MINISTERSTWO EDUKACJI NARODOWEJ
INSTYTUT INFORMATYKI UNIWERSYTETU WROCŁAWSKIEGO
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

VI OLIMPIADA INFORMATYCZNA
1998/1999

WARSZAWA, 1999

MINISTERSTWO EDUKACJI NARODOWEJ
INSTYTUT INFORMATYKI UNIWERSYTETU WROCŁAWSKIEGO
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

VI OLIMPIADA INFORMATYCZNA
1998/1999

WARSZAWA, 1999

Autorzy tekstów:

dr hab. Bogdan S. Chlebus
dr hab. Krzysztof Diks
dr hab. Wojciech Guzicki
Grzegorz Jakacki
mgr Marcin Kubica
dr Krzysztof Loryś
prof. dr hab. Wojciech Rytter
Marcin Sawicki
dr Andrzej Walat

Autorzy programów na dyskietce:

Adam Borowski
Marcin Mucha
Marek Pawlicki
Krzysztof Sobusiak
Marcin Stefaniak
Radosław Szklarczyk
Tomasz Waleń

Opracowanie i redakcja:

dr hab. Krzysztof Diks
Grzegorz Jakacki

Skład: Grzegorz Jakacki

Pozycja dotowana przez Ministerstwo Edukacji Narodowej.

Druk książki został sfinansowany przez PROKOM SOFTWARE S.A.

ISBN 83–906301–5–X

Spis treści

Krzysztof Diks

Wstęp

W roku szkolnym 1998/99 odbyła się VI Olimpiada Informatyczna. Przekazujemy czytelnikom relację z jej przebiegu, a także opracowania zadań wykorzystanych w trakcie Olimpiady. W niniejszej publikacji prezentujemy również krótkie sprawozdanie z X Międzynarodowej Olimpiady Informatycznej. Olimpiada Międzynarodowa odbyła się w Setubal (Portugalia) w październiku ubiegłego roku. W tej imprezie brali udział laureaci V Olimpiady Informatycznej. Oprócz zadań z tegorocznej Olimpiady zamieszczamy także zadania z Olimpiady w Setubal.

W opracowaniu zawarliśmy oficjalne dokumenty Komitetu Głównego: „Sprawozdanie z działań VI Olimpiady Informatycznej", „Regulamin Olimpiady Informatycznej" oraz „Zasady organizacji zawodów w roku szkolnym 1998/1999". Dokumenty te specyfikują wymogi stawiane rozwiązaniom zawodników, formę przeprowadzania zawodów, a także sposób przyznawania nagród i wyróżnień.

Uczestników, przyszłych uczestników i nauczycieli informatyki zapewne najbardziej zainteresują zamieszczone w publikacji opracowania zadań. Na każde opracowanie składa się opis algorytmu oraz program wzorcowy i testy, które posłużyły do sprawdzenia poprawności i efektywności rozwiązań zawodników. Autorami opracowań są pomysłodawcy zadań. Autorami programów wzorcowych są członkowie Jury.

W obecnej formule Olimpiady najważniejszym elementem rozwiązania jest działający program, realizujący właściwie skonstruowany algorytm. Dokumentacja i opis algorytmu stanowią jedynie dodatek do programu i są wykorzystywane przez Jury w sytuacjach spornych lub wyjątkowych. Ocena sprawności programu bazuje na wynikach jego pracy na testach przygotowanych przez Jury. Testy pozwalają zbadać poprawność semantyczną programu oraz efektywność użytego algorytmu.

W niniejszej publikacji staraliśmy się przedstawić opracowania zadań w formie przystępnej dla uczniów. Czasami w tekście występują odwołania do literatury mające na celu zachęcenie ucznia do pogłębienia wiedzy na konkretny temat lub zapoznanie go z problematyką zbyt szeroką, by wyczerpująco poruszać ją na niniejszych stronach. Lista pozycji literaturowych zamieszczona na końcu książki zawiera nie tylko opracowania, do których autorzy odwołują się w swoich tekstach,* ale także pozycje poświęcone ogólnym zagadnieniom związanym z analizą algorymtów, strukturami danych itp., szczególnie polecane jako lektura i źródła problemów dla uczniów zainteresowanych informatyką.

Do książki dołączona jest dyskietka zawierająca programy (w językach Pascal lub C) będące rozwiązaniami wzorcowymi zadań VI Olimpiady Informatycznej oraz testy.

Autorzy i redaktorzy niniejszej pozycji starali się zadbać o to, by do rąk Czytelnika trafiła książka wolna od wad i błędów. Wszyscy, którym pisanie i uruchamianie programów komputerowych nie jest obce, wiedzą, jak trudnym jest takie zadanie. Przepraszając z góry za usterki niniejszej edycji, prosimy P.T. Czytelników o informacje o dostrzeżonych błędach. Pozwoli to nam uniknąć ich w przyszłości.

Książkę tę, i jej poprzedniczki dotyczące zawodów II, III, IV i V Olimpiady, można zakupić:

- *w sieci księgarni „Elektronika" w Warszawie, Łodzi i Wrocławiu,*
- *w niektórych księgarniach technicznych,*
- *w niektórych sklepach ze sprzętem komputerowym,*
- *w Komitetach Okręgowych Olimpiady Informatycznej:*
    - *w Warszawie: Ośrodek Edukacji Informatycznej i Zastosowań Komputerów, 02-026 Warszawa, ul. Raszyńska 8/10, tel. (+22) 8224019*
    - *we Wrocławiu: Instytut Informatyki Uniwersytetu Wrocławskiego, 51-151 Wrocław, ul. Przesmyckiego 20, tel. (+71) 3251271*

* Odwołania do pozycji literaturowych w tekście mają postać numeru pozycji na liście zamieszczonej na końcu książki ujętego w nawiasy kwadratowe [ ].

- *w Toruniu: Wydział Matematyki i Informatyki UMK, 87-100 Toruń, ul. Chopina 12/18 tel. (+56) centr. 6226017, 6226018 lub 6226584 wewn. 36, dr Bolesław Wojdyło,*

- *w sprzedaży wysyłkowej za zaliczeniem pocztowym w Komitecie Głównym Olimpiady Informatycznej. Zamówienia prosimy przesyłać pod adresem 02–026 Warszawa, ul. Raszyńska 8/10.*

Niestety, nakład publikacji o pierwszej Olimpiadzie jest już wyczerpany.

Sprawozdanie z działań VI Olimpiady Informatycznej 1998/1999

Olimpiada Informatyczna została powołana 10 grudnia 1993 roku przez Instytut Informatyki Uniwersytetu Wrocławskiego zgodnie z zarządzeniem nr 28 Ministra Edukacji Narodowej z dnia 14 września 1992 roku.

ORGANIZACJA ZAWODÓW

W roku 1998/1999 odbyły się zawody VI Olimpiady Informatycznej. Olimpiada Informatyczna jest trójstopniowa. Integralną częścią rozwiązania każdego zadania zawodów I, II i III stopnia jest program napisany na komputerze zgodnym ze standardem IBM PC, w języku programowania wysokiego poziomu (Pascal, C, C++). Zawody I stopnia miały charakter otwartego konkursu przeprowadzonego dla uczniów wszystkich typów szkół młodzieżowych.

12 października 1998 r. rozesłano plakaty zawierające zasady organizacji zawodów I stopnia oraz zestaw 4-ch zadań konkursowych do 3350-ciu szkół i zespołów szkół młodzieżowych ponadpodstawowych oraz do wszystkich kuratorów i koordynatorów edukacji informatycznej. Zawody I stopnia rozpoczęły się dnia 26 października 1998 roku. Ostatecznym terminem nadsyłania prac konkursowych był 23 listopada 1998 roku.

Zawody II i III stopnia były dwudniowymi sesjami stacjonarnymi, poprzedzonymi jednodniowymi sesjami próbnymi. Zawody II stopnia odbyły się w trzech okręgach oraz w Krakowie, Katowicach i Sopocie w dniach 9–11.02.1999 r., natomiast zawody III stopnia odbyły się w ośrodku firmy Combidata Poland S.A. w Sopocie w dniach 20–24.04.1999 r.

Uroczystość zakończenia VI Olimpiady Informatycznej odbyła się w dniu 24.04.1999 r. w sali posiedzeń Urzędu Miasta w Sopocie.

SKŁAD OSOBOWY KOMITETÓW OLIMPIADY INFORMATYCZNEJ

Komitet Główny

przewodniczący:
prof. dr hab. inż. Stanisław Waligórski (Uniwersytet Warszawski)

z-cy przewodniczącego:
prof. dr hab. Maciej M. Sysło (Uniwersytet Wrocławski)
dr hab. Krzysztof Diks (Uniwersytet Warszawski)

sekretarz naukowy:
dr Andrzej Walat (OEIiZK)*

kierownik organizacyjny:
Tadeusz Kuran (OEIiZK)

członkowie:
prof. dr hab. Jan Madey (Uniwersytet Warszawski)
prof. dr hab. Wojciech Rytter (Uniwersytet Warszawski)
dr Przemysława Kanarek (Uniwersytet Wrocławski)
dr Krzysztof Loryś (Uniwersytet Wrocławski)
dr Bolesław Wojdyło (Uniwersytet Mikołaja Kopernika w Toruniu)
mgr Jerzy Dałek (Ministerstwo Edukacji Narodowej)
mgr Marcin Kubica (Uniwersytet Warszawski)
mgr Krzysztof Stencel (Uniwersytet Warszawski)
mgr Krzysztof J. Święcicki (Ministerstwo Edukacji Narodowej)

sekretarz Komitetu Głównego:
Monika Kozłowska-Zając

Siedzibą Komitetu Głównego Olimpiady Informatycznej jest Ośrodek Edukacji Informatycznej i Zastosowań Komputerów w Warszawie przy ul. Raszyńskiej 8/10.

Komitet Główny odbył 5 posiedzeń, a Prezydium — 4 zebrania. 29 stycznia 1999 r. przeprowadzono jednodniowe seminarium przygotowujące przeprowadzenie w okręgach zawodów II stopnia.

Komitet Okręgowy w Warszawie

przewodniczący:
dr hab. Krzysztof Diks (Uniwersytet Warszawski)

członkowie:
dr Andrzej Walat (OEIiZK)
mgr Marcin Kubica (Uniwersytet Warszawski)
mgr Adam Malinowski (Uniwersytet Warszawski)
mgr Wojciech Plandowski (Uniwersytet Warszawski)

Siedzibą Komitetu Okręgowego jest Ośrodek Edukacji Informatycznej i Zastosowań Komputerów w Warszawie, ul. Raszyńska 8/10.

Komitet Okręgowy we Wrocławiu

przewodniczący:
dr Krzysztof Loryś (Uniwersytet Wrocławski)

* Ośrodek Edukacji Informatycznej i Zastosowań Komputerów w Warszawie

z-ca przewodniczącego:
dr Helena Krupicka (Uniwersytet Wrocławski)

sekretarz:
inż. Maria Woźniak (Uniwersytet Wrocławski)

członkowie:
dr Przemysława Kanarek (Uniwersytet Wrocławski)
dr Witold Karczewski (Uniwersytet Wrocławski)
mgr Jacek Jagiełło (Uniwersytet Wrocławski)
mgr Paweł Keller (Uniwersytet Wrocławski)

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

Komitet Okręgowy w Toruniu

przewodniczący:
prof. dr hab. Józef Słomiński (Uniwersytet Mikołaja Kopernika w Toruniu)
z-ca przewodniczącego:
dr Mirosława Skowrońska (Uniwersytet Mikołaja Kopernika w Toruniu)

sekretarz:
dr Bolesław Wojdyło (Uniwersytet Mikołaja Kopernika w Toruniu)
członkowie:
dr Krzysztof Skowronek (V Liceum Ogólnokształcące w Toruniu)
mgr Anna Kwiatkowska (IV Liceum Ogólnokształcące w Toruniu)

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

JURY OLIMPIADY INFORMATYCZNEJ

W pracach Jury, które nadzorowali prof. Stanisław Waligórski i dr Andrzej Walat, a którymi kierował mgr Krzysztof Stencel, brali udział pracownicy i studenci Instytutu Informatyki Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego:

Adam Borowski
Grzegorz Jakacki
Marcin Mucha
Marek Pawlicki
Marcin Sawicki
Piotr Sankowski
Marcin Stefaniak
Krzysztof Sobusiak
Tomasz Waleń
Paweł Wolf

ZAWODY I STOPNIA

W VI Olimpiadzie Informatycznej wzięło udział 722 zawodników. Decyzją Komitetu Głównego Olimpiady do zawodów zostało dopuszczonych 2-ch uczniów ze szkół podstawowych:

- S.P. nr 7 im. T. Kościuszki w Sopocie: Paweł Pazderski,

- S.P. nr 6 w Wejherowie: Krzysztof Ślusarski.

Wirusy usunięto z 36-ciu dyskietek.

W VI Olimpiadzie Informatycznej sklasyfikowano 722 zawodników, którzy nadesłali łącznie na zawody I stopnia:

631 rozwiązań zadania	*Muszkieterowie*
459	*Monocyfrowe reprezentacje*
614	*Puste prostopadłościany*
661	*Gra w wielokąty*

2365 rozwiązania łącznie

Z rozwiązaniami:

czterech zadań nadeszły	402 prace
trzech zadań nadeszło	175 prac
dwóch zadań nadeszło	87 prac
jednego zadania nadeszło	58 prac

Zawodnicy reprezentowali 48 województw. Nie było zawodników tylko z województwa bialskopodlaskiego. Województwa jeleniogórskie, konińskie, słupskie reprezentował jeden zawodnik. Najliczniej reprezentowane były następujące województwa:

st.warszawskie	71 zawodników
gdańskie	60
katowickie	56
krakowskie	53
bydgoskie	43
poznańskie	31
wrocławskie	29
łódzkie	24
rzeszowskie	24
toruńskie	20
szczecińskie	18
tarnowskie	18
olsztyńskie	17
kieleckie	14
lubelskie	14
nowosądeckie	14
przemyskie	14
tarnobrzeskie	13
kaliskie	12
białostockie	11
częstochowskie	11
gorzowskie	11
radomskie	11
bielskie	10
zielonogórskie	10

W zawodach I stopnia najliczniej reprezentowane były szkoły:

III L. O. im. Marynarki Wojennej RP w Gdyni	32 zawodników
V L.O. im. A. Witkowskiego w Krakowie	22
VI L.O. im. J. i J. Śniadeckich w Bydgoszczy	16
VIII L.O. im. A. Mickiewicza w Poznaniu	15
IV L.O. im. T. Kościuszki w Toruniu	14
XIV L.O. im. S. Staszica w Warszawie	14
III L.O. im. A. Mickiewicza we Wrocławiu	12

Liceum Ogólnokształcące w Dębicy	9
XIV L.O. im. Polonii Belgijskiej we Wrocławiu	8
I L.O. im. A. Mickiewicza w Białymstoku	7
II L.O. im. K. I. Gałczyńskiego w Olsztynie	7
XXVII L.O. im. T. Czackiego w Warszawie	7
IV L.O. im. Kazimierza Wielkiego w Bydgoszczy	6
VI L.O. im. J. Kochanowskiego w Radomiu	6
IV L.O. im. M. Kopernika w Rzeszowie	6
I L.O. im. M. Skłodowskiej-Curie w Szczecinie	6
VII Liceum Ogólnokształcące w Gdańsku	5
VIII L.O. im. S. Wyspiańskiego w Krakowie	5
I L.O. im. S. Staszica w Lublinie	5
I L.O. im. J. Długosza w Nowym Sączu	5
I Społeczne L. O. w Warszawie	5
V L.O. im. Ks. J. Poniatowskiego w Warszawie	5
XXVIII L.O. im. J. Kochanowskiego w Warszawie	5

Ogólnie najliczniej reprezentowane były miasta:

Warszawa	65 uczniów	Lublin	14
Kraków	48	Rzeszów	13
Gdynia	36	Dębica	10
Bydgoszcz	31	Radom	10
Poznań	28	Białystok	9
Wrocław	28	Częstochowa	9
Szczecin	18	Gliwice	9
Gdańsk	16	Gorzów Wlkp.	8
Łódź	15	Olkusz	8
Olsztyn	15	Zielona Góra	8
Toruń	15	Katowice	7

Zawodnicy uczęszczają do następujących klas:

do klasy I	szkoły średniej	39 zawodników
do klasy II		143
do klasy III		241
do klasy IV		274
do klasy V		23
do klasy VIII	szkoły podstawowej	2

Zawodnicy najczęściej używali następujących języków programowania:

Pascal firmy Borland	558 prace
C/C++ firmy Borland	132 prac

Ponadto pojawiły się:

Watcom C/C++	13 prac
GNU C/C++	5 prac
Ansi C	2 prace
Visual C	2 prace
Sas C/C++	2 prace
DJPG	5 prac

Komputerowe wspomaganie umożliwiło sprawdzenie prac zawodników kompletem 142-ch testów.

Poniższa tabela przedstawia liczby zawodników, którzy uzyskali określone liczby punktów za poszczególne zadania, w zestawieniu ilościowym i procentowym:

	Muszkieterowie		*Monocyfrowe reprezentacje*		*Puste prostopadłościany*		*Gra w wielokąty*	
	liczba zawodn.	czyli	liczba zawodn.	czyli	liczba zawodn.	czyli	liczba zawodn.	czyli
100 pkt.	48	6,6%	117	16,2%	66	9,1%	305	42,2%
99–75 pkt.	38	5,3%	43	6,0%	129	18,0%	199	27,6%
74–50 pkt.	49	6,8%	25	3,5%	130	18,0%	45	6,2%
49–1 pkt.	367	50,8%	224	31,0%	222	30,7%	64	8,9%
0 pkt.	220	30,5%	313	43,3%	175	24,2%	109	15,1%

W sumie za wszystkie 4 zadania:

SUMA	liczba zawodników	czyli
400 pkt.	18	2,5%
399–300 pkt.	92	12,7%
299–200 pkt.	148	20,5%
199–1 pkt.	441	61,1%
0 pkt.	23	3,2%

Prezydium Komitetu Głównego przy udziale Jury rozpatrzyło 6 reklamacji, które w dwóch przypadkach wniosły zmiany do punktacji zawodów I stopnia.

Wszyscy zawodnicy otrzymali listy ze swoimi wynikami oraz dyskietkami zawierającymi ich rozwiązania i testy, na podstawie których oceniano prace.

ZAWODY II STOPNIA

Do zawodów II stopnia zakwalifikowano 168-miu zawodników, którzy osiągnęli w zawodach I stopnia wynik nie mniejszy niż 248 pkt. Zawody II stopnia odbyły się w dniach 9–11 lutego 1999 r. w trzech stałych okręgach oraz w Krakowie, Katowicach i Sopocie:

- w Toruniu, 22-ch zawodników z następujących województw:
  - białostockiego (5),
  - bydgoskiego (11),
  - toruńskiego (4),
  - włocławskiego (2).
- we Wrocławiu, 35-ciu zawodników z następujących województw:
  - częstochowskiego (1),
  - gorzowskiego (2),
  - kaliskiego (1),
  - łódzkiego (5),
  - poznańskiego (6),
  - rzeszowskiego (2),

  - sieradzkiego (1),
  - tarnowskiego (4),
  - wałbrzyskiego (2),
  - wrocławskiego (9),
  - zielonogórskiego (2).
- w Warszawie, 37-miu zawodników z następujących województw:
  - chełmskiego (2),
  - lubelskiego (3),
  - radomskiego (1),
  - rzeszowskiego (7),
  - st.warszawskiego (24),
- w Krakowie, 31 zawodników z następujących województw:
  - bydgoskiego (1),
  - krakowskiego (26),
  - krośnieńskiego (2),
  - nowosądeckiego (2),
- w Katowicach, 23-ch zawodników z następujących województw:
  - katowickiego (13),
  - kieleckiego (2),
  - krośnieńskiego (1),
  - radomskiego (1),
  - rzeszowskiego (3),
- w Sopocie, 19-tu zawodników z następujących województw:
  - gdańskiego (14),
  - koszalińskiego (1),
  - suwalskiego (1),
  - szczecińskiego (3).

Najliczniej w zawodach II stopnia reprezentowane były szkoły:

V L.O. im. A. Witkowskiego w Krakowie	15 uczniów
XIV L.O. im. S. Staszica w Warszawie	9
VI L.O. im. J. i J. Śniadeckich w Bydgoszczy	8
III L.O. im. Marynarki Wojennej RP w Gdyni	7
VIII L.O. im. A. Mickiewicza w Poznaniu	5
XXVII L.O. im. T. Czackiego w Warszawie	5
II L.O. im. Ziemii Olkuskiej w Olkuszu	4
IV L.O. im. M. Kopernika w Rzeszowie	4
III L.O. im. A. Mickiewicza we Wrocławiu	4
I L.O. im. A. Mickiewicza w Białymstoku	3
I L.O. im. M. Kopernika w Gdańsku	3
I L.O. im. M. Kopernika w Łodzi	3
II L.O. im. L. Lisa-Kuli w Rzeszowie	3
IV L.O. im. T. Kościuszki w Toruniu	3
XIV L.O. im. Polonii Belgijskiej we Wrocławiu	3

Ogólnie najliczniej reprezentowane były miasta:

Kraków	26 zawodników
Warszawa	24
Bydgoszcz	11
Rzeszów	9
Wrocław	9
Gdynia	7
Gdańsk	6
Poznań	6
Białystok	4
Łódź	4
Olkusz	4
Radom	4

9 lutego odbyła się sesja próbna, na której zawodnicy rozwiązywali nie liczące się do ogólnej klasyfikacji zadanie *Rakiety*. W dniach konkursowych zawodnicy rozwiązywali zadania: *Grotołazi*, *Lunatyk*, *Bitmapa* i *Lodowisko*, oceniane każde maksymalnie po 100 punktów. Decyzją Komitetu Głównego nie przyznawano punktów uznaniowych za oryginalne rozwiązania.

Do automatycznego sprawdzania 4 zadań konkursowych zastosowano łącznie 150 testów.

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

	Grotołazi		*Lunatyk*		*Bitmapa*		*Lodowisko*	
	liczba zawodn.	czyli	liczba zawodn.	czyli	liczba zawodn.	czyli	liczba zawodn.	czyli
100 pkt.	1	0,6%	4	2,4%	36	21,4%	0	0,0%
99–75 pkt.	3	1,8%	1	0,6%	23	13,7%	0	0,0%
74–50 pkt.	19	11,3%	4	2,4%	9	5,4%	0	0,0%
49–1 pkt.	124	73,8%	79	47,0%	98	58,3%	1	0,6%
0 pkt.	21	12,5%	80	47,6%	2	1,2%	167	99,4%

W sumie za wszystkie 4 zadania przy najwyższym wyniku wynoszącym 400 pkt:

SUMA	liczba zawodników	czyli
400 pkt.	0	0,0%
399–300 pkt.	0	0,0%
299–200 pkt.	9	5,4%
199–1 pkt.	159	94,6%
0 pkt.	0	0,0%

W każdym okręgu na uroczystym zakończeniu zawodów II stopnia zawodnicy otrzymali upominki książkowe ufundowane przez Wydawnictwa Naukowo-Techniczne. Zawodnikom przesłano listy z wynikami zawodów i dyskietkami zawierającymi ich rozwiązania oraz testy, na podstawie których oceniano prace.

ZAWODY III STOPNIA

Zawody III stopnia odbyły się w ośrodku firmy Combidata Poland S.A. w Sopocie w dniach od 20 do 24 kwietnia 1999 r.

W zawodach III stopnia wzięło udział 46-ciu najlepszych uczestników zawodów II stopnia, którzy uzyskali wynik nie mniejszy niż 136 pkt. Zawodnicy reprezentowali nastepujące województwa:

st. warszawskie	10 zawodników
krakowskie	9
wrocławskie	5
rzeszowskie	4
bydgoskie	3
gdańskie	2
poznańskie	2
tarnowskie	2
toruńskie	2
białostockie	1
chełmskie	1
gorzowskie	1
katowickie	1
krośnieńskie	1
łódzkie	1
radomskie	1

Niżej wymienione szkoły miały w finale więcej niż jednego zawodnika:

V L.O. im. A. Witkowskiego w Krakowie	5 zawodników
XIV L.O. im. St. Staszica w Warszawie	5
XXVII L.O. im. T. Czackiego w Warszawie	3
III L.O. im. A. Mickiewicza we Wrocławiu	3
VI L.O. im. J. i J. Śniadeckich w Bydgoszczy	2
VIII L.O. im. A. Mickiewicza w Poznaniu	2
IV L.O. im. M. Kopernika w Rzeszowie	2
IV L.O. im. T. Kościuszki w Toruniu	2

20 kwietnia odbyła się sesja próbna, na której zawodnicy rozwiązywali nie liczące się do ogólnej klasyfikacji zadanie *Trójkolorowe drzewa binarne*. W dniach konkursowych zawodnicy rozwiązywali zadania: *Magazynier*, *Mapa*, *Ołtarze*, *Pierwotek abstrakcyjny*, i *Woda* oceniane każde maksymalnie po 60 punktów. Decyzją Komitetu Głównego nie przyznawano punktów uznaniowych za oryginalne rozwiązania.

Sprawdzanie przeprowadzono korzystając z programu sprawdzającego, przygotowanego przez Marka Pawlickiego, który umożliwił przeprowadzenie pełnego sprawdzenia zadań danego dnia w obecności zawodnika.

Zastosowano łącznie zestaw 69-ciu testów.

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

- *Magazynier*

	liczba zawodników	czyli
60 pkt.	2	4,4%
59–40 pkt.	4	8,7%
39–20 pkt.	22	47,8%
19–1 pkt.	7	15,2%
0 pkt.	11	23,9%

- *Mapa*

	liczba zawodników	czyli
60 pkt.	8	17,4%
59–40 pkt.	3	6,5%
39–20 pkt.	5	10,9%
19–1 pkt.	26	56,5%
0 pkt.	4	8,7%

- *Ołtarze*

	liczba zawodników	czyli
60 pkt.	1	2,2%
59–40 pkt.	2	4,3%
39–20 pkt.	0	0,0%
19–1 pkt.	3	6,5%
0 pkt.	40	87,0%

- *Pierwotek abstrakcyjny*

	liczba zawodników	czyli
60 pkt.	2	4,4%
59–40 pkt.	3	6,5%
39–20 pkt.	22	47,8%
19–1 pkt.	13	28,3%
0 pkt.	6	13,0%

- *Woda*

	liczba zawodników	czyli
60 pkt.	10	21,7%
59–40 pkt.	8	17,4%
39–20 pkt.	16	34,8%
19–1 pkt.	9	19,6%
0 pkt.	3	6,5%

W sumie za wszystkie 5 zadań:

SUMA	liczba zawodników	czyli
300 pkt.	0	0,0%
299–200 pkt.	2	4,4%
199–100 pkt.	22	47,8%
99–1 pkt.	22	47,8%
0 pkt.	0	0,0%

W dniu 24 kwietnia 1999 roku w gmachu Urzędu Miasta w Sopocie ogłoszono wyniki finału VI Olimpiady Informatycznej 1998/99 i rozdano nagrody ufundowane przez: PROKOM Software S.A., Ogólnopolską Fundację Edukacji Komputerowej, Wydawnictwa Naukowo-Techniczne, Olimpiadę Informatyczną. Poniżej zestawiono listę wszystkich laureatów.

(1) **Andrzej Gąsienica-Samek**, laureat I miejsca, 261 pkt.
(notebook, roczne stypendium — PROKOM, książki, roczny abonament — WNT)*

(2) **Krzysztof Onak**, laureat I miejsca, 206 pkt.
(notebook, roczne stypendium — PROKOM, książki — WNT)

(3) **Michał Nowakiewicz**, laureat I miejsca, 192 pkt.
(notebook, roczne stypendium — PROKOM, książki — WNT)

(4) **Marcin Meinardi**, laureat II miejsca, 182 pkt.
(drukarka laserowa HP — PROKOM, książki — WNT)

(5) **Rafał Rusin**, laureat II miejsca, 174 pkt.
(drukarka laserowa HP — PROKOM, książki — WNT)

(6) **Tomasz Malesiński**, laureat II miejsca, 163 pkt.
(drukarka laserowa HP — PROKOM, książki — WNT)

(7) **Michał Rein**, laureat II miejsca, 158 pkt.
(drukarka laserowa HP — PROKOM, książki — WNT)

(8) **Marcin Poturalski**, laureat II miejsca, 148 pkt.
(drukarka laserowa HP — PROKOM, książki — WNT)

(9) **Daniel Jeliński**, laureat II miejsca, 147 pkt.
(drukarka laserowa HP — PROKOM, książki — WNT)

(10) **Mikołaj Zalewski**, laureat II miejsca, 144 pkt.
(drukarka laserowa HP — Olimpiada Informatyczna, książki — WNT)

(11) **Przemysław Broniek**, laureat III miejsca, 137 pkt.
(drukarka atramentowa HP — PROKOM, książki — WNT)

(12) **Gracjan Polak**, laureat III miejsca, 129 pkt.
(drukarka atramentowa HP — PROKOM, książki — WNT)

(13) **Jarosław Duda**, laureat III miejsca, 127 pkt.
(drukarka atramentowa HP — PROKOM, książki — WNT)

(14) **Grzegorz Herman**, laureat III miejsca, 127 pkt.
(drukarka atramentowa HP — PROKOM, książki — WNT)

(15) **Marcin Wojnarski**, laureat III miejsca, 125 pkt.
(drukarka atramentowa HP — PROKOM, książki — WNT)

(16) **Bartosz Nowierski**, laureat III miejsca, 124 pkt.
(drukarka atramentowa HP — PROKOM, książki — WNT)

(17) **Jacek Kalamarz**, laureat III miejsca, 123 pkt.
(drukarka atramentowa HP — PROKOM, książki — WNT)

(18) **Łukasz Anforowicz**, laureat III miejsca, 122 pkt.
(drukarka atramentowa HP — PROKOM, książki — WNT)

(19) **Grzegorz Andruszkiewicz**, laureat III miejsca, 119 pkt.
(drukarka atramentowa HP — Olimpiada Informatyczna, książki — WNT)

Wszyscy finaliści otrzymali elektroniczne notesy menedżerskie ufundowane przez Ogólnopolską Fundację Edukacji Komputerowej.

Ogłoszono komunikat o powołaniu reprezentacji Polski na Olimpiadę Informatyczą Centralnej Europy do Czech oraz Międzynarodową Olimpiadę Informatyczną, która odbędzie się w Turcji. W skład reprezentacji wchodzą:

* W nawiasach () podano informacje o przyznanej nagrodzie (nagrodach) i jej fundatorze.

(1) **Andrzej Gąsienica-Samek**

(2) **Krzysztof Onak**

(3) **Michał Nowakiewicz**

(4) **Marcin Meinardi**

Zawodnikami rezerwowymi zostali:

(5) **Rafał Rusin**

(6) **Tomasz Malesiński**

Zawodnicy zakwalifikowani do zawodów III stopnia Olimpiady mogą być zwolnieni z egzaminu dojrzałości z przedmiotu informatyka na mocy §9 ust. 1 pkt 2 Zarządzenia Ministra Edukacji Narodowej z dnia 14 września 1992 r. Sekretariat olimpiady wystawił łącznie 46 zaświadczeń o zakwalifikowaniu do zawodów III stopnia celem przedłożenia dyrekcji szkół. Laureaci i finaliści mogą być zwolnieni z egzaminów wstępnych do wielu szkół wyższych na mocy uchwał senatów uczelni podjętych na wniosek MEN, zgodnie z art. 141 ust. 1 ustawy z dnia 12 września 1990 r. o szkolnictwie wyższym (Dz.U. nr 65, poz. 385). Sekretariat wystawił łącznie 19 zaświadczeń o uzyskaniu tytułu laureata i 27 zaświadczeń o uzyskaniu tytułu finalisty VI Olimpiady Informatycznej celem przedłożenia władzom szkół wyższych.

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

Nagrody pieniężne za wkład pracy w przygotowanie finalistów Olimpiady przyznano następującym nauczycielom lub opiekunom naukowym laureatów i finalistów:

- **Jan Gąsienica-Samek** (ICL Poland Sp. z o.o. w Warszawie)
  - Andrzej Gąsienica-Samek (laureat I miejsca)
- **Jan Onak** (Zespół Szkół Mechaniczno-Elektrycznych w Tarnowie)
  - Krzysztof Onak (laureat I miejsca)
- **Wanda Jochemczyk** (XIV L.O. im. S. Staszica w Warszawie)
  - Michał Nowakiewicz (laureat I miejsca)
- **Marek Meinardi** (BIPROSTAL S.A., Kraków)
  - Marcin Meinardi (laureat II miejsca)
- **Bogusław Rusin** (F.A.H. 'Minar' w Rzeszowie)
  - Rafał Rusin (laureat II miejsca)
- **Joanna Ewa Łuszcz** (Zespół Szkół Elektrycznych w Białymstoku)
  - Tomasz Malesiński (laureat II miejsca)
- **Mariusz Blank** (Lucent Technologies w Bydgoszczy)
  - Michał Rein (laureat II miejsca)
  - Kamil Blank (finalista)
- **Anna Kwiatkowska** (IV L.O. w Toruniu)
  - Marcin Poturalski (laureat II miejsca)
- **Piotr Zieliński** (student, Poznań)
  - Daniel Jeliński (laureat II miejsca)
  - Bartosz Nowierski (laureat III miejsca)
- **Andrzej Pfeifer** (I L.O. w Krakowie)
  - Mikołaj Zalewski (laureat II miejsca)

- **Andrzej Dyrek** (Uniwersytet Jagielloński w Krakowie)
    - Przemysław Broniek (laureat III miejsca)
    - Grzegorz Herman (laureat III miejsca)
    - Mateusz Gajdzik (finalista)
    - Paweł Kołodziejczyk (finalista)
- **Barbara Olechniewicz** (L.O. w Dębicy)
    - Jarosław Duda (laureat III miejsca)
- **Tomasz Radko** (Katolickie L.O. Zakonu Ojców Pijarów w Krakowie)
    - Marcin Wojnarski (laureat III miejsca)
- **Krzysztof Bakowski** (Zespół Szkół Elektronicznych w Rzeszowie)
    - Jacek Kalamarz (laureat III miejsca)
- **Iwona Waszkiewicz** (VI L.O. im. J. i J. Śniadeckich w Bydgoszczy)
    - Łukasz Anforowicz (laureat III miejsca)
    - Kamil Blank (finalista)
- **Paweł Guraj** (Warszawa)
    - Grzegorz Andruszkiewicz (laureat III miejsca)
- **Jakub Bałamut** (student AGH w Krakowie)
    - Jerzy Bałamut (finalista)
- **Tamara Biniecka** (Szkoła Średnia w Pabianicach)
    - Michał Malinowski (finalista)
- **Antoni Drożdż** (ANBUD - Zakład Budowlany we Wrocławiu)
    - Grzegorz Drożdż (finalista)
- **Zofia Fic** (Gdańsk)
    - Paweł Fic (finalista)
- **Dorota Jurdzińska** (II L.O. im. Piastów Śląskich we Wrocławiu)
    - Łukasz Szkup (finalista)
- **Bartosz Klin** (XXVII L.O. im. T. Czackiego w Warszawie)
    - Stanisław Findeisen (finalista)
    - Adam Koprowski (finalista)
    - Dymitr Pszenicyn (finalista)
- **Stanisław Kwaśniewicz** (II L.O. im. A. Frycza Modrzewskiego we Włodawie)
    - Tomasz Pylak (finalista)
- **Maria Modelska** (Zespół Szkół Mechaniczno-Elektrycznych we Wrocławiu)
    - Maciej Modelski (finalista)
- **Karol Modzelewski** (Politechnika Radomska / VI L.O. w Radomiu)
    - Arkadiusz Paterek (finalista)

- **Marek Noworyta** (Centrozap w Katowicach)
  - Filip Noworyta (finalista)
- **Józef Paradowski** (PROXAN we Wrocławiu)
  - Mariusz Paradowski (finalista)
- **Henryk Pawłowski** (IV Liceum Ogólnokształcące w Toruniu)
  - Łukasz Kamiński (finalista)
- **Ryszard Pieniążek** (Liceum Ogólnokształcące w Brzozowie)
  - Paweł Dąbrowski (finalista)
- **Teresa Pikuta-Byrka** (Liceum Prawno-Ekonomiczne '`Sigma`' we Wrocławiu)
  - Jarosław Byrka (finalista)
- **Jan Przewoźnik** („Integracja" w Gorzowie Wlkp.)
  - Jakub Przewoźnik (finalista)
- **Małgorzata Rostkowska** (XIV L.O. im. S. Staszica w Warszawie)
  - Tomasz Helbing (finalista)
  - Wojciech Jaworski (finalista)
- **Joanna Śmierzchalska** (III L.O. im. Marynarki Wojennej RP w Gdyni)
  - Filip Łukasik (finalista)
- **Piotr Terlecki** (II L.O. w Warszawie)
  - Paweł Terlecki (finalista)
- **Marian Wnuk** (Energopol Warszawa S.A. w Warszawie)
  - Witold Wnuk (finalista)
- **Rafał Wysocki** (XXVII Liceum Ogólnokształcące im. T. Czackiego w Warszawie)
  - Stanisław Findeisen (finalista)

Wszystkim laureatom i finalistom wysłano przesyłki zawierające dyskietki z ich rozwiązaniami oraz testami, na podstawie których oceniono ich prace.

Warszawa, dn. 7 czerwca 1999 roku

Regulamin Olimpiady Informatycznej

§1 WSTĘP

Olimpiada Informatyczna jest olimpiadą przedmiotową powołaną przez Instytut Informatyki Uniwersytetu Wrocławskiego, który jest organizatorem Olimpiady zgodnie z zarządzeniem nr 28 Ministra Edukacji Narodowej z dnia 14 września 1992 roku. W organizacji Olimpiady Instytut będzie współdziałał ze środowiskami akademickimi, zawodowymi i oświatowymi działającymi w sprawach edukacji informatycznej.

§2 CELE OLIMPIADY INFORMATYCZNEJ

(1) Stworzenie motywacji dla zainteresowania nauczycieli i uczniów nowymi metodami informatyki.

(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, Olimpiadę Informatyczną Centralnej Europy i inne międzynarodowe zawody informatyczne.

§3 ORGANIZACJA OLIMPIADY

(1) Olimpiadę przeprowadza Komitet Główny Olimpiady Informatycznej.

(2) Olimpiada Informatyczna jest trójstopniowa.

(3) W Olimpiadzie Informatycznej mogą brać indywidualnie udział uczniowie wszystkich typów szkół średnich dla młodzieży (z wyjątkiem szkół policealnych).

(4) W Olimpiadzie mogą również uczestniczyć — za zgodą Komitetu Głównego — uczniowie szkół podstawowych.

(5) Zestaw zadań na każdy stopień zawodów ustala Komitet Główny, wybierając je drogą głosowania spośród zgłoszonych projektów.

(6) Integralną częścią rozwiązania zadań zawodów I, II i III stopnia jest program napisany na komputerze zgodnym ze standardem IBM PC, w języku programowania wybranym z listy języków ustalanej przez Komitet Główny corocznie przed rozpoczęciem zawodów i ogłaszanej w „Zasadach organizacji zawodów”.

(7) Zawody I stopnia mają charakter otwarty i polegają na samodzielnym rozwiązywaniu przez uczestnika zadań ustalonych dla tych zawodów oraz nadesłaniu rozwiązań pod adresem Komitetu Głównego Olimpiady Informatycznej w podanym terminie.

(8) Liczbę uczestników kwalifikowanych do zawodów II i III stopnia ustala Komitet Główny i podaje ją w „Zasadach organizacji zawodów" na dany rok szkolny.

