MINISTERSTWO EDUKACJI NARODOWEJ
INSTYTUT INFORMATYKI UNIWERSYTETU WROCŁAWSKIEGO
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

VII OLIMPIADA INFORMATYCZNA
1999/2000

WARSZAWA, 2000

MINISTERSTWO EDUKACJI NARODOWEJ
INSTYTUT INFORMATYKI UNIWERSYTETU WROCŁAWSKIEGO
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

VII OLIMPIADA INFORMATYCZNA 1999/2000

WARSZAWA, 2000

Autorzy tekstów:

dr Piotr Chrząstowski-Wachtel
dr hab. Krzysztof Diks
dr hab. Wojciech Guzicki
dr Marcin Kubica
dr Adam Malinowski
Marcin Mucha
prof. dr hab. Wojciech Rytter
Marcin Sawicki
mgr Krzysztof Sobusiak
Tomasz Waleń

Autorzy programów na dyskietce:

Marcin Mucha
Marek Pawlicki
Piotr Sankowski
Marcin Sawicki
mgr Krzysztof Sobusiak
Tomasz Waleń
Paweł Wolff

Opracowanie i redakcja:

dr hab. Krzysztof Diks
Tomasz Waleń

Skład: Tomasz Waleń

Pozycja dotowana przez Ministerstwo Edukacji Narodowej.

Druk książki został sfinansowany przez PROKOM SOFTWARE SA

ISBN 83–906301–6–8

Spis treści

Krzysztof Diks

Wstęp

W roku szkolnym 1999/2000 odbyła się VII Olimpiada Informatyczna. Komitet Główny Olimpiady, w tej corocznej publikacji, przekazuje Czytelnikom oficjalne sprawozdanie z przebiegu Olimpiady, treści zadań z poszczególnych etapów wraz z rozwiązaniami autorskimi, dyskietkę zawierającą wzorcowe programy oraz testy, które posłużyły do sprawdzania rozwiązań zawodników. W prezentowanej książeczce można znaleźć także Regulamin Olimpiady Informatycznej oraz Zasady Organizacji Zawodów w roku szkolnym 1999/2000. Przedstawiamy też krótkie sprawozdanie z XI Międzynarodowej Olimpiady Informatycznej, która odbyła się w październiku 1999 r. w miejscowości Antayla-Belek, w Turcji. Wzięli w niej udział zwycięzcy VI Olimpiady Informatycznej. Dla pełności obrazu załączamy treści zadań z tej olimpiady.

Olimpiada Informatyczna jest olimpiadą, której uczestnicy muszą się wykazać wieloma umiejętnościami informatycznymi. Rozwiązanie zadania olimpijskiego polega na wyłuskaniu i wyspecyfikowaniu rzeczywistego problemu algorytmicznego ukrytego w treści zadania, dyskusji możliwych rozwiązań (algorytmów) i wyborze tego najwłaściwszego, dobraniu odpowiednich struktur danych, zaprogramowaniu rozwiązania w wybranym języku programowania oraz dokładnym przetestowaniu otrzymanego programu. Etapy rozwiązywania zadania olimpijskiego są rzeczywistymi etapami rozwiązywania problemów, na jakie napotyka w swojej pracy każdy informatyk.

Zadania olimijskie nie są łatwe, ale myślę, że są ciekawe, pouczające i przynoszą wiele satysfakcji wszystkim, którzy próbują się z nimi zmierzyć. Chciałbym w tym miejscu podziękować autorom zadań za ich pomysłowość i aktywność w pracy nad zadaniami, oraz za przygotowanie omówień rozwiązań zawartych w tej książeczce. Mam nadzieję, że będą one przydatne wszystkim, którzy przygotowywać się będą do udziału w kolejnych olimpiadach.

Każde zadanie jest opracowywane przez jurorów. Biorą oni na siebie dużą odpowiedzialność, ponieważ to ich rozwiązania są bezpośrednio konfrontowane z rozwiązaniami zawodników. Dlatego bardzo gorąco dziękuję wszystkim jurorom za ich solidność, profesjonalizm i czas poświęcony na przygotowanie wzorcowych rozwiązań. Wyniki ich pracy można znaleźć na załączonej dyskietce.

Tak duża impreza, jak Olimpiada Informatyczna, nie mogłaby się odbyć bez zaangażowania bardzo wielu ludzi, którzy pomagali w organizacji zawodów. Wszystkim im tą drogą składam gorące podziękowania.

Olimpiada Informatyczna ma wielu przyjaciół, którzy wspomagali i uświetniali organizację zawodów. Należą do nich: współorganizator zawodów — firma PROKOM Software SA; firma COMBIDATA Poland Sp. z o.o., która współuczestniczyła w organizacji zawodów II stopnia i była współorganizatorem finałów; uczelnie: Akademia Górniczo-Hutnicza, Politechnika Śląska, Polsko-Japońska Wyższa Szkoła Technik Komputerowych w Warszawie, Uniwersytet im. Mikołaja Kopernika, Uniwersytet Warszawski, Uniwersytet Wrocławski, fundatorzy nagród i upominków: Ogólnopol-

ska Fundacja Edukacji Komputerowej, PWN–Wydawnictwo Naukowe, Wydawnictwa Naukowo–Techniczne, oraz IDG Poland. Wszystkim składam gorące podziękowania.

