 2 2 0 0
2 4 1 1 1 1
```

WYJŚCIE

```
3 1
```

WYJAŚNIENIE

Pozycja (3,1) spełnia wszystkie warunki.

Przemysława Kanarek, Marcin Michalski
Tłumaczenie

BOI, dzień I, 6–05–2005

Magiczne Nawiasy

Zadanie

W języku programowania LISP wszystko jest pisane w nawiasach (TAK JAK TO). W skutek tego w kodzie w LISP-ie niekiedy pojawiają się długie ciągi zamykających nawiasów „)))...”. Niezwykle niemiłą rzeczą dla programisty jest konieczność zliczania nawiasów, tak by liczba nawiasów zamykających dokładnie odpowiadała liczbie otwierających.

Aby uniknąć błędów, w niektórych dialektach LISP-a wprowadzono **magiczny nawias zamykający ']'**, *który zastępuje jeden lub więcej nawiasów zamykających ')', co pozwala zrównoważyć liczbę nawiasów otwierających '('. Teraz to kompilator LISP-a musi policzyć, ile nawiasów zamykających ')' zastępuje magiczny nawias ']'. Jak ma to zrobić?*

Napisz program, który wczyta ciąg znaków: nawiasów otwierających, zamykających i „magicznych” i dla każdego magicznego nawiasu wypisze, ilu otwierającym nawiasom on odpowiada.

Wejście

W pierwszym wierszu pliku wejściowego `LISP.IN` *znajdują się dwie liczby N, M oddzielone pojedynczą spacją ($0 \leqslant N \leqslant 10\,000\,000$, $0 \leqslant M \leqslant 5\,000\,000$). Pierwsza liczba oznacza długość ciągu znaków, natomiast druga jest liczbą magicznych nawiasów występujących w ciągu. W dalszej części wejścia, począwszy od drugiego wiersza, znajduje się ciąg N znaków składający się ze znaków (,),]. Znak ']' występuje dokładnie M razy ($M \leqslant N$). Dla większej przejrzystości ciąg jest podzielony na wiersze — w każdym występują maksymalnie 72 znaki.*

Wyjście

W pierwszym wierszu pliku wyjściowego `LISP.OUT` *powinna znaleźć się liczba 0 lub 1.*
Liczba 0 oznacza, że nie da się poprawnie zbalansować danego ciągu (np. nie można poprawnie zbalansować ciągu złożonego z jednego magicznego nawiasu ']'). W tym przypadku to koniec wyniku.
Liczba 1 oznacza, że ciąg można poprawnie zbalansować. W takim przypadku należy wypisać w wyniku jeszcze M wierszy.
W pierwszym z dodatkowych wierszy znajduje się liczba $C_1 \geqslant 1$ oznaczająca, ile nawiasów jest zastąpionych przez pierwszy magiczny nawias. Drugi dodatkowy wiersz zawiera liczbę $C_2 \geqslant 1$ oznaczającą, ile nawiasów jest zastąpionych przez drugi magiczny nawias, itd.
Jeżeli jest wiele możliwych rozwiązań, wypisz którekolwiek.