(9) O zakwalifikowaniu uczestnika do zawodów kolejnego stopnia decyduje Komitet Główny na podstawie rozwiązań zadań niższego stopnia. Oceny zadań dokonuje Jury powołane przez Komitet i pracujące pod nadzorem przewodniczącego Komitetu i sekretarza naukowego Olimpiady. Zasady oceny ustala Komitet na podstawie propozycji zgłaszanych przez kierownika jury oraz autorów i recenzentów zadań. Wyniki proponowane przez jury podlegają zatwierdzeniu przez Komitet.

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

(11) Zawody II stopnia są przeprowadzane przez komitety okręgowe Olimpiady. Pierwsze sprawdzenie rozwiązań jest dokonywane bezpośrednio po zawodach przez znajdującą się na miejscu część jury. Ostateczną ocenę prac ustala jury w pełnym składzie po powtórnym sprawdzeniu prac.

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

(13) Prace zespołowe, niesamodzielne lub nieczytelne nie będą brane pod uwagę.

§4 KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

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

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

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

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

(5) Komitet Główny powołuje i rozwiązuje komitety okręgowe Olimpiady.

(6) Komitet Główny Olimpiady Informatycznej:

   (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 uczestników wyróżnionych oraz kolejność lokat,

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

   (f) ustala kryteria wyłaniania uczestników uprawnionych do startu w Międzynarodowej Olimpiadzie Informatycznej, Olimpiadzie Informatycznej Centralnej Europy i innych międzynarodowych zawodach informatycznych, publikuje je w „Zasadach organizacji zawodów" oraz ustala ostateczną listę reprezentacji,

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

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

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

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

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

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

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

(14) 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.

(15) 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.

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

§5 KOMITETY OKRĘGOWE

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

(2) Kadencja Komitetu wygasa wraz z kadencją Komitetu Głównego.

(3) Zmiany w składzie Komitetu Okręgowego są dokonywane przez Komitet Główny.

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

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

§6 PRZEBIEG OLIMPIADY

(1) Komitet Główny rozsyła do młodzieżowych szkół średnich oraz kuratoriów oświaty i koordynatorów edukacji informatycznej treść zadań I stopnia wraz z „Zasadami organizacji zawodów”.

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

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

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

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

§7 UPRAWNIENIA I NAGRODY

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

(2) Uczestnicy Olimpiady, którzy zostali zakwalifikowani do zawodów III stopnia są zwolnieni z egzaminu z przygotowania zawodowego z przedmiotu informatyka oraz (zgodnie z zarządzeniem nr 35 Ministra Edukacji Narodowej z dnia 30 listopada 1991 r.) z części ustnej egzaminu dojrzałości z przedmiotu informatyka, jeżeli w klasie, do której uczęszczał zawodnik był realizowany rozszerzony, indywidualnie zatwierdzony przez MEN program nauczania tego przedmiotu.

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

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

(5) Uczestnicy zawodów stopnia II i III otrzymują nagrody rzeczowe.

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

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

§8 FINANSOWANIE OLIMPIADY

(1) Komitet Główny będzie się ubiegał o pozyskanie środków finansowych z budżetu państwa, składając wniosek w tej sprawie do Ministra Edukacji Narodowej i 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) Wyniki zawodów I stopnia Olimpiady są tajne do czasu ustalenia listy uczestników zawodów II stopnia. Wyniki zawodów II stopnia są tajne do czasu ustalenia listy uczestników zawodów III stopnia Olimpiady.

(3) Komitet Główny zatwierdza sprawozdanie z przeprowadzonej Olimpiady w ciągu 2 miesięcy po jej zakończeniu i przedstawia je organizatorowi i Ministerstwu Edukacji Narodowej.

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

Warszawa, 30 września 1998 roku

Zasady organizacji zawodów w roku szkolnym 1998/99

Podstawowym aktem prawnym dotyczącym Olimpiady jest Regulamin Olimpiady Informatycznej, którego pełny tekst znajduje się w kuratoriach oświaty oraz komitetach Olimpiady. Poniższe zasady są uzupełnieniem tego regulaminu, zawierającym szczegółowe postanowienia Komitetu Głównego Olimpiady Informatycznej o jej organizacji w roku szkolnym 1998/99.

§1 WSTĘP

Olimpiada Informatyczna jest olimpiadą przedmiotową powołaną przez Instytut Informatyki Uniwersytetu Wrocławskiego, który jest organizatorem Olimpiady zgodnie z zarządzeniem nr 28 Ministra Edukacji Narodowej z dnia 14 września 1992 roku.

§2 ORGANIZACJA OLIMPIADY

(1) Olimpiadę przeprowadza Komitet Główny Olimpiady Informatycznej.

(2) Olimpiada Informatyczna jest trójstopniowa.

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

(4) Integralną częścią rozwiązania każdego z zadań zawodów I, II i III stopnia jest program napisany na komputerze zgodnym ze standardem IBM PC, w jednym z następujących języków programowania: Pascal, C lub C++.

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

(6) 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.

(7) Do zawodów II stopnia zostanie zakwalifikowanych 160 uczestników, których rozwiązania zadań I stopnia zostaną ocenione najwyżej; do zawodów III stopnia — 40 uczestników, których rozwiązania zadań II stopnia zostaną ocenione najwyżej. Komitet Główny może zmienić podane liczby zakwalifikowanych uczestników o co najwyżej 15%.

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

(9) Terminarz zawodów:

zawody I stopnia — 26.10.98–23.11.98
ogłoszenie wyników: 18.12.98
zawody II stopnia — 9.02.99–11.02.99
ogłoszenie wyników: 8.03.99
zawody III stopnia — 19.04.99–23.04.99

§3 WYMAGANIA DOTYCZĄCE ROZWIĄZAŃ ZADAŃ ZAWODÓW I STOPNIA

(1) Zawody I stopnia polegają na samodzielnym i indywidualnym rozwiązywaniu zadań eliminacyjnych (niekoniecznie wszystkich) i nadesłaniu rozwiązań pocztą, **przesyłką poleconą**, pod adresem:

Olimpiada Informatyczna
Ośrodek Edukacji Informatycznej i Zastosowań Komputerów
ul. Raszyńska 8/10, 02–026 Warszawa
tel. (0–22) 8224019, (0-22) 6685533

w nieprzekraczalnym terminie nadania do 23 listopada 1998 r. (decyduje data stempla pocztowego). Prosimy o zachowanie dowodu nadania przesyłki. Rozwiązania dostarczane w inny sposób nie będą przyjmowane.

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

(3) Rozwiązanie każdego zadania składa się z:

(a) programu (tylko jednego) na dyskietce w postaci źródłowej i skompilowanej,

(b) opisu algorytmu rozwiązania zadania z uzasadnieniem jego poprawności.

(4) Uczestnik przysyła jedną dyskietkę, oznaczoną jego imieniem i nazwiskiem, nadającą się do odczytania na komputerze IBM PC i zawierającą:

- spis zawartości dyskietki w pliku nazwanym SPIS.TRC,
- wszystkie programy w postaci źródłowej i skompilowanej.

Imię i nazwisko uczestnika powinno być podane w komentarzu na początku każdego programu.

(5) Wszystkie nadsyłane teksty powinny być drukowane (lub czytelnie pisane) na kartkach formatu A4. Każda kartka powinna mieć kolejny numer i być opatrzona pełnym imieniem i nazwiskiem autora. Na pierwszej stronie nadsyłanej pracy każdy uczestnik Olimpiady podaje następujące dane:

- imię i nazwisko,
- datę i miejsce urodzenia,
- dokładny adres zamieszkania i ewentualnie numer telefonu,
- nazwę, adres, województwo i numer telefonu szkoły oraz klasę, do której uczęszcza,
- nazwę i numer wersji użytego języka programowania,
- opis konfiguracji komputera, na którym rozwiązywał zadania.

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

Pascal	PAS
C	C
C++	CPP

(7) Opcje kompilatora powinny być częścią tekstu programu. Zaleca się stosowanie opcji standardowych.

§4 UPRAWNIENIA I NAGRODY

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

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

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

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

(5) Komitet Główny ustala składa reprezentacji Polski na XI Międzynarodową Olimpiadę Informatyczną w 1999 roku na podstawie wyników zawodów III stopnia i regulaminu tej Olimpiady Międzynarodowej. Szczegółowe zasady zostaną podane po otrzymaniu formalnego zaproszenia na XI Międzynarodową Olimpiadę Informatyczną.

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

(7) 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 Olimpiady Informatycznej zawiadamia wszystkich uczestników zawodów I i II stopnia o ich wynikach. Każdy uczestnik, który przeszedł do zawodów wyższego stopnia oraz dyrektor jego szkoły otrzymują informację o miejscu i terminie następnych zawodów.

(3) 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 i wyżywienie oraz zwrot kosztów przejazdu.

UWAGA: W materiałach rozsyłanych do szkół, po „Zasadach organizacji zawodów" zostały zamieszczone treści zadań zawodów I stopnia, a po nich — następujące „Wskazówki dla uczestników:"

(1) Przeczytaj uważnie nie tylko tekst zadań, ale i treść „Zasad organizacji zawodów".

(2) Przestrzegaj dokładnie warunków określonych w tekście zadania, w szczególności wszystkich reguł dotyczących nazw plików.

(3) Twój program powinien czytać dane z pliku i zapisywać wyniki do pliku. Nazwy tych plików powinny być takie, jak podano w treści zadania.

(4) Dane testowe są zawsze zapisywane bezbłędnie, zgodnie z warunkami zadania i podaną specyfikacją wejścia. Twój program nie musi tego sprawdzać. Nie przyjmuj żadnych założeń, które nie wynikają z treści zadania.

(5) Staraj się dobrać taką metodę rozwiązania zadania, która jest nie tylko poprawna, ale daje wyniki w jak najkrótszym czasie.

(6) Ocena za rozwiązanie zadania jest określona na podstawie wyników testowania programu i uwzględnia poprawność oraz efektywność metody rozwiązania użytej w programie.

Informacja o X Międzynarodowej Olimpiadzie Informatycznej IOI'98, Setubal, wrzesień 1998

PRZEBIEG I ORGANIZACJA

Dziesiąta Międzynarodowa Olimpiada Informatyczna IOI'98 (International Olympiad in Informatics) odbyła się w Setubal w Portugalii w dniach 5–12 września 1998 roku. Polskę reprezentowali laureaci krajowej V Olimpiady Informatycznej: Tomasz Czajka, Andrzej Gąsienica-Samek, Eryk Kopczyński i Paweł Wolff. Opiekunami polskiej ekipy byli dr Krzysztof Diks (kierownik ekipy) i dr Andrzej Walat (zastępca kierownika ekipy). Pomocą i opieką służyli także prof. dr hab. inż. Stanisław Waligórski, który przebywał w Setubal jako członek Międzynarodowego Komitetu Olimpiad Informatycznych oraz Tadeusz Kuran, nieoceniony kierownik organizacyjny Olimpiady.

Jak co roku, zawodnicy rozwiązywali sześć zadań w dwóch pięciogodzinnych sesjach. Zadania X Olimpiady nie były zbyt trudne w porównaniu do zadań z lat ubiegłych, ale wymagały uwagi. Regulamin oceniania był podobny do regulaminu olimpiady krajowej. Organizatorzy kładli jednak mniejszy nacisk na efektywność rozwiązań.

WYNIKI

Sklasyfikowano 248 zawodników z 65 krajów. Przyznano 22 złote, 40 srebrnych i 59 brązowych medali. Polscy zawodnicy wypadli znakomicie i zdobyli:*

Andrzej Gąsienica-Samek	złoty medal	680 pkt.
Eryk Kopczyński	złoty medal	680 pkt.
Paweł Wolff	złoty medal	650 pkt.
Tomek Czajka	srebrny medal	560 pkt.

Maksymalną liczbę punktów i pierwsze miejsce ex aequo zdobyło czterech zawodników z Chin, Rosji, Rumunii i RPA.

Cztery złote medale zdobyła Słowacja. Trzy złote medale (oprócz Polski) zdobyła tylko ekipa Chin.

* Maksymalna liczba punktów, którą mogli uzyskać zawodnicy, wynosiła 700. Dwóm polskim zawodnikom do zdobycia maksymalnej liczby punktów zabrakło zaliczenia jednego testu.

Zawody I stopnia

opracowania zadań

Grzegorz Jakacki
autor zadania, opracowanie

Marcin Mucha
program wzorcowy

Gra w wielokąty

W grze w wielokąty uczestniczy dwóch graczy. Rekwizytem jest wielokąt wypukły o n wierzchołkach podzielony przez $n-3$ przekątne na $n-2$ trójkąty. Żadne dwie z tych przekątnych nie przecinają się poza wierzchołkami wielokąta. Jeden z trójkątów jest czarny, a pozostałe — białe. Gracze na przemian odcinają od wielokąta po jednym trójkącie, za każdym razem przecinając wielokąt wzdłuż jednej z danych przekątnych. Gracz, który odetnie czarny trójkąt, wygrywa.

PRZYPOMNIENIE: Wielokąt jest wypukły, *jeśli odcinek łączący dowolne dwa jego punkty jest całkowicie zawarty w wielokącie.*

Zadanie

Napisz program, który:

- *czyta z pliku tekstowego* `GRA.IN` *opis rekwizytu do gry,*
- *sprawdza, czy gracz rozpoczynający grę ma strategię wygrywającą,*
- *zapisuje wynik do pliku* `GRA.OUT`*.*

Wejście

Pierwszy wiersz pliku wejściowego `GRA.IN` *zawiera liczbę naturalną n, $4 \leqslant n \leqslant 50000$. Jest to liczba wierzchołków wielokąta. Wierzchołki wielokąta są ponumerowane kolejnymi liczbami od 0 do $n-1$, zgodnie z ruchem wskazówek zegara.*

Następnych $n-2$ wierszy zawiera opisy trójkątów w wielokącie. W wierszu o numerze $i+1$, $1 \leqslant i \leqslant n-2$, znajdują się trzy liczby naturalne a, b, c, oddzielone pojedynczymi odstępami. Są to numery wierzchołków i-tego trójkąta. Pierwszy trójkąt w ciągu jest czarny.

Wyjście

Plik wyjściowy `GRA.OUT` *powinien składać się z jednego wiersza zawierającego jedno słowo:*

- `TAK`*, jeśli gracz rozpoczynający grę ma strategię wygrywającą,*
- `NIE`*, jeśli nie ma.*

Przykład

Dla pliku wejściowego `GRA.IN`*:*

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

poprawną odpowiedzią jest plik tekstowy `GRA.OUT`*:*

```
TAK
```

ROZWIĄZANIE

Zadanie było jednym z najprostszych zadań prezentowanych kiedykolwiek na Olimpiadzie Informatycznej. Już pobieżna analiza przedstawionej gry pokazuje, że:

- jeżeli czarny trójkąt można odciąć w jednym ruchu, to wygrywa rozpoczynający grę,

- w przeciwnym przypadku istnienie strategii wygrywającej dla rozpoczynającego grę zależy wyłącznie od parzystości liczby wierzchołków rekwizytu. Strategia wygrywająca polega na unikaniu odcinania sąsiadów czarnego trójkąta.

Z uwagi na fakt, że wierzchołki wielokąta są ponumerowane kolejno, sprawdzenie, czy czarny trójkąt może zostać odcięty w pierwszym ruchu sprowadza się do ustalenia, czy jego wierzchołkami są trzy kolejne wierzchołki wielokąta. (UWAGA: należy pamiętać, że po wierzchołku nr $n-1$ następuje wierzchołek nr 0, zatem wierzchołki $n-1, 0, 1$ też uważamy za kolejne. Pamiętajmy też, że specyfikacja zadania nie określa porządku, w jakim podane są numery wierzchołków trójkąta w pliku wejściowym.)

Zauważmy, że znalezienie rozwiązania wymaga przeczytania co najwyżej dwóch pierwszych wierszy pliku wejściowego. Programy, które wczytywały cały plik wejściowy były skazane na niepowodzenie przy dużych testach z powodu przekroczenia limitu czasu.

TESTY

Do oceny rozwiązań zawodników użyto 29-ciu testów GRA0.IN–GRA28.IN. By zapobiec przyznawaniu punktów programom, które zawsze generują taki sam plik wyjściowy, testy GRA1.IN–GRA28.IN łączono w dwuelementowe grupy (test GRA$2i$.IN z testem GRA$(2i+1)$.IN). Punkty za każdy test w grupie były przyznawane jedynie wtedy, gdy oba testy zostały zaliczone.

Krzysztof Loryś
autor zadania, opracowanie

Radosław Szklarczyk
program wzorcowy

Monocyfrowe reprezentacje

Niech K będzie cyfrą dziesiętną, różną od 0. K-reprezentacją liczby całkowitej X *nazywamy wyrażenie arytmetyczne o wartości X, w którym wszystkie liczby (w postaci dziesiętnej) składają się wyłącznie z cyfry K. Dopuszczalnymi działaniami w takim wyrażeniu są: dodawanie, odejmowanie, mnożenie i dzielenie. Można też używać nawiasów okrągłych. Dzielenie może wystąpić tylko wtedy, gdy dzielnik dzieli dzielną bez reszty.*

Przykład

Każde z poniższych wyrażeń jest 5-reprezentacją liczby 12:

- $5 + 5 + (5 : 5) + (5 : 5)$
- $(5 + (5)) + 5 : 5 + 5 : 5$
- $55 : 5 + 5 : 5$
- $(55 + 5) : 5$

Długością *K-reprezentacji nazywamy liczbę wystąpień w niej cyfry K. W powyższym przykładzie dwie pierwsze reprezentacje mają długość 6, trzecia — długość 5, a czwarta — długość 4.*

Zadanie

Napisz program, który:

- *z pliku tekstowego* `MON.IN` *wczytuje cyfrę K oraz ciąg liczb,*
- *dla każdej liczby z tego ciągu sprawdza, czy istnieje jej K-reprezentacja nie dłuższa niż 8 i jeśli tak, to znajduje minimalną długość takiej reprezentacji,*
- *zapisuje wyniki do pliku tekstowego* `MON.OUT`.

Wejście

Pierwszy wiersz pliku wejściowego `MON.IN` *zawiera cyfrę K, $K \in \{1, \ldots, 9\}$. Drugi wiersz zawiera liczbę n, $1 \leqslant n \leqslant 10$. W następnych n wierszach znajduje się ciąg liczb naturalnych $a_1, \ldots, a_n$, $1 \leqslant a_i \leqslant 32000$ (dla $i = 1, \ldots, n$), po jednej liczbie w każdym wierszu.*

Wyjście

Plik `MON.OUT` *składa się z n wierszy. Wiersz nr i powinien zawierać:*

- *dokładnie jedną liczbę będącą minimalną długością K-reprezentacji a_i, o ile ta długość nie przekracza 8,*
- *jedno słowo* `NIE`*, jeżeli minimalna długość K-reprezentacji liczby a_i jest większa niż 8.*

Przykład

Dla pliku wejściowego `MON.IN`*:*

```
5
2
12
31168
```

poprawną odpowiedzią jest plik tekstowy `MON.OUT`*:*

```
4
NIE
```

ROZWIĄZANIE

W swej pierwotnej wersji zadanie różniło się od wersji ostatecznej dwoma istotnymi elementami:

- ograniczenie na długość K-reprezentacji było większe i wynosiło 10,
- wynikiem działania programu miało być wyrażenie o minimalnej długości, a nie tylko jego długość.

W niniejszym omówieniu odstąpimy nieco od tradycji prezentowania rozwiązań jedynie ostatecznych wersji zadań i, kierując się jego walorami dydaktycznymi, najpierw omówimy rozwiązanie pierwotnej wersji. Dopiero potem prześledzimy uproszczenia wynikające ze zmian wprowadzonych do treści zadania.

Zadanie w pierwotnej wersji

Zadanie nie jest skomplikowane algorytmicznie i główna trudność leży nie tyle w znalezieniu efektywnego algorytmu (ten daje narzucająca się metoda programowania dynamicznego), co w starannym doborze struktur danych.

Prosta strategia dynamiczna polega na generowaniu K-reprezentacji o coraz większych długościach. Niech $L_K(d)$ będzie zbiorem liczb naturalnych, których minimalna K-reprezentacja ma długość d. Oczywiście $L_K(1)$ zawiera tylko jeden element — liczbę K. Następnie, zakładając, że mamy obliczone zbiory $L_K(d')$, dla $1 \leqslant d' < d$, możemy obliczyć $L_K(d)$ korzystając z prostej własności podanej w poniższym fakcie.

Fakt 1: Liczba s należy do $L_K(d)$ wtedy i tylko wtedy, gdy

- $s = \underbrace{KK\ldots K}_{d}$ lub
- $\forall_{d'<d}\ s \notin L_K(d')$ oraz istnieją l i r takie, że
  - $s = l \odot r$, gdzie $\odot$ jest działaniem ze zbioru $\{+,-,*,/\}$ i
  - $l \in L_K(d_1)$ i $r \in L_K(d_2)$, dla pewnych d_1, d_2 takich, że $d_1 + d_2 = d$.

Struktura danych, w której będziemy pamiętać elementy zbiorów $L_K(d')$, dla $1 \leqslant d' \leqslant d$, powinna umożliwiać szybkie wykonywanie następujących operacji słownikowych:

- sprawdzenie, czy dana liczba s należy już do któregoś z wygenerowanych zbiorów,
- wstawienie s do zbioru;

a także:

- umożliwiać szybki dostęp do wszystkich elementów poszczególnych zbiorów,
- zawierać informacje pozwalające na szybkie odtworzenie minimalnej K-reprezentacji dla wybranej liczby.

Aby racjonalnie zaprojektować taką strukturę powinniśmy na wstępie oszacować liczności zbiorów $L_K(d)$. Niestety nie jest znany prosty wzór określający te wielkości. Jak zwykle w takich przypadkach, warto wyznaczyć je eksperymentalnie (to była jedna z przesłanek umieszczenia zadania w zestawie na pierwszy etap zawodów — w trakcie zawodów II i III stopnia nie ma czasu na przeprowadzanie eksperymentów). Okazuje się, zgodnie z intuicją, iż rozmiar $L_K(d)$ rośnie bardzo szybko (wykładniczo) wraz ze wzrostem

Rys. 1 Wzrost liczności zbiorów $L_K(d)$ dla $K = 8$. Łącznie do zbiorów $L_8(1), \ldots, L_8(9)$ należy 17857 liczb.

d	$\vert L_8(d)\vert$
1	1
2	4
3	13
4	39
5	118
6	355
7	1145
8	3762
9	12420

d (patrz rysunek 1). Dla nas użyteczne jest nie tyle wyznaczenie tempa wzrostu, co konkretnych wartości $|L_K(d)|$.

Wartość $K = 8$ w tabeli z rysunku 1 nie została wybrana przypadkowo. Okazuje się bowiem, że zachodzi następujący fakt:

Fakt 2: Dla $K \neq 8$, wartości $|L_K(d)|$ rosną wolniej niż dla $K = 8$.

Sprawdzenie tego faktu pozostawiamy czytelnikowi. Analizując tabelę możemy wyciągnąć kilka praktycznych wniosków.

Obserwacja 3: Lepiej nie pamiętać $L_K(10)$.

Na rysunku 1 nie podano mocy zbioru $L_8(10)$, ale ekstrapolując można przyjąć rozsądne założenie, że wynosi ona około 40000. Gdybyśmy chcieli zapamiętać wszystkie zbiory od $L_8(1)$ do $L_8(10)$, to struktura powinna być w stanie pomieścić około 60000 elementów, a więc na jeden element moglibyśmy przeznaczyć nie więcej niż 10-11 bajtów. Ponieważ w każdym elemencie struktury musimy przeznaczyć minimum 4 bajty na zapamiętanie liczby (w rzeczywistości liczby z $L_K(10)$ mogą przekraczać wartość pascalowego *MaxLongInt*), pozostanie nam nie więcej niż 7 bajtów — za mało nawet na dwa wskaźniki. W tej sytuacji trudno myśleć o utworzeniu sensownej struktury.

Ponieważ zbiór $L_K(10)$ nie jest nam potrzebny do wyznaczania dalszych zbiorów, możemy zrezygnować z zapamiętywania go. W zamian będziemy musieli jedynie dokonać drobnych modyfikacji programu tak, by przypadek $d = 10$ był odrębnie rozpatrywany.

Obserwacja 4: Czas nie powinien być zasobem krytycznym.

Prosty sposób wyznaczania zbioru $L_K(d)$ polega na rozpatrywaniu wszystkich par liczb, z których jedna należy do $L_K(d_1)$, a druga do $L_K(d_2)$, dla wszystkich d_1 i d_2 sumujących się do d. Takich par jest $\sum_{d_1=1}^{\lfloor d/2 \rfloor} |L_K(d_1)| \cdot |L_K(d-d_1)|$. W najgorszym przypadku ($K = 8$) musimy więc rozważyć 95203 pary, z czego tylko 25081 to pary służące do wyznaczenia $L_8(d)$ (dla $d \leqslant 9$). Wielkości te nie są szokujące, musimy jednak pamiętać, że:

- dla każdej pary będziemy obliczać sumę, różnicę, iloczyn i iloraz,
- dla par rozpatrywanych podczas obliczania $L_K(d)$, dla $d \leqslant 9$, co najmniej trzy z tych wyników (z pominięciem ilorazu w przypadkach, gdy nie jest on liczbą całkowitą) będziemy odszukiwać w strukturze.

Wydaje się, że naszą strukturę danych możemy zbudować w oparciu o drzewa binarnych przeszukiwań, np. mógłby to być las drzew: po jednym drzewie dla każdego zbioru $L_K(d)$. Jeśli w każdym węźle zapamiętamy liczbę typu *longint* oraz dwa wskaźniki na potomków, to pozostanie jeszcze sporo miejsca na informacje, które pozwolą na szybkie odtworzenie K-reprezentacji. Nieco gorzej może być z czasem wyszukiwania. Musimy liczyć się z koniecznością wykonania nawet 100000 wyszukiwań, z czego większość nastąpi w momencie obliczania $L_K(9)$. Wówczas wiele z nich będziemy musieli przeprowadzać we wszystkich dziewięciu drzewach. Co prawda nie są one zbyt duże (patrz tabela na rysunku 1), jednak łączny czas wyszukiwań może okazać się znaczny. Przykładowo, dla $K = 8$ można oczekiwać, że w średnim przypadku jedno wyszukiwanie będzie wymagać wykonania ponad 70 porównań.

Lepszym rozwiązaniem jest zapamiętanie wszystkich zbiorów $L_K(d)$ w jednym drzewie. Aby umożliwić szybki dostęp do elementów poszczególnych zbiorów możemy przeszyć drzewo i utworzyć z jego węzłów 9 list: po jednej na każdy zbiór. To będzie wymagać tylko jednego dodatkowego pola na wskaźnik w każdym węźle. Całe drzewo będzie teraz składać się z mniej niż 18000 węzłów i liczba porównań podczas wyszukiwania znacznie się zmniejszy. Niestety nadal może być zbyt duża. Nawet gdyby drzewo było doskonale zrównoważone, to średnia długość ścieżki wynosiłaby około 14, co przy liczbie wyszukiwań rzędu 100000 nakazuje ostrożność. W rzeczywistości może okazać się, że jest znacznie gorzej. Ponieważ do drzewa dołączamy elementy o coraz dłuższych K-reprezentacjach, należy oczekiwać, że statystycznie będą to coraz większe liczby i wystąpi tendencja do nadmiernego wzrostu prawych poddrzew.

Sytuację można poprawić stosując jeden z rodzajów drzew zrównoważonych (np. drzewa AVL). Proponujemy jednak inne, znacznie szybsze rozwiązanie, oparte na zastosowaniu funkcji haszujących. Główną składową struktury są listy, których początki przechowywane są w tablicy, powiedzmy o nazwie *tab* i rozmiarze m. Wartością $tab[i]$ jest wskaźnik na początek listy, w której pamiętane będą te liczby ze zbiorów $L_K(d)$, dla których funkcja haszująca h przyjmuje wartość i. Załóżmy, że chcemy sprawdzić, czy w strukturze znajduje się liczba s. Najpierw musimy obliczyć wartość $h(s)$, a następnie przeszukać listę, której początek wskazywany jest przez $tab[h(s)]$. Koszt takich operacji jest więc zależny od szybkości obliczania wartości funkcji haszującej oraz od długości powstałych list. Ponieważ oczekiwana długość list jest równa stosunkowi liczby elementów przechowywanych w strukturze do rozmiaru tablicy *tab*, więc w naszym interesie leży, by rozmiar m tablicy był możliwie duży. Jako funkcji haszującej możemy użyć stosunkowo szybko obliczalnej funkcji $h(x) = x \ \mathbf{mod}\ m$. Efektywność takiego rozwiązania najlepiej zilustrować na przykładzie (zob. rysunek 2).

Rys. 2 Rozkład długości list powstałych po umieszczeniu w strukturze wszystkich liczb z $L_8(d)$ (dla $d = 1, \ldots, 9$). Jako m przyjęto 15013.

Długość listy (k)	**Liczba list o długości k**
0	4005
1	5986
2	3520
3	1208
4	263
5	31

Zauważmy, że nie powstała ani jedna lista zawierająca 6 lub więcej elementów. Tak więc każda operacja wyszukiwania elementu w strukturze będzie wymagać nie więcej niż 5 porównań. Jeszcze bardziej zaskakująca dla czytelnika może być średnia liczba porównań przypadająca na jedno wyszukiwanie. Wynosi ona ok. 1,37, wobec 14 w drzewie doskonale zrównoważonym!

Podobnie jak w przypadku drzew, listy można przeszyć, by możliwy był sekwencyjny dostęp do elementów poszczególnych zbiorów $L_K(d)$.

Pozostaje jeszcze do omówienia sposób odtwarzania K-reprezentacji. Wstawiając liczbę do struktury zapamiętujemy informację o tym, na jakiej podstawie dokonujemy wstawienia: czy na podstawie pierwszego punktu faktu 1, czy też na podstawie drugiego. W tym drugim przypadku zapamiętujemy informację o działaniu i jego argumentach (w rzeczywistości wystarcza informacja o działaniu i większym z argumentów). Te informacje pozwalają na odtworzenie K-reprezentacji przy pomocy prostej procedury rekurencyjnej.

Oczywiście można poczynić jeszcze sporo usprawnień przyspieszających działanie algorytmu, jak na przykład przerywanie generowania zbiorów $L_K(d)$ w momencie umieszczenia w strukturze wszystkich liczb a_i (patrz treść zadania). Nie omawiamy ich, pozostawiając to inwencji czytelnika.

Zadanie w ostatecznej wersji

Ograniczenie długości K-reprezentacji do 8 radykalnie upraszcza zadanie. Można sprawdzić, że dla każdego K, liczb mających K-reprezentację o długości nie większej niż 8 jest mniej niż 5500. Dzięki temu dowolna ze struktur rozpatrywanych wyżej pozwala na zadowalająco efektywną implementację algorytmu. Z kolei zrezygnowanie z konieczności wypisywania K-reprezentacji na rzecz wypisywania tylko ich długości pozwala na zastosowanie tricku polegającego na tzw. *preprocessingu*, który jest często używany podczas zawodów programistycznych. Polega on na uprzednim utworzeniu, często kosztem żmudnych obliczeń, struktur danych i umieszczeniu ich jako stałych w programie. Dzięki temu można znacznie przyspieszyć działanie programu, a często nawet sprowadzić je do odczytania odpowiednich informacji ze struktury.

W przypadku naszego zadania bazujemy na fakcie, iż w przedziale [1..32000] nie ma zbyt wielu liczb posiadających K-reprezentacje o długości nie większej od 8. Ilustruje to poniższa tabela.

K	$\sum_{d=1}^{8} \lvert L_K(d) \cap [1..32000] \rvert$
1	418
2	1382
3	2066
4	2763
5	2623
6	2950
7	3210
8	2937
9	2730

Możemy utworzyć 9 tablic, po jednej dla każdego K, których elementami będą rekordy typu:

```
record
  liczba : integer;
  długość : byte
end
```

W rekordach K-tej tablicy, w polu *liczba*, pamiętana jest liczba posiadająca K-reprezentację o długości nie większej od 8, a w polu *długość* minimalna długość takiej reprezentacji. Tablice te możemy umieścić w sekcji stałych programu. Wówczas jego działanie sprowadzi się do odszukania w odpowiedniej tablicy liczb a_i. Jeśli zadbamy, by rekordy w tablicy były uporządkowane rosnąco względem pola *liczba*, to naturalnym będzie zastosowanie wyszukiwania binarnego.

Zauważmy jeszcze na koniec, że na wszystkie tablice potrzebujemy w sumie zaledwie 63237 bajtów (21079 trzybajtowych rekordów).

TESTY

Do oceny rozwiązań użyto 13-tu testów MON0.IN-MON12.IN. Testy MON0.IN i MON1.IN są testami poprawnościowymi. Testy MON2.IN-MON10.IN zawierają pytania o K-reprezentacje dla kolejnych wartości K. I tak MON2.IN zawiera pytanie o 1-reprezentację, MON3.IN — pytania o 2-reprezentacje, ..., MON10.IN — pytania o 9-reprezentacje. Celem tych testów była, poza sprawdzeniem poprawności, ocena efektywności programów. Dlatego liczby, o które pytamy w kolejnych testach z tej grupy, są coraz większe i mają coraz dłuższe minimalne K-reprezentacje. W testach MON11.IN i MON12.IN pytamy o 9-reprezentacje liczb jeszcze większych. Dla niektórych z tych liczb odpowiedź jest negatywna.

Wojciech Guzicki
autor zadania, opracowanie

Marcin Mucha
program wzorcowy

Muszkieterowie

W czasach Ludwika XIII i jego potężnego ministra, kardynała Richelieu, w gospodzie pod Pełnym Antałkiem raczyło się jadłem i winem n muszkieterów. Wina było pod dostatkiem, a że muszkieterowie byli skorzy do zwady, doszło do wielkiej awantury, w której każdy muszkieter obraził wszystkich pozostałych.

Musiało dojść do pojedynków, ale kto miał się z kim pojedynkować i w jakiej kolejności? Uzgodnili (po raz pierwszy od awantury zrobili coś zgodnie), że staną na okręgu w ustalonej kolejności i będą ciągnąć losy. Wylosowany muszkieter toczył pojedynek ze swoim sąsiadem z prawej. Przegrany „odpadał z gry", a dokładniej — służący wynosili jego zwłoki. Sąsiadem wygrywającego stawał się następny muszkieter, stojący za przegranym.

Po latach, gdy historycy czytali wspomnienia zwycięzcy, spostrzegli, że ostateczny wynik zależał w istotny sposób od kolejności wylosowanych pojedynków. Zauważyli bowiem, że dotychczasowe treningi fechtunku pokazywały, kto z kim mógł wygrać pojedynek. Okazało się, że — mówiąc językiem matematycznym — relacja „A wygrywa z B" nie była przechodnia! Mogło się zdarzyć, że muszkieter A walczył lepiej od muszkietera B, B lepiej od C, a C lepiej od A. Oczywiście wśród takiej trójki wybór pierwszego pojedynku decydował o ostatecznym wyniku! Jeśli pierwsi będą walczyć A i B, to ostatecznie wygra C, ale jeśli pierwszymi będą B i C, to ostatecznie wygra A. Historycy, zafascynowani tym odkryciem, postanowili sprawdzić, którzy muszkieterowie mogli przeżyć — od tego zależał przecież los Francji, a wraz z jej losem, los całej cywilizowanej Europy!

Zadanie

Na okręgu stoi n osób, ponumerowanych kolejno liczbami od 1 do n. Osoby te toczą $n-1$ pojedynków. W pierwszej rundzie jedna z tych osób, powiedzmy o numerze i — toczy pojedynek z sąsiadem z prawej, tzn. z osobą o numerze $i+1$ (lub, jeśli $i=n$, to z osobą o numerze 1). Przegrany odpada z gry, sąsiadem zwycięzcy staje się następna w kolejności osoba. Dana jest też tabela możliwych wyników pojedynków w postaci macierzy A. Jeśli $A_{i,j}=1$, to osoba o numerze i zawsze wygrywa z osobą o numerze j. Jeżeli $A_{i,j}=0$, to osoba o numerze i przegrywa z osobą o numerze j. Mówimy, że osoba k może wygrać *zawody, jeżeli istnieje taki ciąg $n-1$ losowań, w wyniku którego będzie ona zwycięzcą ostatniego pojedynku.*

Napisz program, który:

- *wczyta z pliku tekstowego* `MUS.IN` *macierz A,*
- *wyznaczy numery osób, które mogą wygrać zawody,*
- *zapisze je do pliku tekstowego* `MUS.OUT`.

Wejście

W pierwszym wierszu pliku tekstowego `MUS.IN` *dana jest liczba całkowita n spełniająca nierówność $3 \leqslant n \leqslant 100$. W każdym z następnych n wierszy znajduje się jedno słowo złożone z n znaków* `0` *lub* `1`*. Znak stojący na j-tym miejscu w i-tym z tych wierszy oznacza wartość $A_{i,j}$ (tzn. jeśli tym znakiem jest jedynka, to $A_{i,j}=1$, a jeśli zero, to $A_{i,j}=0$). Oczywiście $A_{i,j}=1-A_{j,i}$ dla $i \neq j$. Przyjmujemy, że $A_{i,i}=1$, dla każdego i.*

Wyjście

W pierwszym wierszu pliku wyjściowego `MUS.OUT` *należy zapisać liczbę m tych osób, które mogą wygrać zawody. W następnych m wierszach należy zapisać numery tych osób w kolejności rosnącej, po jednym numerze w wierszu.*

Przykład

Dla pliku wejściowego `MUS.IN`*:*

```
7
1111101
0101100
0111111
```

```
0001101
0000101
1101111
0100001
```

kolejność pojedynków: 1–2, 1–3, 5–6, 7–1, 4–6, 6–1 daje ostateczną wygraną osobie o numerze 6. Można sprawdzić, że jeszcze tylko osoby o numerach 1 i 3 mogą wygrać zawody. Zatem plik wyjściowy `MUS.OUT` *powienien wyglądać następująco:*

```
3
1
3
6
```

ROZWIĄZANIE

Wprowadźmy oznaczenia (i oraz j są tu liczbami naturalnymi z przedziału od 1 do n):

$$i \prec j \Leftrightarrow \text{ osoba } j \text{ wygrywa z osobą } i,$$

oraz

$$i \oplus j = \begin{cases} i+j & \text{gdy } i+j \leqslant n \\ i+j-n & \text{gdy } i+j > n. \end{cases}$$

Zauważmy najpierw, że algorytm polegający na sprawdzeniu wszystkich kolejności losowań będzie działał w czasie rzędu $n!$. Dla większych n taki program nie daje szans na uzyskanie wyniku — czas jego działania będzie nieprawdopodobnie długi. Jednak dla n mniejszych od 10 taki program może dość szybko dać odpowiedź i, gdy brakuje nam lepszego pomysłu, powinniśmy oczywiście taki program napisać.