Krzysztof Diks

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, V i VI Olimpiady, można zakupić:

- *w sieci księgarni „Elektronika” w Warszawie, Łodzi i Wrocławiu,*
- *w niektórych księgarniach,*
- *w Komitecie Okręgowym Olimpiady Informatycznej w Warszawie: Ośrodek Edukacji Informatycznej i Zastosowań Komputerów, 02-026 Warszawa, ul. Raszyńska 8/10, tel. (+22) 8226048,*
- *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 przebiegu VII Olimpiady Informatycznej 1999/2000

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

ORGANIZACJA ZAWODÓW

W roku szkolnym 1999/2000 odbyły się zawody VII 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. 4 października 1999 r. rozesłano plakaty zawierające zasady organizacji zawodów I stopnia oraz zestaw 4 zadań konkursowych do 3350 szkół i zespołów szkół młodzieżowych ponadpodstawowych oraz do wszystkich kuratorów i koordynatorów edukacji informatycznej. Zawody I stopnia rozpoczęły się dnia 18 października 1999 roku. Ostatecznym terminem nadsyłania prac konkursowych był 15 listopada 1999 roku. Zawody II i III stopnia były dwudniowymi sesjami stacjonarnymi, poprzedzonymi jednodniowymi sesjami próbnymi. Zawody II stopnia odbyły się w trzech okręgach, w Warszawie, Wrocławiu i Toruniu, oraz w Krakowie, Katowicach i Sopocie, w dniach 8–10 lutego 2000 r., natomiast zawody III stopnia odbyły się w ośrodku firmy Combidata Poland S.A., w Sopocie, w dniach 11–15 kwietnia 2000 r. Uroczystość zakończenia VII Olimpiady Informatycznej odbyła się w dniu 15 kwietnia 2000 r. w sali posiedzeń Urzędu Miasta w Sopocie.

SKŁAD OSOBOWY KOMITETÓW OLIMPIADY INFORMATYCZNEJ

Komitet Główny

przewodniczący:
dr hab. Krzysztof Diks, prof. UW (Uniwersytet Warszawski)

z-cy przewodniczącego:
prof. dr hab. Maciej M. Sysło (Uniwersytet Wrocławski)
dr Andrzej Walat (Ośrodek Edukacji Informatycznej i Zastosowań Komputerów)

członek prezydium Komitetu Głównego:
dr hab. inż. Stanisław Waligórski, prof. UW (Uniwersytet Warszawski)

sekretarz naukowy:
dr Marcin Kubica (Uniwersytet Warszawski)

kierownik Jury:
dr Krzysztof Stencel (Uniwersytet Warszawski)

kierownik organizacyjny:
Tadeusz Kuran (Ośrodek Edukacji Inormatycznej i Zastosowań Komputerów)

członkowie:
prof. dr hab. Zbigniew Czech (Politechnika Śląska)
mgr Jerzy Dałek (Ministerstwo Edukacji Narodowej)
dr Przemysława Kanarek (Uniwersytet Wrocławski)
dr Krzysztof Loryś (Uniwersytet Wrocławski)
dr hab. Jan Madey, prof. UW (Uniwersytet Warszawski)
prof. dr hab. Wojciech Rytter (Uniwersytet Warszawski)
mgr Krzysztof J. Święcicki (Ministerstwo Edukacji Narodowej)
dr Maciej Ślusarek (Uniwersytet Jagielloński)
dr Bolesław Wojdyło (Uniwersytet Mikołaja Kopernika w Toruniu)

sekretarz Komitetu Głównego
Monika Kozłowska-Zając (Ośrodek Edukacji Informatycznej i Zastosowań Komputerów)

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. 28 i 29 stycznia 2000 r. przeprowadzono seminarium przygotowujące przeprowadzenie zawodów II stopnia oraz omawiające dalszy rozwój Olimpiady.

Komitet Okręgowy w Warszawie

przewodniczący:
dr Wojciech Plandowski (Uniwersytet Warszawski)

członkowie:

dr Marcin Kubica (Uniwersytet Warszawski)
dr Adam Malinowski (Uniwersytet Warszawski)
dr Andrzej Walat (Ośrodek Edukacji Informatycznej i Zastosowań Komputerów)

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)

z-ca przewodniczącego:

dr Helena Krupicka (Uniwersytet Wrocławski)

sekretarz:

inż. Maria Woźniak (Uniwersytet Wrocławski)

członkowie:

mgr Jacek Jagiełło (Uniwersytet Wrocławski)
dr Tomasz Jurdziński (Uniwersytet Wrocławski)
dr Przemysława Kanarek (Uniwersytet Wrocławski)
dr Witold Karczewski (Uniwersytet Wrocławski)

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

Komitet Okręgowy w Toruniu

przewodniczący:

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:

mgr Anna Kwiatkowska (IV Liceum Ogólnokształcące w Toruniu)
dr Krzysztof Skowronek (V Liceum Ogólnokształcące w Toruniu).

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

JURY OLIMPIADY INFORMATYCZNEJ

W pracach Jury, które nadzorował dr hab. Krzysztof Diks, a którymi kierował dr Krzysztof Stencel, brali udział pracownicy i studenci Instytutu Informatyki Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego:

Adam Borowski
mgr Marcin Engel
Andrzej Gąsienica-Samek
Marcin Mucha
Marek Pawlicki
Piotr Sankowski
Marcin Sawicki
Marcin Stefaniak
Tomasz Waleń
Paweł Wolff

ZAWODY I STOPNIA

W VII Olimpiadzie Informatycznej wzięło udział 757 zawodników. Decyzją Komitetu Głównego Olimpiady do zawodów zostało dopuszczonych 2 uczniów gimnazjum i 3 uczniów szkół podstawowych:

- Gimnazjum nr 1 im. M. Konopnickiej w Gdyni
  - Piotr Kowalczyk
  - Remigiusz Modrzejewski
- S. P. nr 12 w Piotrkowie Trybunalskim
  - Łukasz Krysiak
- S. P. nr 42 w Gliwicach
  - Leszek Knoll
- Prywatna S. P. nr 2 w Sosnowcu
  - Michał Koziak

W VII Olimpiadzie Informatycznej sklasyfikowano 757 zawodników.