Przykład

Przykładowe wejście opisuje 8-znakowy ciąg z dwoma magicznymi nawiasami. Przykładowe wyjście opisuje jedno z możliwych rozwiązań: pierwszy magiczny nawias zastępuje trzy nawiasy zamykające, a drugi — jeden. I rzeczywiście, po wstawieniu odpowiedniej liczby nawiasów w miejsce nawiasów magicznych otrzymujemy poprawnie ponawiasowane wyrażenie: `((((()))))`. *Podkreślone nawiasy są nawiasami wstawionymi w miejsce nawiasów magicznych.*

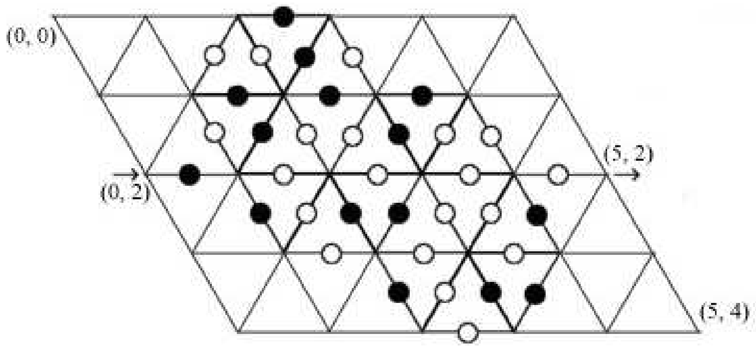
WEJŚCIE	WYJŚCIE
`8 2`	`1`
`(((((])]`	`3`
	`1`

Przemysława Kanarek, Marcin Michalski
Tłumaczenie

BOI, dzień I, 6–05–2005

Plansza

Rozważmy następującą planszę złożoną z trójkątów równobocznych:



Każdy wierzchołek na planszy jest opisany przez dwie współrzędne x, y w sposób przedstawiony na rysunku. Niektóre krawędzie są oznaczone białym lub czarnym kółkiem. Obowiązują dwie podstawowe zasady poruszania się po planszy:

- *Wolno przechodzić tylko po krawędziach oznaczonych kółkami.*
- *Podczas wędrówki po planszy trzeba na zmianę przechodzić po krawędziach oznaczonych białymi i czarnymi kółkami; tzn. przez krawędź z białym kółkiem można przejść tylko wówczas, gdy poprzedni ruch był przez krawędź z czarnym kółkiem, i na odwrót. W pierwszym ruchu można przejść albo przez krawędź z białym kółkiem, albo przez krawędź z czarnym kółkiem.*

Zadanie

Napisz program, który obliczy długość najkrótszej ścieżki z punktu wejściowego planszy do punktu wyjściowego. Długość ścieżki to liczba krawędzi (lub, równoznacznie, kółek), przez które ta droga przechodzi. Możesz założyć, że taka droga zawsze istnieje.

Wejście

Plik wejściowy nazywa się `MAZE.IN` *Pierwszy wiersz zawiera dwie liczby całkowite W i H oznaczające odpowiednio szerokość i wysokość planszy ($1 \leqslant W, H \leqslant 500$). Drugi wiersz zawiera cztery liczby całkowite: X_1 Y_1 X_2 Y_2 ($0 \leqslant X1, X2 \leqslant W$; $0 \leqslant Y_1, Y_2 \leqslant H$); ($X_1, Y_1$) są współrzędnymi punktu wejściowego planszy, a (X_2, Y_2) — współrzędnymi punktu wyjściowego.*

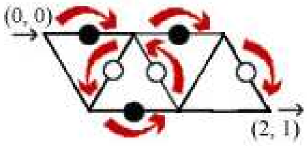
W kolejnych $2H + 1$ wierszach znajdują się opisy krawędzi: wiersze nieparzyste (3-ci, 5-ty itd.) opisują krawędzie poziome, a wiersze parzyste: (4-ty, 6-ty itd.) opisują pozostałe krawędzie niepoziome. Każdy wiersz zawiera ciąg liter n, w oraz b, gdzie n oznacza, że na *danej krawędzi nie ma kółka, a w i b oznaczają, że na danej krawędzi jest białe (w) lub czarne (b) kółko. Pomiędzy literami zapisanymi w wierszu nie ma odstępów. Oczywiście każdy wiersz nieparzysty zawiera dokładnie W liter, a każdy wiersz parzysty $2W+1$ liter.*

Wyjście

Twój program powinien wypisać jedną liczbę całkowitą (oznaczającą długość najkrótszej ścieżki od punktu wejściowego do punktu wyjściowego planszy) w pierwszym (i jedynym) wierszu pliku `MAZE.OUT`.