Rozwiązanie „autorskie" polega na zauważeniu najpierw, że wystarczy rozważyć wszystkie pojedynki rozgrywane na odcinkach, a nie na okręgu, tzn. wśród osób o numerach

$$i, i \oplus 1, i \oplus 2, \ldots, i \oplus (n-1)$$

dla $i = 1, 2, \ldots, n$. W takim przypadku nie uznajemy osób i oraz $i \oplus (n-1)$ za sąsiadów. Teraz dla każdych i, j znajdujemy zbiór $Z_{i,j}$ osób, które mogą wygrać zawody na odcinku

$$i, i \oplus 1, \ldots, j.$$

Na przykład $Z_{2,6}$ to zbiór osób, które mogą wygrać zawody na odcinku $2, 3, 4, 5, 6$, a zbiór $Z_{n-3,2}$ to zbiór osób, które mogą wygrać pojedynek na odcinku $n-3, n-2, n-1, n, 1, 2$. Zauważmy, że szukane rozwiązanie jest sumą zbiorów $Z_{i,i\oplus(n-1)}$, dla $1 = 1, 2, \ldots, n$. Zbiory $Z_{i,j}$ znajdujemy „przekątnymi". Najpierw kładziemy $Z_{i,i} = \{i\}$. Następnie dla kolejnych s znajdujemy zbiory $Z_{i,i\oplus s}$. I tak dla $s = 1$ mamy

$$Z_{i,i\oplus 1} = \begin{cases} \{i\} & \text{gdy } A_{i,i\oplus 1} = 1 \\ \{i \oplus 1\} & \text{gdy } A_{i,i\oplus 1} = 0. \end{cases}$$

Dla większych s korzystamy z wyznaczonych dotychczas zbiorów Z. Można to zrobić dwiema metodami:

(1) Chcemy wyznaczyć $Z_{i,j}$. Wtedy dla dowolnego k znajdującego się na odcinku $i, \ldots, j$ rozpatrujemy zbiory $X = Z_{i,k}$ oraz $Y = Z_{k\oplus 1,j}$. Następnie dla każdego $x \in X$ i $y \in Y$ rozważamy wynik pojedynku x z y i zwycięzcę dołączamy do zbioru $Z_{i,j}$. Ten sposób obliczania zbioru $Z_{i,j}$ wymaga czasu $O(l_{i,j}^3)$, gdzie $l_{i,j}$ jest długością odcinka $i, i \oplus 1, \ldots, j$. Ponieważ musimy wyznaczyć n^2 takich zbiorów, więc całkowita złożoność tego algorytmu wyniesie $O(n^5)$.

(2) Dla każdego a z odcinka $i, \ldots, j$ sprawdzamy, czy a należy do $Z_{i,j}$. Dla $a = i$ korzystamy z następującego prostego faktu:

   ◦ $a \in Z_{i,j}$ wtedy i tylko wtedy, gdy istnieje b na odcinku $i \oplus 1, \ldots, j$ takie, że $b \prec a$ oraz $b \in Z_{i\oplus 1,j}$.

Dla $a = j$ korzystamy z podobnego faktu. Wreszcie dla pozostałych a korzystamy z kolejnego faktu:

- $a \in Z_{i,j}$ wtedy i tylko wtedy, gdy $a \in Z_{i,a}$ oraz $a \in Z_{a,j}$.

Ten sposób wyznaczania $Z_{i,j}$ wymaga czasu $O(l_{i,j})$ i cały algorytm będzie miał złożoność $O(n^3)$. Opisany algorytm działa w czasie $O(n^3)$ i wymaga dość dużej pamięci: musimy zapamiętać n^2 zbiorów, z których każdy ma co najwyżej 100 elementów. Istnieje inny algorytm, działający również w czasie $O(n^3)$, ale wymagający znacznie mniej pamięci. Jest on praktycznie dużo szybszy od algorytmu opisanego wyżej.

W tym algorytmie również rozpatrujemy pojedynki odbywające się na odcinkach od i do j. W tablicy kwadratowej wymiaru $n \times n$ zapisujemy, czy możliwy jest taki przebieg pojedynków, by muszkieterowie o numerach i oraz j stanęli obok siebie (tzn. taki przebieg, że wszyscy muszkieterowie rozdzielający muszkieterów i oraz j giną). Jeśli taki przebieg pojedynków jest możliwy, to kładziemy $T[i,j] = \mathbf{true}$. Oczywiście $T[i,j]$ i $T[j,i]$ oznaczają co innego: $T[i,j] = \mathbf{true}$ oznacza, że giną wszyscy muszkieterowie z odcinka od i do j; $T[j,i]$ oznacza, że giną wszyscy muszkieterowie z odcinka od j do i (w obu przypadkach bez muszkieterów i oraz j). Tablicę T wypełniamy również „przekątnymi". Najpierw kładziemy $T[i, i \oplus 1] = \mathbf{true}$. Następnie, jeśli j nie jest postaci $i \oplus 1$, to kładziemy $T[i,j] = \mathbf{true}$ wtedy i tylko wtedy, gdy istnieje numer k w odcinku od i do j taki, że $T[i,k] = T[k,j] = \mathbf{true}$ oraz któryś z muszkieterów i lub j potrafi wygrać z muszkieterem k. Po zapełnieniu całej tablicy szukamy zbioru możliwych zwycięzców. Jeśli może zostać tylko dwóch muszkieterów i oraz j (czyli mamy jednocześnie $T[i,j] = T[j,i] = \mathbf{true}$), to zwycięzca pojedynku i z j jest dołączany do zbioru zwycięzców.

TESTY

Do oceny rozwiązań zawodników użyto 20-tu testów MUS0.IN–MUS19.IN. Test MUS0.IN jest testem z treści zadania. Testy można podzielić na sześć typów:

(1) test klasyczny: każdy muszkieter może wygrać, muszkieter o większym numerze wygrywa, gdy różnica numerów jest parzysta (zakładamy tu, że liczba n jest nieparzysta);

(2) test rosnący: muszkieter o większym numerze zawsze wygrywa z muszkieterem o mniejszym numerze;

(3) test losowy: tablica wyników pojedynków wygenerowana losowo, każdy może wygrać;

(4) test prawie losowy: tablica wyników wygenerowana losowo, nie każdy może wygrać;

(5) splot dwóch testów: z dwóch testów (jeden dla k muszkieterów, drugi dla l muszkieterów) robimy trzeci (dla $k \cdot l$ muszkieterów) dzieląc muszkieterów na k grup po l muszkieterów. Każdy muszkieter dostaje dwa numery: numer grupy i numer w grupie. Jeśli muszkieterowie i oraz j mają ten sam numer grupy, to wynik jest taki sam, jak wynik walki w teście drugim muszkieterów o takich samych numerach, co numery i i j w tej grupie. Jeśli mają inne numery grup, to wynik jest taki, jak wynik walki muszkieterów w teście pierwszym o takich samych numerach, co numery grup muszkieterów i i j.

(6) testy, w których wszyscy muszkieterowie mogą być zwycięzcami, ale jeden wybrany muszkieter może wygrać w jeden tylko sposób (jest tylko jeden ciąg pojedynków, którego zwycięzcą jest ten wybrany muszkieter).

Testy od MUS1.IN do MUS19.IN były wybrane w następujący sposób:

- MUS1.IN — klasyczny dla 3 muszkieterów,
- MUS2.IN — rosnący dla 3 muszkieterów,
- MUS3.IN — klasyczny dla 39 muszkieterów,
- MUS4.IN — splot: rosnący dla 2 muszkieterów z klasycznym dla 3 muszkieterów (3 zwycięzców),
- MUS5.IN — klasyczny dla 75 muszkieterów,
- MUS6.IN — splot: prawie losowy dla 3 muszkieterów z klasycznym dla 3 muszkieterów (3 zwycięzców),

- MUS7.IN — klasyczny dla 99 muszkieterów,
- MUS8.IN — prawie losowy dla 10 muszkieterów (9 zwycięzców),
- MUS9.IN — splot: klasyczny dla 5 muszkieterów z prawie losowym dla 10 muszkieterów (45 zwycięzców),
- MUS10.IN — splot: losowy dla 15 muszkieterów z prawie losowym dla 5 muszkieterów (71 zwycięzców),
- MUS11.IN — splot: klasyczny dla 5 muszkieterów z prawie losowym dla 20 muszkieterów (95 zwycięzców),
- MUS12.IN — splot: prawie losowy dla 9 muszkieterów z losowym dla 10 muszkieterów (80 zwycięzców),
- MUS13.IN — splot: klasyczny dla 10 muszkieterów z rosnącym dla 10 muszkieterów (9 zwycięzców),
- MUS14.IN — jedna permutacja dla 10 muszkieterów,
- MUS15.IN — prawie losowy dla 8 muszkieterów (6 zwycięzców),
- MUS16.IN — jedna permutacja dla 15 muszkieterów,
- MUS17.IN — prawie losowy dla 8 muszkieterów (6 zwycięzców),
- MUS18.IN — jedna permutacja dla 20 muszkieterów,
- MUS19.IN — prawie losowy dla 8 muszkieterów (5 zwycięzców).

SYMULACJE LOSOWE

Warto jeszcze zwrócić uwagę na to, że można próbować wyznaczyć zbiór możliwych zwycięzców za pomocą symulacji losowych. Losujemy wiele możliwych przebiegów gry i za każdym razem znajdujemy zwycięzcę. Symulacja jednej gry wymaga n losowań. Trzeba tylko opracować dobry sposób usuwania pokonanych muszkieterów. W programie MUS-LOS.PAS został zastosowany najprostszy sposób: usuwanie elementu z tablicy. W tym celu wykorzystano typ *string*, w którym procedura *Delete* usuwa dany wyraz bardzo szybko. Można również utworzyć drzewo binarne złożone ze wszystkich muszkieterów i w czasie logarytmicznym znajdować i usuwać pokonanego muszkietera. Czas symulacji przebiegu jednych zawodów jest zatem rzędu $O(n^2)$ lub $O(n \log n)$, można więc dość szybko przeprowadzić nawet kilka tysięcy takich symulacji. Program symulacyjny nie daje zawsze odpowiedzi poprawnej. Jego istotną zaletą jest szybkość oraz to, że uzyskany zbiór zwycięzców jest podzbiorem prawdziwego zbioru zwycięzców. Nie jest możliwe błędne zakwalifikowanie kogoś jako zwycięzcy, co mogłoby się zdarzyć przy innych błędnych algorytmach.

Taki program symulacyjny może dać prawidłową odpowiedź tylko wtedy, gdy każdy muszkieter, który może być zwycięzcą, wygrywa w dostatecznie wielu rozgrywkach (tzn. istnieje dostatecznie wiele ciągów kolejnych losowań, w wyniku których zostaje on zwycięzcą). Chodzi o to, by prawdopodobieństwo wygrania było dość duże. Popatrzmy na kilka przykładów.

W teście MUS8.IN na łączną liczbę 1814400 ciągów losowań (zauważmy, że liczba takich ciągów jest równa $(n!/2)$: losujemy najpierw jednego muszkietera z n, potem jednego z $n-1$ itd.; nie musimy losować, gdy zostanie dwóch muszkieterów) poszczególni muszkieterowie wygrywają następującą liczbę razy:

Nr	**Liczba wygranych**	**Częstość**
1	96178	5,30%
2	95334	5,25%
3	572743	31,57%
4	0	0,00%
5	525663	28,97%
6	74312	4,10%
7	54026	2,98%
8	125756	6,93%
9	194912	10,74%
10	75476	4,16%

Test MUS8.IN wymagał tylko 44 symulacji. Niektóre testy wymagały jednak znacznie większej liczby symulacji. Na przykład test MUS15.IN, w którym muszkieter o numerze 1 wygrywał tylko w 36 przypadkach na łączną liczbę 20160 (tj. tylko w 0,18% przypadków), wymagał już 700 symulacji. Te symulacje zostały wykonane bardzo szybko: 0,11 sekundy na procesorze 386 (33MHz). Podobnie testy MUS17.IN i MUS19.IN wymagały tylko, odpowiednio, 305 i 491 symulacji. Test MUS3.IN wymagał tylko 191 symulacji (0,33 sek.), test MUS5.IN tylko 273 symulacji (1,10 sek.), test MUS7.IN tylko 711 symulacji (3,18 sek.), natomiast test MUS9.IN aż 9124 symulacji (14,01 sek.). Testy MUS14.IN, MUS16.IN i MUS18.IN zostały zaprojektowane specjalnie w ten sposób, by symulacja losowa nie mogła dać poprawnego wyniku. W tych testach jeden z muszkieterów może wygrać tylko w jeden sposób (zachęcam Czytelnika do znalezienia macierzy wyników pojedynków, która by to gwarantowała!). Oczywiście prawdopodobieństwo trafienia akurat na ten jeden sposób jest bardzo małe. Na przykład w teście MUS14.IN dziewięciu zwycięzców zostało znalezionych dopiero po 234119 symulacjach (ok. 55 sek.). W tym teście muszkieter o numerze 1 wygrywa tylko w jednym przypadku (na 1814400 wszystkich przypadków), a muszkieter o numerze 2 w 8 przypadkach. W teście MUS16.IN dziewięciu zwycięzców (na piętnastu) znaleziono dopiero po 514870 symulacjach (ponad 3 minuty). Podobnie zachowywały się niektóre duże testy: MUS10.IN wymagał 136355 symulacji (prawie 6 minut); w testach MUS11.IN i MUS12.IN kilkaset tysięcy symulacji (kilkanaście minut) nie wystarczyło do znalezienia wszystkich zwycięzców.

Taki sposób rozwiązania zadania, mimo iż nie jest w pełni poprawny, powinien być brany pod uwagę. Istnieje wiele problemów, których nie umiemy rozwiązać w inny sposób niż przez symulacje losowe. Istnieją też problemy, w których rozwiązanie znajdujemy w sposób losowy, a o poprawności znalezionego losowania możemy się przekonać w sposób deterministyczny. Przykładem takiego zadania, mającego duże znaczenie praktyczne, jest poszukiwanie tzw. niereszt kwadratowych (zachęcam Czytelnika do dowiedzenia się z podręcznika teorii liczb, co to są niereszty kwadratowe!). Przykładem bliższym nam może być zadanie „Jedynki i zera” z II Olimpiady Informatycznej. Chodziło w nim o to, by dla danej liczby n znaleźć jej wielokrotność mającą w zapisie dziesiętnym co najwyżej 100 cyfr: zer i jedynek. Jednym ze sposobów rozwiązania jest generowanie losowe liczb 100-cyfrowych składających się tylko z zer i jedynek i dzielenie ich przez n. Z chwilą, gdy uzyskamy resztę 0, kończymy dzielenie, jeśli nie uzyskamy zera, to losujemy następną liczbę (warto rozważyć oddzielnie przypadek liczb składających się z samych jedynek). Okazuje się, że na ogół wystarczyło wygenerować stosunkowo niewiele takich liczb (najwyżej ok. 3000). Program tak napisany działał szybko i dawał poprawną odpowiedź, gdy odpowiednia wielokrotność została znaleziona. Trudnym problemem jest jednak ustalenie maksymalnej liczby losowań.

Powróćmy jeszcze na chwilę do zadania o muszkieterach. Kusząca jest hipoteza:

$$\text{jeśli } a \text{ może wygrać zawody oraz } a \prec b, \text{ to } b \text{ też może wygrać zawody,}$$

która mogłaby znacznie przyspieszyć wyznaczenie szukanego zbioru za pomocą symulacji. Istnieje jednak kontrprzykład na tę hipotezę i zachęcam Czytelnika do znalezienia go.

Bogdan S. Chlebus
autor zadania, opracowanie

Marek Pawlicki
program wzorcowy

Puste prostopadłościany

Prostopadłościan nazwiemy regularnym, *gdy:*

- *jednym z jego wierzchołków jest punkt o współrzędnych (0, 0, 0),*
- *krawędzie wychodzące z tego wierzchołka leżą na dodatnich półosiach układu współrzędnych,*
- *krawędzie te mają długości nie większe niż 10^6.*

Dany jest zbiór A punktów przestrzeni, których wszystkie współrzędne są całkowite i należą do przedziału $[1..10^6]$. Szukamy prostopadłościanu regularnego o maksymalnej objętości, który w swoim wnętrzu nie zawiera żadnego punktu ze zbioru A. Punkt należy do wnętrza prostopadłościanu *jeżeli jest punktem prostopadłościanu, ale nie jego ściany.*

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `PUS.IN` *współrzędne punktów ze zbioru A,*
- *wyznaczy jeden z prostopadłościanów regularnych o maksymalnej objętości, nie zawierający w swoim wnętrzu punktów ze zbioru A,*
- *zapisze wynik do pliku tekstowego* `PUS.OUT`*.*

Wejście

W pierwszym wierszu pliku wejściowego `PUS.IN` *znajduje się jedna całkowita, nieujemna liczba n, $n \leqslant 5000$, będąca liczbą elementów zbioru A.*

W kolejnych n wierszach pliku `PUS.IN` *znajdują się trójki liczb całkowitych z przedziału $[1..10^6]$ będące współrzędnymi (odpowiednio x, y i z) punktów ze zbioru A. Liczby w wierszu pooddzielane są pojedynczymi odstępami.*

Wyjście

W jedynym wierszu pliku wyjściowego `PUS.OUT` *powinny znaleźć się trzy liczby całkowite oddzielone pojedynczymi odstępami, będące współrzędnymi (odpowiednio x, y i z) tego wierzchołka znalezionego prostopadłościanu regularnego, który ma wszystkie współrzędne dodatnie.*

Przykład

Dla pliku wejściowego `PUS.IN`*:*

```
4
3 3 300000
2 200000 5
90000 3 2000
2 2 1000
```

poprawną odpowiedzią jest plik tekstowy `PUS.OUT`*:*

```
1000000 200000 1000
```

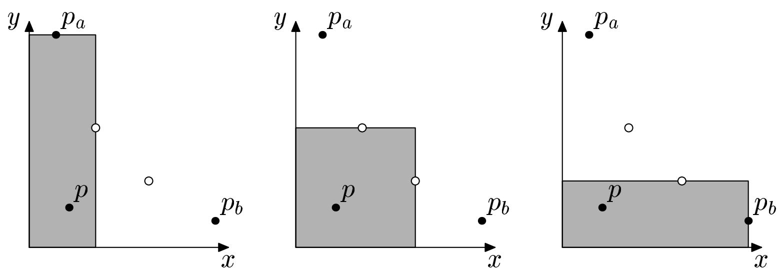
ROZWIĄZANIE

Regularny prostopadłościan, którego wnętrze nie zawiera punktów ze zbioru A, nazywamy *pustym*. Pusty prostopadłościan jest *maksymalny*, gdy nie jest właściwie zawarty w żadnym innym regularnym pustym prostopadłościanie. W trakcie obliczeń, na zmiennej *maxobj* pamiętamy największą objętość maksymalnego pustego prostopadłościanu spośród rozpatrywanych dotychczas, a jego wierzchołek o wszystkich współrzędnych różnych od 0 przechowujemy na zmiennej *pkt*. Uaktualnianie tych zmiennych polega na porównaniu wartości *maxobj* z objętością rozpatrywanego właśnie prostopadłościanu, i ewentualnym przypisaniu im nowych wartości, gdy badany prostopadłościan ma większą objętość. W trakcie wykonywania algorytmu będziemy odwoływali się do pewnego zbioru punktów $B \subseteq A$, który nazywamy *brzegiem*. Punkty należące do brzegu pamiętamy w porządku rosnącym względem współrzędnej x i jednocześnie w porządku malejącym względem współrzędnej y. Niech $p_1, \ldots, p_n$ będą punktami zbioru A uporządkowanymi niemalejąco względem współrzędnej z. Dla punktu p_i, niech p_i^x, p_i^y oraz p_i^z będą jego współrzędnymi.

Punkty A przeglądamy kolejno, zaczynając od p_1. Najpierw ustalamy brzeg na $B = \{p_1\}$. Maksymalny pusty prostopadłościan o podstawie $[0..10^6] \times [0..10^6]$ ma wysokość p_1^z.* Zapamiętujemy objętość tego prostopadłościanu na zmiennej *maxobj*, a jego wierzchołek o wszystkich współrzędnych różnych od 0 na zmiennej *pkt*. Wyższe puste prostopadłościany muszą mieć albo współrzędne x wnętrza nie większe od p_1^x albo współrzędne y wnętrza nie większe od p_1^y. Rozważmy punkt p_2. Jeżeli $p_2^x > p_1^x$ oraz $p_2^y > p_1^y$, to punkt p_2 nie może leżeć na ścianie żadnego pustego prostopadłościanu o większej objętości — zatem nie ma wpływu na rozwiązanie. Jeżeli $p_2^x < p_1^x$ oraz $p_2^y < p_1^y$ to punkt p_2 zastępuje p_1 w tym sensie, że każdy prostopadłościan o wysokości większej niż p_2^z musi mieć albo współrzędne x wnętrza nie większe od p_2^x, albo współrzędne y wnętrza nie większe od p_2^y. Kładziemy zatem $B = \{p_2\}$. Jednocześnie obliczamy objętość dwóch maksymalnych pustych prostopadłościanów, które mają wysokość p_2^z i dla których p_1 leży na jednej ze ścian. Jeżeli objętość któregoś z nich jest większa niż wartość *maxobj*, to przypisujemy nowe wartości *maxobj* i *pkt*. Jeżeli $p_2^x \leqslant p_1^x$ oraz $p_2^y \geqslant p_1^y$, to najwyższy pusty prostopadłościan, którego współrzędne x są nie większe od p_1^x ma wysokość p_2^z. Obliczamy objętość tego prostopadłościanu i ewentualnie uaktualniamy wartości zmiennych *maxobj* oraz *pkt*. Aktualizujemy także B — $B := B \cup \{p_2\}$. W przypadku $p_2^x \geqslant p_1^x$ oraz $p_2^y \leqslant p_1^y$ postępujemy podobnie.

Przypuśćmy, że rozważamy kolejny punkt p_{i+1}. Jeżeli istnieje taki punkt $p \in B$, że jednocześnie $p_{i+1}^x > p^x$ oraz $p_{i+1}^y > p^y$, to punkt p_{i+1} nie leży na ścianie żadnego pustego prostopadłościanu o większej objętości, a zatem nie wpływa na rozwiązanie. Przypuśćmy, że takiego punktu p nie ma. Zatem istnieją puste prostopadłościany o wysokości p_{i+1}^z, na których ścianach leżą punkty z B. Niech p_a będzie punktem z B takim, że $p_a^x < p_{i+1}^x$ oraz p_a ma największą współrzędną x spośród punktów w B o tej własności. Podobnie, niech p_b będzie odpowiednim punktem z B takim, że $p_b^y < p_{i+1}^y$ oraz p_b ma największą współrzędną y spośród punktów w B o tej własności. Te punkty znajdujemy przeglądając kolejno wszystkie elementy B. Rozważmy pary (s, t) kolejnych punktów z B począwszy od $s = p_a$, a kończąc na $t = p_b$, i odpowiednie puste prostopadłościany o wysokości p_{i+1}^z, na których ścianach leżą s i t. Obliczamy objętości wszystkich tych prostopadłościanów i jeżeli trzeba, uaktualniamy *maxobj* oraz *pkt*. Potem usuwamy te punkty z B, które są pomiędzy p_a i p_b, a na ich miejsce wstawiamy p_{i+1} (zob. rysunek 1). Jeżeli p_a lub p_b nie istnieje, to znaczy p_{i+1} jest krańcowym punktem w B — postępujemy podobnie.

Rys. 1 Ilustracja działania algorytmu przy rozpatrywaniu punktu $p = p_{i+1}$. Oś OZ jest prostopadła do płaszczyzny rysunku. Usuwane punkty brzegu zaznaczono symbolem ∘. Kolorem szarym zaznaczono podstawy rozpatrywanych prostopadłościanów (wysokości tych prostopadłościanów są równe p^z).



Po rozpatrzeniu ostatniego punktu p_n rozwiązanie będzie na zmiennej *pkt*. Asymptotyczny koszt algorytmu szacujemy przez $O(n^2)$, ponieważ rozpatrując kolejny punkt p_{i+1} potrzebujemy przejrzeć elementy w B w celu znalezienia punktów p_a oraz p_b, a rozmiar tego zbioru możemy oszacować ogólnie tylko przez n.

* Dla prostoty opisu zakładamy, że do zbioru A należy punkt $(1, 1, 10^6)$.

Rozwiązanie wzorcowe nie jest asymptotycznie najszybsze. Można je ulepszyć tak, by otrzymać pesymistyczny koszt $O(n \log n)$. W tym celu przechowujemy brzeg w wierzchołkach drzewa poszukiwań binarnych. Używamy drzewo zrównoważonego, które daje operacje wyszukiwania, usuwania i wstawiania z kosztem $O(\log n)$. Implementacja takich drzew jest czasochłonna, a uzyskane przyspieszenie nie jest znaczące dla rozmiaru danych określonych w specyfikacji zadania.

INNE ROZWIĄZANIA

Istnieją rozwiązania „naiwne", które opierają się na mniej systematycznym przeszukiwaniu zbioru wszystkich pustych prostopadłościanów.

Oto pierwsze z nich. Rozważamy wszystkie możliwe trójki punktów $T \subseteq A$. Każda z nich poprzez maksymalne wartości współrzędnych swoich elementów wyznacza regularny prostopadłościan, na którego ścianach leżą punkty z T. Sprawdzamy, czy ten prostopadłościan jest pusty przeglądając kolejno punkty z A. Podobnie, każda para punktów $P \subseteq A$ i każda współrzędna (na przykład x), wyznacza jednoznacznie prostopadłościan, którego krawędź x ma długość 10^6, a punkty z P leżą na jego ścianach. Sprawdzamy każdy z takich prostopadłościanów. Podobnie, każdy punkt $p \in A$, dla każdej pary współrzędnych (na przykład x i y), wyznacza jednoznacznie prostopadłościan, którego krawędzie x i y mają długość 10^6 i na którego ścianie leży p. Sprawdzamy wszystkie takie prostopadłościany. Złożoność tego algorytmu wynosi $O(n^4)$.

Oto podobne inne rozwiązanie. Rozważamy wszystkie pary punktów $\{p, q\} \subseteq A$. Ich rzuty na płaszczyznę XY wyznaczają boki prostokąta $R_{p,q}$, którego dwie krawędzie leżą na osiach X i Y a pozostałe zawierają punkty p i q. Przeglądamy wszystkie punkty z A i znajdujemy taki r, który ma minimalną wysokość spośród tych, których rzuty należą do $R_{p,q}$. Sprawdzamy pusty prostopadłościan, na którego ścianach leżą punkty p, q, r. Czas pracy takiego algorytmu wynosi $O(n^3)$.

TESTY

Do oceny rozwiązań zawodników użyto 9-ciu testów PUS0.IN–PUS8.IN. Test PUS0.IN był testem z treści zadania. Połowa z testów była wykonywana na małych zbiorach danych i sprawdzała poprawność rozwiązania. Pozostałe testy sprawdzały czas pracy rozwiązania na dużych danych losowych. Algorytm o koszcie $O(n^4)$ zaliczał test PUS5.IN, ale nie zaliczał testu PUS6.IN. Algorytm o koszcie $O(n^3)$ zaliczał test PUS6.IN, ale nie zaliczał testu PUS7.IN. Algorytm wzorcowy i algorytm o koszcie $O(n \log n)$ przechodziły wszystkie testy.

Zawody II stopnia

opracowania zadań

Marcin Sawicki
autor zadania, opracowanie

Marcin Mucha
program wzorcowy

Bitmapa

Dana jest prostokątna bitmapa o rozmiarach $n \times m$. Każdy piksel bitmapy jest albo biały, albo czarny, przy czym co najmniej jeden jest biały. Piksel w i-tym wierszu i j-tej kolumnie bitmapy nazywamy pikselem (i,j). Odległość dwóch pikseli $p_1 = (i_1, j_1)$ oraz $p_2 = (i_2, j_2)$ określamy jako

$$d(p_1, p_2) = |i_1 - i_2| + |j_1 - j_2|.$$

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego* `BIT.IN` *opis bitmapy,*
- *dla każdego piksela oblicza odległość do najbliższego piksela białego,*
- *zapisuje wyniki w pliku tekstowym* `BIT.OUT`*.*

Wejście

W pierwszym wierszu pliku tekstowego `BIT.IN` *znajduje się para liczb całkowitych n, m oddzielonych pojedynczym odstępem, $1 \leqslant n \leqslant 182$, $1 \leqslant m \leqslant 182$.*

W każdym z kolejnych n wierszy pliku `BIT.IN` *zapisano dokładnie jedno słowo zero-jedynkowe o długości m — opis jednego wiersza bitmapy. Na pozycji j w wierszu $(i+1)$, $1 \leqslant i \leqslant n$, $1 \leqslant j \leqslant m$, znajduje się '1' wtedy i tylko wtedy, gdy piksel (i,j) jest biały.*

Wyjście

W i-tym wierszu pliku wyjściowego `BIT.OUT`*, $1 \leqslant i \leqslant n$, należy wypisać m liczb całkowitych $f(i,1), \ldots, f(i,m)$ pooddzielanych pojedynczymi odstępami i takich, że $f(i,j)$ jest odległością piksela (i,j) od najbliższego piksela białego.*

Przykład

Dla pliku wejściowego `BIT.IN`*:*

```
3 4
0001
0011
0110
```

poprawną odpowiedzią jest plik tekstowy `BIT.OUT`*:*

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

ROZWIĄZANIE

Zacznijmy od bardziej abstrakcyjnego sformułowania problemu. Niech X będzie dowolnym zbiorem. Jego elementy będą dla nas *punktami*, dlatego też cały X będziemy nazywać *przestrzenią*.

Definicja 1: *Metryka* lub inaczej *odległość* w przestrzeni X jest to funkcja dwu zmiennych $d : X \times X \longrightarrow [0; +\infty)$, która dla dowolnych punktów p, q, r spełnia następujące warunki:

(1) $d(p,q)=0 \iff p=q$

(2) $d(p,q)=d(q,p)$

(3) $d(p,q)+d(q,r)\geqslant d(p,r)$

Pierwsze dwa z tych warunków są dość oczywiste, a trzeci, nazywany *nierównością trójkąta*, oznacza, że gdy idziemy z p do r przez jakiś pośredni punkt q, to nie możemy nic zaoszczędzić w stosunku do bezpośredniej drogi z p do r.

Znanym ze szkoły przykładem metryki jest *odległość euklidesowa* na płaszczyźnie określona dla punktów $p=(x_1,y_1)$ oraz $q=(x_2,y_2)$ wzorem

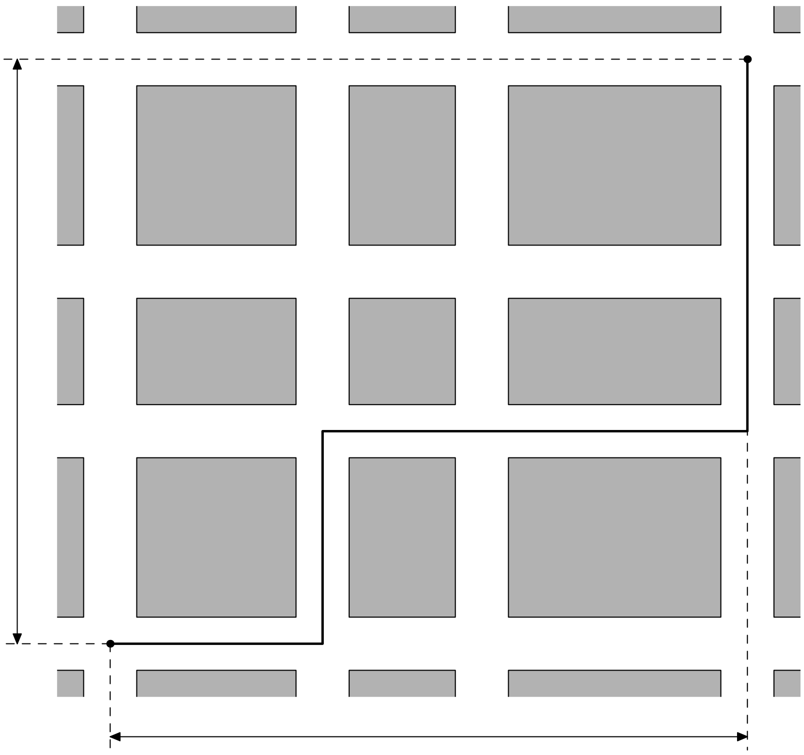
$$d_e(p,q)=\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$$

Także podany w treści zadania wzór określa metrykę, którą nazywa się *metryką miejską* lub *taksówkarską*:

$$d_m(p,q)=|x_1-x_2|+|y_1-y_2|$$

Skąd taka nazwa? Wyobraźmy sobie miasto, w którym ulice biegną tylko w dwu wzajemnnie prostopadłych kierunkach. Jaką odległość musi ponokać samochód, by z jednego skrzyżowania dostać się na inne? Odpowiedź znajdziesz na rysunku 1.

Rys. 1 Metryka miejska.



Jeśli mamy określoną metrykę, możemy też określić *odległość punktu $p\in X$ od zbioru $A\subset X$* wzorem

$$d(p,A)=\inf_{q\in A} d(p,q).$$

Stosujemy to samo co poprzednio oznaczenie $d(\cdot,\cdot)$. Oczywiście nie prowadzi to do nieporozumień, bo $d(p,q)=d(p,\{q\})$.

Nasze zadanie można więc sformułować tak: dla skończonej przestrzeni $X = \{1, 2, \ldots, n\} \times \{1, 2, \ldots, m\}$ z metryką miejską d_m oraz zadanego zbioru punktów białych $B \subset X$ należy, wyznaczyć każdą z odległości $d_m(x, B)$, dla $x \in X$.

Rozwiązanie wzorcowe wykorzystuje specyficzną własność metryki miejskiej w naszej przestrzeni. Jeśli rozpatrzymy graf $G = (X, E)$, którego wierzchołkami będą punkty zbioru X, a krawędzie będą łączyły każdy punkt z jego bezpośrednimi sąsiadami (czterema, chyba że leży on na brzegu bitmapy), to odległość dowolnych dwu punktów $d_m(p, q)$ równa jest ich odległości w rozważanym grafie, to znaczy długości najkrótszej łączącej je ścieżki. Fakt ten jest intuicyjnie oczywisty, a podanie formalnego dowodu może być dla Czytelnika pouczającym ćwiczeniem (dla uproszczenia proponuję rozważać nieskończony zbiór $X = \mathbf{Z} \times \mathbf{Z}$, gdzie $\mathbf{Z}$ oznacza zbiór wszystkich liczb całkowitych).

Do znajdowania najkrótszych ścieżek w grafie możemy z powodzeniem zastosować przeszukiwanie wszerz (BFS). Kolejkę zainicjujemy jednak nie jednym punktem, lecz od razu wszystkimi punktami białymi. Dla każdego z tych punktów szukana odległość d_m wynosi oczywiście 0. Natomiast dla dowolnego innego punktu jest ona o 1 większa, niż odległość jego poprzednika w drzewie (lub raczej lesie) BFS.

Aby udowodnić poprawność naszego algorytmu wystarczy nieznacznie zmodyfikować dowód znanego faktu, że zwyczajny BFS (startujący z jednego punktu) poprawnie oblicza odległości od tego punktu. Zainteresowanych odsyłam do znakomitej książki [12], do rozdziału 23.2 i zawartego w nim twierdzenia 23.4.

Złożoność czasowa naszego algorytmu jest taka sama, jak zwykłego BFS, czyli $O(|V| + |E|)$. U nas $|V| = |X| = nm$, zaś $|E| \leqslant 2|X|$, zatem wynosi ona $O(nm)$. Jest więc liniowa ze względu na rozmiar danych.

INNE ROZWIĄZANIA

Naiwne rozwiązanie mogłoby polegać na znalezieniu dla każdego punktu p z osobna najbliższego mu punktu białego. Można by użyć algorytmu BFS lub po prostu przeglądać kolejno zbiory tych punktów, których odległość d_m od p wynosi odpowiednio $1, 2, 3, \ldots$ Łatwo wyznaczyć te zbiory. Są to brzegi kwadratów o bokach nachylonych pod kątem 45° do osi układu współrzędnych i środkach w punkcie p. Można te zbiory traktować jako okręgi w metryce d_m o środku w punkcie p. Oznaczmy te zbiory przez $C_m(p, R)$ ($R \geqslant 0$ — promień). Koszt takiego rozwiązania w pesymistycznym przypadku wyniósłby $\Omega(n^2 m^2)$ (np. dla jednego białego punktu w rogu planszy).

Istotne ulepszenie powyższego algorytmu opiera się na spostrzeżeniu, że z nierówności trójkąta dla dowolnej metryki d łatwo wynika nierówność

$$|d(p, A) - d(q, A)| \leqslant d(p, q)$$

Oznacza to, że można wykorzystać ideę programowania dynamicznego: jeżeli p i q są sąsiadami, a zatem $d(p, q) = 1$, to do obliczenia $d(q, B)$ możemy wykorzystać wartość $d(p, B)$. Wiemy, że $d(q, B)$ może różnić się od $d(p, B)$ o jeden lub zero. Wystarczy zatem w poszukiwaniu punktów białych przejrzeć najpierw zbiór $C_m(q, R-1)$, a następnie $C_m(q, R)$, gdzie $R = d(p, B)$. Obniża to koszt naszego rozwiązania do $O(nm \min(n, m))$, gdyż jak nietrudno spostrzec, $|C_m(p, R)| \leqslant 2 \min(n, m)$ (nawet dla $R > \min(n, m)$ nasze kwadratowe „okręgi" są zbiorami skończonymi!).

Ostatni z przedstawionych pomysłów jest o tyle ciekawy, że pozwala rozwiązać zadanie także w przypadku metryki euklidesowej d_e, a taka była oryginalna wersja prezentowanego zadania (stąd ograniczenie $1 \leqslant n, m \leqslant 182$ — wtedy kwadrat odległości naroży bitmapy spełnia $d_e^2 = 2 \cdot 181^2 = 65522 < 2^{16}$, czyli mieści się w jednym słowie maszynowym). Przypuśćmy mianowicie, że znamy odległość $R = d_e(p, B)$, a q jest sąsiadem p. Zatem $d_e(p, q) = 1$ i dla wyznaczenia $d_e(q, B)$ wystarczy w poszukiwaniu punktów ze zbioru B przejrzeć te punkty $x \in X$, dla których $R - 1 \leqslant d_e(x, q) < R + 1$ (warunek ten wyznacza na płaszczyźnie pierścień kołowy).

Jak wyznaczyć te punkty efektywnie? Posortujmy punkty przestrzeni X po wielkości $d_e(x, p_0)$, gdzie $p_0 = (1, 1)$ (w praktyce do porównań będziemy używać liczby d_e^2, która jest całkowita). Teraz łatwo znajdziemy te z nich, których odległości od p_0 spełniają określoną nierówność, np. przy pomocy wyszukiwania binarnego.

Przesuwając te punkty o wektor $q - p_0$, uzyskamy jedną z ćwiartek szukanego pierścienia. Pozostałe jego punkty wyznaczymy przez symetrie. Można by poczynić dodatkowe oszczędności zauważając, że taki pierścień kołowy na siatce punktów o współrzędnych całkowitych ma nie tylko symetrię czworokątną, lecz nawet ośmiokątną (cztery osie symetrii).

I tym razem widzimy, że punktów spełniających warunek $R - 1 \leqslant d_e(x, q) < R + 1$ jest na pewno nie więcej niż $K \cdot \min(m, n)$ (proste, choć nieco żmudne rozważania geometryczne wykazują, że można przyjąć np. $K = 8$). W takim razie koszt pesymistyczny głównej części naszego algorytmu nie zmieni się w stosunku do algorytmu dla metryki miejskiej i wyniesie $O(nm \min(n, m))$. Dodatkowy narzut czasowy na posortowanie punktów X na samym początku wynosi $\Theta(nm(\log n + \log m)) = O(nm \min(n, m))$, o ile przyjmiemy $\log n = O(m)$ i $\log m = O(n)$. Możemy zatem powiedzieć, że wszystkie obliczenia wykonamy w czasie $O(nm \min(n, m))$.