Z rozwiązaniami:

czterech zadań nadeszło	360 prac
trzech zadań nadeszło	201 prac
dwóch zadań nadeszło	111 prac
jednego zadania nadeszło	85 prac

Kolejność województw pod względem liczby uczestników jest następująca:

mazowieckie	111 zawodników
śląskie	97
małopolskie	89
podkarpackie	61
pomorskie	60
kujawsko-pomorskie	55
wielkopolskie	52
dolnośląskie	42
łódzkie	38
lubelskie	35
lubuskie	24
świętokrzyskie	22
podlaskie	21
warmińsko-mazurskie	20
zachodniopomorskie	20
opolskie	10

W zawodach I stopnia najliczniej reprezentowane były szkoły:

III L.O. im. Marynarki Wojennej RP w Gdyni	35 zawodników
V L.O. im. A Witkowskiego w Krakowie	33
XIV L.O. im. St. Staszica w Warszawie	13
VIII L.O. im. A. Mickiewicza w Poznaniu	12
I L.O. im. St. Staszica w Lublinie	11
II L.O. im. C. K. Norwida w Tychach	9
VI L.O. im. J. i J. Śniadeckich w Bydgoszczy	9
IV L.O. im. T. Kościuszki w Toruniu	8
II L.O. im. J. Chreptowicza w Ostrowcu Świętokrzyskim	7
L.O. im. M. Skłodowskiej–Curie w Pile	7
VI L.O. im. J. Kochanowskiego w Radomiu	7
VIII L.O. im. M. Skłodowskiej–Curie w Katowicach	6
Zespół Szkół w Kańczudze	6
I L.O. im. A. Mickiewicza w Białymstoku	5
I L.O. im. A. Osuchowskiego w Cieszynie	5
I L.O. im. M. Skłodowskiej–Curie w Szczecinie	5
I L.O. im. St. Staszica w Zgierzu	5
I L.O. im. T. Kościuszki w Lesznie	5
I L.O. im. Ziemii Kujawskiej we Włocławku	5
II L.O. im. Królowej Jadwigi w Siedlcach	5
XIV L.O. im. Polonii Belgijskiej we Wrocławiu	5

Ogólnie najliczniej reprezentowane były miasta:

Warszawa	66 uczniów	Gliwice	8
Kraków	58	Gorzów Wlkp.	8
Gdynia	43	Białystok	7
Bydgoszcz	27	Ostrowiec Św.	7
Poznań	25	Piła	7
Lublin	20	Głogów	6
Częstochowa	15	Kańczuga	6
Łódź	15	Koszalin	6
Rzeszów	15	Opole	6
Wrocław	14	Płock	6
Radom	13	Stalowa Wola	6
Szczecin	13	Toruń	6
Tychy	13	Włocławek	6
Katowice	11	Zielona Góra	6
Gdańsk	10	Elbląg	5
Inowrocław	10	Legnica	5
Siedlce	10	Leszno	5
Cieszyn	9	Olsztyn	5
Kielce	9	Oława	5
Mielec	9	Zgierz	5

Zawodnicy uczęszczali do następujących klas:

do klasy I	gimnazjum	2 zawodników
do klasy VIII	szkoły podstawowej	3
do klasy I	szkoły średniej	45
do klasy II		183
do klasy III		256
do klasy IV		252
do klasy V		16

Zawodnicy najczęściej używali następujących języków programowania:

Pascal firmy Borland	529 prac
C/C++ firmy Borland	190 prac

Ponadto pojawiły się :

Watcom C/C++	3 prace
GNU C/C++	11 prac
Ansi C	5 prac
Visual C	11 prac
Symantec C/C++	1 praca
DJGPP	7 prac

Komputerowe wspomaganie umożliwiło sprawdzenie prac zawodników kompletem 154-ch testów.

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

	Gdzie zbudować browar?		*Wirusy*		*Narciarze*		*Paski*	
	liczba zawodn.	czyli	liczba zawodn.	czyli	liczba zawodn.	czyli	liczba zawodn.	czyli
100 pkt.	58	7,7%	12	1,6%	114	15,1%	1	0,1%
99–75 pkt.	51	6,7%	6	0,8%	55	7,3%	0	0,0%
74–50 pkt.	16	2,1%	24	3,2%	114	15,1%	0	0,0%
49–1 pkt.	358	47,3%	189	25,0%	224	29,5%	130	17,2%
0 pkt.	274	36,2%	526	69,4%	250	33,0%	626	82,7%

W sumie za wszystkie 4 zadania:

SUMA	liczba zawodników	czyli
400 pkt.	0	0,0%
399–300 pkt.	8	1,1%
299–200 pkt.	68	9,0%
199–1 pkt.	539	71,2%
0 pkt.	142	18,7%

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 191 zawodników, którzy osiągnęli w zawodach I stopnia wynik nie mniejszy niż 118 pkt.