Przykład

WEJŚCIE	WYJŚCIE	UWAGI

`2 1`
`0 0 2 1`
`bb`
`nwwnw`
`bn`
	`6`	*Prosta plansza. Jedyną z możliwych najkrótszych ścieżek jest:*
		$(0,0)\to(1,0)\to(0,1)\to(1,1)\to(1,0)\to(2,0)\to(2,1)$
		Poniższy rysunek przedstawia planszę oraz najkrótsza ścieżkę:
		

`5 4`
`0 2 5 2`
`nnbnn`
`nnnwwbwnnnn`
`nbbbn`
`nnwbwwbwwnn`
`bwwww`
`nnbwbbwwbnn`
`nwwwn`
`nnnnbwbbnnn`
`nnwnn`
	`22`	*Ten opis zawiera planszę przedstawioną na rysunku w treści zadania. Najkrótszą ścieżką jest:*
		$(0,2)\to(1,2)\to(1,1)\to(2,1)\to(2,0)\to(3,0)\to(3,1)\to(3,2)\to(4,1)\to(3,1)\to(3,0)\to(2,0)\to(2,1)\to(1,1)\to(1,2)\to(1,3)\to(2,3)\to(2,4)\to(3,4)\to(3,3)\to(4,3)\to(4,2)\to(5,2)$
		(Długość: 22)

Przemysława Kanarek, Marcin Michalski
Tłumaczenie

BOI, dzień II, 7–05–2005

Starożytny rękopis

Zadanie

Nadbałtyccy archeolodzy pracują obecnie nad bardzo ważnym projektem. Ostatnio znaleźli starożytny rękopis, który może okazać się kluczowy dla zrozumienia kultury występującej na badanych przez nich terenach. Rękopis zawiera wiele rycin, więc archeolodzy z grubsza orientują się, jaka jest jego treść.

Jednak obok rycin w rękopisie znajdują się także zapiski, z którymi naukowcy mają poważny problem. Nie dość, że używany w nich język jest bardzo stary, to jeszcze fragmenty rękopisu uległy zniszczeniu, a niektóre litery wyblakły. Stąd naukowcy nie są w stanie zrozumieć w pełni treści tych zapisków.

Jeden z naukowców stwierdził, że słowa występujące w rękopisie przypominają mu język, w którym w żadnym słowie nie może wystąpić kolejno więcej niż V_C *samogłosek (analogicznie* C_C *spółgłosek) oraz nie może być więcej niż* V_E *równych kolejnych samogłosek (analogicznie* C_E *spółgłosek).*

Naukowiec ten opuścił grupę badawczą poszukując dokładniejszych informacji. Pozostali, oczekując na jego powrót, postanowili sprawdzić, czy w rękopisie nie występują fragmenty przeczące postawionej hipotezie i ile pracy ich czeka przy jego odczytywaniu. Chcą dowiedzieć się, na ile różnych sposobów można odszyfrować rękopis. Trzeba im w tym pomóc!

Uwaga: samogłoski to: „aeiou", natomiast spółgłoski, to pozostałe 21 liter alfabetu.

Wejście

Plik wejściowy nazywa się `ANCIENT.IN`*. W pierwszym wierszu znajdują się cztery liczby całkowite* V_E, V_C, C_E *oraz* C_C ($1 \leqslant V_E \leqslant V_C \leqslant 4$, $1 \leqslant C_E \leqslant C_C \leqslant 4$) *oddzielone pojedynczymi spacjami. W drugim wierszu znajduje się jedno słowo wybrane ze starożytnego rękopisu składające się z najwyżej 15 małych liter alfabetu łacińskiego. Każda brakująca litera w słowie (o ile takie występują) jest zastąpiona znakiem „*".*

Wyjście

Plik wyjściowy nazywa się `ANCIENT.OUT`*. Powinna w nim znajdować się jedna liczba mówiąca, na ile sposobów można odtworzyć słowo zgodnie z podanymi ograniczeniami. Możesz założyć, że wynik będzie maksymalnie liczbą całkowitą 64-bitową ze znakiem. Gdyby okazało się, że hipoteza dotycząca języka rękopisu jest fałszywa i słowa nie da się odtworzyć zgodnie z podanymi ograniczeniami, wówczas należy wypisać wynik 0.*

Przykład

WEJŚCIE	**WYJŚCIE**
`1 1 1 1` `a**`	`105`
`1 1 1 1` `b*i`	`0`
`1 2 1 2` `ancient`	`1`
`4 4 4 4` `man****ipt`	`261870`
`2 2 2 2` `*boi*`	`546`

Przemysława Kanarek, Marcin Michalski
Tłumaczenie

BOI, dzień II, 7–05–2005

Podróż autobusem

Zadanie

Mamy N miast oraz M jednokierunkowych, bezpośrednich linii autobusowych pomiędzy tymi miastami. Miasta są ponumerowane liczbami od 1 do N. Podróżny, który znajduje się w mieście 1 w czasie 0, chce dotrzeć do miasta P w czasie T. Dokładnie o tej godzinie i w tym miejscu ma spotkanie. Jeśli przyjedzie wcześniej, to musi poczekać na przystanku.