Inspiracją dla tego zadania był następujący problem: jak szybko projektować efektowne stereogramy? Do wygenerowania stereogramu potrzebna jest mapa głębi poszczególnych pikseli. W najprostszym przypadku jest ona dwu- lub kilkupoziomowa, jednak utworzone na tej podstawie stereogramy są mało ciekawe. Z drugiej strony, utworzenie mapy głębi na podstawie projektu przestrzennego (np. metodą śledzenia promieni) może być bardzo pracochłonne.

Dobrym pomysłem może być tworzenie stereogramu na podstawie czarno-białego rysunku, na którym białe plamy oznaczają obszary płaskie (morze), natomiast czarne — wypukłości (lądy). Wysokość nad poziomem morza zmienia się płynnie z odległością od morza (niekoniecznie liniowo). Utworzenie mapy głębi na podstawie projektu wymaga właśnie rozwiązania naszego zadania.

Pomysł ten zaprogramowałem w javie. Działający applet można znaleźć na stronie internetowej Olimpiady Informatycznej.

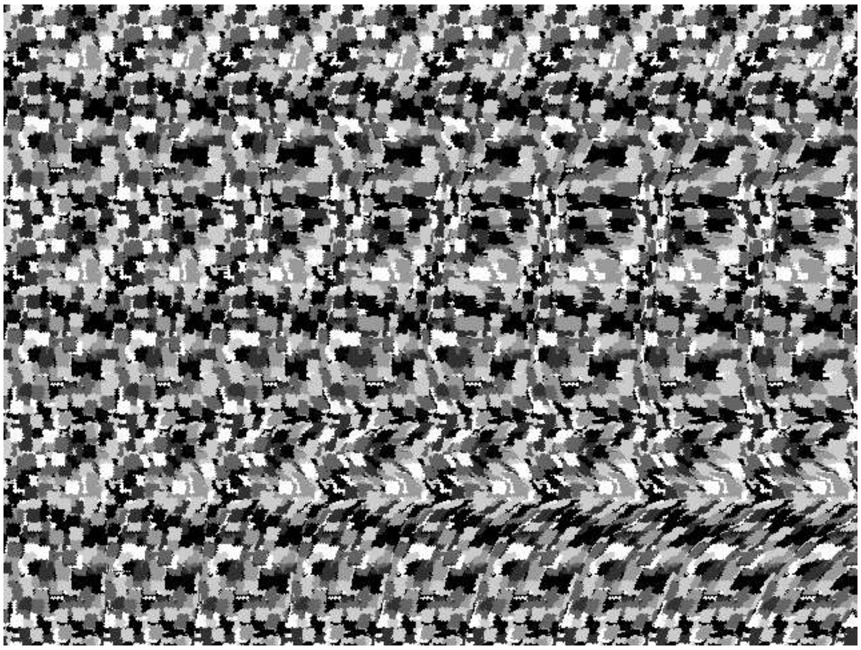
TESTY

Rozwiązania zawodników były testowane przy pomocy 16 testów BIT0.IN–BIT15.IN. Test BIT0.IN był testem z treści zadania. Kolejne cztery testy to bitmapy 1×1, 182×1, 1×182 i 182×182. Każda z nich miała po jednym białym pikselu. Następne osiem testów stanowiły bitmapy maksymalnego rozmiaru, składające się z czarnego pola, otoczonego wąską białą nieregularną obwódką. Są to testy wydajnościowe dla algorytmu opisanego w dyskusji, gdyż koszt wykonania jednego kroku jest tam proporcjonalny do obliczanej odległości, a zatem duży dla dużych czarnych obszarów. Nieregularny kształt obwódki był zbyt trudny dla prostych algorytmów heurystycznych. Ostatnie trzy testy miały również największy możliwy rozmiar i składały się z białych napisów lub prostych rysunków na czarnym tle.

Rys. 2 Projekt stereogramu

Rys. 3 Mapa głębi wygenerowana na podstawie projektu z rysunku 2

Rys. 4 Otrzymany stereogram



Marcin Kubica
autor zadania, opracowanie

Krzysztof Sobusiak
program wzorcowy

Grotołazi

Drużyna grotołazów organizuje trening w największej jaskini Bajtogór. Trening polega na przebyciu drogi od komory leżącej najwyżej, do komory leżącej najniżej. Grotołazi mogą posuwać się tylko w dół, tzn. kolejne odwiedzane komory muszą leżeć coraz niżej. Dodatkowo każdy z nich ma wyjść z najwyższej komory innym korytarzem oraz każdy z nich ma wejść do najniższej komory innym korytarzem. Pozostałe korytarze grotołazi mogą pokonywać wspólnie. Ilu grotołazów może odbyć trening jednocześnie?

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego* `GRO.IN` *opis jaskini,*
- *oblicza największą liczbę grotołazów, którzy mogą odbyć trening jednocześnie,*
- *zapisuje wynik w pliku tekstowym* `GRO.OUT`*.*

Wejście

W pierwszym wierszu pliku tekstowego `GRO.IN` *jest zapisana jedna liczba całkowita n ($2 \leqslant n \leqslant 200$), równa liczbie komór w jaskini. Komory są ponumerowane liczbami od 1 do n w taki sposób, że komora z większym numerem leży niżej od komory z numerem mniejszym. (Najwyższa komora ma numer 1, a najniższa n.) W wierszach o numerach $2, 3, \ldots, n$ są opisane korytarze wychodzące z kolejnych komór $1, 2, \ldots, n-1$, do komór o wyższych numerach (w wierszu o numerze i znajduje się opis korytarzy wychodzących z komory o numerze $i-1$). Każdy z tych wierszy zawiera ciąg nieujemnych liczb całkowitych pooddzielanych pojedynczymi odstępami. Pierwsza liczba w wierszu, m, $0 \leqslant m \leqslant (n-i+1)$, jest liczbą korytarzy prowadzących z komory do komór o wyższych numerach, natomiast kolejne m liczb to numery komór, do których te korytarze prowadzą.*

Wyjście

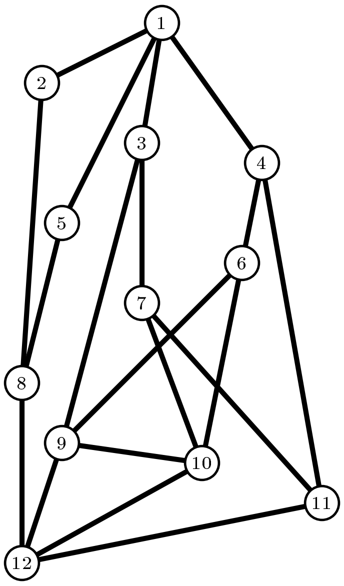
W jedynym wierszu pliku `GRO.OUT` *Twój program powinien zapisać jedną liczbę całkowitą, równą maksymalnej liczbie grotołazów mogących wziąć jednocześnie udział w treningu.*

Przykład

Dla pliku wejściowego `GRO.IN`*:*

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

opisującego następującą jaskinię:



poprawną odpowiedzią jest plik tekstowy `GRO.OUT`*:*
```
3
```

ROZWIĄZANIE

Warunek nałożony w treści zadania na drogi grotołazów mówi, że mają oni wyjść z najwyższej komory różnymi korytarzami i wejść do najniższej komory różnymi korytarzami. Pomiędzy rozejściem się z najwyższej komory i spotkaniem w najniższej komorze ich drogi mogą przebiegać dowolnie. Nazwijmy komory połączone korytarzami bezpośrednio z najwyższą komorą komorami górnymi, a komory połączone korytarzami bezpośrednio z najniższą komorą komorami dolnymi. (Przyjmujemy przy tym, że komora najwyższa nie jest dolną, a komora najniższa nie jest górną, nawet jeśli są połączone bezpośrednim korytarzem.) Każda droga wiodąca z najwyższej do najniższej komory wiedzie albo korytarzem łączącym bezpośrednio najwyższą i najniższą komorę, albo poprzez pewną komorę górną oraz pewną komorę dolną. Zauważmy, że wynik zależy nie tyle od przebiegu korytarzy łączących komory górne z dolnymi, co od tego, które pary komór dolnych i górnych są połączone drogami. Pozwala nam to podzielić obliczenie wyniku na dwie fazy. Najpierw wyznaczamy komory górne i dolne oraz ustalamy pary komór (górna, dolna) połączone drogami. Następnie obliczamy wynik na podstawie rezultatów pierwszej fazy algorytmu (uwzględniając ewentualny korytarz łączący bezpośrednio najwyższą i najniższą komorę).

Spójrzmy na komory i korytarze jak na wierzchołki i krawędzie grafu skierowanego. Pierwsza faza algorytmu sprowadza się do obliczenia domknięcia zwrotno-przechodniego grafu. (Domknięcie takie jest grafem, który zawiera dokładnie takie krawędzie $v \rightarrow w$, że $v = w$, lub z v można dojść do w.) Rozmiar danych ($2 \leqslant n \leqslant 200$) pozwala nam na przechowywanie grafu w postaci tablicy sąsiedztwa.*

```
const MAXN = 200;
var
   T : array [1..MAXN, 1..MAXN] of boolean;
   n : word;
```

Domknięcie zwrotno-przechodnie grafu możemy obliczyć korzystając z wersji algorytmu Floyda-Warshalla (por. [12], p. 26.2, [12], p. 3.5). Dużą zaletą tego algorytmu jest prostota jego implementacji.

```
var
```

* Fragmenty programu przytaczane w tym rozdziale pochodzą z rozwiązania zamieszczonego na załączonej dyskietce.

```
  k, i, j : word;
begin
  for k := 1 to n do
    for i := 1 to n do
      if T[i, k] then
        for j := 1 to n do
          T[i, j] := T[i, j] or T[k, j];
  for i := 1 to n do T[i, i] := true
end
```

W dalszej części interesować nas będzie tylko to, czy z danej komory górnej można dojść do danej komory dolnej. Zbiory komór górnych i dolnych możemy pamiętać w postaci tablic logicznych o wartościach równych **true** dla indeksów należących do zbioru. Zbiory te możemy wyznaczyć następująco:

```
var
  A, B : array [1..MAXN] of boolean;
  i : word;
begin
  for i := 1 to n do begin
    A[i] := T[1, i];
    B[i] := T[i, n]
  end
end
```

Wyobraźmy sobie teraz graf dwudzielny (nieskierowany), którego wierzchołkami są komory górne i dolne, a krawędzie są postaci $v - w$, dla takich komór górnych v i komór dolnych w, że z v można dojść do w. Przez każdą górną i dolną komorę może przejść co najwyżej jeden grotołaz. Tak więc szukana liczba grotołazów jest maksymalnym rozmiarem skojarzenia w naszym grafie dwudzielnym. (Skojarzenie to taki zbiór $\{v_1 - w_1, \ldots, v_k - w_k\}$ krawędzi grafu dwudzielnego, że $v_1, \ldots, v_k, w_1, \ldots, w_k$ są parami różne.) Algorytm znajdowania skojarzenia o maksymalnej liczności w grafie dwudzielnym jest opisany np. w [18], p. 4.3, oraz w [12], p. 27.3.

Potrzebne nam będzie pojęcie *ścieżki rozszerzającej*. Niech S będzie pewnym skojarzeniem. *Ścieżką rozszerzającą* skojarzenie S nazwiemy każdą taką ścieżkę prostą $v_1 - v_2 - \cdots - v_{2l}$ (dla $1 \leqslant l$), że: v_1 i v_{2l} nie są końcami żadnej krawędzi z S, $v_2 - v_3, \ldots, v_{2l-2} - v_{2l-1} \in S$, oraz $v_1 - v_2, \ldots, v_{2l-1} - v_{2l} \notin S$. Inaczej mówiąc, ścieżka rozszerzająca to taka ścieżka prosta łącząca wierzchołki nie pojawiające się w skojarzeniu, której krawędzie na przemian nie należą i należą do skojarzenia. Zauważmy, że

$$S' = S \backslash \{v_2 - v_3, \ldots, v_{2l-2} - v_{2l-1} \in S\} \cup \{v_1 - v_2, \ldots, v_{2l-1} - v_{2l} \notin S\}$$

jest też skojarzeniem i to mającym o jeden element więcej niż S. Okazuje się, że zawsze, gdy skojarzenie nie ma maksymalnej liczności, to istnieje względem niego ścieżka rozszerzająca (por. [18], p. 4.3).

Zauważmy, że pusty zbiór krawędzi jest skojarzeniem. Nasz algorytm polega na ciągłym rozszerzaniu (początkowo pustego) skojarzenia, aż do osiągnięcia skojarzenia o maksymalnej liczności. Do rozwiązania pozostaje jeszcze tylko jeden problem — jak znaleźć ścieżkę rozszerzającą. W zależności od skonstruowanego do tej pory skojarzenia S skierujemy nasz graf dwudzielny w następujący sposób. Niech $v - w$ będzie krawędzią, gdzie v jest komorą górną, a w komorą dolną. Jeżeli $v - w \in S$, to $v \leftarrow w$, a w przeciwnym przypadku $v \rightarrow w$. Zauważmy, że w tak zorientowanym grafie, ścieżki rozszerzające są dokładnie ścieżkami prowadzącymi od pewnej komory górnej, nie występującej w skojarzeniu, do pewnej komory dolnej, także nie występującej w skojarzeniu. Możemy więc szukać takich ścieżek np. za pomocą przeszukiwania w głąb.

Konstruowane skojarzenie możemy reprezentować za pomocą dwóch tablic zawierających dla każdej komory górnej numer skojarzonej z nią komory dolnej i dla każdej komory dolnej numer skojarzonej z nią komory górnej (lub 0, gdy dana komora nie należy do skojarzenia).

```
  var SA, SB : array [1..MAXN] of word;
```

Poniższa funkcja szuka ścieżki rozszerzającej zaczynającej się w komorze o numerze zadanym jako argument (zakładamy, że jest to komora górna nie występująca w skojarzeniu). Korzysta ona z pomocniczej tablicy *OB* służącej do zaznaczania odwiedzonych już wierzchołków. Jej wynikiem jest **true**, gdy taką ścieżkę udało się znaleźć. Równocześnie funkcja ta rozszerza dotychczasowe skojarzenie.

```
var
  OB : array [1..MAXN] of boolean;
function Powieksz(i : word) : boolean;
var j : word;
begin
  for j := i to n do
    if T[i, j] and B[j] and not OB[j] and (SA[i] ≠ j) then begin
      { Krawędź i–j należy do grafu, ale nie do skojarzenia,
        j jest nieodwiedzoną jeszcze komorą dolną. }
      OB[j] := true;
      if (SB[j] = 0) or Powieksz(SB[j]) then begin
        { Ścieżka znaleziona }
        SA[i] := j;
        SB[j] := i;
        Powieksz := true;
        Exit
      end
    end;
  Powieksz := false
end;
```

Zwróć uwagę, że funkcja ta korzysta z „leniwego” obliczania warunków logicznych.

Niech *ns* będzie zmienną przechowującą liczność konstruowanego skojarzenia. Następująca procedura konstruuje skojarzenie o maksymalnej liczności. Na zakończenie, wyliczając wynik na zmiennej *ns*, uwzględnia ona również przypadek, gdy najwyższa i najniższa komora są połączone bezpośrednim korytarzem.

```
var
  ns : word;
procedure Skojarz;
var
  i, j : word;
  pow : boolean;
begin
  ns := 0;
  for i := 1 to n do begin
    SA[i] := 0; SB[i] := 0
  end;
  for j := 1 to n do OB[j] := false;
  repeat
    pow := false;
    for i := 1 to n do
      if A[i] and (SA[i] = 0) then begin
        { Szukamy ścieżki rozszerzającej zaczynającej się w i }
        if Powieksz(i) then begin
          pow := true;
          Inc(ns);
          for j := 1 to n do OB[j] := false
        end
      end
  until not pow;
  if T[1, n] then Inc(ns)
end;
```

Jak łatwo zauważyć, pierwsza faza algorytmu działa w czasie $O(n^3)$. Znalezienie ścieżki rozszerzającej wymaga co najwyżej tyle czasu, co przeszukanie całego grafu, czyli $O(n^2)$. Ścieżek rozszerzających może być znalezionych co najwyżej $n-2$. Tak więc cały algorytm działa w czasie $O(n^3)$.

Należy nadmienić, że opisany algorytm obliczania najliczniejszego skojarzenia nie jest optymalny. Znany jest inny, bardziej skomplikowany algorytm (por. [18], p. 4.3) o czasie działania $O(n^{5/2})$. Jednak przy takim rozmiarze danych jak w zadaniu, taka różnica nie jest istotna, gdyż dominującą częścią programu jest wczytywanie danych z dysku.

Zastanówmy się nad innymi możliwymi rozwiązaniami tego zadania. Możemy je podzielić na takie, w których wyliczane jest domknięcie zwrotno-przechodnie grafu komór i korytarzy, oraz na takie, w których konstruowane są całe drogi grotołazów.

Domknięcie zwrotno-przechodnie grafu możemy wyliczać inaczej niż za pomocą algorytmu Warshalla. Możemy dla każdej komory górnej wyznaczać komory dolne, do których możemy dojść np. za pomocą przeszukiwania w głąb. Nie wpływa to w istotny sposób na czas działania programu.

Jak inaczej możemy znaleźć maksymalne skojarzenie? Jedna z możliwości, to rekurencyjne przeglądanie z nawrotami wszystkich możliwych skojarzeń. Algorytm taki daje poprawne wyniki i stosunkowo łatwo go zaimplementować. Niestety czas jego działania jest wykładniczy. Tak więc czas obliczeń jest akceptowalny tylko dla niewielkich testów.

Zastanówmy się nad programami, które starają się zbudować skojarzenie o maksymalnej liczności w sposób zachłanny. Możliwych jest kilka sposobów wyboru kolejnej krawędzi skojarzenia. Można wybierać „pierwszą wolną” krawędź, losową, lub wybierać krawędź o końcach będących wierzchołkami jak najmniejszego stopnia. Ta ostatnia technika daje najlepsze rezultaty. Niestety zawodzi dla większych testów losowych oraz dla testów odpowiednio „złośliwych”. W przypadku losowego wyboru krawędzi możemy wielokrotnie konstruować losowe skojarzenie i wybierać najbardziej liczne. Taka technika polepsza rezultaty tylko w przypadku niewielkich testów.

Algorytmy konstruujące drogi grotołazów, bez wyliczenia domknięcia grafu, dają generalnie gorsze rezultaty. Możemy rekurencyjnie (z nawrotami) poszukiwać maksymalnej liczby dróg grotołazów. Algorytm taki ma koszt wykładniczy i, jak poprzednio, ma szansę działać w rozsądnym czasie tylko dla małych testów. Stosowanie heurystyk mających na celu ograniczenie liczby przeszukiwanych dróg nie zmienia w istotny sposób sytuacji.

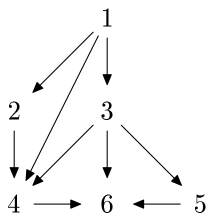
Reasumując, rozwiązania nie wyliczające domknięcia zwrotno-przechodniego grafu i nie korzystające ze ścieżek rozszerzających miały szansę na zaliczenie tylko części testów — zwykle niewielkich.

TESTY

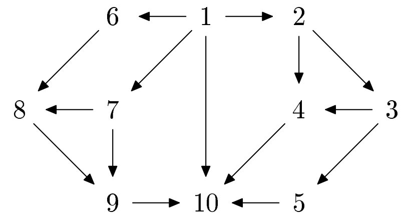
Do sprawdzenia rozwiązań zawodników użyto 11-tu testów GRO0.IN–GRO10.IN. Test GRO0.IN był testem z treści zadania. Testy GRO1.IN–GRO4.IN są niewielkimi testami poprawnościowymi (zob. ilustracje).

Testy GRO5.IN–GRO8.IN są testami losowymi różnej wielkości, przy czym testy GRO5.IN i GRO8.IN, z pominięciem najwyższej i najniższej komory, to grafy dwudzielne. Test GRO9.IN to duży test eliminujący rozwiązania zachłanne. Test GRO10.IN to graf o 200 komorach i wszystkich możliwych korytarzach. Poniższa tabelka przedstawia liczbową charakterystykę testów. Wszystkie testy można znaleźć na załączonej dyskietce.

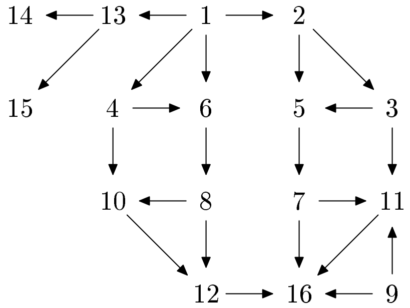
nr testu	l. komór	l. korytarzy	wynik
1	6	9	2
2	10	15	3
3	16	23	2
4	18	43	8
5	100	248	48
6	200	400	43
7	200	498	85
8	200	498	95
9	198	628	98
10	200	19900	199



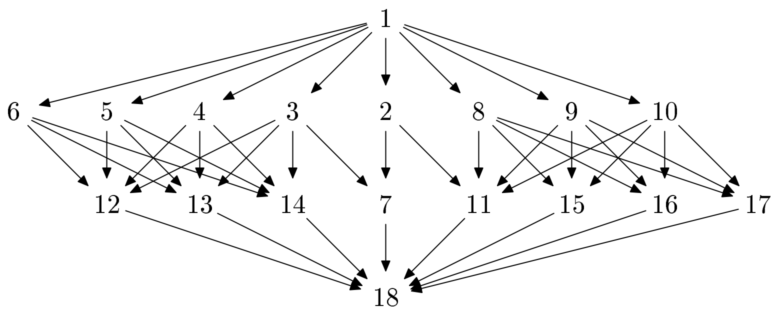
GRO1.IN



GRO2.IN



GRO3.IN



GRO4.IN

Wojciech Rytter
autor zadania, opracowanie

Adam Borowski
program wzorcowy

Lunatyk

Płaski dach budynku ma kształt kwadratu o rozmiarach $3^k \times 3^k$ i bokach równoległych do kierunków północ-południe oraz wschód-zachód. Dach pokryto kwadratowymi kaflami o boku 1, ale jeden kafel został wyrwany i w tym miejscu jest dziura. Kafle na dachu tworzą prostokątną siatkę, wobec tego ich pozycje można określić za pomocą współrzędnych. Kafel położony w południowo-zachodnim rogu ma współrzędne $(1,1)$. Pierwsza współrzędna rośnie przy przemieszczaniu się na wschód, a druga przy przemieszczaniu się na północ.

Lunatyk przemierza dach przemieszczając się w każdym kroku z kafla, na którym aktualnie się znajduje, na kafel sąsiedni od wschodu (E), zachodu (W), południa (S) lub północy (N). Wędrówka lunatyka po dachu zaczyna się zawsze od kafla w rogu południowo-zachodnim, a opisem drogi jest słowo d_k złożone z liter N, S, E, W oznaczających, odpowiednio, krok na północ, na południe, na wschód i na zachód. Dla $k = 1$ opisem drogi lunatyka jest słowo

$$d_1 = EENNWSWN.$$

Dla $k = 2$ opisem drogi lunatyka jest słowo

$$\begin{aligned} d_2 = & NNEESWSEENNEESWSEEEENNWSWNNEENNWSW\text{-} \\ & NNEENNWSWNWWWSSENESSSSWWNENWWSSW\text{-} \\ & WNENWNEENNWSWN. \end{aligned}$$

Spójrz na rysunek przedstawiający drogi lunatyka po dachach o rozmiarach 3×3 i 9×9.

Ogólnie, dla dowolnego $k \geqslant 1$, opisem drogi lunatyka po dachu o rozmiarach $3^{k+1} \times 3^{k+1}$ jest słowo:

$$d_{k+1} = a(d_k)\ E\ a(d_k)\ E\ d_k\ N\ d_k\ N\ d_k\ W\ c(d_k)\ S\ b(d_k)\ W\ b(d_k)\ N\ d_k$$

gdzie funkcje a, b i c oznaczają następujące przemianowania liter określających kierunki:

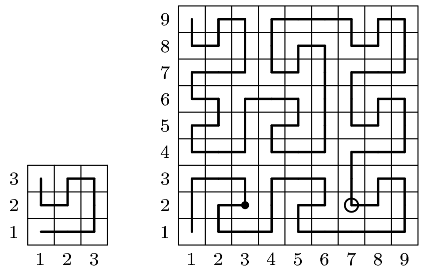
$$\begin{array}{llllll} a: & E \to N & W \to S & N \to E & S \to W \\ b: & E \to S & W \to N & N \to W & S \to E \\ c: & E \to W & W \to E & N \to S & S \to N \end{array}$$

np. $a(SEN) = WNE$, $b(SEN) = ESW$, $c(SEN) = NWS$.

Zaczynamy obserwować lunatyka w momencie, gdy znajduje się na kaflu o współrzędnych (u_1, u_2). Po ilu krokach lunatyk wpadnie do dziury, która znajduje się w kwadracie o współrzędnych (v_1, v_2)?

Przykład

Na rysunku przedstawiono drogi lunatyka po dachu o rozmiarach 3×3 oraz po dachu o rozmiarach 9×9. W drugim przypadku zaznaczono punkt, od którego rozpoczynamy obserwację lunatyka i dziurę w dachu. Lunatyka dzieli od dziury 20 kroków.



Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego* `LUN.IN` *liczbę naturalną k określającą rozmiary dachu ($3^k \times 3^k$), pozycję lunatyka w chwili, gdy rozpoczynamy jego obserwację oraz pozycję dziury,*
- *oblicza liczbę kroków, które dzielą lunatyka od dziury,*
- *zapisuje wynik w pliku tekstowym* `LUN.OUT`.

Wejście

W pierwszym wierszu pliku tekstowego `LUN.IN` *jest zapisana jedna liczba naturalna k, $1 \leqslant k \leqslant 60$, określająca rozmiary dachu ($3^k \times 3^k$).*

W każdym z kolejnych dwóch wierszy pliku `LUN.IN` *są zapisane dwie liczby naturalne x, y oddzielone odstępem, $1 \leqslant x \leqslant 3^k$, $1 \leqslant y \leqslant 3^k$. Liczby w drugim wierszu pliku* `LUN.IN` *są współrzędnymi kafla, na którym stoi lunatyk; liczby w trzecim wierszu pliku* `LUN.IN` *są współrzędnymi dziury.*

Możesz założyć, że dane są tak dobrane, iż po pewnej liczbie kroków lunatyk wpadnie do dziury.

Wyjście

Jedyny wiersz pliku wyjściowego `LUN.OUT` *powinien zawierać liczbę kroków, które dzielą lunatyka od dziury.*

Przykład

Dla pliku wejściowego `LUN.IN`:

```
2
3 2
7 2
```

poprawną odpowiedzią jest plik tekstowy `LUN.OUT`:

```
20
```

ROZWIĄZANIE

Zadanie sprowadza się do znalezienia odległości $DIST(k,x,y)$ od punktu (1,1) do zadanego punktu (x,y) w kwadracie $3^k \times 3^k$. Jeśli policzymy tę odległość dla punktu, od którego rozpoczynamy obserwację lunatyka i punktu położenia dziury, to wystarczy policzyć ich różnicę, aby otrzymać wynik końcowy.

Uwaga. W celu uproszczenia opisu rozwiązania przyjmujemy, że współrzędne są liczone od 0, a nie od 1, jak to jest w treści zadania.

Zadanie ma charakter rekurencyjny. Najistotniejszą cechą jest to, że mamy do czynienia z bardzo dużymi kwadratami i wędrowanie po takich kwadratach „małymi krokami” wymaga zbyt dużego czasu.

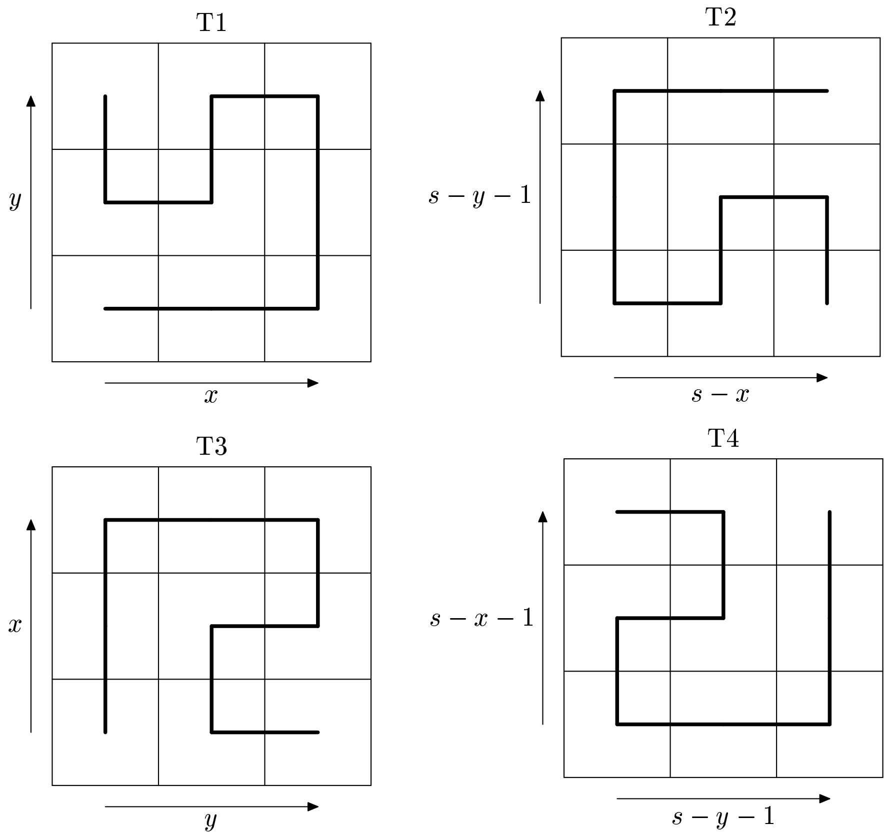
Możemy zaoszczędzić czas „przeskakując” przez całe duże podkwadraty, o ile wiemy, że lunatyk przejdzie w nich wszystkie pola. Tego typu przeskoki są pokazane na rysunku 3.

Kwadrat o rozmiarach $3^k \times 3^k$ nazwiemy kwadratem stopnia k. Składa się on z 9 podkwadratów stopnia $k-1$, które nazwiemy podkwadratami bazowymi. Każdy taki podkwadrat odpowiada parze (i,j) w naturalny sposób, gdzie $0 \leqslant i,j \leqslant 2$. Oznaczmy go przez $SUB(i,j)$. W każdym z tych bazowych podkwadratów lunatyk chodzi tak samo, z dokładnością do odpowiedniej transformacji współrzędnych. Transformacje te są podane na rysunku 1 i zapamiętane w tablicy TR. W tablicy NUM pamiętamy, w jakiej kolejności lunatyk odwiedza podkwadraty bazowe (zob. rysunek 2). Jeśli punkt (x,y) jest w podkwadracie bazowym o współrzędnych (baz_x, baz_y), to lunatyk chodzi tam zmieniając współrzędne zgodnie z $TR(baz\_x, baz\_y)$. Wcześniej możemy przeskoczyć $NUM(baz\_x, baz\_y)$ podkwadratów stopnia $k-1$.

Istnieje prosty sposób na policzenie współrzędnych (baz_x, baz_y) bazowego podkwadratu, w którym jest punkt (x,y), oraz współrzędnych punktu (wew_x, wew_y), będących współrzędnymi punktu (x,y) wewnątrz podkwadratu.

$$\begin{aligned}(baz\_x, baz\_y) &= (x \ div\ 3^{k-1}, y \ div\ 3^{k-1})\\(wew\_x, wew\_y) &= (x \ mod\ 3^{k-1}, y \ mod\ 3^{k-1})\end{aligned}$$

Rys. 1 Transformacje współrzędnych (s oznacza długość boku podkwadratu): $T1 : (x,y) \mapsto (x,y)$, $T2 : (x,y) \mapsto (s-x-1, s-y-1)$, $T3 : (x,y) \mapsto (y,x)$, $T4 : (x,y) \mapsto (s-y-1, s-x-1)$.



Wynika z tego następujący ogólny schemat liczenia odległości $DIST(k,x,y)$.

function $DIST$ $(k,\ x,\ y)$: *integer*;
begin
  if $k = 0$ **then** $DIST := 0$
  else begin
    $(baz\_x,\ baz\_y) := (x$ **div** $3^{k-1},\ y$ **div** $3^{k-1})$;
    $(wew\_x,\ wew\_y) := (x$ **mod** $3^{k-1},\ y$ **mod** $3^{k-1})$;
    oblicz (x', y') jako wynik transformacji $TR(baz\_x, baz\_y)$
    zastosowanej do punktu (wew_x, wew_y);
    $DIST := DIST(k-1, x', y') + NUM(baz\_x, baz\_y) \cdot 3^{k-1}$
  end
end

Zobaczmy, w jaki sposób wykonywane jest obliczanie $DIST(3,x,y)$ dla punktu $(x,y) = (9,10)$.

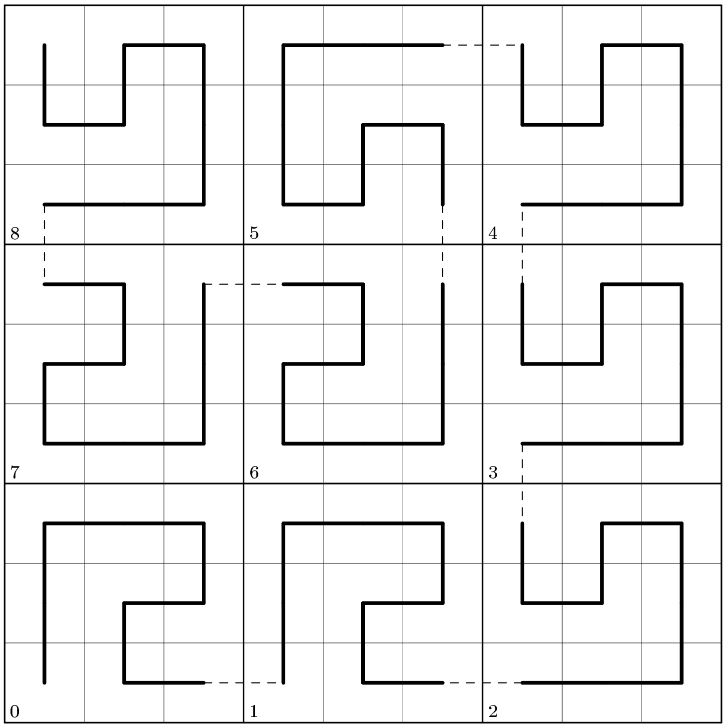
- Obliczamy $(baz\_x,\ baz\_y) := (1,1)$, $(wew\_x,\ wew\_y) := (0,1)$.
- Stosujemy transformację $TR(1,1)$ do $(0,1)$ w podkwadracie o boku $s = 9$ i otrzymujemy punkt $(x', y') = (8-1, 8-0) = (7,8)$. Mamy $NUM(1,1) = 6$. Zatem:

$$DIST(3,9,10) = DIST(2,7,8) + 6 \cdot 9^2$$

- Podobnie obliczamy $DIST(2,7,8) = DIST(1,1,2) + 4 \cdot 9$
- Ostatecznie $DIST(1,1,2) = 5$ i jako wynik otrzymujemy

$$DIST(3,9,10) = 6 \cdot 9^2 + 4 \cdot 9 + 5.$$

Rys. 2 Kolejność odwiedzania podkwadratów bazowych oraz tablice TR i NUM.



Tablica $TR[i, j]$

j \ i	0	1	2
2	T1	T2	T1
1	T4	T4	T1
0	T3	T3	T1

Tablica $NUM[i, j]$

j \ i	0	1	2
2	8	5	4
1	7	6	3
0	0	1	2

Zamiast za każdym razem pracowicie liczyć operacje **div** i **mod** wygodniej jest raz przedstawić x i y w systemie trójkowym; wtedy w każdej iteracji błyskawicznie odczytujemy pozycję podkwadratu bazowego. Tego typu chwyt algorytmiczny zastosowano w programie wzorcowym. Ma to jednak wyłącznie charakter techniczny i nie zmienia istotnie sposobu obliczania $DIST$.

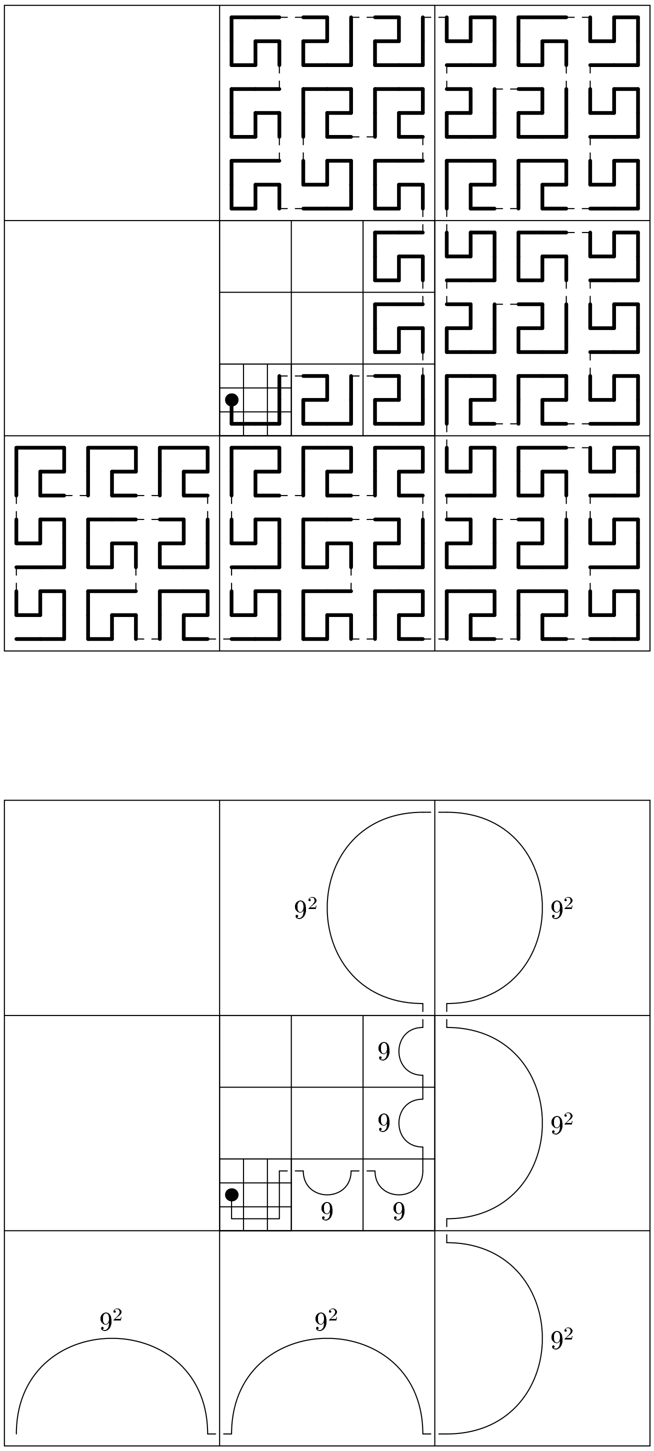
TESTY

Do oceny rozwiązań zawodników użyto 11-tu testów LUN0.IN–LUN10.IN:

- LUN0.IN — $k = 2$, test z treści zadania,
- LUN1.IN — $k = 1$, najmiejszy możliwy rozmiar dachu, lewy dolny i prawy dolny róg planszy to, odpowiednio, początek i koniec obserwowanej drogi lunatyka,

- LUN2.IN, LUN3.IN — testy poprawnościowe,
- LUN4.IN — $k = 30$, test losowy,
- LUN5.IN — $k = 60$, maksymalny rozmiar planszy, lewy dolny i prawy górny róg,
- LUN6.IN — $k = 60$, największy możliwy wynik, lewy dolny i lewy górny róg,
- LUN7.IN–LUN9.IN — testy losowe, duże k,
- LUN10.IN — $k = 6$, dwa sąsiednie pola.