Zawody II stopnia odbyły się w dniach 8-10 lutego 2000 r. w trzech stałych okręgach oraz w Krakowie, Katowicach i Sopocie:

- w Toruniu — 24 zawodników z następujących województw:
  - kujawsko-pomorskie (11),
  - łódzkie (3),
  - mazowieckie (3),
  - pomorskie (6),

  - warmińsko-mazurskie (1),
- we Wrocławiu — 44 zawodników z następujących województw:
  - dolnośląskie (9),
  - lubuskie (7),
  - łódzkie (4),
  - małopolskie (7),
  - opolskie (2),
  - śląskie (5),
  - wielkopolskie (9),
  - zachodniopomorskie (1),
- w Warszawie — 49 zawodników z następujących województw:
  - lubelskie (8),
  - łódzkie (3),
  - małopolskie (9),
  - mazowieckie (21),
  - podkarpackie (2),
  - podlaskie (6),
- w Krakowie — 36 zawodników z następujących województw:
  - lubelskie (2),
  - małopolskie (21),
  - podkarpackie (12),
  - świętokrzyskie (1),
- w Katowicach — 21 zawodników z następujących województw:
  - małopolskie (8),
  - podkarpackie (1),
  - śląskie (12),
- w Sopocie — 17 zawodników z następujących województw:
  - kujawsko-pomorskie (5),
  - łódzkie (1),
  - pomorskie (9),
  - warmińsko-mazurskie (2).

W zawodach II stopnia najliczniej reprezentowane były szkoły:

V L.O. im. A. Witkowskiego w Krakowie	30 uczniów
III L.O. im. Marynarki Wojennej RP w Gdyni	12
I L.O. im. St. Staszica w Lublinie	7
VI L.O. im. J. Kochanowskiego w Radomiu	6
VI L.O. im. J. i J. Śniadeckich w Bydgoszczy	6
XIV L.O. im. St. Staszica w Warszawie	6
I L.O. im. A. Mickiewicza w Białymstoku	5
IV L.O. im. T. Kościuszki w Toruniu	5
XIV L.O. im. Polonii Belgijskiej we Wrocławiu	5
II L.O. im. L. Lisa-Kuli w Rzeszowie	4
VIII L.O. im. A. Mickiewicza w Poznaniu	3

Ogólnie najliczniej reprezentowane były miasta:

Kraków	36 zawodników
Warszawa	15
Gdynia	13
Bydgoszcz	11
Lublin	7
Białystok	6
Łódź	6
Radom	6
Rzeszów	6
Wrocław	6
Toruń	5
Gorzów Wlkp.	4
Poznań	4
Katowice	3
Stalowa Wola	3
Żywiec	3

8 lutego odbyła się sesja próbna, na której zawodnicy rozwiązywali nie liczące się do ogólnej klasyfikacji zadanie *Podpisy*. W dniach konkursowych zawodnicy rozwiązywali zadania: *Automorfizmy* – maksymalnie 60 pkt, *Trójramienny dźwig* – maksymalnie 40 pkt, *Labirynt studni* i *Kod* oceniane, każde, maksymalnie po 50 punktów. Ośmiu zawodników nie stawiło się na zawody.

Do automatycznego sprawdzania 4 zadań konkursowych zastosowano łącznie 142 testy.

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

- *Automorfizmy*

	liczba zawodników	czyli
60 pkt.	2	2,4%
59–40 pkt.	4	4,8%
39–20 pkt.	27	32,5%
19–1 pkt.	38	45,8%
0 pkt.	12	14,5%

- *Trójramienny dźwig*

	liczba zawodników	czyli
40 pkt.	30	16,4%
39–30 pkt.	15	8,2%
29–20 pkt.	18	9,8%
19–1 pkt.	55	30,1%
0 pkt.	65	35,5%

- *Kod*

	liczba zawodników	czyli
50 pkt.	27	14,8%
49–30 pkt.	3	1,6%
29–20 pkt.	4	2,2%
19–1 pkt.	91	49,7%
0 pkt.	58	31,7%

- *Labirynt studni*

	liczba zawodników	czyli
50 pkt.	6	3,3%
49–30 pkt.	17	9,3%
29–20 pkt.	37	20,2%
19–1 pkt.	54	29,5%
0 pkt.	69	37,7%

W sumie za wszystkie 4 zadania, przy najwyższym wyniku wynoszącym 200 pkt.:

SUMA	liczba zawodników	czyli
200 pkt.	1	0,5%
199–100 pkt.	22	12,0%
99–50 pkt.	51	27,9%
49–1 pkt.	94	51,4%
0 pkt.	15	8,2%

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 i 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 11 do 15 kwietnia 2000 r.

W zawodach III stopnia wzięło udział 44 najlepszych uczestników zawodów II stopnia, którzy uzyskali wynik nie mniejszy niż 71 pkt. Zawodnicy pochodzili z następujących województw:

małopolskie	9 zawodników
podkarpackie	6
kujawsko-pomorskie	4
mazowieckie	4
pomorskie	4
śląskie	4
świętokrzyskie	4
łódzkie	3
lubelskie	2
warmińsko-mazurskie	2
wielkopolskie	2
dolnośląskie	1
opolskie	1
podlaskie	1

Niżej wymienione szkoły miały w finale więcej niż jednego zawodnika:

V L.O. im. A. Witkowskiego w Krakowie	6 zawodników
XIV L.O. im. St. Staszica w Warszawie	3
III L.O. im. A. Mickiewicza w Tarnowie	2
VI L.O. im. J. i J. Śniadeckich w Bydgoszczy	2
III L.O. im. Marynarki Wojennej RP w Gdyni	2
IV L.O. im. M. Kopernika w Rzeszowie	2
I L.O. im. J. Śniadeckiego w Pabianicach	2

11 kwietnia odbyła się sesja próbna, na której zawodnicy rozwiązywali nie liczące się do ogólnej klasyfikacji zadanie *Lollobrygida*. W dniach konkursowych zawodnicy rozwiązywali zadania: *Jajka*, *Po–łamana* oceniane maksymalnie po 60 punktów oraz *Agenci*, *Promocja* i *Powtórzenia* oceniane maksymalnie po 40 punktów.