Dla każdej linii autobusowej i znamy oczywiście jej miasto początkowe i docelowe: s_i oraz t_i. Znamy także czasy odjazdu i przyjazdu, ale jedynie z pewnym przybliżeniem: wiemy, że autobus odjeżdża z s_i w przedziale czasowym $[a_i,b_i]$ i przybywa do t_i w przedziale czasowym $[c_i,d_i]$ (oba przedziały są domknięte).

Podróżny nie lubi przesiadywać na przystankach, szuka więc takiego połączenia, by zminimalizować maksymalny możliwy czas oczekiwania na przystankach, a jednocześnie mieć pewność, że nie spóźni się nigdzie na przesiadkę (tzn. przy każdej przesiadce najpóźniejszy czas przyjazdu autobusu, którym przyjeżdża, nie może być późniejszy niż najwcześniejszy czas odjazdu autobusu, na który przesiada się)

Licząc czas oczekiwania na przystankach podróżny musi wziąć pod uwagę najwcześniejszy możliwy czas przyjazdu autobusu, którym przyjeżdża, oraz najpóźniejszy możliwy czas odjazdu autobusu, na który przesiada się.

Napisz program, który pomoże podróżnemu zaplanować odpowiednią trasę.

Wejście

Plik wejściowy nazywa się `TRIP.IN`*. W jego pierwszym wierszu znajdują się liczby całkowite N $(1 \leqslant N \leqslant 50\,000)$, M $(1 \leqslant M \leqslant 100\,000)$, P $(1 \leqslant P \leqslant N)$ oraz T $(0 \leqslant T \leqslant 1\,000\,000\,000)$.*

W kolejnych M wierszach są opisane linie autobusowe. Każdy opis linii składa się z liczb całkowitych s_i, t_i, a_i, b_i, c_i, d_i, gdzie s_i oraz t_i oznaczają miasto początkowe i docelowe linii i, natomiast a_i, b_i, c_i, d_i oznaczają czas odjazdu i przyjazdu opisany jak w treści zadania $(1 \leqslant s_i \leqslant N,\ 1 \leqslant t_i \leqslant N,\ 0 \leqslant a_i \leqslant b_i < c_i \leqslant d_i \leqslant 1\,000\,000\,000)$.

Wyjście

Jedyny wiersz pliku wyjściowego `TRIP.OUT` *powinien zawierać maksymalny możliwy sumaryczny czas oczekiwania przy najlepiej zaplanowanej podróży. Jeżeli nie można zagwarantować dojazdu do P w czasie T, to w wierszu należy napisać -1.*

Przykład

WEJŚCIE	**WYJŚCIE**

`3 6 2 100`
`1 3 10 20 30 40`
`3 2 32 35 95 95`
`1 1 1 1 7 8`
`1 3 8 8 9 9`
`2 2 98 98 99 99`
`1 2 0 0 99 101`
	`32`

Najbardziej pesymistyczny przypadek oznaczający najdłuższy czas oczekiwania przy najlepiej zaplanowanej podróży dla powyższych danych, to:

Czas:	**Podróżny:**
0...1	**Czeka w mieście 1**
1...7	*Jedzie autobusem 3 z 1 do 1*
7...8	**Czeka w mieście 1**
8...9	*Jedzie autobusem 4 z 1 do 3*
9...35	**Czeka w mieście 3**
35...95	*Jedzie autobusem 2 z 3 do 2*
95...98	**Czeka w mieście 2**
98...99	*Jedzie autobusem 5 z 2 do 2*
99...100	**Czeka w mieście 2**

Sumaryczny czas oczekiwania na przystankach:
$1+1+26+3+1=32$

WEJŚCIE	**WYJŚCIE**

`3 2 2 100`
`1 3 0 0 49 51`
`3 2 50 51 100 100`
	`-1`

Przemysława Kanarek, Marcin Michalski
Tłumaczenie

BOI, dzień II, 7–05–2005

Wielokąt

Zadanie

Zadanie polega na znalezieniu wielokąta wypukłego, którego boki maja podane długości.