Rys. 3 Symulacja drogi lunatyka do punktu $(x, y) = (9, 10)$ za pomocą „przeskakiwania" przez podkwadraty, które lunatyk przechodzi w całości (przypominamy, że współrzędne liczymy od 0). W tym wypadku „przeskakujemy" 6 podkwadratów stopnia 2 i 4 stopnia 1.



Wojciech Rytter
autor zadania, opracowanie

Adam Borowski
program wzorcowy

Rakiety

Na płaskiej mapie wyznaczono dwa rozłączne, n-elementowe zbiory punktów: R i W. Żadne trzy punkty ze zbioru $R \cup W$ nie są współliniowe. W punktach ze zbioru R rozlokowane są rakiety typu ziemia-ziemia, natomiast w punktach ze zbioru W znajdują się obiekty wroga, które trzeba zniszczyć. Rakiety mogą lecieć jedynie w linii prostej, a tory rakiet nie mogą się przecinać. Chcemy dla każdej rakiety wyznaczyć cel, który powinna zniszczyć.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego* `RAK.IN` *współrzędne punktów ze zbiorów R i W,*
- *wyznacza zbiór n parami nie przecinających się odcinków i takich, że jeden koniec każdego odcinka należy do zbioru R, podczas gdy drugi należy do zbioru W,*
- *zapisuje wynik w pliku tekstowym* `RAK.OUT`.

Wejście

W pierwszym wierszu pliku tekstowego `RAK.IN` *jest zapisana jedna liczba naturalna n, $1 \leqslant n \leqslant 10000$, równa liczności zbiorów R i W.*

W każdym z kolejnych $2n$ wierszy pliku `RAK.IN` *jest zapisana para liczb całkowitych z przedziału $[-10000, 10000]$ oddzielonych pojedynczym odstępem. Są to współrzędne punktu na płaszczyźnie (najpierw współrzędna x, potem współrzędna y). Pierwsze n z tych wierszy zawiera współrzędne punktów ze zbioru R, natomiast ostatnie n wierszy zawiera współrzędne punktów ze zbioru W. W wierszu o numerze $i+1$ znajdują się współrzędne punktu r_i, natomiast w wierszu o numerze $i+n+1$ znajdują się współrzędne punktu w_i, $1 \leqslant i \leqslant n$.*

Wyjście

Plik `RAK.OUT` *powinien składać się z n wierszy. W i-tym wierszu należy zapisać jedną liczbę naturalną k_i taką, że odcinek $\overline{r_i w_{k_i}}$ należy do wyznaczonego zbioru odcinków (innymi słowy, że rakieta z punktu r_i ma zniszczyć obiekt położony w punkcie w_{k_i}).*

Przykład

Dla pliku wejściowego `RAK.IN`*:*

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

poprawną odpowiedzią jest na przykład plik `RAK.OUT`*:*

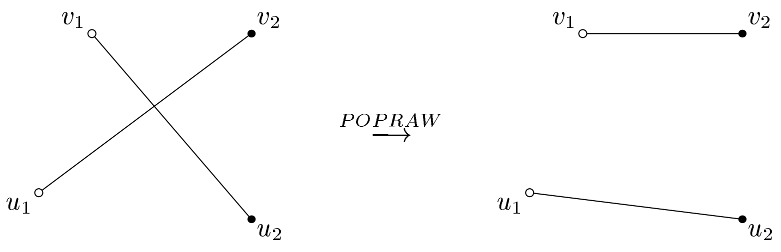
```
2
1
4
3
```

ROZWIĄZANIE

Pokolorujmy punkty z R na czarno, a punkty z W na biało. Przez proste i odcinki rozumiemy dalej tylko proste i odcinki zawierające dwa punkty różnych kolorów. Przez punkty rozumiemy tylko punkty z $R \cup W$. Naszym celem jest znalezienie n nieprzecinających się odcinków.

Jeżeli mamy dwa przecinające się odcinki $\overline{u_1v_1}$, $\overline{u_2v_2}$, to możemy wykonać operację *POPRAW* pokazaną na rysunku 1.

Rys. 1 Operacja $POPRAW(u_1,u_2,v_1,v_2)$.



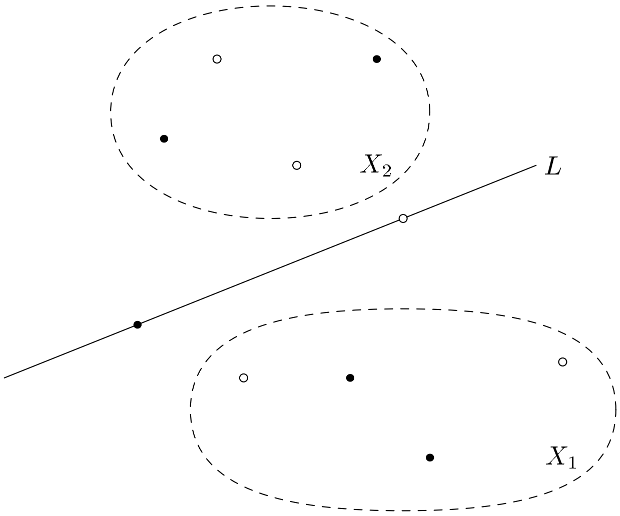
Oto **algorytm brutalny**:

```
utwórz w dowolny sposób n odcinków zawierających wszystkie punkty
while (istnieją dwa przecinające się odcinki u1v1, u2v2) do
  POPRAW(u1, u2, v1, v2)
```

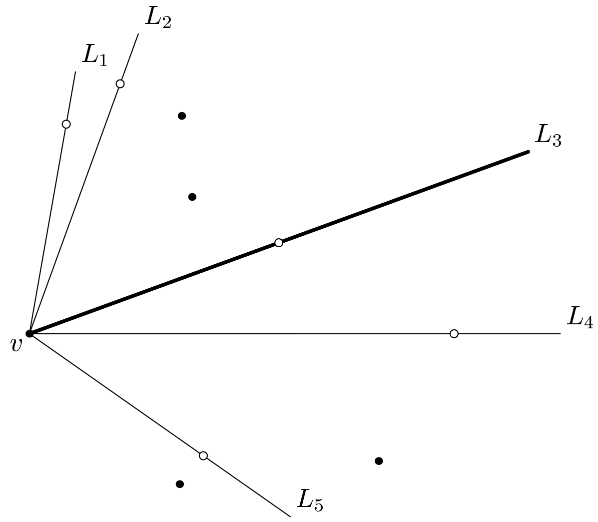
Oczywistym jest, że jeśli algorytm ten ma własność stopu, to jest poprawny. Algorytm ma własność stopu, gdyż każda wykonana operacja *POPRAW* zmniejsza sumaryczną długość odcinków. Dla wielu zestawów danych algorytm ten zadziała błyskawicznie, ale dla pewnych złośliwych przypadków może pracować bardzo długo.

Zrównoważonym zbiorem punktów będziemy nazywać taki zbiór, w którym jest tyle samo punktów czarnych, co białych. Podamy teraz algorytm korzystający z pojęcia *prostej równoważącej*. Jest to taka prosta, która dzieli zbiór niezawartych w niej punktów na dwa zbiory zrównoważone (zob. rysunek 2). Załóżmy, że mamy funkcję $PROSTA\_ROW(X)$ znajdującą prostą równoważącą dla niepustego podzbioru X. Wtedy wynik obliczamy wywołując $ODCINKI\_W\_ZBIORZE(R \cup W)$.

Rys. 2 Prosta równoważąca.



Rys. 3 Ilustracja graficzna działania funkcji $PROSTA\_ROW$, $L = L_3$.



```
procedure ODCINKI_W_ZBIORZE(X);
begin
  if X ≠ ∅ then begin
    L := PROSTA_ROW(X);
    połącz dwa  punkty leżące na L;
    niech X1, X2 będą zbiorami punktów leżących, odpowiednio,
    po jednej i po drugiej stronie L;
    wykonaj rekurencyjnie ODCINKI_W_ZBIORZE(X1);
    wykonaj rekurencyjnie ODCINKI_W_ZBIORZE(X2)
  end
end
```

Prostą równoważącą można obliczyć w następujący sposób:

```
1:  function PROSTA_ROW(X) : prosta;
2:  begin
3:    wybierz punkt v leżący najbardziej na lewo;
4:    { przyjmijmy dla uproszczenia, że v jest czarny }
5:    niech L_1, L_2, ..., L_4n będą prostymi przechodzącymi przez v
6:    posortowanymi względem ich współczynników kierunkowych;
7:    niech L ∈ {L_1, L_2, ... L_n} będzie pierwszą
8:    prostą równoważącą;
9:    PROSTA_ROW := L
10: end
```

Na rysunku 3 przedstawiono ideę działania funkcji *PROSTA_ROW*.

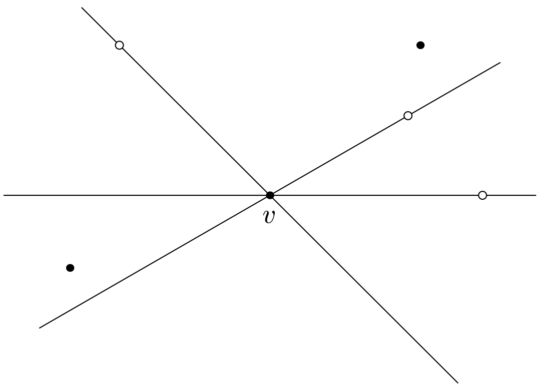
W instrukcji w wierszu 7 zawsze znajdzie się odpowiednia prosta. Wynika to z następującego faktu:

Fakt 1: Jeśli mamy ciąg punktów czarnych i białych, w którym liczba czarnych jest o jeden mniejsza niż białych, to znajdzie się punkt taki, że zarówno zbiór punktów występujących przed nim, jak i zbiór punktów występujących po nim w tym ciągu są zbiorami zrównoważonymi.

Uzasadnienie: Gdyby istniał ciąg będący kontrprzykładem, to usuwając z niego dwa sąsiednie punkty różnych kolorów otrzymalibyśmy „krótszy kontrprzykład". W końcu otrzymalibyśmy kontrprzykład długości 1. Ale ciąg długości 1, spełniający założenie, składa się tylko z punktu białego i po obu jego stronach są zbiory zrównoważone.

Obserwacja 2: Szukając prostej równoważącej zaczynamy od punktu najbardziej na lewo, natomiast nie możemy wystartować z dowolnego punktu v. Rysunek 4 pokazuje dlaczego.

Rys. 4 Nie możemy wybrać dowolnego punktu v w funkcji *PROSTA_ROW*. Nie ma równoważącej prostej przechodzącej przez punkt v na rysunku.



Funkcję *PROSTA_ROW* można łatwo zaimplementować w czasie $O(n \log n)$ (potrzeba trochę prostej geometrii: liczenie kątów). W sensie złożoności czasowej „wąskim gardłem" jest sortowanie prostych względem ich współczynników kierunkowych. Możemy pozbyć się sortowania korzystając z bardziej skomplikowanego algorytmu używającego liczenia mediany w czasie liniowym (co nie jest trywialne). Znajdujemy punkt biały w o tej własności, że po obydwu stronach prostej (v, w) jest tyle samo punktów białych (z dokładnością do 1). Jeżeli prosta (v, w) jest równoważąca, to kończymy. W przeciwnym razie poszukiwanie prostej wynikowej ograniczamy do zbioru leżącego po tej stronie prostej (v, w), po której jest mniej czarnych punktów. Dla tego zbioru powtarzamy postępowanie rekurencyjnie.

Cały algorytm ma złożoność czasową $O(n^2 \log n)$, gdy korzystamy z sortowania, oraz $O(n^2)$ gdy zastępujemy sortowanie skomplikowanym liczeniem mediany w czasie liniowym.

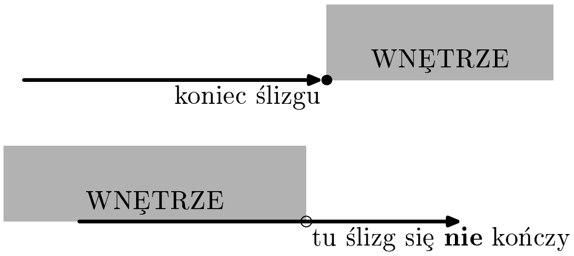
TESTY

Do oceny rozwiązań zawodników użyto 11-tu testów RAK0.IN–RAK10.IN. Największą trudnością było wygenerowanie testów dużych rozmiarów, w których żadne trzy punkty nie byłyby współliniowe. Zachęcamy czytelników do napisania efektywnego generatora takich testów.

Marcin Kubica **Adam Borowski**
autor zadania, opracowanie program wzorcowy

Lodowisko

Na największym lodowisku w Bajtocji zorganizowano zawody w ślizganiu. Lodowisko ma kształt kwadratu o rozmiarach 10000×10000. Zawodnik rozpoczyna ślizgi z punktu startowego wskazanego przez sędziów i ma za zadanie zakończyć ślizgi w punkcie docelowym, również wskazanym przez sędziów. Punkty startowy i docelowy są różne. Ślizgi mogą odbywać się tylko w kierunkach równoległych do boków lodowiska. Na lodowisku rozstawiono przeszkody. Każda przeszkoda jest graniastosłupem, którego podstawy są wielokątami o bokach równoległych do boków lodowiska. Każde dwa kolejne boki podstawy są zawsze do siebie prostopadłe. Przeszkody nie mają wspólnych punktów. Każdy ślizg kończy się przy pierwszym zetknięciu ze ścianą pewnej przeszkody, która to ściana jest prostopadła do kierunku ślizgu i ogranicza wnętrze przeszkody w kierunku tego ślizgu. Innymi słowy, zatrzymanie następuje wyłącznie na ścianie, na którą się „wpada" lub w punkcie docelowym. Wypadnięcie z lodowiska oznacza dyskwalifikację. Ślizgi mogą odbywać się wzdłuż ściany przeszkody.



Czy zawodnik wykonujący ślizgi zgodnie z podanymi regułami może dotrzeć z punktu startowego do punktu docelowego? Jesli tak, to jaka jest najmniejsza liczba ślizgów, które musi wykonać?

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego* `LOD.IN` *opisy lodowiska i przeszkód oraz współrzędne punktów startowego i docelowego,*
- *sprawdza, czy ślizgając się można z punktu startowego dotrzeć do punktu docelowego, a jeżeli tak, to oblicza najmniejszą liczbę ślizgów potrzebną do tego,*
- *zapisuje wynik w pliku tekstowym* `LOD.OUT`*.*

Wejście

Plan lodowiska jest naniesiony na siatkę prostokątną o rozmiarach 10000×10000. Dolny lewy róg siatki ma współrzędne $(0, 0)$. Górny prawy róg siatki ma współrzędne $(10000, 10000)$. W pierwszym wierszu pliku wejściowego `LOD.IN` *znajdują się dwie liczby całkowite z_1 i z_2 oddzielone pojedynczym odstępem, $0 \leqslant z_1, z_2 \leqslant 10000$. Para (z_1, z_2) to współrzędne punktu startowego. W drugim wierszu pliku* `LOD.IN` *znajdują się dwie liczby całkowite t_1 i t_2 oddzielone pojedynczym odstępem, $0 \leqslant t_1, t_2 \leqslant 10000$. Para (t_1, t_2) to współrzędne punktu docelowego. Trzeci wiersz pliku* `LOD.IN` *zawiera jedną liczbę naturalną s, $1 \leqslant s \leqslant 2500$. Jest to liczba przeszkód. Kolejne wiersze zawierają opisy s przeszkód. Każdy opis przeszkody rozpoczyna się od wiersza zawierającego jedną dodatnią liczbę całkowitą r równą liczbie ścian bocznych (boków podstawy) przeszkody. W kolejnych r wierszach znajdują się po dwie liczby całkowite x i y oddzielone pojedynczym odstępem. Są to kolejne współrzędne wierzchołków podstawy przeszkody przy obchodzeniu jej zgodnie z ruchem wskazówek zegara (tzn. przy obchodzeniu przeszkody wnętrze znajduje się zawsze po prawej stronie obchodzącego). Łączna liczba ścian bocznych w przeszkodach nie przekracza 10000.*

Wyjście

Twój program powinien zapisać w jednym wierszu pliku wyjściowego `LOD.OUT`*:*

- *jedno słowo 'NIE', gdy nie jest możliwe dotarcie z punktu startowego do punktu docelowego lub*

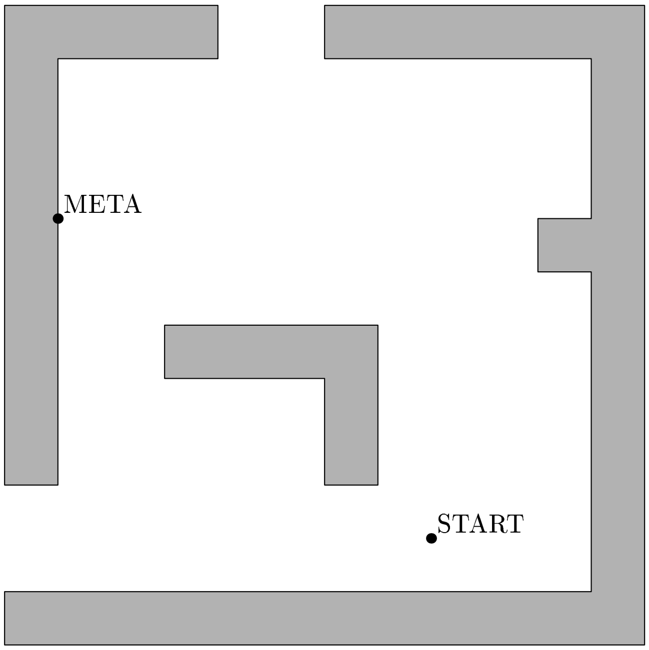
- *najmniejszą liczbę ślizgów potrzebną w tym celu, w przeciwnym przypadku.*

Przykład

Dla pliku wejściowego `LOD.IN`*:*

```
40 10
5 40
3
6
0 15
0 60
20 60
20 55
5 55
5 15
12
30 55
30 60
60 60
60 0
0 0
0 5
55 5
55 35
50 35
50 40
55 40
55 55
6
30 25
15 25
15 30
35 30
35 15
30 15
```

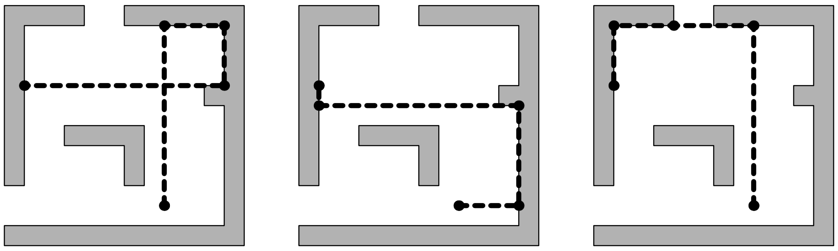
opisującego następujący układ przeszkód:



poprawną odpowiedzią jest plik tekstowy `LOD.OUT`*:*

4

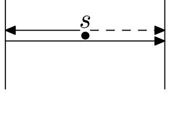
Oto możliwe ciągi ślizgów długości 4:



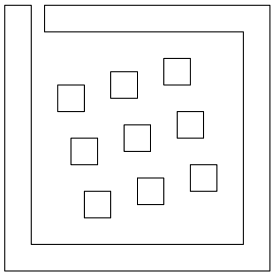
ROZWIĄZANIE

Stosunkowo łatwo można zauważyć, że mamy tu do czynienia z pewnym grafem. Wierzchołkami tego grafu są możliwe punkty zatrzymania się zawodnika oraz punkt startowy i docelowy, a krawędzie odpowiadają możliwym ślizgom. Nasze zadanie polega na znalezieniu (długości) najkrótszej ścieżki łączącej punkt startowy z docelowym. Możemy posłużyć się w tym celu algorytmem przeszukiwania grafu wszerz (por. [18], p. 2.3, [9] p. 1.5.3, [12] p. 23.2). Trudność tego zadania polega jednak na konstrukcji grafu.

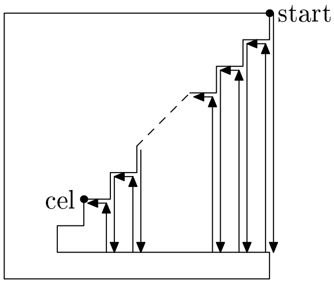
Zastanówmy się, ile wierzchołków może mieć nasz graf. Niech n będzie liczbą wierzchołków przeszkód ($n \leqslant 10\,000$). Ile może być możliwych punktów zatrzymania nie będących wierzchołkami przeszkód? Zapomnijmy na chwilę o punkcie startowym i docelowym. Zauważmy, że jeżeli zatrzymaliśmy się nie w wierzchołku przeszkody, to bez straty ogólności możemy założyć, że zaczęliśmy ten ślizg wzdłuż pewnej ściany. Jeżeli tak nie jest, to znaczy, że poprzedni ślizg został wykonany w przeciwną stronę, po czym „odbiliśmy się” od ściany. Jest to możliwe jedynie wtedy, gdy poprzedni ślizg zaczynał się w punkcie startowym. Wówczas jednak mogliśmy od razu zacząć ślizgać się w drugą stronę.



Powtarzając to rozumowanie odpowiednią liczbę razy znajdziemy taką ścianę, że ślizgając się wzdłuż niej zatrzymamy się w danym punkcie. Zauważmy, że stojąc przy ścianie możemy zacząć się ślizgać równolegle do niej tylko w dwóch kierunkach. Jeśli więc każdej ścianie przyporządkujemy punkty (leżące poza nią) w których zatrzymamy się, jeżeli zaczniemy się ślizgać równolegle do niej, to każdej ścianie przyporządkujemy co najwyżej dwa punkty. Jest to jednak oszacowanie górne, którego nie da się osiągnąć. Zaczynając ślizg przy skrajnie lewej ścianie i ślizgając się równolegle do niej wypadniemy poza lodowisko (jeśli jest kilka skrajnie lewych ścian, to wypadniemy poza lodowisko zaczynając ślizg przy najwyższej z nich i ślizgając się w górę oraz przy najniższej z nich i ślizgając się w dół). Podobnie dla skrajnie prawej, skrajnie górnej i skrajnie dolnej ściany przeszkód. Stąd punktów zatrzymania nie będących wierzchołkami przeszkód może być co najwyżej $2n-8$. Uwzględniając punkt startowy i docelowy musimy doliczyć jeszcze co najwyżej sześć punktów — punkt startowy i docelowy też mogą być wierzchołkami naszego grafu, a dodatkowo z punktu startowego możemy zacząć się ślizgać w dowolnym z czterech kierunków. Tak więc punktów zatrzymania może być co najwyżej $3n-2$. Poniższy rysunek pokazuje, jak można się zbliżyć do tego oszacowania.

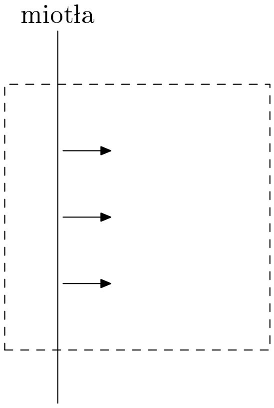


Jednym z błędów, jaki można było popełnić, było zbyt małe ograniczenie na liczbę punktów zatrzymania, np. 10 000 lub 20 000. Również liczba ślizgów niezbędnych, aby dotrzeć do punktu docelowego, może być większa od 10 000. Ilustruje to poniższy rysunek.



Najprostsze rozwiązanie, jakie się nasuwa, polega na skonstruowaniu grafu, którego wierzchołkami są możliwe punkty zatrzymania, a krawędzie odpowiadają możliwym ślizgom, oraz wyznaczeniu minimalnej liczby ślizgów potrzebnych do osiągnięcia punktu docelowego za pomocą przeszukiwania wszerz. Zarówno liczba wierzchołków, jak i krawędzi grafu jest liniowa ze względu na liczbę wierzchołków przeszkód n. Możemy więc przeszukać graf w czasie $O(n)$. Jednak konstrukcja grafu jest bardziej kosztowna. Kluczowym jej elementem, zarówno przy wyznaczaniu wierzchołków jak i krawędzi grafu, jest obliczenie punktów zatrzymania dla ślizgu o zadanym kierunku i punkcie początkowym. Najprościej możemy tego dokonać przeglądając za każdym razem wszystkie ściany prostopadłe do kierunku ślizgu. Wówczas jednak nasz program będzie działał w czasie rzędu $O(n^2)$.

Algorytm ten możemy przyspieszyć korzystając z techniki zamiatania (por. [9] p. 8.4, [12] p. 35.2). Opiera się ona na następującym pomyśle. Mamy dokonać pewnych obliczeń geometrycznych na płaszczyźnie. Po płaszczyźnie „przesuwamy" prostą (zwaną miotłą) — przykładowo niech to będzie prosta pionowa, którą przesuwamy z lewa na prawo. Staramy się zredukować nasz problem rozwiązując go w danej chwili tylko w obrębie miotły. Ponadto nie rozważamy każdego możliwego położenia miotły, tylko przesuwając ją „zatrzymujemy" się, gdy miotła napotyka punkty istotne dla obliczeń. Zwykle z miotłą związana jest pewna struktura danych, którą aktualizujemy w miarę przesuwania miotły.



Za pomocą jednego zamiatania możemy np. wyznaczyć dla każdej pionowej ściany punkty zatrzymania dla pionowych ślizgów zaczynających się przy tych ścianach. W strukturze danych odpowiadającej miotle przechowujemy poziome ściany przecinające miotłę (a dokładniej ich współrzędne Y). Miotłę zatrzymujemy, gdy napotyka ona wierzchołki ścian. Początkowo miotła jest pusta. Gdy miotła napotyka pewne wierzchołki przeszkód — są to równocześnie końce pewnej pionowej ściany (ścian), jak i końce poziomych ścian — wykonujemy dla nich następujące operacje (w podanej kolejności):

- dla każdego wierzchołka będącego lewym końcem poziomej ściany wstawiamy tę ścianę do miotły,
- dla każdego wierzchołka leżącego we wklęsłym kącie ściany (kąt) ślizg zaczynający się przy pionowej ścianie przylegającej do tego wierzchołka i skierowany przez ten wierzchołek zatrzyma się w nim,
- dla każdego wierzchołka leżącego w wypukłym kącie ściany (narożnik) ślizg zaczynający się przy pionowej ścianie przylegającej do tego wierzchołka i skierowany przez ten wierzchołek zatrzyma się na najbliższej poziomej ścianie należącej do miotły, leżącej, odpowiednio, powyżej lub poniżej danego wierzchołka,
- dla każdego wierzchołka będącego prawym końcem poziomej ściany usuwamy tę ścianę z miotły.

Aby wyznaczyć miejsca zatrzymania miotły, przed przystąpieniem do zamiatania sortujemy wierzchołki przeszkód względem ich współrzędnych X. Współrzędne te są liczbami całkowitymi od 0 do 10 000 — możemy więc zastosować np. sortowanie przez zliczanie o koszcie $O(n)$.

Stosując drugie zamiatanie (w pionie) wyznaczamy pozostałe wierzchołki naszego grafu. Nie mamy jednak wyznaczonych wszystkich jego krawędzi. Lukę tę uzupełnimy wprowadzając dwie poprawki do opisanego algorytmu.

Po pierwsze, jeżeli zaczynamy ślizg równoległy do ściany, przy której stoimy i zatrzymujemy się w punkcie, który leży na innej ścianie, lecz nie na jej końcu, to odbijając się od niej prostopadle zatrzymamy się w wierzchołku ściany, przy której staliśmy. Podobnie, jeżeli zaczynając ślizg z punktu startowego zatrzymamy się na pewnej ścianie, to odbijając się od niej prostopadle zatrzymamy się w tym samym punkcie, co gdybyśmy z punktu startowego ślizgali się w przeciwnym kierunku.

Po drugie, w przypadku ślizgów równoległych do ściany, przy której stoimy, punkt zatrzymania nie zależy od tego, w którym miejscu ściany zaczynaliśmy ślizg. Możemy więc dla każdego punktu zatrzymania nie będącego wierzchołkiem ściany pamiętać, na jakiej ścianie on leży, po czym poprowadzić z tego punktu krawędzie do punktów zatrzymania, dla ślizgów równoległych do ściany zaczynających się w jej końcach.

Efektywność opisanego algorytmu zależy od sposobu zaimplementowania miotły i operacji na niej. Miotła jest tu zbiorem liczb całkowitych (z przedziału od 0 do 10 000) z następującymi operacjami:

- inicjuj miotłę jako pusty zbiór,
- wstaw liczbę do zbioru,
- usuń liczbę ze zbioru,
- dla zadanej liczby znajdź najmniejszą liczbę większą od niej i należącą do miotły,

- dla zadanej liczby znajdź największą liczbę mniejszą od niej i należącą do miotły.

Jeżeli zaimplementujemy miotłę jako listę, to koszt czasowy operacji na niej będzie wynosił $O(n)$, a cały algorytm będzie działał w czasie $O(n^2)$. Mimo, że jest to ten sam rząd co w przypadku poprzedniego algorytmu, to takie rozwiązanie ma szansę być bardziej efektywne, gdyż miotła zwykle zawiera istotnie mniej niż $(n/2)$ ścian.

Możemy tu jednak zastosować jedną z wielu słownikowych struktur danych realizującą powyższe operacje w czasie $O(\log n)$. Daje to koszt całkowity algorytmu rzędu $O(n \log n)$.

Pozwólmy sobie na drobną dygresję. W przypadku prawie wszystkich zadań olimpijskich podawane jest górne ograniczenie na rozmiar danych wejściowych. W przypadku wielu zadań należy wykorzystać taką czy inną strukturę służącą do przechowywania danych, przy czym rozmiar tej struktury potrafimy ograniczyć na podstawie ograniczenia rozmiaru danych wejściowych. W takich przypadkach należy się zastanowić nad wykorzystaniem struktury danych będącej drzewem zrównoważonym, często o ustalonym kształcie, pamiętanym w statycznej tablicy. Podstawowe operacje na takich strukturach są zwykle proste do zaimplementowania, a działają w czasie $O(\log n)$.

Najbardziej znanym przykładem takiej struktury danych jest kopiec (por. [9] p. 2.6, [12] r. 7). Pozwala on na wstawianie elementów i usuwanie elementu najmniejszego (największego) w czasie $O(\log n)$. Kopiec jest w pełni zrównoważonym drzewem binarnym, w którego węzłach są przechowywane liczby, przy czym wartość przechowywana w ojcu jest mniejsza (większa) od wartości przechowywanych w jego synach. Istotny jest sposób pamiętania kopca w tablicy. Kształt kopca n-elementowego jest ustalony i zajmuje on pierwszych n elementów tablicy (indeksowanej od 1). Wartość przechowywana w korzeniu znajduje się pod indeksem 1. Niech wartość przechowywana w danym węźle znajduje się w i-tym elemencie tablicy. Jeżeli $2i > n$, to dany węzeł jest liściem. Jeżeli $2i = n$, to dany węzeł ma jednego syna, którego wartość jest pamiętana w n-tym elemencie tablicy. Jeżeli $2i < n$, to dany węzeł ma dwóch synów, a ich wartości są pamiętane w $2i$-tym i $(2i+1)$-szym elemencie tablicy. Jak łatwo zauważyć, wartość przechowywana w ojcu danego węzła znajduje się w $\lfloor (i/2) \rfloor$-tym elemencie tablicy.

Innym przykładem takiej struktury jest drzewo BST o ograniczonej wysokości. Przechowujemy je w tablicy, podobnie jak kopiec, przy czym elementy tablicy zawierają, oprócz wartości przechowywanych w węzłach, znaczniki wskazujące, czy dany element jest wykorzystany. Strukturę tę stosujemy, gdy najpierw znamy wszystkie elementy, które należy wstawić do drzewa, po czym są one sukcesywnie z niego usuwane. Po posortowaniu elementów tworzymy w pełni zrównoważone drzewo BST. Usuwanie elementów nie zwiększa wysokości drzewa, więc możemy to robić tak samo, jak w przypadku zwykłych drzew BST, zaznaczając zwalniane elementy.

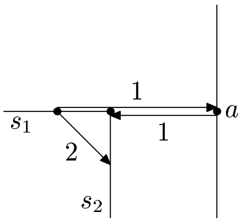
W naszym przypadku możemy skorzystać z drzewa całkowitoliczbowego. Jest to struktura służąca do pamiętania zbiorów (lub multizbiorów) złożonych z liczb całkowitych z określonego przedziału (powiedzmy k-elementowego). Jest to drzewo binarne o głębokości $\lceil \log_2 k \rceil$, mające wszystkie możliwe węzły do głębokości $\lceil \log_2 k \rceil - 1$ oraz k liści na głębokości $\lceil \log_2 k \rceil$. W węzłach tego drzewa przechowujemy liczby całkowite (od 0 do k). Liście znajdujące się najgłębiej reprezentują elementy należące do zbioru — jeżeli element należy do zbioru, to w liściu znajduje się 1, a w przeciwnym przypadku 0. W każdym węźle wewnętrznym pamiętamy sumę wartości przechowywanych w liściach poddrzewa o korzeniu w tym węźle. Inaczej mówiąc, w każdym węźle wewnętrznym pamiętamy sumę wartości przechowywanych w jego dzieciach. Drzewo takie pamiętamy, podobnie jak kopiec, w $2^{\lceil \log_2 k \rceil} - 1 + k$-elementowej tablicy liczb całkowitych. Drzewo całkowitoliczbowe pozwala nam na wykonywanie następujących operacji:

- inicjacja drzewa — w czasie $O(k)$,
- wstawienie elementu — w czasie $O(\log k)$,
- usunięcie elementu — w czasie $O(\log k)$,
- wybór najmniejszego (największego) elementu w zbiorze — w czasie $O(\log k)$,
- wybór najmniejszego (największego) elementu w zbiorze większego (mniejszego) od zadanej liczby — w czasie $O(\log k)$,
- podanie liczby elementów w zbiorze — w czasie $O(1)$.

Implementując miotłę jako drzewo całkowitoliczbowe i stosując zamiatanie otrzymujemy rozwiązanie działające w czasie $O(n \log n)$. Nie jest to jednak jeszcze rozwiązanie wzorcowe.

Główną wadą takiego rozwiązania jest zbyt duży rozmiar konstruowanego grafu. Postaramy się zastąpić punkty zatrzymania, jako wierzchołki grafu, ścianami na których się zatrzymujemy. Przypomnijmy, że w przypadku ślizgu równoległego do ściany, przy której stoimy nie jest istotne, z którego dokładnie punktu przy tej ścianie zaczynamy ślizg. W przypadku ślizgu prostopadłego do ściany (nie zaczynającego się w punkcie startowym) możliwe są następujące przypadki:

- zaczynamy ślizg w wierzchołku ściany — wówczas jest to ślizg równoległy do drugiej ściany o końcu w tym wierzchołku,
- punkt startowy nie leży przy żadnej ścianie, zaczynamy ślizg w punkcie, który nie jest wierzchołkiem i do którego dostaliśmy się jednym ślizgiem z punktu startowego — przypadek taki możemy pominąć, gdyż ma on o jeden ślizg więcej niż gdybyśmy z punktu startowego ślizgali się w przeciwnym kierunku,
- punkt startowy leży przy ścianie, ale nie w wierzchołku, zaczynamy ślizg w punkcie, który nie jest wierzchołkiem i do którego dostaliśmy się jednym ślizgiem z punktu startowego, prostopadłym do ściany, przy której leży punkt startowy — przypadek taki możemy pominąć, gdyż prowadzi on nas z powrotem do punktu startowego,
- zaczynamy ślizg w punkcie a, który nie jest wierzchołkiem i do którego dostaliśmy się w wyniku ślizgu zaczynającego się przy pewnej ścianie s_1, równoległego do niej — wówczas drugi ślizg kończy się w wierzchołku łączącym ściany s_1 i s_2.



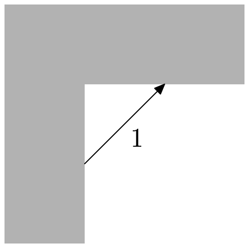
Możemy wówczas „zapomnieć” o punkcie a, pamiętając jedynie, że w dwóch ślizgach możemy się dostać z punktu leżącego przy ścianie s_1 do punktu leżącego przy ścianie s_2.

Możemy więc posłużyć się grafem, którego wierzchołkami są ściany przeszkód, a krawędzie mają długość 1 lub 2. Dodatkowo dodajemy trzy specjalne wierzchołki — dwa reprezentujące punkt startowy (dla ślizgów pionowych i poziomych) i jeden reprezentujący punkt docelowy. Krawędź długości 1 prowadząca od jednej ściany do drugiej oznacza, że stojąc przy pierwszej ścianie i ślizgając się równolegle do niej, zatrzymujemy się na drugiej ścianie. Krawędź długości 2 prowadząca od jednej ściany do drugiej oznacza, że stojąc przy pierwszej ścianie i ślizgając się równolegle do niej, a następnie odbijając się prostopadle od pewnej ściany, zatrzymujemy się na drugiej ścianie. Graf taki ma co najwyżej 10 003 wierzchołki, a z każdego wierzchołka wychodzą co najwyżej dwie krawędzie długości 1 i dwie krawędzie długości 2.

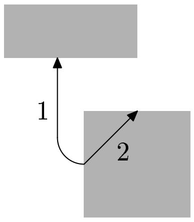
Zastanówmy się, jak można wyznaczyć krawędzie tego grafu. Podobnie jak poprzednio korzystamy dwukrotnie z zamiatania — pionowo i poziomo. Zamiatając poziomo tworzymy krawędzie odpowiadające pionowym ślizgom, a zamiatając pionowo, odpowiadające poziomym ślizgom. Dla ustalenia uwagi opiszemy poziome zamiatanie.

Struktura danych miotły zawiera ściany przecinające prostopadle miotłę, uporządkowane wedle ich współrzędnych Y. Początkowo miotła jest pusta. Strukturę tę implementujemy jako drzewo całkowitoliczbowe oraz tablicę zawierającą informację, która ściana przecina w danym miejscu miotłę. Wierzchołki ścian sortujemy (przez zliczanie — w czasie $O(n)$) ze względu na ich współrzędne X. Miotłę zatrzymujemy na wierzchołkach ścian. Gdy miotła napotyka pewne wierzchołki przeszkód, wykonujemy dla nich następujące operacje:

- dla każdego wierzchołka będącego lewym końcem poziomej ściany wstawiamy tę ścianę do miotły,
- jeżeli punkt docelowy leży na miotle, to wstawiamy do miotły odpowiadający mu wierzchołek,
- dla każdego wierzchołka leżącego we wklęsłym kącie ściany (kąt) prowadzimy krawędź długości 1, od pionowej ściany o końcu w tym wierzchołku, do poziomej ściany o końcu w tym wierzchołku,



- dla każdego wierzchołka leżącego w wypukłym kącie ściany (narożnik), w zależności od tego, czy jest to górny, czy dolny koniec pionowej ściany, znajdujemy najbliższą poziomą ścianę przecinającą miotłę (lub punkt docelowy) i leżącą, odpowiednio, powyżej lub poniżej niego; następnie prowadzimy krawędź długości 1 od pierwszej ściany do drugiej, oraz krawędź długości 2 od pierwszej ściany do ściany stykającej się z nią w danym wierzchołku,



- dla punktu startowego znajdujemy najbliższe poziome ściany przecinające miotłę (lub punkt docelowy), leżące powyżej i poniżej; następnie prowadzimy krawędź długości 1 od punktu startowego do tych ścian,
- dla każdego wierzchołka będącego prawym końcem poziomej ściany, usuwamy tę ścianę z miotły,
- jeżeli punkt docelowy leży na miotle, to usuwamy z miotły odpowiadający mu wierzchołek.