Sprawdzanie przeprowadzono korzystając z programu sprawdzającego, przygotowanego przez Marka Pawlickiego i Piotra Sankowskiego, który umożliwił częściowe sprawdzenie zadań z danego dnia w obecności zawodnika.

Zastosowano łącznie 76 testów.

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

- *Jajka*

	liczba zawodników	czyli
60 pkt.	2	4,6%
59–40 pkt.	0	0,0%
39–20 pkt.	10	22,7%
19–1 pkt.	18	40,9%
0 pkt.	14	31,8%

- *Po–łamana*

	liczba zawodników	czyli
60 pkt.	8	18,2%
59–40 pkt.	7	15,9%
39–20 pkt.	3	6,8%
19–1 pkt.	6	13,6%
0 pkt.	20	45,5%

- *Agenci*

	liczba zawodników	czyli
40 pkt.	2	4,5%
39–30 pkt.	1	2,3%
29–20 pkt.	9	20,5%
19–1 pkt.	19	43,2%
0 pkt.	13	29,5%

- *Powtórzenia*

	liczba zawodników	czyli
40 pkt.	11	25,0%
39–30 pkt.	2	4,5%
29–20 pkt.	14	31,8%
19–1 pkt.	8	18,2%
0 pkt.	9	18,2%

- *Promocja*

	liczba zawodników	czyli
40 pkt.	2	4,6%
39–30 pkt.	4	9,1%
29–20 pkt.	18	40,9%
19–1 pkt.	6	13,6%
0 pkt.	14	31,8%

W sumie za wszystkie 5 zadań:

SUMA	liczba zawodników	czyli
240 pkt.	0	0,0%
239–160 pkt.	3	6,8%
159–80 pkt.	18	40,9%
79–1 pkt.	22	50,0%
0 pkt.	1	2,3%

W dniu 15 kwietnia 2000 roku w gmachu Urzędu Miasta w Sopocie ogłoszono wyniki finału VII Olimpiady Informatycznej 1999/2000 i rozdano nagrody ufundowane przez: PROKOM Software S.A., Ogólnopolską Fundację Edukacji Komputerowej, Wydawnictwa Naukowo-Techniczne, PWN-Wydawnictwo Naukowe, IDG Poland i Olimpiadę Informatyczną. Poniżej zestawiono listę wszystkich laureatów wraz z przyznanymi nagrodami:

(1) **Tomasz Czajka**, laureat I miejsca, 223 pkt., L.O. im. K.E.N. w Stalowej Woli, (notebook – PROKOM, roczny abonament na książki – WNT)

(2) **Krzysztof Onak**, laureat I miejsca, 204 pkt., III L.O. im. A. Mickiewicza w Tarnowie, (notebook – PROKOM)

(3) **Tomasz Malesiński**, laureat I miejsca, 178 pkt., Z. S. Elektronicznych im. J. Groszkowskiego w Białymstoku, (notebook – PROKOM)

(4) **Grzegorz Stelmaszek**, laureat II miejsca, 150 pkt., XIV L.O. im. Polonii Belgijskiej we Wrocławiu, (notebook – PROKOM)

(5) **Rafał Rusin**, laureat II miejsca, 145 pkt., II L.O. im. L. Lisa-Kuli w Rzeszowie, (drukarka laserowa – PROKOM, prenumerata roczna PC World Komputer – IDG Poland)

(6) **Grzegorz Herman**, laureat II miejsca, 141 pkt., V L.O. im. A. Witkowskiego w Krakowie, (drukarka laserowa – PROKOM, prenumerata roczna PC World Komputer – IDG Poland)

(7) **Jan Jeżabek**, laureat II miejsca, 139 pkt., V L.O. im. A. Witkowskiego w Krakowie, (drukarka laserowa – PROKOM, prenumerata roczna PC World Komputer – IDG Poland)

(8) **Przemysław Broniek**, laureat II miejsca, 137 pkt., V L.O. im. A. Witkowskiego w Krakowie, (drukarka laserowa – PROKOM, prenumerata roczna PC World Komputer – IDG Poland)

(9) **Jakub Pięđel**, laureat II miejsca, 126 pkt., XIV L.O. im. St. Staszica w Warszawie, (drukarka laserowa – PROKOM, prenumerata roczna PC World Komputer – IDG Poland)

(10) **Krzysztof Kluczek**, laureat III miejsca, 115 pkt., L.O. im. St. Żeromskiego w Bartoszycach, (drukarka laserowa – PROKOM, prenumerata roczna PC World Komputer – IDG Poland)

(11) **Krzysztof Ocetkiewicz**, laureat III miejsca, 114 pkt., VI L.O. im. J. i J. Śniadeckich w Bydgoszczy, (drukarka laserowa – PROKOM, prenumerata roczna PC World Komputer – IDG Poland)