W tym zadaniu zakładamy, że wielokąt jest wypukły wtedy, gdy wszystkie jego kąty wewnętrzne są ostro większe od 0 i ostro mniejsze od 180.

Wejście

W pierwszym wierszu pliku wejściowego `POLY.IN` *znajduje się jedna liczba całkowita* N ($3 \leqslant N \leqslant 1\,000$)*, oznaczająca liczbę boków wielokąta. Każdy z kolejnych* N *wierszy zawiera liczbę całkowitą* a_i*, oznaczającą długość jednego z boków wielokąta* ($1 \leqslant a_i \leqslant 10,000$).

Wyjście

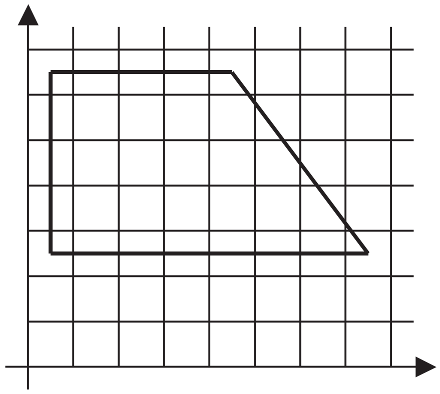
Jeżeli można skonstruować wielokąt wypukły o podanych bokach, to plik wyjściowy `POLY.OUT` *powinien zawierać dokładnie* N *wierszy. Każdy z nich powinien zawierać dwie liczby rzeczywiste* x_i, y_i ($|x_i|,|y_i| \leqslant 10\,000\,000$)*, takie że po połączeniu odcinkami punktów* (x_i, y_i) *i* (x_{i+1}, y_{i+1})*, dla* $1 \leqslant i < N$*, oraz punktów* (x_N, y_N) *i* (x_1, y_1)*, otrzymamy wielokąt wypukły. Długości boków tego wielokąta muszą być równe liczbom podanym w wejściu, jednak kolejność nie musi być zachowana.*

Wierzchołki utworzonego wielokąta można wypisać zgodnie z kierunkiem ruchu wskazówek zegara lub przeciwnie.

Jeżeli nie można stworzyć takiego wielokąta, należy wypisać: `NO SOLUTION` *w jedynym wierszu pliku wyjściowego.*

Przykład

WEJŚCIE	**WYJŚCIE**
`4`	`0.5  2.5`
`7`	`7.5  2.5`
`4`	`4.5  6.5`
`5`	`0.5  6.5`
`4`	



Ocenianie

Program sprawdzający uznaje dwie długości za równe, gdy różnią się o mniej niż 0.001*. Każdy format zmiennoprzecinkowy jest akceptowalny.*

Bibliografia

[1] *I Olimpiada Informatyczna 1993/1994*. Warszawa–Wrocław, 1994.

[2] *II Olimpiada Informatyczna 1994/1995*. Warszawa–Wrocław, 1995.

[3] *III Olimpiada Informatyczna 1995/1996*. Warszawa–Wrocław, 1996.

[4] *IV Olimpiada Informatyczna 1996/1997*. Warszawa, 1997.

[5] *V Olimpiada Informatyczna 1997/1998*. Warszawa, 1998.

[6] *VI Olimpiada Informatyczna 1998/1999*. Warszawa, 1999.

[7] *VII Olimpiada Informatyczna 1999/2000*. Warszawa, 2000.

[8] *VIII Olimpiada Informatyczna 2000/2001*. Warszawa, 2001.

[9] *IX Olimpiada Informatyczna 2001/2002*. Warszawa, 2002.