Zamiatania wykonują się w czasie $O(n \log n)$. Minimalną liczbę potrzebnych ślizgów wyznaczamy za pomocą zmodyfikowanego algorytmu przeszukiwania wszerz. (Możemy też posłużyć się algorytmem Dijkstry, por. [18] p. 3.3, [9] p. 7.6, [12] p. 25.2. Jest to jednak mniej efektywne i bardziej skomplikowane rozwiązanie.) W przypadku zwykłego przeszukiwania wszerz posługujemy się kolejką wierzchołków grafu. W naszym przypadku posługujemy się trzema kolejkami — pierwsza zawiera aktualnie przeglądane wierzchołki, do drugiej wstawiamy wierzchołki, do których prowadzą krawędzie długości 1, a do trzeciej — wierzchołki, do których prowadzą krawędzie długości 2. Na początku do pierwszej kolejki wstawiamy wierzchołki reprezentujące punkt startowy. W momencie, gdy pierwsza kolejka staje się pusta, przenosimy do niej wierzchołki z drugiej kolejki, a do drugiej kolejki przenosimy wierzchołki z trzeciej kolejki. Jednocześnie zwiększa się o 1 odległość od punktu startowego wierzchołków znajdujących się w pierwszej kolejce. Przeszukiwanie kończymy, gdy osiągniemy punkt docelowy, lub gdy wszystkie trzy kolejki będą puste. (W ostatnim przypadku punktu docelowego nie można osiągnąć.)

Przeszukiwanie grafu działa w czasie $O(n)$. Tak więc cały algorytm działa w czasie $O(n \log n)$. Rozwiązanie to znajduje się na załączonej dyskietce.

TESTY

Do oceny rozwiązań zawodników użyto 26-ciu testów. Test LOD0.IN był testem z treści zadania. Testy dla tego zadania zostały podzielone na grupy. Rozwiązanie uzyskiwało punkty za testy z danej grupy tylko wtedy, gdy przechodziło wszystkie testy z tej grupy. Dzięki temu rozwiązania podające stale wynik 'NIE' nie otrzymywały żadnych punktów. Testy należące do tej samej grupy mają w nazwie tę samą liczbę (numer grupy) z ew. dodaną literą 'a' lub 'b'.

Testy z grupy 1 i 2 to proste testy oceniające poprawność rozwiązania.

Testy z grup 3–5 i 11–12 to testy charakteryzujące się dużą liczbą możliwych punktów zatrzymania oraz stosunkowo niewielką długością ścieżki prowadzącej do punktu docelowego (o ile taka ścieżka istnieje).

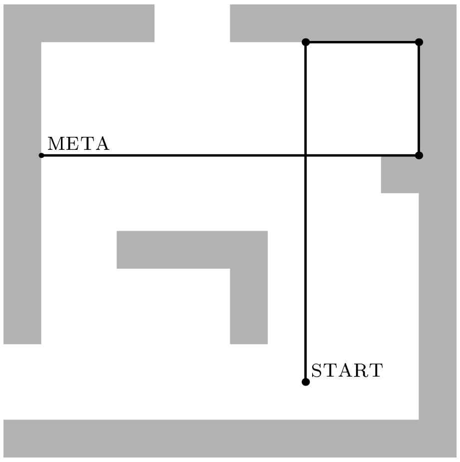
Niektóre z nich to rysunki zaadoptowane jako testy. Testy te miały na celu wychwycenie prostych, acz nieefektywnych rozwiązań.

Testy z grup 7–8, 10 i 13 to testy o zróżnicowanej liczbie możliwych punktów zatrzymania i różnej długości ścieżek prowadzących do punktów docelowych (o ile ścieżki takie istnieją), acz nie większej niż 10 000 ślizgów. Testy te miały na celu zróżnicowanie rozwiązań ze względu na ich efektywność. Testy nr 6 i 9 charakteryzują się dużą liczbą możliwych punktów zatrzymania oraz ścieżką prowadzącą do punktu docelowego dłuższą niż 10 000 ślizgów. Testy te sprawdzają również efektywność rozwiązań.

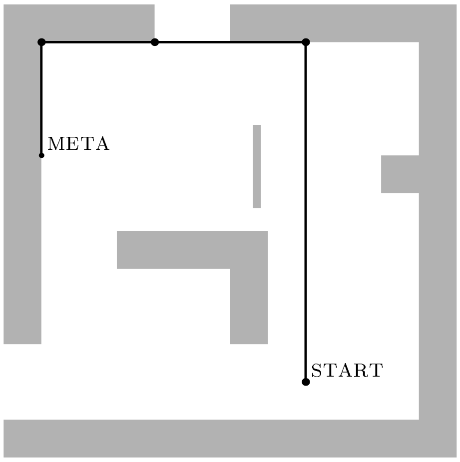
W poniższej tabeli są zestawione wielkości liczbowe charakteryzujące testy.

Test	L. ścian	Wynik
1	328	NIE
1a	328	13
2	326	32
2a	124	NIE
3	7 870	398
3a	7 870	NIE
4	9 804	195
5	9 988	65
6	9 396	14 060
7	10 000	2 500
7a	10 000	2 500
7b	10 000	2 500

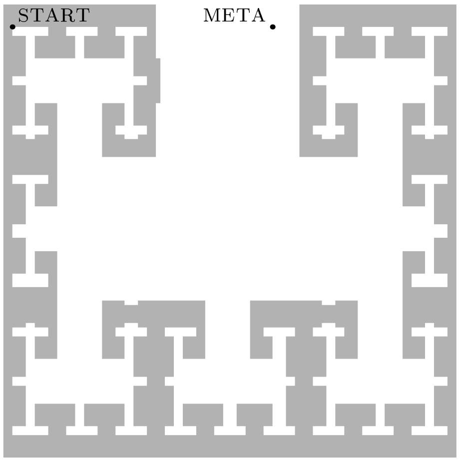
Test	L. ścian	Wynik
8	2 506	484
8a	4 290	NIE
9	10 000	14 987
9a	12	3
10	8 886	211
10a	10 000	2 623
10b	9 996	4 995
11	9 994	629
11a	10 000	NIE
12	7 730	404
12a	9 054	444
13	7 606	2 851
13a	7 606	NIE



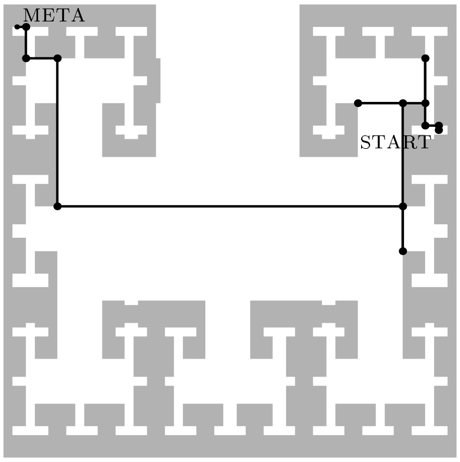
LOD0.IN



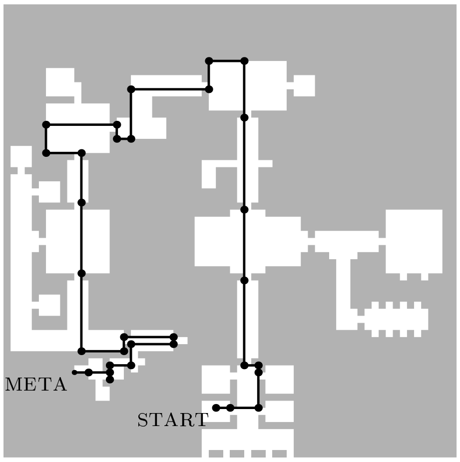
LOD0A.IN



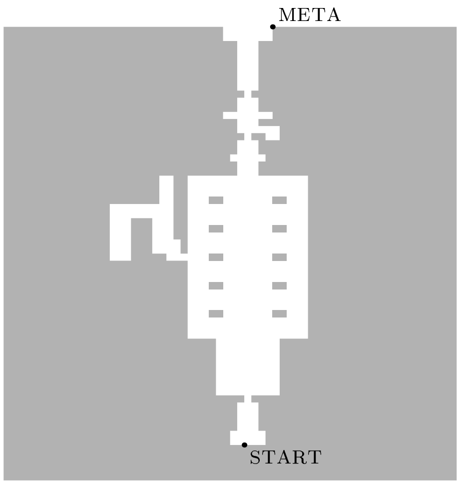
LOD1.IN



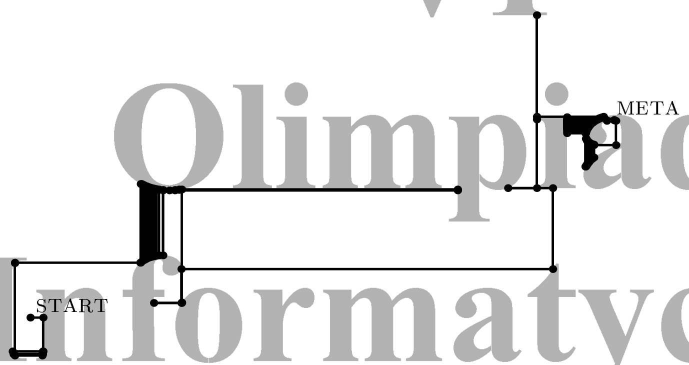
LOD1A.IN



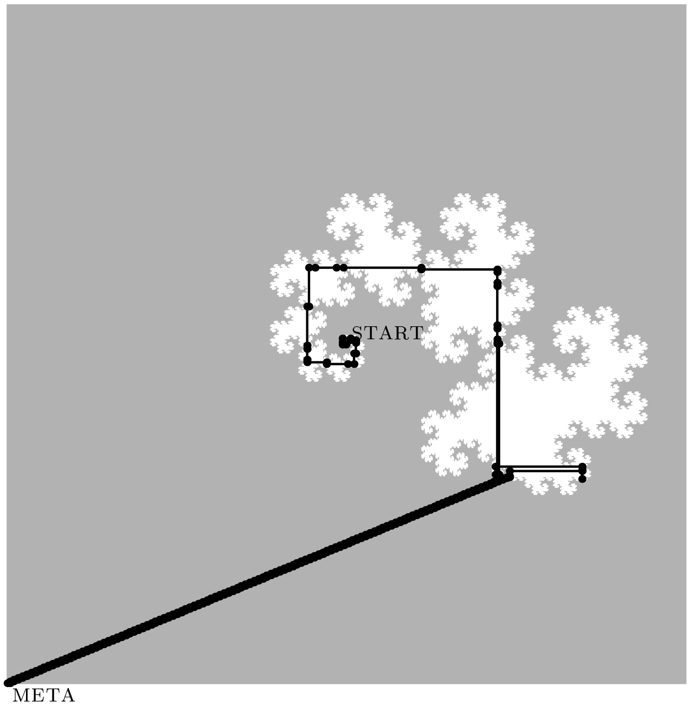
LOD2.IN



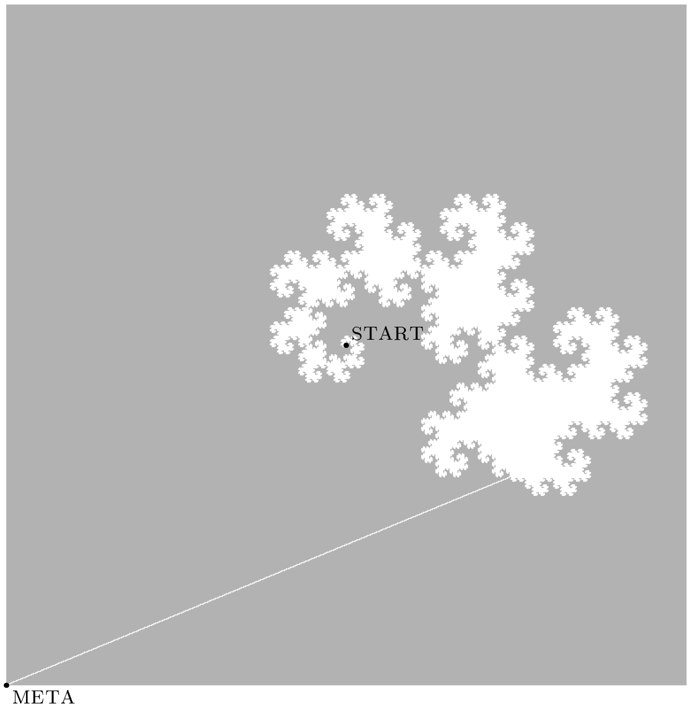
LOD2A.IN



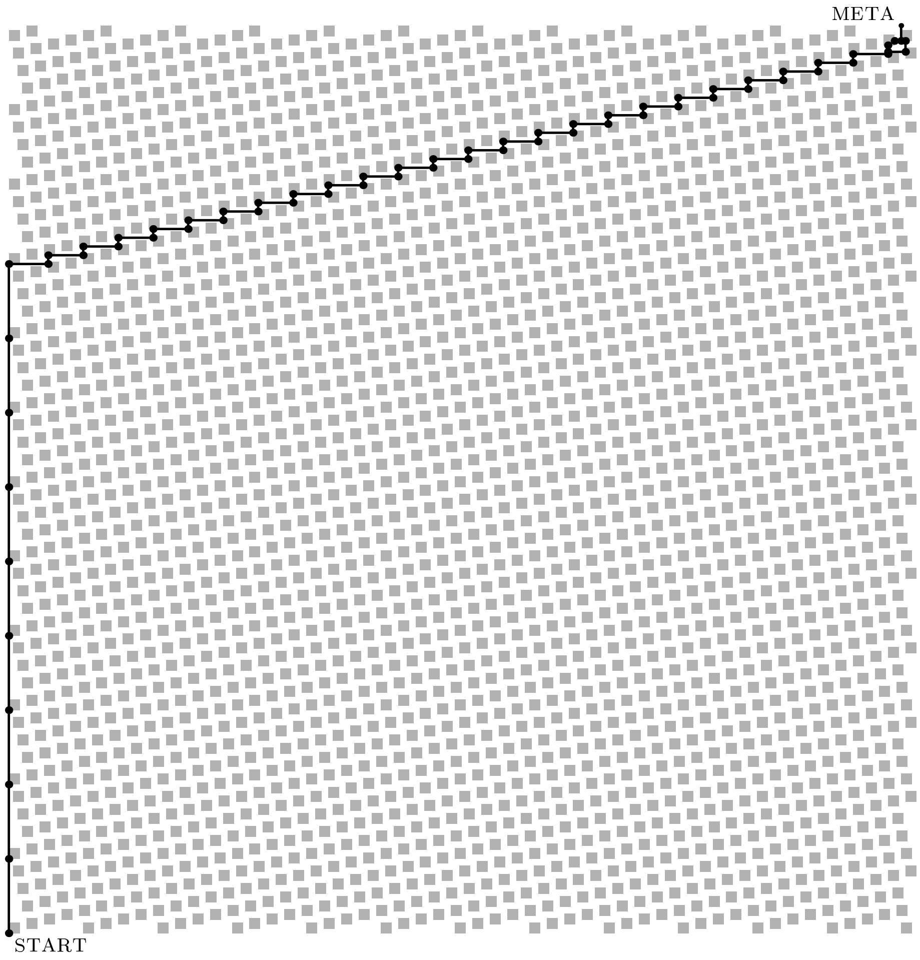
LOD4.IN



LOD3.IN



LOD3A.IN



LOD5.IN

Zawody III stopnia

opracowania zadań

Krzysztof Loryś
autor zadania, opracowanie

Marcin Stefaniak
program wzorcowy

Magazynier

Podłoga magazynu jest prostokątem podzielonym na $n \times m$ kwadratowych pól. Dwa pola są sąsiednie*, jeśli mają wspólny bok. Na jednym z pól stoi paczka. Każde inne pole jest albo wolne, albo zajęte przez skrzynię, której magazynier nie jest w stanie poruszyć. Magazynier musi przepchnąć paczkę z pola początkowego P na pole docelowe K. Magazynier może przemieszczać się po wolnych polach, przechodząc z pola, na którym stoi, na dowolne z wolnych pól sąsiednich. Jeśli magazynier stoi na polu sąsiadującym z polem, na którym jest paczka, to może ją przepchnąć na pole sąsiednie z przeciwnej strony paczki, jeżeli jest ono wolne.*

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `MAG.IN` *plan magazynu, początkowe położenia magazyniera i paczki oraz docelowe położenie paczki,*
- *obliczy minimalną liczbę przesunięć paczki przez granice pól, wymaganą do ustawienia jej w położeniu docelowym lub stwierdzi, że to jest niemożliwe,*
- *zapisze wynik w pliku tekstowym* `MAG.OUT`*.*

Wejście

W pierwszym wierszu pliku wejściowego `MAG.IN` *są zapisane dwie dodatnie liczby całkowite n i m oddzielone pojedynczym odstępem, $n, m \leqslant 100$. Są to rozmiary magazynu. W każdym z kolejnych n wierszy znajduje się jedno m-literowe słowo utworzone z liter* `S`, `M`, `P`, `K`, `w`*. Litera na i-tej pozycji j-tego z tych słów oznacza stan pola o współrzędnych (i, j):*

- `S` — *skrzynia,*
- `M` — *początkowa pozycja magazyniera,*
- `P` — *początkowa pozycja paczki,*
- `K` — *docelowa pozycja paczki,*
- `w` — *wolne pole.*

Każda z liter `M`, `P` *i* `K` *występuje w pliku* `MAG.IN` *dokładnie raz.*

Wyjście

Twój program powinien zapisać w pliku wyjściowym `MAG.OUT`*:*

- *dokładnie jedno słowo* `NIE`*, jeżeli paczki nie da się umieścić w położeniu docelowym,*
- *dokładnie jedną liczbę całkowitą, równą minimalnej liczbie przesunięć paczki przez granice pól, wymaganą do umieszczenia paczki w położeniu docelowym, jeżeli paczkę da się tam przepchnąć.*

Przykład

Dla pliku wejściowego `MAG.IN`*:*

```
10 12
SSSSSSSSSSSS
SwwwwwwwSSSS
SwSSSSwwSSSS
SwSSSSwwSKSS
SwSSSSwwSwSS
```

```
SwwwwwPwwwww
SSSSSSSwSwSw
SSSSSSMwSwww
SSSSSSSSSSSS
SSSSSSSSSSSS
```

poprawną odpowiedzią jest plik tekstowy `MAG.OUT`*:*

```
7
```

ROZWIĄZANIE

Zadanie oparte jest na popularnej swego czasu grze komputerowej *Sokoban*. Scenariusz gry jest taki sam, jak w naszym zadaniu: mamy magazyn, magazyniera i paczki; ten sam jest cel: paczki należy przesunąć na wyznaczone pola docelowe; identyczne są zasady ruchów: paczki można tylko pchać i to tylko w sytuacji, gdy za paczką znajduje się wolna przestrzeń. Jedyna różnica tkwi w liczbie paczek: w Sokobanie liczba ta nie jest z góry niczym ograniczona (oczywiście poza powierzchnią magazynu), w naszym zadaniu została ograniczona do 1. Ograniczenie to ma kolosalny wpływ na złożoność problemu. Udowodniono, że w wersji bez ograniczenia liczby paczek, stwierdzenie, czy wszystkie paczki można przesunąć na wyznaczone pozycje, jest problemem NP-zupełnym, co oznacza, że nie należy spodziewać się istnienia algorytmu rozwiązującego ten problem w czasie wielomianowym (względem rozmiaru powierzchni magazynu). Istnieje natomiast prosty algorytm, który działa w czasie wielomianowym, gdy liczba paczek k jest ograniczona przez stałą. Opiszemy pokrótce ten algorytm dla $k = 1$, a następnie przedstawimy modyfikacje prowadzące do liniowego czasu działania.

Stanem gry nazwiemy parę pól magazynu takich, że na pierwszym polu znajduje się magazynier, a na drugim — paczka. Stan nazwiemy *końcowym*, jeśli pole z paczką jest polem docelowym. Możliwy przebieg gry wygodnie jest opisać w terminologii grafowej. Konstruujemy graf skierowany G_S, którego wierzchołkami są stany gry. Z wierzchołka u prowadzimy krawędź do wierzchołka v wtedy i tylko wtedy, gdy z u można przejść do v w jednym ruchu magazyniera. Pytanie, czy można paczkę przesunąć na pozycję docelową, sprowadza się do pytania o istnienie ścieżki prowadzącej od stanu początkowego do stanu końcowego. Gdy zainteresowani jesteśmy obliczeniem najmniejszej liczby przesunięć paczek, możemy nadać krawędziom wagi: 1 — jeśli odpowiada ona przejściu między stanami z przesunięciem którejś z paczek; 0 — jeśli przejście odbywa się bez przesuwania paczek. Odpowiedź uzyskamy obliczając długość najtańszej ścieżki od stanu początkowego do jednego ze stanów końcowych.

Niestety, algorytm ten jest niepraktyczny. Graf stanów ma bowiem N^2 wierzchołków, gdzie N jest liczbą pól w magazynie. Jest to nieakceptowalne, gdy tak jak w naszym zadaniu, N może się równać 10000. Graf G_S możemy łatwo zmodyfikować tak, by miał jedynie $O(N)$ wierzchołków. Odbędzie się to jednak kosztem bardziej skomplikowanego wyznaczania krawędzi. Zauważmy bowiem, że z naszego punktu widzenia istotne są tylko te stany, w których pole magazyniera sąsiaduje z polem paczki, a takich pól jest mniej niż $4N$. Tylko z tych stanów wychodzą krawędzie o wadze 1, a więc odpowiadające przesunięciu paczki. Pozostałe krawędzie, o wadze 0, będą łączyły pary stanów różniące się jedynie pozycją magazyniera i będą odpowiadały „obejściu" paczki przez magazyniera (wykonanym jedynie po to, by zmienić kierunek pchania).

Krawędzie takie można wyznaczyć różnymi metodami. Poniżej zasygnalizujemy dwie z nich. Ponieważ interesują nas teraz ruchy magazyniera, w których nie przesuwa on paczki, wygodnie jest posłużyć się innym grafem, nazwijmy go G_M, którego wierzchołkami są pola magazynu nie zajęte przez skrzynie, a krawędzie łączą wierzchołki odpowiadające sąsiadującym polom. Zwracamy uwagę, że rozważany w tej chwili graf G_M jest nową konstrukcją i nie należy go mylić z grafem stanów G_S.

Metoda pierwsza. Dla każdej pary stanów u, v z grafu G_S różniących się jedynie pozycjami magazyniera sprawdzamy, czy w G_M istnieje ścieżka łącząca te pozycje i nie przechodząca przez pole zajęte przez paczkę.

Metodę tę można sprowadzić do zastosowania procedury przechodzenia grafu G_M w głąb: najpierw usuwamy z G_M wierzchołek (wraz z incydentnymi krawędziami), na którym stoi paczka, a następnie rozpoczynamy przechodzenie G_M od pola zajętego przez magazyniera w stanie u. Krawędź między stanami istnieje, jeśli uda nam się dojść do pola zajętego przez magazyniera w stanie v. Co prawda procedura przechodzenia grafu w głąb działa w czasie liniowym, ale konieczność wykonania jej dla $O(N)$ par stanów prowadzi do czasu kwadratowego.

Bardziej uważna analiza pozwala wyznaczyć krawędzie G_S w znacznie krótszym czasie. Zauważamy, że parę stanów, w których paczka zajmuje to samo pole, należy połączyć krawędzią wtedy i tylko wtedy, gdy w G_M istnieją co najmniej dwie rozłączne ścieżki pomiędzy polami odpowiadającymi pozycjom magazyniera. Jedna ścieżka zawsze istnieje — jest to ścieżka długości 2, wiodąca jedynie poprzez pole zajęte przez paczkę. Jeśli więc ma być możliwość obejścia paczki, musi istnieć w G_M druga ścieżka. Grafy, w których zachodzi taka własność, nazywamy dwuspójnymi.

Definicja 1: Graf $G' = (V', E')$ jest *dwuspójny*, jeśli pomiędzy każdą parą jego wierzchołków istnieją co najmniej dwie łączące je ścieżki, które poza końcami nie mają żadnych wspólnych wierzchołków.

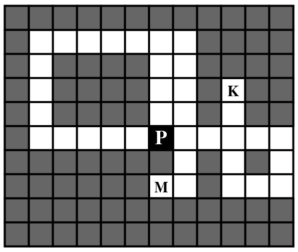
Metoda druga. Rozbijamy graf G_M na *dwuspójne składowe*, tj. na maksymalne podgrafy, z których każdy jest grafem dwuspójnym. Parę stanów u, v z grafu G_S, różniących się jedynie pozycjami magazyniera, łączymy krawędzią wtedy i tylko wtedy, gdy pola zajęte przez magazyniera w u i v należą do jednej dwuspójnej składowej z G_M.

Algorytm wyznaczający dwuspójne składowe polega na sprytnym wykorzystaniu algorytmu przechodzenia grafu wgłąb i działa w czasie liniowym. Nie będziemy go omawiać w tym miejscu, a zainteresowanego czytelnika odsyłamy do [12].

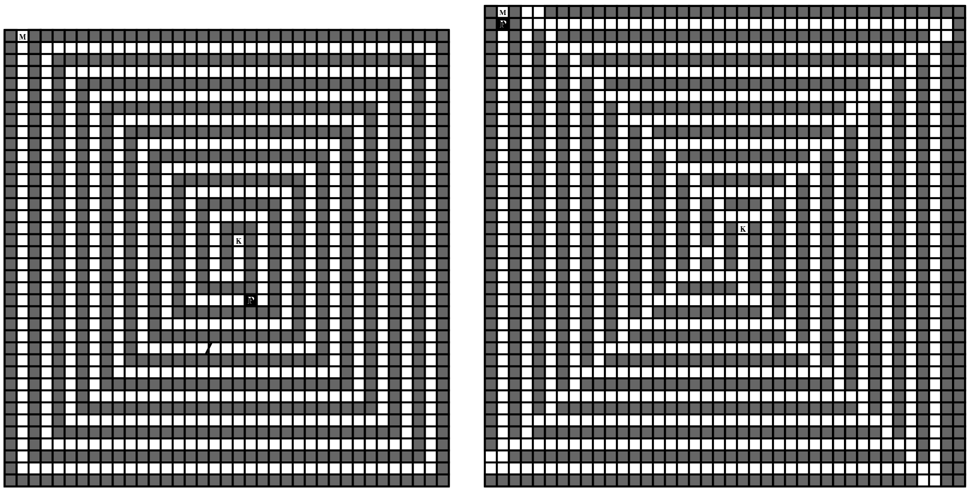
TESTY

Do oceny rozwiązań użyto 18 testów MAG0.IN–MAG17.IN. Test MAG0.IN (rysunek 1) jest testem z treści zadania. Pozostałe można podzielić na kategorie w zależności od kształtu magazynu. Poniżej ilustrujemy kilka z nich.

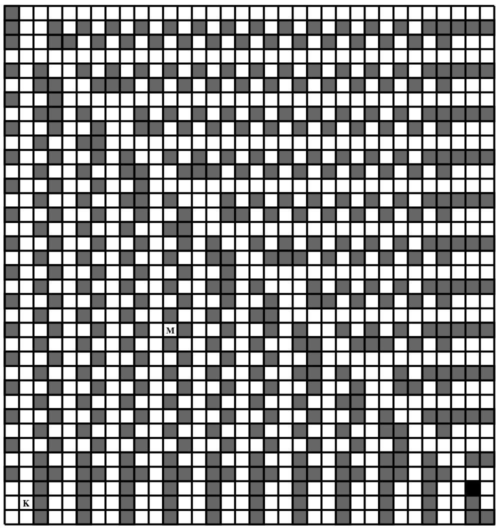
Rys. 1 Test MAG0.IN. Dla przejrzystości pola zajęte przez skrzynie zamalowane są kolorem szarym, pole zajęte przez paczkę — kolorem czarnym, a pola wolne — kolorem białym.



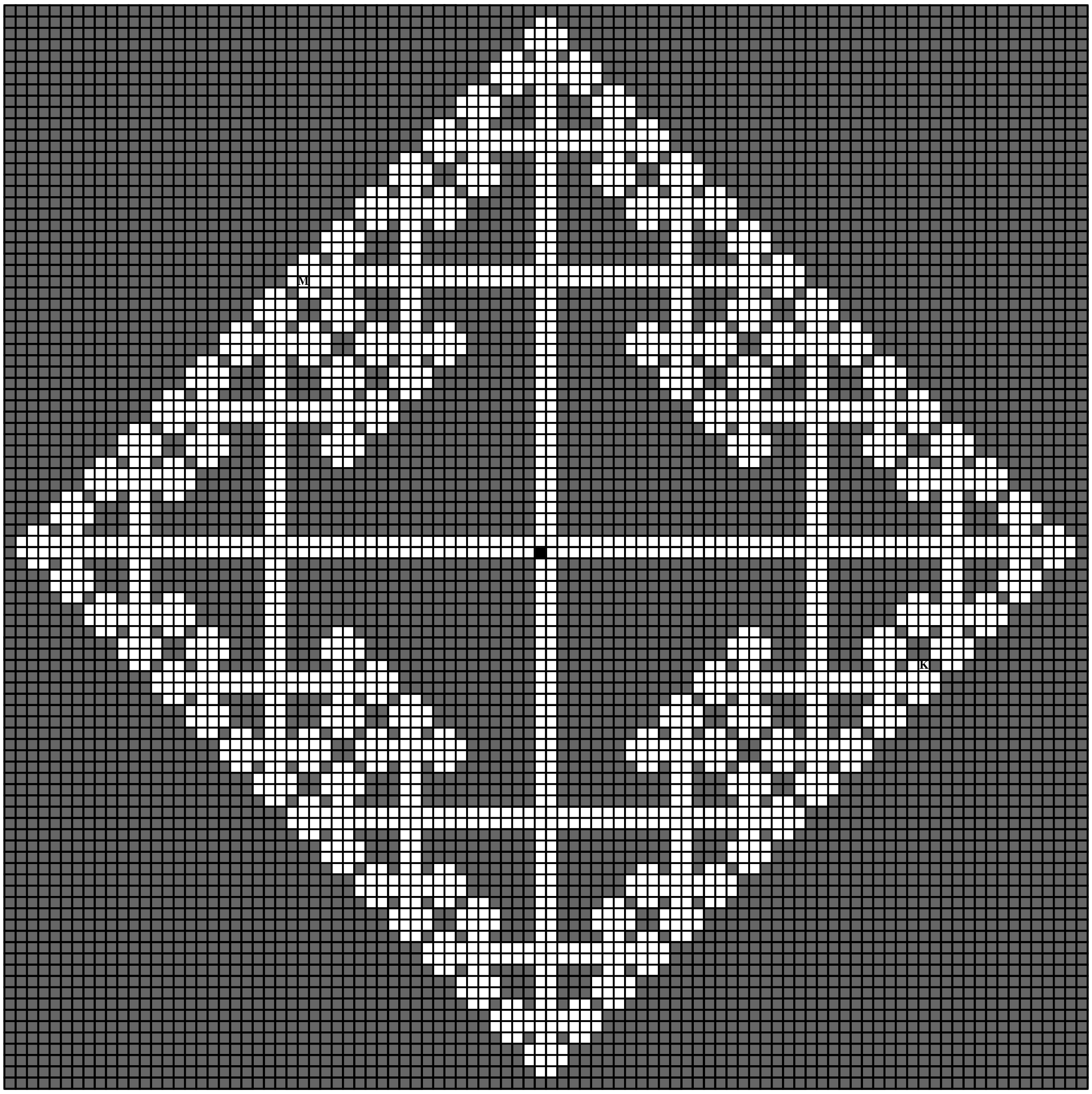
Rys. 2 Testy „spiralne”. Tego typu testami są MAG2.IN-MAG5.IN. Podobnymi testami są też MAG14.IN i MAG15.IN — w nich jednak korytarze mają podwójną szerokość.



Rys. 3 Testy wymuszające zawracanie. Zwróć uwagę, że magazynier będzie musiał przepychać paczkę do wszystkich komór znajdujących się na prawym i dolnym obrzeżu magazynu, ponieważ tylko tam może obejść paczkę i zmienić kierunek pchania. Tego typu testami są MAG16.IN i MAG17.IN.



Rys. 4 Test „fraktalny". Tego typu testami są MAG8.IN i MAG9.IN.



Marcin Sawicki
autor zadania, opracowanie

Marek Pawlicki
program wzorcowy

Mapa

Po nowym podziale administracyjnym Bajtocji przygotowywana jest mapa demograficzna kraju. Z powodów technicznych do kolorowania mapy można użyć tylko kilku kolorów. Mapę należy pokolorować w taki sposób, żeby gminy o zbliżonym zaludnieniu (tj. liczbie mieszkańców) były pokolorowane tym samym kolorem. Dla danego koloru k niech $A(k)$ będzie taką liczbą, że wśród gmin o kolorze k jest tyle samo gmin o zaludnieniu większym lub równym $A(k)$, co gmin o zaludnieniu mniejszym lub równym $A(k)$. Błędem pokolorowania *gminy pokolorowanej kolorem k nazywamy wartość bezwzględną różnicy liczby $A(k)$ i zaludnienia gminy.* Błędem sumarycznym *nazywamy sumę wszystkich błędów pokolorowania. Jak pokolorować mapę, żeby błąd sumaryczny był najmniejszy?*

Zadanie

Napisz program, który:

- *wczyta z pliku wejściowego* `MAP.IN` *zaludnienia gmin Bajtocji,*
- *obliczy minimalny błąd sumaryczny,*
- *zapisze wynik w pliku tekstowym* `MAP.OUT`*.*

Wejście

W pierwszym wierszu pliku wejściowego `MAP.IN` *znajduje się jedna liczba całkowita n równa liczbie gmin Bajtocji, $10 < n < 3000$. W drugim wierszu jest zapisana liczba m kolorów użyta do pokolorowania mapy, $2 \leqslant m \leqslant 10$. W każdym z następnych n wierszy znajduje się po jednej nieujemnej liczbie całkowitej. Są to zaludnienia gmin Bajtocji. Populacja Bajtocji nie przekracza 2^{30}.*

Wyjście

Twój program powinien zapisać w pierwszym i jedynym wierszu pliku wyjściowego `MAP.OUT` *jedną liczbę całkowitą będącą równą minimalnemu błędowi sumarycznemu, jaki można uzyskać przy kolorowaniu mapy.*

Przykład

Dla pliku wejściowego `MAP.IN`*:*

```
11
3
21
14
6
18
10
2
15
12
3
2
2
```

poprawną odpowiedzią jest plik tekstowy `MAP.OUT`*:*
`15`

MODYFIKACJA TREŚCI ZADANIA

Podczas trwania zawodów definicję liczby $A(k)$ zmieniono w następujący sposób:
Liczba $A(k)$ to dowolna liczba rzeczywista taka, że:

- *co najmniej połowa spośród gmin pokolorowanych kolorem k ma $A(k)$ lub więcej mieszkańców,*
- *co najmniej połowa spośród gmin pokolorowanych kolorem k ma $A(k)$ lub mniej mieszkańców,*

ROZWIĄZANIE

Liczba $A(k)$, zdefiniowana w treści zadania, jest *medianą* multizbioru* liczb mieszkańców gmin, oznaczonych na mapie kolorem k. Mediana multizbioru o parzystej liczbie elementów może nie być wyznaczona jednoznacznie. Mediana multizbioru liczb całkowitych nie musi nawet (choć może) być liczbą całkowitą. Zauważmy jednak, że wartość błędu sumarycznego jest dobrze określona mimo to. Po prostu nie zależy od tego, którą z możliwych wartości mediany wybierzemy do obliczeń (Czytelnik zechce to sprawdzić!).

Oznaczmy przez $L[1..n]$ tablicę liczb mieszkańców poszczególnych gmin. Załóżmy, że wartości te są posortowane niemalejąco. Łatwo zauważyć, że minimalny błąd sumaryczny jest osiągany dla takiego pokolorowania mapy, w którym każdy z kolorów jest użyty do pomalowania gmin ze spójnego fragmentu tablicy L. Dlatego wprowadzimy dalsze oznaczenia: niech $W[g,j]$ będzie sumą błędów pokolorowania gmin $g, g+1, \ldots, j$ przy założeniu, że wszystkie one (i tylko one) zostały pokolorowane tym samym kolorem. Czyli $W[g,j] = |L[g]-A| + |L[g+1]-A| + \cdots + |L[j]-A|$, gdzie A jest medianą liczb $L[g], \ldots, L[j]$. Możemy przyjąć np. $A = L\left[\left\lfloor \frac{g+j+1}{2} \right\rfloor\right]$. Przyjmijmy na razie, że znamy wartości $W[g,j]$ dla wszystkich $1 \leqslant g \leqslant j \leqslant n$.

Zbliżamy się do rozwiązania naszego problemu metodą programowania dynamicznego. Niech $s[g,i]$ będzie sumą błędów optymalnego pokolorowania przedziału $g \ldots n$ przy pomocy i kolorów. Oznaczenie to ma sens tylko dla $g \leqslant n-i+1$, zaś dla $1 \leqslant g \leqslant m-i$ nie będzie dla nas użyteczne. Oczywiście $s[g,1] = W[g,n]$. Zachodzi także (co jest dla nas kluczowe) równość

$$s[g,i+1] = \min_{j=g,\ldots,n-i} (W[g,j] + s[j+1,i])\,.$$

W istocie, jeśli przedział $g \ldots n$ został pokolorowany optymalnie $i+1$ kolorami, a pierwszym kolorem pokolorowano gminy od g do j włącznie, to gminy od $j+1$ do n również muszą być pokolorowane optymalnie, tyle że za pomocą i kolorów. Przecież pozostawiając nie zmienione j, za to poprawiając pokolorowanie gmin od $j+1$-szej do n-tej, poprawilibyśmy pokolorowanie całego przedziału $g \ldots n$, wbrew założeniu, że było ono optymalne!

Potrzebujemy znaleźć wartość $s[1,m]$. Teraz już bardzo łatwo ułożyć algorytm, który to zrobi:

```
for g := 1 to n do
  s[g,1] := W[g, n]
for i := 2 to m do
  for g := m − i + 1 to n − i + 1 do begin
    s[g,i] := +∞
    for j := g to n − i + 1 do
      s[g,i] := min(s[g,i], W[g,j] + s[j + 1, i − 1])
  end
end
```

* Multizbiór (ang. *multiset, bag*) jest strukturą danych podobną do zbioru, z tym że elementy mogą w nim występować wielokronie, np. $\{1,3,3,5\}$, $\{1,1,3,5\}$.

Widoczne jest typowe dla programowania dynamicznego wykorzystanie wyników częściowych, czyli wartości z tablicy s, dla obliczenia kolejnych wyników częściowych oraz wyniku ostatecznego.

Algorytm ten działa oczywiście w czasie $O(mn^2)$ i potrzebuje $O(n)$ słów pamięci, gdyż nie musimy przechowywać całej tablicy s, a jedynie dwa ostatnie jej wiersze. Gdybyśmy potrzebowali również odtworzyć znalezione optymalne pokolorowanie, potrzebowalibyśmy dodatkowej tablicy o rozmiarze $m \times n$ do przechowywania tych wartości j, dla których w najbardziej wewnętrznej pętli zostało znalezione minimum.

Jeśli tablica L nie była na wejściu posortowana, to możemy ją posortować w czasie $O(n \log n)$, który oczywiście jest zaniedbywalny z punktu widzenia naszej analizy.