(12) **Paweł Parys**, laureat III miejsca, 106 pkt., L.O. im. St. Staszica w Tarnowskich Górach, (drukarka atramentowa – PROKOM, książka „Matematyka konkretna" – PWN)

(13) **Tomasz Szydło**, laureat III miejsca, 106 pkt., IV L.O. im. M. Kopernika w Rzeszowie, (drukarka atramentowa – PROKOM, książka „Matematyka konkretna" – PWN)

(14) **Tomasz Dobrowolski**, laureat III miejsca, 105 pkt., IX L.O. w Gdańsku, (drukarka atramentowa – PROKOM, książka „Matematyka konkretna" – PWN)

(15) **Władysław Bodzek**, laureat III miejsca, 102 pkt., I Katolickie L.O. im. Jana III Sobieskiego w Bydgoszczy, (drukarka atramentowa – PROKOM, książka „Matematyka konkretna" – PWN)

(16) **Michał Adamaszek**, laureat III miejsca, 99 pkt., V L.O. w Bielsku-Białej, (drukarka atramentowa – PROKOM, książka „Matematyka konkretna" – PWN)

(17) **Wojciech Matyjewicz**, laureat III miejsca, 99 pkt., III L. O. im. A. Mickiewicza w Tarnowie, (drukarka atramentowa – PROKOM, książka „Matematyka konkretna" – PWN)

(18) **Paweł Walter**, laureat III miejsca, 96 pkt., V L.O. im. A. Witkowskiego w Krakowie, (drukarka atramentowa – PROKOM, książka „Matematyka konkretna" – PWN)

(19) **Piotr Skibiński**, laureat III miejsca, 92 pkt., IV L.O. im. M. Kopernika w Rzeszowie, (drukarka atramentowa – Olimpiada Informatyczna, książka „Matematyka konkretna" – PWN)

Wszyscy finaliści otrzymali książki ufundowane przez WNT, a ci którzy nie byli laureatami otrzymali zegarki CASIO ufundowane przez Ogólnopolską Fundację Edukacji Komputerowej.

Ogłoszono komunikat o powołaniu reprezentacji Polski, na Olimpiadę Informatyczną Centralnej Europy w Cluj w Rumunii oraz na Międzynarodową Olimpiadę Informatyczną w Pekinie w Chinach, w składzie:

(1) **Tomasz Czajka**

(2) **Krzysztof Onak**

(3) **Tomasz Malesiński**

(4) **Grzegorz Stelmaszek**

zawodnikami rezerwowymi zostali:

(5) **Rafał Rusin**

(6) **Grzegorz Herman**

oraz na Bałtycką Olimpiadę Informatyczną w Sztokholmie, w składzie:

(1) **Tomasz Malesiński**

(2) **Grzegorz Stelmaszek**

(3) **Grzegorz Herman**

(4) **Jakub Piędel**

(5) **Krzysztof Kluczek**

(6) **Krzysztof Ocetkiewicz**

zawodnikami rezerwowymi zostali:

(7) **Paweł Parys**

(8) **Władysław Bodzek**

Sekretariat olimpiady wystawił łącznie 44 zaświadczenia o zakwalifikowaniu do zawodów III stopnia celem przedłożenia dyrekcji szkoły.

Sekretariat wystawił łącznie 19 zaświadczeń o uzyskaniu tytułu laureata i 25 zaświadczeń o uzyskaniu tytułu finalisty VII 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.

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

- **Klemens Czajka** (Huta Stalowa Wola)
  - Tomasz Czajka (laureat I miejsca)
- **Jan Onak** (Zespół Szkół Mechaniczno-Elektrycznych w Tarnowie)
  - Krzysztof Onak (laureat I miejsca)
- **Joanna Ewa Łuszcz** (Zespół Szkół Elektrycznych w Białymstoku)
  - Tomasz Malesiński (laureat I miejsca)
- **Lech Stelmaszek** (Pracownia zastosowań informatyki "INFOS", Wrocław)
  - Grzegorz Stelmaszek (laureat II miejsca)

- **Bogusław Rusin** (F.A.H. „Minar” w Rzeszowie)
  - Rafał Rusin (laureat II miejsca)
- **Andrzej Dyrek** (Uniwersytet Jagielloński w Krakowie)
  - Grzegorz Herman (laureat II miejsca)
  - Jan Jeżabek (laureat II miejsca)
  - Przemysław Broniek (laureat II miejsca)
  - Paweł Walter (laureat III miejsca)
  - Michał Kosek (finalista)
  - Grzegorz Łukasik (finalista)
- **Katarzyna Lewińska–Piędel** (Zakład Sitodruku „Ma Print” w Otwocku)
  - Jakub Piędel (laureat II miejsca)
- **Zdzisław Kluczek** (Key Electronics w Bartoszycach)
  - Krzysztof Kluczek (laureat III miejsca)
- **Iwona Waszkiewicz** (VI L.O. w Bydgoszczy)
  - Krzysztof Ocetkiewicz (laureat III miejsca)
  - Roman Łomowski (finalista)
- **Krzysztof Żelazowski** (Z. S. O. im. St. Staszica w Tarnowskich Górach)
  - Paweł Parys (laureat III miejsca)
- **Janusz Szydło** (Rzeszów), **Mariusz Kraus** (IV L.O. w Rzeszowie)
  - Tomasz Szydło (laureat III miejsca)
- **Waldemar Dobrowolski** (ZR RADMOR S.A. w Gdyni)
  - Tomasz Dobrowolski (laureat III miejsca)
- **Lech Bodzek** (Poczta Polska, Centralny Ośrodek Rozliczeniowy w Bydgoszczy)
  - Władysław Bodzek (laureat III miejsca)
- **Anna Kowalska** (V L.O. w Bielsku-Białej)
  - Michał Adamaszek (laureat III miejsca)
- **Paweł Guraj** (Warszawa)
  - Grzegorz Andruszkiewicz (laureat III miejsca)
- **Janusz Matyjewicz** (AMPLI S.A. w Tarnowie)
  - Wojciech Matyjewicz (laureat III miejsca)
- **Aleksander Skibiński** (ARCUS – biuro projektów w Rzeszowie)