[10] *X Olimpiada Informatyczna 2002/2003*. Warszawa, 2003.

[11] *XI Olimpiada Informatyczna 2003/2004*. Warszawa, 2004.

[12] A. V. Aho, J. E. Hopcroft, J. D. Ullman. *Projektowanie i analiza algorytmów komputerowych*. PWN, Warszawa, 1983.

[13] L. Banachowski, A. Kreczmar. *Elementy analizy algorytmów*. WNT, Warszawa, 1982.

[14] L. Banachowski, K. Diks, W. Rytter. *Algorytmy i struktury danych*. WNT, Warszawa, 1996.

[15] J. Bentley. *Perełki oprogramowania*. WNT, Warszawa, 1992.

[16] I. N. Bronsztejn, K. A. Siemiendiajew. *Matematyka. Poradnik encyklopedyczny*. PWN, Warszawa, wydanie XIV, 1997.

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest. *Wprowadzenie do algorytmów*. WNT, Warszawa, 1997.

[18] D. Harel. *Algorytmika. Rzecz o istocie informatyki*. WNT, Warszawa, 1992.

[19] J. E. Hopcroft, J. D. Ullman. *Wprowadzenie do teorii automatów, języków i obliczeń*. PWN, Warszawa, 1994.

[20] W. Lipski. *Kombinatoryka dla programistów*. WNT, Warszawa, 2004.

[21] E. M. Reingold, J. Nievergelt, N. Deo. *Algorytmy kombinatoryczne*. WNT, Warszawa, 1985.

[22] R. Sedgewick. *Algorytmy w C++. Grafy*. RM, 2003.

[23] M. M. Sysło. *Algorytmy*. WSiP, Warszawa, 1998.

[24] M. M. Sysło. *Piramidy, szyszki i inne konstrukcje algorytmiczne*. WSiP, Warszawa, 1998.

[25] M. M. Sysło, N. Deo, J. S. Kowalik. *Algorytmy optymalizacji dyskretnej z programami w języku Pascal*. PWN, Warszawa, 1993.

[26] R. J. Wilson. *Wprowadzenie do teorii grafów*. PWN, Warszawa, 1998.

[27] N. Wirth. *Algorytmy + struktury danych = programy*. WNT, Warszawa, 1999.

[28] Ronald L. Graham, Donald E. Knuth, Oren Patashnik. *Matematyka konkretna*. PWN, Warszawa, 1996.

[29] K. A. Ross, C. R. B. Wright. *Matematyka dyskretna*. PWN, Warszawa, 1996.

[30] Donald E. Knuth. *Sztuka programowania*. WNT, Warszawa, 2002.

[31] Steven S. Skiena, Miguel A. Revilla. *Wyzwania programistyczne*. WSiP, Warszawa, 2004.

[32] A. Silberschatz, P. B. Galvin. *Podstawy systemów operacyjnych*. WNT, Warszawa, 2000.

Niniejsza publikacja stanowi wyczerpujące źródło informacji o zawodach XII Olimpiady Informatycznej, przeprowadzonych w roku szkolnym 2004/2005. Książka zawiera informacje dotyczące organizacji, regulaminu oraz wyników zawodów. Zawarto w niej także opis rozwiązań wszystkich zadań konkursowych. Do książki dołączony jest dysk CD-ROM zawierający wzorcowe rozwiązania i testy do wszystkich zadań XII Olimpiady Informatycznej.

Książka zawiera też zadania z XVI Międzynarodowej Olimpiady Informatycznej oraz XI Bałtyckiej Olimpiady Informatycznej.

XII Olimpiada Informatyczna to pozycja polecana szczególnie uczniom przygotowującym się do udziału w zawodach następnych Olimpiad oraz nauczycielom informatyki, którzy znajdą w niej interesujące i przystępnie sformułowane problemy algorytmiczne wraz z rozwiązaniami. Książka może być wykorzystywana także jako pomoc do zajęć z algorytmiki na studiach informatycznych.

Olimpiada Informatyczna
jest organizowana przy współudziale

PROKOM
SOFTWARE SA

ISBN 83–922946–0–2