Pozostaje problem znalezienia wartości W. Naiwny algorytm wyznaczania każdego $W[g,j]$ z osobna wprost z definicji (jako odpowiedniej sumy) wymaga czasu $\Omega(n^3)$, co istotnie spowalnia nasz program. Okazuje się, że i tym razem w sukurs przychodzi nam technika programowania dynamicznego: zachodzi zależność

$$W[g,j+1] = W[g,j] + L[j+1] - L\left[\left\lfloor \frac{g+j+1}{2} \right\rfloor\right],$$

która pozwala wyznaczyć całą tablicę W w czasie $O(n^2)$. Zamiast dowodzić tej równości formalnie, proponuję przyjrzeć się rysunkowi 1.

Nietrudno też spostrzec, że nie potrzebujemy trzymać wartości $W[g,j]$ w żadnej tablicy: jeżeli będziemy obliczać je w miarę potrzeby w wewnętrznej pętli algorytmu, to co prawda każdą z nich obliczymy $(m-1)$-kronie, lecz złożoność czasowa całego algorytmu na tym nie ucierpi (nadal będzie wynosić $O(mn^2)$), zaś złożoność pamięciowa pozostanie na poziomie $O(n)$. Przy podanych ograniczeniach na n i m jest to istotne usprawnienie.

INNE ROZWIĄZANIA

Wspomniano już o rozwiązaniu o złożoności czasowej $\Omega(n^3)$, obliczającym W wprost z definicji. Taki algorytm potrzebuje również $\Omega(n^2)$ słów pamięci na przechowywanie tablicy W. Jeśli chcielibyśmy zaoszczędzić pamięć, to koszt wzrósłby do $\Omega(mn^3)$. Można też ułożyć algorytm przeglądający wszystkie możliwe podziały tablicy L na m bloków; działałby on w czasie $\Omega(n^{m-1})$. Jakość poszczególnych algorytmów łatwo ocenić wiedząc, że w Polsce jest około 2500 gmin, a do kolorowania map demograficznych używa się zazwyczaj 7 kolorów.

INNY WARIANT ZADANIA

W praktyce kartograficznej nie minimalizuje się sumy odległości danych od ich mediany, lecz od ich średniej arytmetycznej. Oczywiście wymaga to użycia arytmetyki zmiennopozycyjnej. Zwróćmy uwagę, że dla rozwiązania tego zadania nie musimy modyfikować głównej części algorytmu, a jedynie sposób obliczania wartości $W[g,j]$.

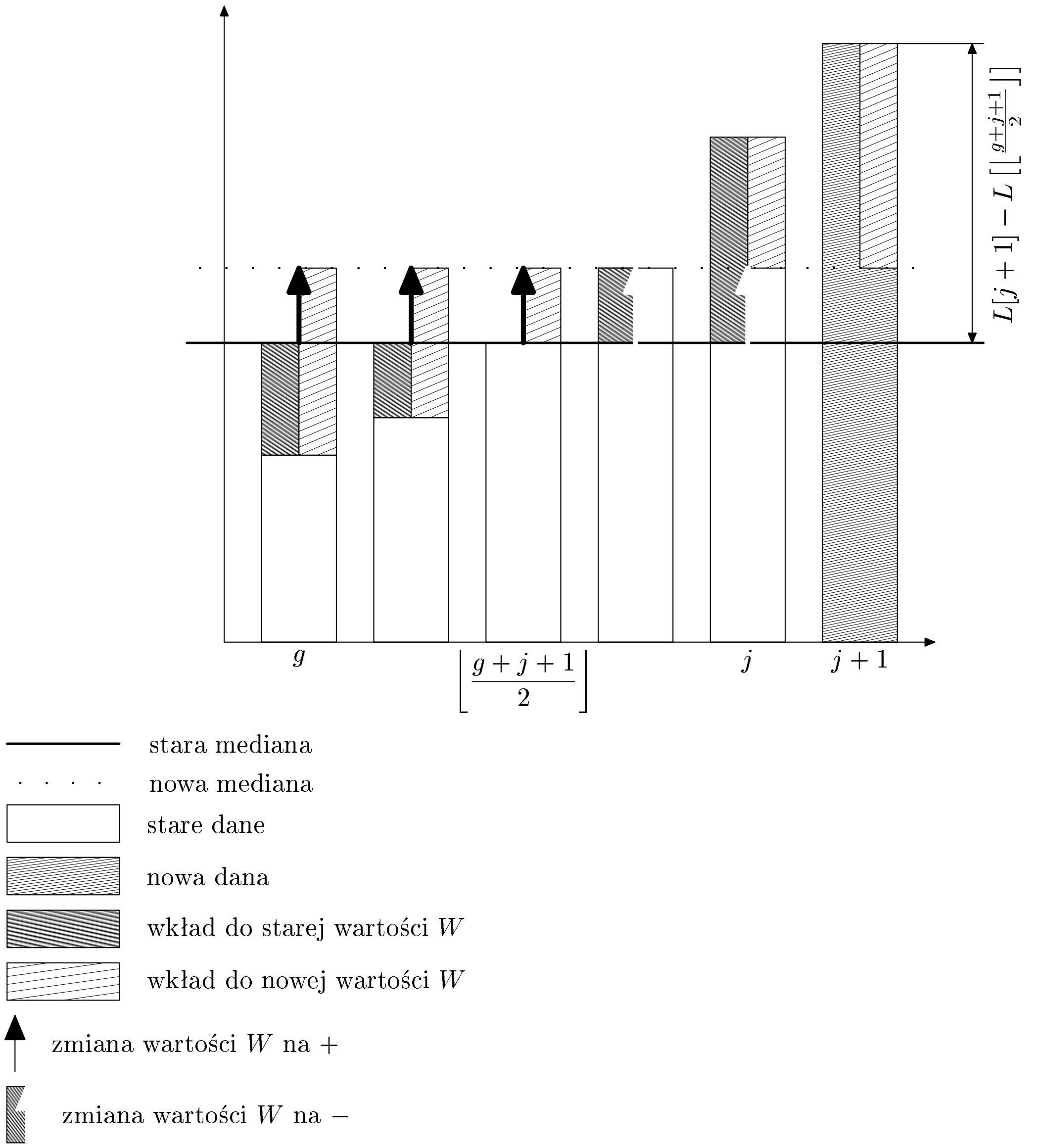
Tym razem również można wyznaczyć wszystkie wartości $W[g,j]$ w łącznym czasie $O(n^2)$ Co więcej tak samo jak poprzednio nie potrzebujemy trzymać ich wszystkich w tablicy. Jednak obliczenie $W[g,j+1]$ na podstawie $W[g,j]$ nie jest, jak sądzę, możliwe w czasie stałym (najkrócej mówiąc, jeśli $L[j+1]$ jest stosunkowo duże, to wartość średnia może przeskoczyć kilka spośród liczb L na raz). Daje się to natomiast zrobić w zamortyzowanym czasie stałym.* Znalezienie odpowiedniego algorytmu może być ciekawym ćwiczeniem.

TESTY

Do oceny rozwiązań użyto 10-ciu testów MAP0.IN–MAP9.IN. Test MAP0.IN był testem z treści zadania. Testy MAP1.IN–MAP5.IN sprawdzały poprawność programu. Był wśród nich zestaw danych o minimalnym rozmiarze, zestaw, w którym wszystkie gminy miały po 1 mieszkańcu, dwa proste testy losowe i jeden,

* Definicję i omówienie pojęcia kosztu zamortyzowanego znajdzie Czytelnik w książce [12], rozdział 18.

Rys. 1 Indukcyjne obliczanie wartości $W[g,j]$.



w którym populacja całej Bajtocji wynosiła $2^{30} - 2$. Następne cztery testy były losowe i coraz większe — zaprojektowane tak, by algorytm działający w czasie $\Omega(n^{m-1})$ przechodził co najwyżej pierwszy z nich, a działający w czasie $\Omega(n^3)$ — co najwyżej dwa pierwsze.

Grzegorz Jakacki
autor zadania, opracowanie

Tomasz Waleń
program wzorcowy

Ołtarze

Według chińskich wierzeń ludowych złe duchy mogą poruszać się tylko po linii prostej. Ma to istotne znaczenie przy budowie świątyń. Świątynie są budowane na planach prostokątów o bokach równoległych do kierunków północ-południe oraz wschód-zachód. Żadne dwa z tych prostokątów nie mają punktów wspólnych. Po środku jednej z czterech ścian jest wejście, którego szerokość jest równa połowie długości tej ściany. W centrum świątyni (na przecięciu przekątnych prostokąta) znajduje się ołtarz. Jeśli znajdzie się tam zły duch, świątynia zostanie zhańbiona. Tak może się zdarzyć, jeśli istnieje półprosta (w płaszczyźnie równoległej do powierzchni terenu), która biegnie od ołtarza w centrum świątyni przez otwór wejściowy aż do nieskończoności, nie przecinając i nie dotykając po drodze żadnej ściany, tej lub innej świątyni.

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `OLT.IN` *opisy świątyń,*
- *sprawdzi, które świątynie mogą zostać zhańbione,*
- *zapisze ich numery w pliku tekstowyn* `OLT.OUT`*.*

Wejście

W pierwszym wierszu pliku wejściowego `OLT.IN` *jest zapisana jedna liczba naturalna* n*,* $1 \leqslant n \leqslant 1000$*, będąca liczbą świątyń.*

W każdym z kolejnych n *wierszy znajduje się opis jednej świątyni (w* i*-tym z tych wierszy opis świątyni numer* i*). Opis świątyni składa się z czterech nieujemnych liczb całkowitych, nie większych niż 8000, oraz jednej litery E, W, S lub N. Pierwsze dwie liczby, to współrzędne północno-zachodniego narożnika świątyni, a dwie następne, to współrzędne przeciwległego, południowo-wschodniego narożnika. Określając współrzędne punktu podajemy najpierw jego długość geograficzną (która rośnie z zachodu na wschód), a następnie — szerokość geograficzną, która rośnie z południa na północ. Piąty element opisu wskazuje ścianę, na której znajduje się wejście do świątyni (E — wschodnią, W — zachodnią, S — południową, N — północną). Kolejne elementy opisu świątyni są pooddzielane pojedynczymi odstępami.*

Wyjście

W kolejnych wierszach pliku tekstowego `OLT.IN` *Twój program powinien zapisać w porządku rosnącym numery świątyń, które mogą zostać zhańbione przez złego ducha, każdy numer w osobnym wierszu. Jeżeli takich świątyń nie ma, to w pliku* `OLT.OUT` *należy zapisać jedno słowo* `BRAK`*.*

Przykład

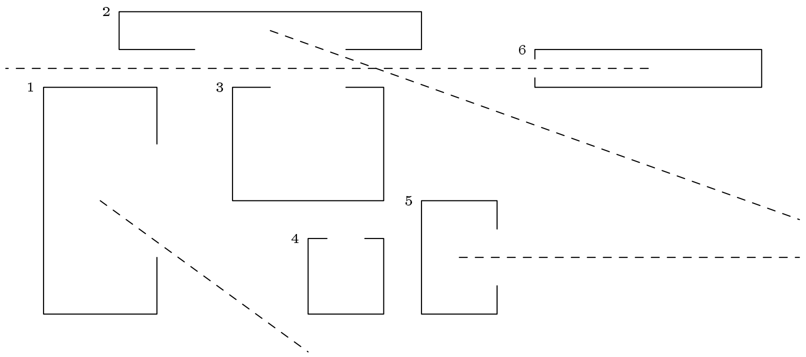
Dla pliku wejściowego `OLT.IN`*:*

```
6
1 7 4 1 E
3 9 11 8 S
6 7 10 4 N
8 3 10 1 N
11 4 13 1 E
14 8 20 7 W
```

poprawną odpowiedzią jest plik tekstowy `OLT.OUT`*:*

```
1
2
5
6
```

Na rysunku pokazano układ świątyń opisany w pliku `OLT.IN`*. Linie przerywane pokazują możliwe drogi inwazji złych duchów.*



ROZWIĄZANIE

Zadanie reprezentuje dziedzinę leżącą na pograniczu algorytmiki i geometrii, tzw. geometrię obliczeniową. Jednym z bodźców, który przyczynił się do jej rozwoju, było pojawienie się problemów „z życia wziętych", związanych m.in. z zagadnieniami transportowymi, analizą obrazu, czy produkcją układów scalonych wysokiej skali integracji. Na taki właśnie z życia wzięty problem natrafiłem w Chinach. Świątynie występujące w zadaniu są bardzo silnym uproszczeniem rzeczywistości. Chińska świątynia przypomina park lub ogród, w którym znajduje się kilka pawilonów mieszczących przedmioty kultu. Świątynia posiada często oś symetrii, na której znajdują się główne zabudowania oraz brama. W poprzek bramy, na terenie świątyni lub poza nią, znajduje się zazwyczaj specjalny mur. Złe duchy i wszelkie „wpływy" poruszają się bowiem, według Chińczyków, wyłącznie po linii prostej. Ściana pełni przeto rolę ekranu zabezpieczającego świątynie przed wpływami świata zewnętrznego. Z podobnych przyczyn do wielu domostw wchodzi się przez wysoki próg lub zza węgła.

Przy wyznaczaniu lokalizacji nowej świątyni Chińczycy posługują się starożytną sztuką *Feng Shui* łączącą elementy wróżbiarstwa, radiestezji i planowania przestrzennego. Nam, nie mającym pojęcia o tej skompilkowanej metodzie, pozostaje rozwiązanie zadania przy pomocy geometrii obliczeniowej.

Skupmy się na tym, w jaki sposób sprawdzić, czy pewna wybrana świątynia S może zostać zhańbiona. Wyobraźmy sobie, że stoimy na ołtarzu tej świątyni ze wzrokiem utkwionym w lewej framudze otworu wejściowego (patrząc z wnętrza świątyni). Następnie zaczynamy powoli obracać się zgodnie z ruchem wskazówek zegara (w prawo). Linia naszego wzroku prowadzi ze świątyni przez otwór drzwiowy i przesuwa się po fragmentach innych świątyń. Natrafienie wzrokiem na prześwit pomiędzy zabudowaniami oznacza odnalezienie potencjalnej drogi inwazji złego ducha.

Niech O oznacza punkt położenia ołtarza w świątyni S. Każdą półprostą o wierzchołku O nazywać będziemy *drogą inwazji*. Mówimy, że droga inwazji jest *zabezpieczona przez świątynię* T, jeżeli ma punkt wspólny ze ścianą świątyni T. Zbiór wszystkich dróg inwazji zabezpieczanych przez świątynię T nazywamy *obszarem zasłanianym przez świątynię* T. *Polem widzenia świątyni* S nazywamy zbiór wszystkich dróg inwazji nie zabezpieczanych przez tę świątynię.

Każdy obszar zabezpieczany przez świątynię jest jednoznacznie wyznaczany przez dwie skrajne drogi inwazji. Jeżeli przyjmiemy uporządkowanie dróg inwazji zgodnie z ruchem wskazówek zegara, to wygodnie pierwszą z tych dróg nazywać *lewym skrajem*, natomiat drugą — *prawym skrajem*.

Niech X będzie zbiorem wszystkich obszarów zabezpieczanych przez świątynie. **Bez szkody możemy ograniczyć się do niepustych obszarów będących przecięciami obszarów z X z polem widzenia świątyni S.** Dalej tylko te obszary będziemy nazywać obszarami zasłanianymi.

Algorytm sprawdzający, czy świątynia S może zostać zhańbiona wygląda następująco:

(1) Dla każdej świątyni różnej od S wyznaczamy lewy i prawy skraj obszaru, który ona zabezpiecza i zapamiętujemy je, jeżeli obszar ma niepuste przecięcie z polem widzenia świątyni S.

(2) Sortujemy skraje zgodnie z porządkiem wyznaczonym przez ruch wskazówek zegara.

(3) Sprawdzamy, czy pierwszy i ostatni skraj pokrywa się odpowiednio z lewym i prawym skrajem pola widzenia (jeśli nie, to istnieje niezabezpieczona droga inwazji).

(4) Ustawiamy licznik na zero i przeglądamy kolejne skraje. Natrafiając na lewy skraj zwiększamy licznik, natrafiając na prawy — zmniejszamy. Napotkawszy sytuację, w której wartość licznika wynosi zero, a kolejny skraj ma kierunek różny od ostatniego skraju, kończymy sprawdzanie (wykryliśmy niezabezpieczoną drogę inwazji).

(5) Jeżeli sprawdzanie nie zakończyło się w poprzednich punktach, to każda droga inwazji jest zabezpieczona.

Optymalna implementacja algorytmu ma złożoność czasową $O(n \log n)$, pochodzącą od operacji sortowania. Ponieważ analogiczne sprawdzenie trzeba wykonać dla każdej świątyni osobno, ogólny koszt czasowy optymalnej implementacji wynosi $O(n^2 \log n)$. Umiejętna implementacja algorytmu pozwala uzyskać koszt pamięciowy $O(1)$.

Program wzorcowy do sortowania wykorzystuje hybrydę algorytmu sortowania przez kopcowanie (Heapsort) oraz przez wstawianie (Insertionsort). Z tego powodu pesymistyczna złożoność dla tej implementacji wynosi $O(n^3)$, jednak dla losowych danych oczekiwany czas pracy algorytmu wynosi $O(n^2 \log n)$. Podobnie, pesymistyczny koszt pamięciowy wynosi $O(n)$, jednak dla danych losowych oczekiwany koszt pamięciowy jest rzędu $\log n$ (co w zasadzie i tak nie ma znaczenia ze względu na niskie ograniczenie n).

Kilku słów wymaga jeszcze omówienie szczegółów związanych z reprezentacją półprostych oraz sprawdzaniem ich położenia. Ponieważ operujemy wyłącznie na półprostych o wspólnym wierzchołku, półprostą można wygodnie reprezentować przez dowolny leżący na niej wektor (pozwala to także uniknąć obliczeń zmiennopozycyjnych). Do ustalenie kolejności półprostych względem porządku wyznaczanego przez kierunek ruchu wskazówek zegara można wykorzystać iloczyn wektorowy (zob. [12]).

INNE ROZWIĄZANIA

Zauważmy, że jeżeli nie musielibyśmy każdorazowo sortować skrajów, to można by przyspieszyć podane rozwiązanie. W istocie istnieje algorytm o złożoności czasowej $O(n^2)$, który dla każdego z n punktów na płaszczyźnie wyznacza uporządkowanie pozostałych $n-1$ punktów w porządku wyznaczonym przez ruch wskazówek zegara. Zastosowanie go do wstępnego posortowania narożników świątyń pozwoliłoby zredukować całkowity koszt czasowy rozwiązania do $O(n^2)$. Niestety algorytm ten jest zbyt skomplikowany, by go tu opisywać, nie wspominając o perspektywach zakodowania go podczas 5-godzinnej sesji zawodów Olimpiady.

TESTY

Do sprawdzenia rozwiązań zawodników użyto 11-tu testów OLT0.IN–OLT10.IN.

- OLT0.IN — test z treści zadania;
- OLT1.IN — mały test poprawnościowy;
- OLT2.IN — test z odpowiedzią „BRAK”;
- OLT3.IN — mały test losowy;
- OLT4.IN–OLT10.in — duże testy wydajnościowe.

Wojciech Rytter
autor zadania, opracowanie

Marcin Mucha
program wzorcowy

Pierwotek abstrakcyjny

Kod genetyczny *pierwotka abstrakcyjnego (*Primitivus recurencis*) jest ciągiem liczb naturalnych* $K = (a_1, \ldots, a_n)$. Cechą *pierwotka nazywamy każdą parę liczb* (l, r), *które występują kolejno w kodzie genetycznym, tzn. istnieje i takie, że* $l = a_i$, $r = a_{i+1}$. *U pierwotków nie występują cechy postaci* (p, p).

Zadanie

Napisz program, który:

- *wczyta listę cech z pliku tekstowego* `PIE.IN`,
- *obliczy długość najkrótszego kodu genetycznego pierwotka zawierającego podane cechy,*
- *zapisze wynik w pliku tekstowym* `PIE.OUT`.

Wejście

W pierwszym wierszu pliku wejściowego `PIE.IN` *zapisana jest jedna dodatnia liczba całkowita n. Jest to liczba różnych cech pierwotka. W każdym z kolejnych n wierszy znajduje się para liczb naturalnych l i r oddzielonych pojedynczym odstępem,* $1 \leqslant l \leqslant 1000$, $1 \leqslant r \leqslant 1000$. *Para* (l, r) *jest jedną z cech pierwotka. Cechy w pliku wejściowym nie powtarzają się.*

Wyjście

Twój program powinien zapisać w pierwszym i jedynym wierszu pliku tekstowego `PIE.OUT` *dokładnie jedną liczbę całkowitą, równą długości najkrótszego kodu genetycznego pierwotka zawierającego cechy z pliku* `PIE.IN`.

Przykład

Dla pliku wejściowego `PIE.IN`:

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

poprawną odpowiedzią jest plik tekstowy `PIE.OUT`:

```
15
```

Wszystkie cechy z pliku `PIE.IN` *są zawarte np. w następującym kodzie genetycznym:*

$$(8, 5, 1, 4, 2, 3, 9, 6, 4, 5, 7, 6, 2, 8, 6)$$

ROZWIĄZANIE

Zadanie ma charakter teoriografowy. Tworzymy graf skierowany G, którego wierzchołkami są liczby występujące w cechach pierwotka. Niech V oznacza zbiór tych liczb. W naszym zadaniu $V \subseteq \{1, \ldots, 1000\}$.

Każda cecha odpowiada jednej krawędzi. Niech E będzie zbiorem krawędzi grafu. Mamy zatem graf $G = (V, E)$, $|V| \leqslant 1000$, $|E| = m$.

Przez *ścieżkę* w grafie G rozumiemy taki ciąg krawędzi, że początek każdej kolejnej krawędzi jest końcem poprzedniej oraz żadna krawędź na ścieżce nie powtarza się (ścieżka może być cyklem). Mówimy, że zbiór ścieżek S *pokrywa* graf G, jeżeli każda krawędź grafu G należy do pewnej ścieżki ze zbioru S.

Obserwacja 1: Niech q będzie minimalną liczbą skierowanych, krawędziowo rozłącznych ścieżek, które pokrywają G. Wtedy rozwiązaniem zadania jest liczba $m + q$.

Zaczniemy od algorytmu A, który konstruuje najmniejszy zbiór ścieżek pokrywających graf G, a następnie składa te ścieżki w kod K, który jest najkrótszym kodem dla danego zestawu cech. Potem pokażemy, że można obliczyć liczbę q tych ścieżek bez faktycznego ich konstruowania (algorytm B).

Oznaczmy przez *path*(v) operację polegającą na znalezieniu nieprzedłużalnej ścieżki o początku w v i usunięciu jej krawędzi z grafu. Taką operację można wykonać startując z v i idąc dowolnie po grafie, jak długo się da, za każdym razem usuwając krawędź, po której przeszliśmy.

Niech $\gamma = (v_1, \ldots, v_{r-1}, v_r, v_1)$ będzie cyklem, a $\gamma' = (v'_1, \ldots, v'_{p-1}, v'_p)$ ścieżką o wspólnym wierzchołku u, $u = v_i = v'_j$, dla pewnych i, j, $1 \leqslant i \leqslant r$, $1 \leqslant j \leqslant p$. Operację wklejania cyklu γ w ścieżkę γ' definiujmy jako operację polegającą na zastąpieniu wierzchołka u na ścieżce γ' przez ścieżkę (cykl) $v_i, v_{i+1}, \ldots, v_r, v_1, \ldots, v_{i-1}, v_i$ i oznaczamy ją przez *paste*(γ,γ').

Oznaczmy przez $d^+(v)$, $d^-(v)$ liczby krawędzi, odpowiednio, wchodzących do i wychodzących z wierzchołka v. Liczby te nazywamy stopniem wejściowym i stopniem wyjściowym wierzchołka. Algorytm A konstruuje najkrótszy kod w sposób zachłanny i wygląda następująco:

```
function DługośćKoduA : integer;
begin
  S := ∅;
  while E ≠ ∅ do begin
    weźmy  dowolny wierzchołek v taki, że d⁺(v) < d⁻(v);
    jeśli takiego wierzchołka nie ma, to weźmy dowolny wierzchołek taki,
    że d⁻(v) > 0;
    γ := path(v);
    if γ jest cyklem i ma wspólny wierzchołek z pewną ścieżką γ' ∈ S
    then
      paste(γ, γ');
    else
      S := S ∪ {γ}
  kod := kod powstały w wyniku sklejenia ścieżek* z S
  DługośćKoduA = |kod|
end
```

Rysunki 1 i 2 przedstawiają działanie algorytmu na przykładzie grafu odpowiadającego 14-tu cechom.

Algorytm A „liczy za dużo”, ponieważ nie interesują nas elementy zbioru S, a tylko ich liczba. Najsłabszą stroną algorytmu jest to, że musimy wczytać wszystkie cechy pierwotka do pamięci. Tak więc potrzebujemy pamięci rzędu m, a może się zdarzyć, że np. $|V| = 1000$, $m = 990000$.

Pokażemy teraz algorytm, w którym wystarczy pamięć rzędu liczby wierzchołków. Algorytm ten opiera się na analizie spójnych składowych grafu cech.

Słabo spójną składową (w skrócie: *składową*) grafu skierowanego nazywamy każdy maksymalny (w sensie zawierania) podzbiór jego wierzchołków taki, że między dowolnymi dwoma jego elementami istnieje ścieżka nieskierowana (zapominamy o orientacji krawędzi). Na rysunku 1 przedstawiono trzy spójne składowe przykładowego grafu.

Zdefiniujmy *stopień niezrównoważenia wierzchołka* v: $stn(v) = |d^+(v) - d^-(v)|$. Przez *stopień niezrównoważenia grafu* G (ozn. $stn(G)$) będziemy rozumieć sumę stopni niezrównoważenia jego wierzchołków. Mówimy, że graf jest zrównoważony gdy jego stopień niezrównoważenia wynosi zero.

* Tutaj ścieżki traktujemy jako ciągi wierzchołków

Obserwacja 2: Załóżmy, że graf G składa się z jednej spójnej składowej i $stn(G) \neq 0$. Wtedy w funkcji każdy obliczony cykl zostaje wklejony w ścieżkę, a każda ścieżka dodana do S zmniejsza stopień niezrównoważenia grafu o 2. Zatem po zakończeniu działania algorytmu $|S| = stn(G)/2$.

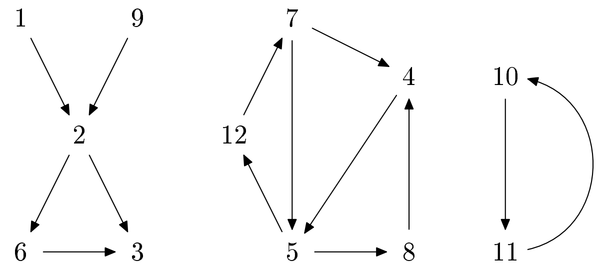
Z obserwacji tej wynika następujący wzór, w którym $G_1,\ G_2, \dots G_r$ oznaczają składowe grafu G:

$$\text{dł. najkrótszego kodu} = \sum_{i=1}^{r} \max\{stn(G_i)/2,\ 1\} \tag{1}$$

W algorytmie wzorcowym liczymy $deficyt(v) = \max\{d^+(v) - d^-(v), 0\}$. Przez *deficyt grafu* rozumiemy sumę deficytów jego wierzchołków. Wtedy można zastąpić wzór (1) następującym równoważnym wzorem:

$$\text{dł. najkrótszego kodu} = \sum_{i=1}^{r} \max\{deficyt(G_i),\ 1\} \tag{2}$$

Rys. 1 Graf reprezentuje 14 cech oraz ścieżki i cykle skonstruowane w algorytmie zachłannego chodzenia.



P1: 1 ➤ 2 ➤ 6 ➤ 3

P2: 9 ➤ 2 ➤ 3

P3: 7 ➤ 4 ➤ 5 ➤ 8 ➤ 4

C1: 7 ➤ 5 ➤ 12

C2: 10 11

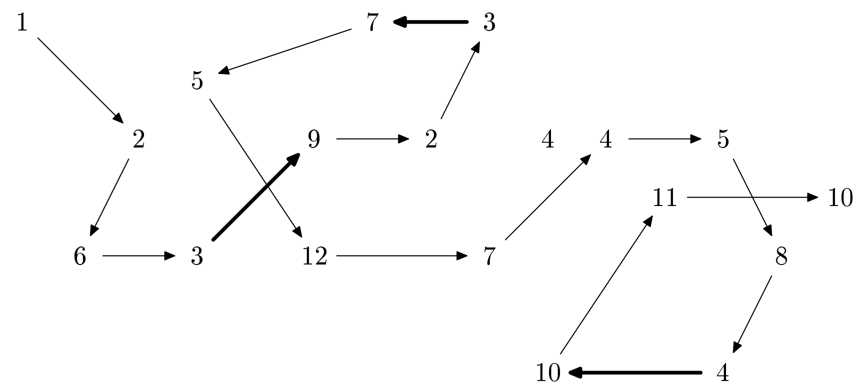
tablica stopni

i	1	2	3	4	5	6	7	8	9	10	11
$d^-[i]$	1	2	0	1	2	1	2	1	1	1	1
$d^+[i]$	0	2	2	2	2	1	1	1	0	1	1

Korzystając z powyższych wzorów konstruujemy algorytm B, który z pliku wczytuje każdą cechę tylko raz, ale nie zapamiętuje jej, a jedynie aktualizuje tablice stopni d^+ i d^- oraz informację o składowych grafu rozpiętego na dotychczas wczytanych krawędziach (cechach).

```
function DługośćKoduB : integer
begin
  for v := 1 to 1000 do begin
    d+[v] := 0;
    d-[v] := 0
  end
```

Rys. 2 Minimalny kod pierwotka abstrakcyjnego dla zbioru cech odpowiadającego grafowi z poprzedniego rysunku. Końcowy ciąg jest złożeniem ścieżek P_1, P_2, P_3 i cykli C_1, C_2. Specjalnie dodanymi połączeniami są $(3,9)$, $(3,7)$, $(4,10)$. Cykl C_1 został wklejony w ścieżkę P_3, natomiast cykl C_2 występuje samodzielnie. Krawędzie pogrubione zostały dodane w wyniku scalania ścieżek.



  { Graf nie zawiera żadnych krawędzi i składa się z 1000
    jednowierzchołkowych składowych. }
  while (nie wczytano wszystkich cech) **do begin**
    wczytaj kolejną cechę (u, v);
    $d^-[u] := d^-[u] + 1$; $d^+[v] := d^+[v] + 1$;
    if u i v należą do różnych składowych
    then połącz te składowe w jedną
  end
  DługośćKoduB := oblicz wynik korzystając ze wzoru (1) lub (2)
end

Żeby efektywnie zaimplementować powyższy algorytm musimy szybko umieć stwierdzać, do jakiej składowej należy dany wierzchołek i łączyć różne składowe w jedną. Jest to klasyczny problem sumowania zbiorów rozłącznych (ang. *find-union*). Różne efektywne rozwiązania tego problemu znajdzie czytelnik w książkach [9] oraz [12]. W programie wzorcowym zastosowano metodę łączenia wg rangi z kompresją ścieżek [12]. Tak zaimplementowany algorytm działa w czasie prawie liniowym ze względu na m i używa niewielkiej ilości pamięci (kilka tablic długości 1000).

TESTY

Do sprawdzenia rozwiązań zawodników użyto 12-tu testów PIE0.IN–PIE11.IN. Test PIE0.IN był testem z treści zadania. Testy PIE1.IN–PIE6.IN to testy poprawnościowe. Testy PIE7.IN–PIE11.IN to testy sprawdzające wydajność (czasową i pamięciową) rozwiązań.

Marcin Kubica
autor zadania, opracowanie

Marcin Mucha
program wzorcowy

Trójkolorowe drzewa binarne

Drzewo *składa się z wierzchołka, do którego podczepiono zero, jedno lub dwa poddrzewa, zwane dziećmi.* Specyfikacją *drzewa nazywamy ciąg cyfr. Jeżeli drzewo składa się z wierzchołka, do którego podczepiono:*

- *zero dzieci, to jego specyfikacja jest ciągiem jednoelementowym, którego jedynym elementem jest cyfra '0';*
- *jedno dziecko, to jego specyfikacja jest ciągiem rozpoczynającym się od cyfry '1'; po której następuje specyfikacja dziecka;*
- *dwoje dzieci, to jego specyfikacja jest ciągiem rozpoczynającym się od cyfry '2', po której następuje najpierw specyfikacja jednego, a potem drugiego dziecka.*

Każdy wierzchołek drzewa trzeba pomalować na czerwono, zielono lub niebiesko. Należy jednak trzymać się dwóch zasad:

- *wierzchołek i jego dziecko nie mogą być pomalowane na ten sam kolor,*
- *jeżeli wierzchołek ma dwoje dzieci, to muszą one być pomalowane różnymi kolorami.*

Ile wierzchołków można pomalować na zielono?

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `TRO.IN` *specyfikację drzewa,*
- *obliczy maksymalną i minimalną liczbę wierzchołków, które można pomalować na zielono,*
- *zapisze wyniki w pliku tekstowym* `TRO.OUT`*.*

Wejście

Pierwszy i jedyny wiersz pliku wejściowego `TRO.IN` *zawiera słowo o długości nie przekraczającej 10000 znaków, będące specyfikacją pewnego drzewa.*

Wyjście

Twój program powinien zapisać w pierwszym i jedynym wierszu pliku wyjściowego `TRO.OUT` *dokładnie dwie liczby całkowite oddzielone pojedynczym odstępem, odpowiednio maksymalną i minimalną liczbą wierzchołków, które można pomalować na zielono.*

Przykład

Dla pliku wejściowego `TRO.IN`*:*
`1122002010`
poprawną odpowiedzią jest plik tekstowy `TRO.OUT`*:*
`5 2`

ROZWIĄZANIE

Drzewo, zgodnie ze swoją definicją, może zawierać w sobie poddrzewa. Zastanówmy się, jaka może być minimalna/maksymalna liczba wierzchołków danego koloru w drzewie w zależności od kształtu tego drzewa i możliwej liczby wierzchołków danego koloru w jego poddrzewach. W treści zadania jest mowa o kolorze zielonym. Zauważmy jednak, że wybór konkretnego koloru nie ma znaczenia dla wyniku —

wszystkie kolory są traktowane tak samo. Jeśli jednak wiemy, jakiego koloru jest korzeń drzewa, to minimalna/maksymalna liczba wierzchołków danego koloru zależy od tego, czy jest to taki sam kolor, jak kolor korzenia, czy też inny kolor. Możemy więc uogólnić problem i dla zadanego drzewa wyliczyć cztery liczby: minimalną/maksymalną liczbę wierzchołków, które można pokolorować danym kolorem, takim samym/innym niż kolor korzenia. Dla każdego wierzchołka v naszego drzewa wyznaczymy cztery liczby opisujące poddrzewo o korzeniu w tym wierzchołku: a_v, b_v, c_v, d_v, gdzie a_v to minimalna liczba wierzchołków o tym samym kolorze co v, b_v to maksymalna liczba wierzchołków o tym samym kolorze co v, c_v to minimalna liczba wierzchołków tego samego koloru, innego niż kolor v, d_v to maksymalna liczba wierzchołków tego samego koloru, innego niż kolor v. Liczby te możemy wyliczyć w następujący sposób:

- Załóżmy, że v jest liściem (tzn. nie ma dzieci). Wówczas oczywiście $a_v = b_v = 1$ i $c_v = d_v = 0$.
- Załóżmy, że v ma jedno dziecko (o korzeniu) v'. Kolor v' musi się różnić od koloru v. Stąd $a_v = c_{v'} + 1$ oraz $b_v = d_{v'} + 1$. Kolor, który różni się od koloru v może być albo kolorem v', albo trzecim kolorem — różnym od kolorów v i v'. Stąd $c_v = \min(a_{v'}, c_{v'})$ oraz $d_v = \max(b_{v'}, d_{v'})$.
- Załóżmy, że v ma dwoje dzieci (o korzeniach) v' i v''. Wówczas v, v' i v'' muszą mieć trzy różne kolory. Stąd $a_v = c_{v'} + c_{v''} + 1$, $b_v = d_{v'} + dv'' + 1$, $c_v = \min(a_{v'} + c_{v''}, c_{v'} + a_{v''})$ oraz $d_v = \max(b_{v'} + d_{v''}, d_{v'} + b_{v''})$.

Zgodnie z powyższymi wzorami możemy wyznaczać liczby a_v, b_v, c_v, d_v idąc w górę drzewa, począwszy od liści, aż do korzenia. Jeżeli v jest korzeniem całego drzewa, to wynikiem są liczby $\min(a_v, c_v)$ i $\max(b_v, d_v)$

Powyższy algorytm można zaimplementować na kilka sposobów. Najprostszy wydaje się program zawierający rekurencyjną procedurę, która wczytuje specyfikację drzewa i daje w wyniku liczby a_v, b_v, c_v i d_v. Format wejścia jest taki, że każde wywołanie takiej procedury wczytuje z pliku wejściowego opis odpowiedniego poddrzewa. Wadą takiego rozwiązania jest to, że dla drzew o dużej wysokości może nastąpić przepełnienie stosu.

Pamięć potrzebną na stos można zmniejszyć konstruując drzewo, w którego węzłach znajdują się wyliczane wartości. Wówczas procedura rekurencyjna wyliczająca wartości przechowywane w węzłach może mieć tylko jeden argument, nie mieć zmiennych lokalnych i zajmować istotnie mniej pamięci na stosie. Mimo to, dla danych postaci

$$\underbrace{1\ldots1}_{9\,999\text{razy}}\,0$$

nastąpi przepełnienie stosu.

Problemy z wielkością stosu można rozwiązać konstruując własny stos na wyliczane wartości i przerabiając procedurę rekurencyjną na iteracyjną. Możemy też najpierw wczytać specyfikację drzewa do tablicy, a następnie wyliczyć wynik iteracyjnie, korzystając z pomocniczego stosu.

Potraktujmy 0 jako stałą reprezentującą czwórkę $\langle 1, 1, 0, 0\rangle$, a 1 i 2, odpowiednio, jako jedno- i dwuargumentową operację wyliczającą czwórkę liczb $\langle a_v, b_v, c_v, d_v\rangle$ dla wierzchołka na podstawie czwórek liczb opisujących jego dzieci. Zauważmy wówczas, że format danych wejściowych jest wyrażeniem zapisanym w notacji polskiej. (W notacji polskiej najpierw zapisujemy operator, a potem jego argumenty. Np. $* + 3\ 2 - 3\ 1$ to $(3+2)*(3-1)$.) Jeżeli więc odwrócimy zapis danych wejściowych, to uzyskamy wyrażenie zapisane w odwrotnej notacji polskiej (z odwróconą kolejnością argumentów 2-ki, co nie zmienia wyniku). (W odwrotnej notacji polskiej najpierw zapisujemy argumenty, a potem operator. Np. $3\ 2 + 3\ 1 - *$ to $(3 + 2) * (3 - 1)$.) Wartość takich wyrażeń możemy wyliczać przetwarzając od lewej do prawej kolejne symbole w następujący sposób:

- jeśli dany symbol jest stałą, to włóż na stos wartość reprezentowaną przez tę stałą,
- jeśli dany symbol jest operacją, to zastosuj ją do wartości zdjętych z wierzchołka stosu, po czym wynik włóż na stos.

Po przetworzeniu całego wyrażenia na stosie powinna znajdować się jedna wartość będąca wynikiem wyrażenia.