    ◦ Piotr Skibiński (laureat III miejsca)

- **Marek Adamiak** (AUTOSAN S. A. w Sanoku)
    - Marek Adamiak (finalista)
- **Krzysztof Stefański** (VIII L.O. im. A. Mickiewicza w Poznaniu)
    - Jan Chmiel (finalista)
- **Kazimierz Cwalina** (Ministerstwo Gospodarki w Warszawie)
    - Karol Cwalina (finalista)
- **Mirosław Modzelewski** (VI L.O. w Radomiu)
    - Jacek Erd (finalista)
- **Jan Gibek** (Elektrownia Skawina S. A. w Skawinie)
    - Marek Gibek (finalista)
- **Zbigniew Gocaliński** (KWK Makoszowy — stacja sejsmoakustyki w Zabrzu)
    - Igor Gocaliński (finalista)
- **Małgorzata Tatar** (Z. S. Elektronicznych i Telekomunikacyjnych w Olsztynie)
    - Tomasz Kolinko (finalista)
- **Edyta Pobiega** (II L.O. im. J. Chreptowicza w Ostrowcu Świętokrzyskim)
    - Piotr Kowalski (finalista)
- **Stanisław Kwaśniewicz** (II L.O. we Włodawie)
    - Michał Kozłowski (finalista)
- **Anna Hertleb** (I L.O. im. J. Śniadeckiego w Pabianicach)
    - Michał Malinowski (finalista)
- **Michał Malinowski** (Dobroń)
    - Piotr Malinowski (finalista)
- **Halina Michalska** (Lewin Brzeski)
    - Michał Michalski (finalista)
- **Marek Noworyta** (Centrozap S. A. w Katowicach)
    - Filip Noworyta (finalista)
- **Ryszard Szubartowski** (Gdyńskie Liceum Autorskie i III L.O. im. Marynarki Wojennej RP w Gdyni)
    - Tomasz Pyra (finalista)
    - Dominik Wojtczak (finalista)
    - Marcin Zawadzki (finalista)

- **Marek Sadziak** (EL-KAM S. C. w Warszawie)
  - Sławomir Sadziak (finalista)
- **Alicja Skalska** (V L.O. w Bydgoszczy)
  - Kamil Skalski (finalista)
- **Ewa Stawicka** (Łask-Kolumna)
  - Paweł Stawicki (finalista)
- **Mirosław Kulik** (I L.O. im. St. Staszica w Lublinie)
  - Stanisław Wierchoła (finalista)
- **Tomasz Radko** (Katolickie L.O. Księży Pijarów w Krakowie)
  - Marcin Wojnarski (finalista)

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

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

Warszawa, dn. 7 czerwca 2000 roku

Regulamin Olimpiady Informatycznej

§1 WSTĘP

Olimpiada Informatyczna jest olimpiadą przedmiotową powołaną przez Instytut Informatyki Uniwersytetu Wrocławskiego, który jest organizatorem Olimpiady, zgodnie z zarządzeniem nr 28 Ministra Edukacji Narodowej z dnia 14 września 1992 roku (Dz. Urz. MEN nr 7 z 1992 r. poz. 31) z późniejszymi zmianami (zarządzenie Ministra Edukacji Narodowej nr 19 z dnia 20 października 1994 r., Dz. Urz. MEN nr 5 z 1994 r. poz. 27). W organizacji Olimpiady Instytut współdziała ze środowiskami akademickimi, zawodowymi i oświatowymi działającymi w sprawach edukacji informatycznej.

§2 CELE OLIMPIADY INFORMATYCZNEJ

(1) Stworzenie motywacji dla zainteresowania nauczycieli i uczniów 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 w języku programowania i środowisku wybranym z listy języków i środowisk 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 rozwiązywaniu 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 w szczególności: przewodniczący, dwóch wiceprzewodniczących, sekretarz naukowy, kierownik Jury i kierownik organizacyjny.

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

(5) Komitet powołuje i rozwiązuje komitety okręgowe Olimpiady.

(6) Komitet:

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

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

   (c) udziela wyjaśnień w sprawach dotyczących Olimpiady,

   (d) ustala listy laureatów i 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 i 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 Warszawie w Ośrodku Edukacji Informatycznej i Zastosowań Komputerów 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 na podstawie zaświadczenia wydanego przez Komitet, najwyższą ocenę z informatyki na zakończenie nauki w klasie, do której uczęszczają.

(2) Uczestnicy Olimpiady, którzy zostali zakwalifikowani do zawodów III stopnia są zwolnieni z egzaminu z przygotowania zawodowego z przedmiotu informatyka oraz (zgodnie z zarządzeniem nr 35 Ministra Edukacji Narodowej z dnia 30 listopada 1991 r.) 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. Zaświadczenia podpisuje przewodniczący Komitetu. Komitet prowadzi rejestr wydanych zaświadczeń.