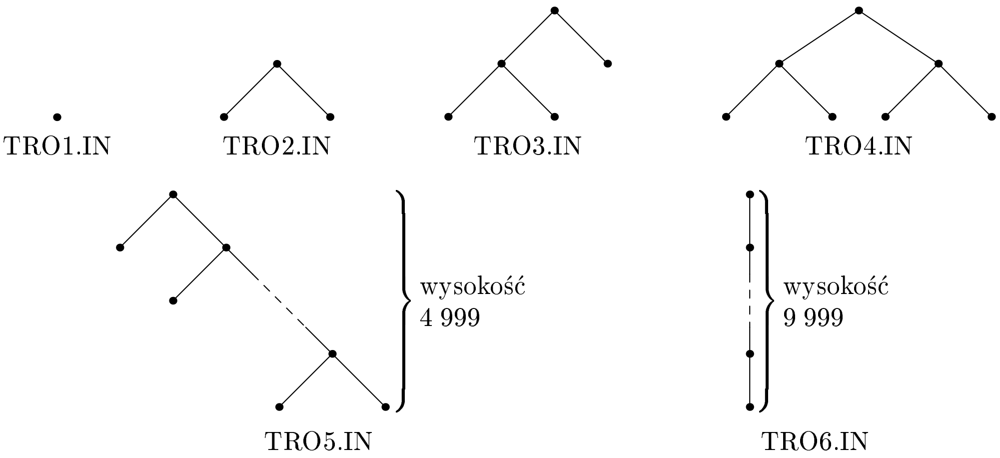
Rozwiązanie takie zastosowano w programie załączonym na dyskietce. Poniżej przedstawiamy główną procedurę wyliczającą liczby a_v, b_v, c_v i d_v dla korzenia całego drzewa. Korzysta ona z tablicy *data* zawierającej dane wejściowe długości n, oraz ze stosu zaimplementowanego jako jednokierunkowa lista rekordów.

```
type
  stack = ↑stack_node;
  stack_node = record
    min_z, min_nz, max_z, max_nz : word;
    prev : stack;
  end;
procedure gen_ziel (var s : stack);
var
  i : word;
  p, q, r : stack;
begin
  for i := n downto 1 do begin
    new(p);
    case data[i] of
      '0' : begin
        p↑.min_nz := 0; p↑.min_z := 1;
        p↑.max_nz := 0; p↑.max_z := 1;
      end;
      '1' : begin
        q := pop(s);
        p↑.min_nz := min(q↑.min_nz, q↑.min_z);
        p↑.min_z := q↑.min_nz + 1;
        p↑.max_nz := max(q↑.max_nz, q↑.max_z);
        p↑.max_z := q↑.max_nz + 1;
        dispose(q);
      end;
      '2' : begin
        q := pop(s);
        r := pop(s);
        p↑.min_nz :=
          min(q↑.min_nz + r↑.min_z, q↑.min_z + r↑.min_nz);
        p↑.min_z := q↑.min_nz + r↑.min_nz + 1;
        p↑.max_nz :=
          max(q↑.max_nz + r↑.max_z, q↑.max_z + r↑.max_nz);
        p↑.max_z := q↑.max_nz + r↑.max_nz + 1;
        dispose(q); dispose(r)
      end
    end;
    push(s, p)
  end
end;
```

Zastanówmy się, czy nie można rozwiązać tego zadania znajdując pokolorowanie wierzchołków o najmniejszej i największej liczbie zielonych wierzchołków. Zauważmy, że w przypadku wierzchołków mających jedno lub dwoje dzieci, jeżeli kolor danego wierzchołka jest określony, to jego dzieci możemy pokolorować na dwa różne sposoby. Ponadto w całym drzewie liści jest o jeden więcej niż wierzchołków mających dwoje dzieci. Jeżeli więc n jest liczbą wierzchołków w drzewie, to przynajmniej $\lceil (n-1)/2 \rceil$ wierzchołków ma jedno lub dwoje dzieci. Stąd liczba możliwych pokolorowań jest wykładnicza ze względu na rozmiar drzewa. Tak więc programy sprawdzające wszystkie kolorowania drzewa dają wyniki w rozsądnym czasie jedynie dla bardzo małych drzew. Okazuje się też, że nie da się wyznaczyć kolorowania o maksymalnej lub minimalnej liczbie zielonych wierzchołków w sposób zachłanny. Tak więc kluczem do rozwiązania są zależności między liczbami a_v, b_v, c_v i d_v.

TESTY

Do sprawdzenia rozwiązań zawodników użyto 13-tu testów TRO0.IN–TRO12.IN. Test TRO0.IN był testem z treści zadania. Testy TRO1.IN–TRO4.IN to małe testy poprawnościowe, a TRO5.IN i TRO6.IN to duże testy poprawnościowe. Zadaniem testów TRO1.IN–TRO4.IN było zbadanie ogólnej poprawności programu. Testy TRO5.IN i TRO6.IN badały, czy w programach nie występuje błąd przepełnienia stosu. Testy te przedstawiono na poniższych rysunkach. Testy TRO7.IN–TRO12.IN to testy losowe różnej wielkości. Ich zadaniem było zbadanie efektywności rozwiązań.



nr testu	l. wierzch.	maks. l. w. ziel.	min. l. w. ziel.
1	1	1	0
2	3	1	1
3	5	2	1
4	7	3	2
5	9 999	4 999	2 500
6	10 000	5 000	0
7	50	22	12
8	500	222	99
9	5 000	2 110	1 275
10	10 000	4 707	1 465
11	10 000	4 359	2 330
12	10 000	4 147	2 655

Grzegorz Jakacki
autor zadania, opracowanie

Marcin Mucha
program wzorcowy

Woda

Na prostokątnej szachownicy składającej się z $n \times m$ pól ustawiono $n \cdot m$ prostopadłościanów — na każdym polu jeden prostopadłościan. Podstawa każdego prostopadłościanu pokrywa się z jednym polem szachownicy i ma powierzchnię jednego cala kwadratowego. Prostopadłościany na sąsiednich polach ściśle przylegają do siebie i nie tworzą żadnych szczelin. Na tę konstrukcję spadł ulewny deszcz. W niektórych miejscach utworzyły się zastoiska wody.

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `WOD.IN` *rozmiary szachownicy oraz wysokości prostopadłościanów ustawionych na poszczególnych polach,*
- *obliczy maksymalną objętość wody, która po deszczu może pozostać w zastoiskach,*
- *zapisze wyniki w pliku tekstowym* `WOD.OUT`*.*

Wejście

W pierwszym wierszu pliku wejściowego `WOD.IN` *są zapisane dwie dodatnie liczby całkowite n i m, $1 \leqslant n \leqslant 100$, $1 \leqslant m \leqslant 100$. Są to rozmiary szachownicy. W każdym z kolejnych n wierszy znajduje się m liczb całkowitych z przedziału [1..10000]: i-ta liczba w j-tym wierszu jest wyrażoną w calach wysokością prostopadłościanu stojącego na przecięciu i-tej kolumny i j-tego wiersza szachownicy.*

Wyjście

Twój program powinien zapisać w pierwszym i jedynym wierszu pliku wyjściowego `WOD.OUT` *jedną liczbę całkowitą równą maksymalnej objętości wody (wyrażoną w calach sześciennych), która może zebrać się w zastoiskach konstrukcji.*

Przykład

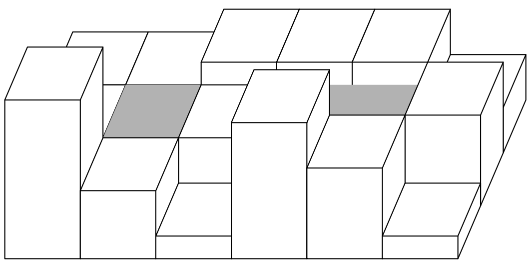
Dla pliku wejściowego `WOD.IN`*:*

```
3 6
3 3 4 4 4 2
3 1 3 2 1 4
7 3 1 6 4 1
```

poprawną odpowiedzią jest plik tekstowy `WOD.OUT`*:*

```
5
```

Poniższy rysunek przedstawia szachownicę po deszczu.



ROZWIĄZANIE

Zacznijmy od ustalenia terminologii. *Wysokością* pola nazwiemy wysokość stojącego na nim prostopadłościanu. *Brzegiem* szachownicy nazwiemy zbiór tych pól, które mają boki wspólne z mniej niż czterema innymi polami.

Wyobraźmy sobie szachownicę jako pasmo górskie z kraterami, które powoli zalewa powódź. Woda podnosząc się zaczyna zalewać stoki gór położone przy brzegu szachownicy, ale do pewnego momentu kratery pozostają suche, chociaż ich dna znajdują się poniżej poziomu wody. Gdy wreszcie poziom wody zrównuje się z najniższym punktem korony jakiegoś krateru (punktem przelewowym), woda momentalnie zalewa krater. Zauważmy, że jeżeli woda przykryje szachownicę, po czym zupełnie opadnie, to poziom wody w każdym kraterze będzie dokładnie taki sam, jak w momencie jego zalania (zaniedbujemy parowanie). Oczywiście wszystko jedno, czy szachownicę zaleje powódź, czy deszcz.

Zaprojektujemy algorytm, który wykorzystuje powyższe obserwacje. Oznaczmy przez $h[p]$ wysokość pola p, a przez $l[p]$ *poziom wody* nad polem p po deszczu (tzn. odległość zwierciadła wody od szachownicy nad polem p). Jeżeli nad polem nie tworzy się zastoisko, to przyjmujemy $l[p] = h[p]$.

Zauważmy, że wartość $l[p]$ można interpretować jako poziom wody w momencie zalania. Dla pól, które leżą na stokach „odpływowych" jest to po prostu ich wysokość. Pola sortujemy niemalejąco względem wysokości i przeglądamy w tej kolejności (co odpowiada zalewaniu szachownicy przez podnoszącą się wodę). Jeżeli napotkamy pole „odpływowe" p, przez które woda może przelać się do krateru, „zalewamy" krater (tj. wszystkim polom p' krateru ustawiamy $l[p'] = h[p]$). Obszar krateru wyznaczamy na podstawie wysokości pól, jako maksymalny spójny obszar pól zawierający p, o wysokościach nie przekraczających $h[p]$. Zauważmy, że każde pole leżące na stoku „odpływowym" można traktować tak, jakby było „przelewowe". Co jednak robić, jeżeli przeglądając pola natkniemy się na pole leżące na dnie krateru? Nic. Wartość l dla tych pól zostanie wpisana w momencie zalania krateru. Algorytm przedstawia się następująco:

(1) Posortuj pola niemalejąco względem wysokości.

(2) Kolejno dla każdego pola p:

- jeżeli pole p należy do brzegu szachownicy lub sąsiaduje z polem p', dla którego obliczyliśmy już $l[p']$, to $l[p] := h[p]$ i „zalej krater" za pomocą algorytmu przeszukiwania grafu (np. DFS lub BFS*);
- w pozostałych przypadkach nie rób nic.

(3) Oblicz $\Sigma_p l[p] - h[p]$.

Zajmijmy się poprawnością opisanego algorytmu. *Ścieżką* nazwiemy taki ciąg pól, że każde dwa kolejne pola w tym ciągu mają wspólny bok, natomiast *wysokością ścieżki* — maksimum z wysokości jej pól.

Rozważmy ścieżki łączące pole p z brzegiem szachownicy. Spośród tych ścieżek wyróżnijmy taką, która ma minimalną wysokość i oznaczmy tę wysokość przez $l'[p]$. Widać, że:

- poziom wody na polu nie może być większy niż $l'[p]$, gdyż woda odpłynęłaby wyróżnioną ścieżką,
- poziom wody na polu nie może być mniejszy niż $l'[p]$, ponieważ wszystkie ścieżki, którymi woda mogłaby ujść z pola, mają wysokość przynajmniej $l'[p]$.

Tak więc poziom wody na polu p jest równy $l'[p]$. Jako ćwiczenie dla Czytelnika pozostawiamy udowodnienie faktu, że dla dowolnego pola p wielkość $l[p]$ obliczana przez opisany algorytm jest równa $l'[p]$.

Sortowanie pól można zrealizować w czasie $O(nm \log(nm))$, każdorazowe przeszukiwanie grafu można wykonać w czasie proporcjonalnym do rozmiaru zalewanego krateru. Łączny czas potrzebny na zalanie wszystkich kraterów nie przekroczy zatem $O(nm)$. Sumaryczny koszt czasowy algorytmu wynosi $O(nm \log(nm))$ i nie jest liniowy jedynie z powodu konieczności posortowania pól.

Czytelnik zechce sprawdzić, że ograniczenie algorytmu do przeglądania jedynie tych pól, dla których wartość l nie została obliczona, nie poprawia asymptotycznej złożoności czasowej.

---

* Opis algorytmów BFS i DFS można znaleźć w [12]

Złożoność pamięciowa algorytmu wynosi $O(nm)$. Sortowanie można zrealizować w pamięci $O(1)$ (tzn. używając tylko stałej liczby komórek pamięci ponad rozmiar danych wejściowych), natomiast przeszukiwanie grafu i składowanie wartości l wymaga pamięci rozmiaru $O(nm)$.

Implementacja wzorcowa korzysta z algorytmu sortowania przez kopcowanie (Heapsort) oraz algorytmu przeszukiwania grafu wszerz (BFS).

INNE ROZWIĄZANIA

Narzucającym się rozwiązaniem jest symulacja spływu wody. Oznaczmy przez h_{max} maksimum z wysokości pól szachownicy. Wiadomo, że dla dowolnego pola p zachodzi $h[p] \leqslant l'[p] \leqslant h_{max}$. Można skonstruować algorytm, który będzie wyznaczał wartości l' dla wszystkich pól metodą kolejnych przybliżeń. Dla każdego pola przechowywać będziemy wartość $w[p]$, początkowo równą h_{max}, która w miarę postępu obliczeń będzie „poprawiana":

- jeśli p leży na brzegu szachownicy, to $w[p] := h[p]$;
- jeśli pole ma sąsiada p' takiego, że $w[p'] < w[p]$, to $w[p] := max(h[p], w[p'])$.

Algorytm polega na „poprawianiu" wartości w dopóty, dopóki istnieje pole p, dla którego można „poprawić" wartość $w[p]$.

Jeżeli będziemy za każdym razem przeglądać wszystkie wierzchołki, to algorytm może wymagać $O(nm)$ „poprawek" dla całej planszy. W takim przypadku czas działania algorytmu wyniósłby $O((nm)^2)$.

TESTY

Do sprawdzania rozwiązań zawodników użyto 12-tu testów WOD0.IN–WOD12.IN:

- WOD0.IN — test z treści zadania;
- WOD1.IN — jeden wiersz;
- WOD2.IN — jedna kolumna;
- WOD3.IN — spirala na planszy 10×10;
- WOD4.IN — spirala na planszy 40×40;
- WOD5.IN — spirala na planszy 100×100;
- WOD6.IN — mały test poprawnościowy;
- WOD7.IN — maksymalny test poprawnościowy;
- WOD8.IN–WOD12.IN — testy losowe.

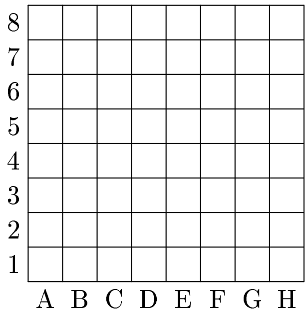
X Międzynarodowa Olimpiada Informatyczna IOI'98, Setubal, wrzesień 1998

teksty zadań

Krzysztof Diks, Andrzej Walat
przekład

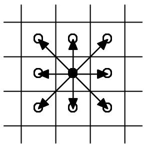
Camelot

Wieki temu Król Artur i Rycerze Okrągłego stołu spotykali się każdego pierwszego dnia roku, aby celebrować swoją przyjaźń. Na pamiątkę tych wydarzeń zaproponowano grę planszową dla jednego gracza, w której figury reprezentują jednego króla i wielu rycerzy. Na początku gry figury rozmieszczane są na różnych polach planszy, każda na innym. Plansza jest tablicą o wymiarach 8 × 8 kwadratowych pól.



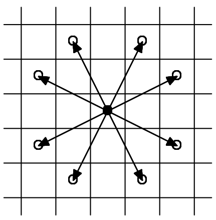
Rysunek 1. Plansza.

Król może się poruszać na każde pole, z • na ∘, jak pokazano na rysunku 2, ale nie może wyjść poza planszę.



Rysunek 2. Wszystkie możliwe ruchy króla.

Rycerz może skoczyć z pola • na każde z pól ∘, jak to widać na rysunku 3, ale nie może wyjść poza planszę.



Rysunek 3. Wszystkie możliwe ruchy rycerza.

W czasie gry gracz może umieścić więcej niż jedną figurę na tym samym polu. Przyjmuje się, że pola planszy są dostatecznie wielkie, by żadna figura nie była przeszkodą w swobodnym ruchu innej figury. Celem gracza jest przesunięcie figur, w najmniejszej możliwej liczbie ruchów, w taki sposób, żeby zebrać je wszystkie na jednym polu. Aby to osiągnąć, gracz musi poruszać figury tak, jak zostało to opisane wyżej. Ponadto, gdy tylko król i jeden lub więcej rycerzy znajduje się na tym samym polu, gracz może się zdecydować wykonywać ruchy królem i jednym z rycerzy jednocześnie tak, jakby to był ruch tylko rycerzem, aż do zebrania wszystkich figur na jednym polu. Jednoczesny ruch rycerza i króla liczymy jako jeden ruch.

Zadanie

Napisz program obliczający minimalną liczbę ruchów, które gracz musi wykonać, aby zebrać wszystkie figury razem.

Dane wejściowe

Plik `CAMELOT.IN` *zawiera początkową konfigurację planszy, zakodowaną jako napis (character string). Taki napis opisuje do 64 różnych pól planszy, najpierw pozycję króla, a potem rycerzy. Każda pozycja jest parą litera-cyfra. Litera oznacza współrzędną poziomą, cyfra współrzędną pionową.*

Przykład danych wejściowych

```
D4A3A8H1H8
```

Król znajduje się na D4. Jest 4 rycerzy znajdujących się na A3, A8, H1, H8.

Dane wyjściowe

Plik `CAMELOT.OUT` *ma zawierać jeden wiersz z jedną liczbą całkowitą, oznaczającą minimalną liczbę ruchów doprowadzającą do zgromadzenia figur razem.*

Wyjście dla przykładowych danych

```
10
```

Ograniczenia

0 ⩽ liczba rycerzy ⩽ 63

Krzysztof Diks, Andrzej Walat
przekład

Gwiaździste niebo

Wysoko na niebie gwiazdy tworzą układy mające różne kształty. Układ jest niepustą grupą gwiazd sąsiadujących w pionie, poziomie lub po przekątnej. Układ nie może być częścią większego układu. Dwa układy są podobne, gdy składają się z takiej samej liczby gwiazd i mają taki sam kształt, bez względu na ich orientacje. W ogólnym przypadku może być osiem różnych orientacji określonego układu co ilustruje rysunek 1.

```
* * *         * * *           *         *
*     *           *       *     *       *
*                 *         * * *       *
                *                       * * *

  * * *       * * *       *               *
*     *       *           *     *           *
      *       *           * * *             *
                *                       * * *
```

Rysunek 1. Osiem podobnych układów

Mapą nieba jest dwuwymiarowa macierz zer i jedynek. Jedynka (1) oznacza pojedynczą gwiazdę, a zero (0) puste miejsce.

Zadanie

Na danej mapie zaznacz małymi literami alfabetu wszystkie układy gwiazd. Układy podobne muszą być zaznaczone tą samą literą; układy niepodobne należy zaznaczyć różnymi literami. Zaznaczenie układu polega na zastąpieniu każdej jedynki w układzie tą samą małą literą alfabetu.

Dane wejściowe

Dwa pierwsze wiersze pliku `STARRY.IN` *zawierają odpowiednio szerokość* W *i wysokość* H *mapy nieba. Sama mapa jest zapisana w kolejnych* H *wierszach, po* W *znaków w każdym wierszu.*

Przykładowe wejście

```
23
15
10001000000000010000000
01111100011111000101101
01000000010001000111111
00000000010101000101111
00000111010001000000000
00001001011111000000000
10000001000000000000000
00101000000111110010000
00001000000100010011111
00000001110101010100010
00000100110100010000000
00010001110111110000000
00100001110000000100000
00001000100001000100101
00000001110001000111000
```

W tym przypadku mapa nieba ma szerokość 23 i wysokość 15. Dla jasności, niebo przedstawione na mapie pokazano na poniższym rysunku:

```
*       *                     *
  * * * * *       * * * * *       *   * *   *
  *               *       *       * * * * * *
                  *   *   *       *   * * * *
          * * *   *       *
        *     *   * * * * *
*             *
    *   *             * * * * *     *
        *             *       *     * * * * *
              * * *   *   *   *   *       *
          *     * *   *       *
      *       * * *   * * * * *
    *         * * *               *
        *       *         *       *     *   *
              * * *       *       * * *
```

Rysunek 2. Obraz nieba

Dane wyjściowe

Plik `STARRY.OUT` *zawiera opis tej samej mapy, co plik* `STARRY.IN`*, z tą różnicą, że każdy układ został zaznaczony zgodnie z opisem w treści zadania.*

Przykładowe wyjście

```
a000a0000000000b0000000
0aaaaa000ccccc000d0dd0d
0a0000000c000c000dddddd
000000000c0b0c000d0dddd
00000eee0c000c000000000
0000e00e0ccccc000000000
b000000e000000000000000
00b0f000000ccccc00a0000
0000f000000c000c00aaaaa
0000000ddd0c0b0c0a000a0
00000b00dd0c000c0000000
000g000ddd0ccccc0000000
00g0000ddd0000000e00000
0000b000d0000f000e00e0b
0000000ddd000f000eee000
```

Jest to jeden z możliwych wyników dla przykładowych danych. Ten plik wyjściowy odpowiada następującemu obrazowi nieba:

```
a       a                     b
  a a a a a       c c c c c       d   d d   d
  a               c       c       d d d d d d
                  c   b   c       d   d d d d
          e e e   c       c
        e     e   c c c c c
b             e
    b   f             c c c c c     a
        f             c       c     a a a a a
              d d d   c   b   c   a       a
          b     d d   c       c
      g       d d d   c c c c c
    g         d d d               e
        b       d         f       e     e   b
              d d d       f       e e e
```

Rysunek 3. Obraz nieba z zaznaczonymi układami gwiazd.

Ograniczenia

- *0 ⩽ W (szerokość mapy nieba) ⩽ 100*
- *0 ⩽ H (wysokość mapy nieba) ⩽ 100*
- *0 ⩽ liczba układów ⩽ 500*
- *0 ⩽ liczba niepodobnych układów ⩽ 26 (a..z)*
- *1 ⩽ liczba gwiazd w układzie ⩽ 160*

Krzysztof Diks, Andrzej Walat
przekład

Kontakt

Dr Astro Insky pracuje w centrum radioastronomii. Właśnie odkryła bardzo dziwne emisje mikrofal z samego centrum galaktyki. Czy ta emisja pochodzi od pozaziemskiej formy inteligencji? Czy też jest to tylko zwykła pulsacja gwiazd?

Zadanie

Pomóż Dr Insky odkryć prawdę dostarczając jej narzędzia do analizowania sekwencji występujących w zebranych przez nią danych. Dr Insky chce znaleźć sekwencje o długości pomiędzy A i B (włącznie), które najczęściej powtarzają się w pliku danych. W każdym przypadku należy znaleźć N największych różnych częstości (częstość sekwencji to liczba jej wystąpień). Sekwencje mogą na siebie zachodzić i tylko takie sekwencje, które występują przynajmniej, raz są brane pod uwagę.

Dane wejściowe

Plik `CONTACT.IN` *zawiera ciąg danych o następującym formacie:*

- *Pierwszy wiersz — liczba całkowita A określająca minimalną długość sekwencji.*
- *Drugi wiersz — liczba całkowita B określająca maksymalną długość sekwencji.*
- *Trzeci wiersz — liczba całkowita N określająca liczbę szukanych różnych częstości.*
- *Czwarty wiersz — ciąg znaków 0 lub 1 zakończony znakiem 2.*

Przykładowe wejście:

```
2
4
10
01010010010001000111101100001010011001111000010010011110010000002
```

Oczekujemy znalezienia dziesięciu największych częstości sekwencji nie krótszych niż 2 i nie dłuższych niż 4 w danym ciągu zer i jedynek:

```
0101001001000100011110110000101001100111100001001001111001000000
```

Dane wyjściowe

Plik wyjściowy `CONTACT.OUT` *powinien mieć co najwyżej N wierszy, zawierających co najwyżej N największych częstości i sekwencji o tych wartościach. Wiersze te powinny być uporządkowane według malejących wartości i mieć postać:*

$$\textit{częstość sekw sekw} \ \ldots \ \textit{sekw}$$

gdzie częstość *jest liczbą wystąpień wymienionych po niej sekwencji.*

W każdym wierszu sekwencje muszą być uporządkowane według malejących długości. Sekwencje tej samej długości należy wypisać w porządku malejącym (arytmetycznie). W przypadku, gdy jest mniej niż N różnych częstości, plik wyjściowy będzie miał mniej niż N wierszy.

Przykładowe wyjście

Dla przykładowego wejścia poprawnym plikiem wyjściowym jest:

```
23 00
15 10 01
12 100
11 001 000 11
```

```
10 010
8 0100
7 1001 0010
6 0000 111
5 1000 110 011
4 1100 0011 0001
```

Ograniczenia

Plik wejściowy może mieć rozmiar do 2 megabajtów. Parametry A, B i N spełniają nierówności: $0 < N \leqslant 20$, $0 < A \leqslant B \leqslant 12$

Krzysztof Diks, Andrzej Walat
przekład

Lampy

Do oświetlenia przyjęcia z okazji IOI'98 użyjemy N kolorowych lamp ponumerowanych od 1 do N. Lampy są podłączone do czterech przycisków:

- *przycisk 1 — naciśnięcie tego przycisku zmienia stan wszystkich lamp: WŁĄCZONE zostaną WYŁĄCZONE, a WYŁĄCZONE zostaną WŁĄCZONE;*
- *przycisk 2 — zmienia stan wszystkich lamp o numerach nieparzystych;*
- *przycisk 3 — zmienia stan wszystkich lamp o numerach parzystych;*
- *przycisk 4 — zmienia stan wszystkich lamp o numerach postaci $3K + 1$ (dla $K \geqslant 0$), tzn. 1,4,7...*

Na początku wszystkie lampy są WŁĄCZONE.

Zadanie

Napisz program, który dla danych: liczby naciśnięć przycisków C oraz informacji o końcowym stanie pewnych lamp, znajdzie wszystkie możliwe końcowe konfiguracje N lamp zgodne z podaną informacją.

Dane wejściowe

W pliku `PARTY.IN` *są cztery wiersze określające: liczbę N lamp, liczbę C naciśnięć przycisków i końcowy stan wybranych lamp. Pierwszy wiersz zawiera liczbę N, a drugi liczbę C. W trzecim wierszu znajdują się numery wybranych lamp, które na końcu muszą być WŁĄCZONE. Kolejne liczby w wierszu są oddzielone pojedynczymi odstępami, zaś -1 oznacza koniec danych w wierszu. W czwartym wierszu znajdują się numery tych wybranych lamp, które na końcu muszą być WYŁĄCZONE. Kolejne liczby w wierszu są oddzielone pojedynczymi odstępami, zaś -1 oznacza koniec danych w wierszu.*

Przykładowe wejście

```
10
1
-1
7 -1
```

W tym przypadku jest 10 lamp i wolno nacisnąć tylko raz jeden z przycisków. Lampa 7 musi być na końcu wyłączona.

Dane wyjściowe

Plik `PARTY.PUT` *musi zawierać wszystkie końcowe (bez powtórzeń) konfiguracje lamp, zgodne z danymi informacjami. Każdą możliwą konfigurację należy zapisać w osobnym wierszu. Można je wypisać w dowolnym porządku.*

Każdy wiersz ma N znaków (zer lub jedynek), gdzie pierwszy znak określa stan lampy numer 1, a ostatni — stan lampy numer N. Znak 0 (zero) oznacza, że lampa jest WYŁĄCZONA, a 1 (jedynka) oznacza, że jest WŁĄCZONA.

Przykładowe wyjście

```
0000000000
0110110110
0101010101
```

Dla przykładowego wejścia możliwe są trzy konfiguracje końcowe: albo wszystkie lampy są WYŁĄCZONE; albo lampy 1, 4,7, 10 są WYŁĄCZONE, a lampy 2, 3, 5, 6, 8, 9 są WŁĄCZONE; albo lampy 1, 3, 5, 7, 9 są WYŁĄCZONE, a lampy 2, 4, 6, 8, 10 WŁĄCZONE.

Ograniczenia

Parametry N i C spełniają nierówności: $10 \leqslant N \leqslant 100$, $1 \leqslant C \leqslant 1000$. Liczba wybranych lamp, które mają być na końcu WŁĄCZONE nie jest większa niż 2. Liczba wybranych lamp, które mają być na końcu WYŁĄCZONE nie jest większa niż 2. Dla każdych danych wejściowych istnieje przynajmniej jedna możliwa konfiguracja końcowa.

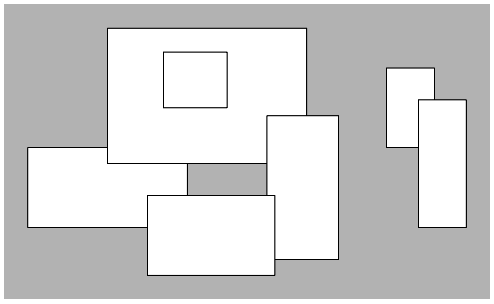
Krzysztof Diks, Andrzej Walat
przekład

Obraz

Na ścianie powieszono pewną liczbę prostokątów (plakatów, fotografii, itp.). Wszystkie boki prostokątów są pionowe lub poziome. Każdy z prostokątów może być całkowicie lub częściowo zasłonięty przez inny. Długość brzegu sumy prostokątów nazywamy obwodem.

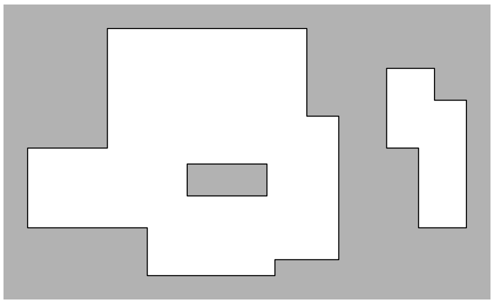
Zadnie

Napisz program obliczający obwód. Na rysunku 1 pokazano przykładowy zestaw 7 prostokątów.



Rys.1 Zestaw 7 prostokątów

Odpowiadający mu brzeg składa się ze wszystkich odcinków przedstawionych na rysunku 2.



Rys. 2 Brzeg zestawu prostokątów z rysunku 1

Wierzchołki wszystkich prostokątów mają współrzędne całkowite.

Dane wejściowe

Pierwszy wiersz pliku `PICTURE.IN` *zawiera liczbę prostokątów zawieszonych na ścianie. W każdym z kolejnych wierszy znajdują się współrzędne wierzchołków jednego prostokąta, dolnego-lewego i górnego-prawego. Wartości współrzędnych są dane jako uporządkowane pary liczb całkowitych: współrzędna x-owa, a następnie współrzędna y-owa.*

Przykładowe wejście

```
7
-15 0 5 10
-5 8 20 25
15 -4 24 14
```

```
0 -6 16 4
2 15 10 22
30 10 36 20
34 0 40 16
```

Powyższe odpowiada przykładowi z rysunku 1.

Dane wyjściowe

Plik `PICTURE.OUT` *powinien składać się z pojedynczego wiersza zawierającego jedną nieujemną liczbę całkowitą równą obwodowi zestawu danych prostokątów.*

Przykładowe wyjście

```
228
```

Powyższe jest zawartością pliku wyjściowego dla przykładowych danych.

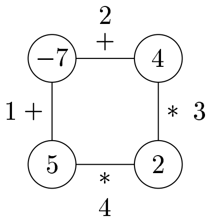
Ograniczenia

$0 \leqslant$ liczba prostokątów < 5000. Wszystkie współrzędne są z przedziału $[-10000, 10000]$, a każdy dany prostokąt ma dodatnie pole. Uwaga: wynik może wymagać reprezentacji 32-bitowej ze znakiem.

Krzysztof Diks, Andrzej Walat
przekład

Wielokąt

Wielokąt jest grą dla jednej osoby. Zaczyna się od wielokąta o N wierzchołkach, jak na rysunku 1, gdzie $N = 4$. Każdy wierzchołek jest etykietowany liczbą całkowitą, a każda krawędź jest etykietowana znakiem dodawania „+”, albo znakiem mnożenia „”. Krawędzie są ponumerowane od 1 do N.*



Rysunek 1. Graficzne przedstawienie wielokąta

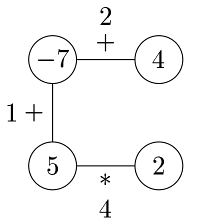
W pierwszym ruchu usuwamy jedną krawędź. Kolejne ruchy składają się z następujących kroków:

- *usuwamy jedną krawędź E i wierzchołki $V1$ i $V2$ połączone krawędzią E oraz*
- *zastępujemy je nowym wierzchołkiem o etykiecie, która jest wynikiem działania przypisanego krawędzi E na etykietach $V1$ i $V2$.*

Gra się kończy, gdy nie ma już żadnej krawędzi, a jej wynikiem jest etykieta jedynego pozostałego wierzchołka.

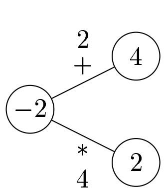
Przykładowa gra

Popatrz na wielokąt na rysunku 1. Gracz zaczyna od usunięcia krawędzi 3. Skutek jest widoczny na rysunku 2.



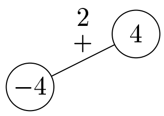
Rysunek 2. Usunięcie krawędzi 3

Następnie gracz usuwa krawędź 1,



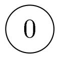
Rysunek 3. Usunięcie krawędzi 1

następnie — krawędź 4,



Rysunek 4. Usunięcie krawędzi 4

i w końcu — krawędź 2. Wynikiem jest 0.



Rysunek 5. Usunięcie krawędzi 2

Zadanie

Napisz program, który dla danego wielokąta oblicza największy możliwy do osiągnięcia wynik i wypisuje każdą krawędź, której usunięcie w pierwszym ruchu może prowadzić do uzyskania najwyższego możliwego wyniku w grze.

Dane wejściowe

Opis wielokąta znajduje się w pliku POLYGON.IN, składającym się z dwóch wierszy. W pierwszym wierszu jest liczba N. Drugi wiersz zawiera etykiety krawędzi $1, \ldots, N$, na przemian z etykietami wierzchołków (najpierw wierzchołka wspólnego dla krawędzi 1 i 2, następnie dla wierzchołka wspólnego dla krawędzi 2 i 3, i tak dalej, a na końcu — wierzchołka wspólnego dla krawędzi N i 1). Wszystkie te etykiety są pooddzielane pojedynczymi odstępami. Etykieta krawędzi jest reprezentowana albo przez literę `t` *oznaczjącą „+”, albo* `x` *oznaczającą „*”.*

Przykładowe wejście

```
4
t -7 t 4 x 2 x 5
```

Ten plik wejściowy zawiera opis wielokąta na rysunku 1. Drugi wiersz zaczyna się od etykiety krawędzi 1.

Dane wyjściowe

Twój program musi zapisać w pierwszym wierszu pliku `POLYGON.OUT` *największy możliwy do uzyskania wynik gry dla danego wielokąta, a w drugim wierszu listę wszystkich krawędzi, mających tę właściwość, że po usunięciu w pierwszym ruchu którejkolwiek z nich można następnie uzyskać najwyższy wynik. Numery tych krawędzi należy wypisać w porządku rosnącym, pooddzielane pojedynczymi odstępami.*

Wyjście dla przykładowego wejścia

```
33
1 2
```

Taki powinien być wynik dla wielokąta z rysunku 1.

Ograniczenia

$3 \leqslant N \leqslant 50$. Dla dowolnej sekwencji ruchów, etykiety wielokątów będą się mieściły w zakresie $[-32768, 32767]$.

Literatura

Poniżej oprócz pozycji cytowanych w niniejszej publikacji zamieszczono inne opracowania polecane zawodnikom Olimpiady Informatycznej.

[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] A. V. Aho, J. E. Hopcroft, J. D. Ullman *Projektowanie i analiza algorytmów komputerowych*, PWN, Warszawa 1983

[7] L. Banachowski, A. Kreczmar *Elementy analizy algorytmów*, WNT, Warszawa 1982

[8] L. Banachowski, A. Kreczmar, W. Rytter *Analiza algorytmów i struktur danych*, WNT, Warszawa 1996

[9] L. Banachowski, K. Diks, W. Rytter *Algorytmy i struktury danych*, WNT, Warszawa 1996

[10] J. Bentley *Perełki oprogramowania*, WNT, Warszawa 1992

[11] I. N. Bronsztejn, K. A. Siemiendiajew *Matematyka. Poradnik encyklopedyczny*, PWN, Warszawa 1996

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest *Wprowadzenie do algorytmów*, WNT, Warszawa 1997

[13] *Elementy informatyki. Pakiet oprogramowania edukacyjnego*, Instytut Informatyki Uniwersytetu Wrocławskiego, OFEK, Wrocław–Poznań 1993.

[14] *Elementy informatyki: Podręcznik (cz. 1), Rozwiązania zadań (cz. 2), Poradnik metodyczny dla nauczyciela (cz. 3)*, pod redakcją M. M. Sysły, PWN, Warszawa 1996

[15] G. Graham, D. Knuth, O. Patashnik *Matematyka konkretna*, PWN, Warszawa 1996

[16] D. Harel *Algorytmika. Rzecz o istocie informatyki*, WNT, Warszawa 1992

[17] J. E. Hopcroft, J. D. Ullman *Wprowadzenie do teorii automatów, języków i obliczeń*, PWN, Warszawa 1994

[18] W. Lipski *Kombinatoryka dla programistów*, WNT, Warszawa 1989

[19] E. M. Reingold, J. Nievergelt, N. Deo *Algorytmy kombinatoryczne*, WNT, Warszawa 1985

[20] K. A. Ross, C. R. B. Wright *Matematyka dyskretna*, PWN, Warszawa 1996

[21] M. M. Sysło *Algorytmy*, WSiP, Warszawa 1998

[22] M. M. Sysło *Piramidy, szyszki i inne konstrukcje algorytmiczne*, WSiP 1998

[23] M. M. Sysło, N. Deo, J. S. Kowalik *Algorytmy optymalizacji dyskretnej z programami w języku Pascal*, PWN, Warszawa 1993

[24] R. J. Wilson *Wprowadzenie do teorii grafów*, PWN, Warszawa 1998

[25] N. Wirth *Algorytmy + struktury danych = programy*, WNT, Warszawa 1999

Niniejsza publikacja stanowi wyczerpujące źródło informacji o zawodach VI Olimpiady Informatycznej, przeprowadzonych w roku szkolnym 1998/1999. Książka zawiera zarówno informacje dotyczące organizacji, regulaminu oraz wyników zawodów. Zawarto także opis rozwiązań wszystkich zadań konkursowych.

VI Olimpiada Informatyczna to pozycja polecana szczególnie uczniom przygotowującym się do udziału w zawodach następnych Olimpiad oraz nauczycielom informatyki, którzy znajdą w niej interesujące i przystępnie sformułowane problemy algorytmiczne wraz z rozwiązaniami.

Olimpiada Informatyczna
jest organizowana przy współudziale

PROKOM SOFTWARE S.A.

ISBN 83–906301–5–X