(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 wypłacaną z budżetu Olimpiady.

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

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

§8 FINANSOWANIE OLIMPIADY

(1) Komitet Główny będzie się ubiegał o pozyskanie środków finansowych z budżetu państwa, składając wniosek w tej sprawie do Ministra Edukacji Narodowej i 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 dwóch 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, 17 września 1999 roku

Zasady organizacji zawodów w roku szkolnym 1999/2000

Podstawowym aktem prawnym dotyczącym Olimpiady jest Regulamin Olimpiady Informatycznej, którego pełny tekst znajduje się w kuratoriach. Poniższe zasady są uzupełnieniem tego Regulaminu, zawierającym szczegółowe postanowienia Komitetu Głównego Olimpiady Informatycznej o jej organizacji w roku szkolnym 1999/2000.

§1 WSTĘP

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

§2 ORGANIZACJA OLIMPIADY

(1) Olimpiadę przeprowadza Komitet Główny Olimpiady Informatycznej.

(2) Olimpiada Informatyczna jest trójstopniowa.

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

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

(5) Zawody I stopnia mają charakter otwarty i polegają na samodzielnym 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 co najwyżej o 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 — 18.10.1999–15.11.1999
ogłoszenie wyników: 20.12.1999
zawody II stopnia — 8.02.2000–10.02.2000
ogłoszenie wyników: 6.03.2000
zawody III stopnia — 11.04.2000–15.04.2000

§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 adres:

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

w nieprzekraczalnym terminie nadania do 15 listopada 1999 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,
- programy w postaci źródłowej i skompilowanej do wszystkich rozwiązanych zadań.

Zaleca się kompilowanie do trybu rzeczywistego MS-DOS. Imię i nazwisko uczestnika musi 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 rozwiązał 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 skład reprezentacji Polski na XII Międzynarodową Olimpiadę Informatyczną w 2000 roku na podstawie wyników zawodów III stopnia i regulaminu tej Olimpiady Międzynarodowej. Szczegółowe zasady zostaną podane po otrzymaniu formalnego zaproszenia na XII 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.

Strona Olimpiady:
`www.oi.pjwstk.waw.pl`
oraz
`www.mimuw.edu.pl/oi`

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 teksty zadań, ale i treść „Zasad organizacji zawodów”.

(2) Przestrzegaj dokładnie warunków określonych w zadaniu, 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.

(5) Nie przyjmuj żadnych założeń, które nie wynikają z treści zadania.

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

(7) Ocena za rozwiązanie zadania jest określona na podstawie wyników testowania programu i uwzględnia poprawność oraz efektywność metody rozwiązania użytej w programie. W szczególności programy poprawne, lecz działające zbyt długo — zwłaszcza dla dużych rozmiarów danych — mogą zostać ocenione nisko.

Informacja o XI Międzynarodowej Olimpiadzie Informatycznej IOI'99, Antayla-Belek Turcja, październik 1999

PRZEBIEG I ORGANIZACJA

XI Międzynarodowa Olimpiada Informatyczna odbyła się w dniach 9–16 paździenika 1999 r., w śródziemnomorskim kurorcie Belek koło miasta Antalya w Turcji. Wzięło w niej udział 257 uczniów reprezentujących 65 krajów z całego świata. Polskę reprezentowali: Andrzej Gąsienica-Samek, Marcin Meinardi, Michał Nowakiewicz i Krzysztof Onak. W dwóch sesjach zawodnicy rozwiązywali 2 razy po trzy zadania. Były to jedne z najtrudniejszych zestawów zadań, jakie zdarzyły się na olimpiadach międzynarodowych. Zawodnicy mogli zdobyć maksymalnie 600 punktów, natomiast średnia liczba zdobytych punktów wyniosła 135. Polacy w tej trudnej konkurencji spisali się znakomicie. Andrzej Gąsienica-Samek zdobył medal złoty (trzeci w karierze) za 370 punktów, Marcin Meinardi i Michał Nowakiewicz zdobyli medale srebrne za, odpowiednio, 294 i 240 punktów, zaś Krzysztof Onak zdobył medal brązowy za 210 punktów. Największą liczbę punktów uzyskał Hong Chen z Chin – 480. Drugie miejsce zajęli Mathijs Vogelzang z Holandii i Roman Pastoukhov z Rosji – obaj po 457 punktów. Należy jeszcze wspomnieć, że Tomasz Czajka, który podczas tych zawodów reprezentował Wielką Brytanię, zdobył także medal złoty z wynikiem 350 punktów.

Na końcowych stronach tej książeczki przedstawiamy zadania z XI Międzynarodowej Olimpiady Informatycznej. Zainteresowanych większa liczbą szczegółów na temat tej olimpiady odsyłamy do strony internetowej:

`http://www.ioi99.org.tr/`

Zawody I stopnia

opracowania zadań

Wojciech Guzicki
autor zadania, opracowanie

Tomasz Waleń
program wzorcowy

Gdzie zbudować browar?

Mieszkańcy bajtockiej wyspy Abstynencja bardzo lubią piwo bezalkoholowe. Do tej pory piwo bezalkoholowe sprowadzano z Polski, ale w tym roku w jednym z miast na Abstynencji zostanie wybudowany browar. Wszystkie miasta wyspy leżą na wybrzeżu i są połączone autostradą obiegającą wyspę wzdłuż brzegu. Inwestor budujący browar zebrał informacje o zapotrzebowaniu na piwo, tj. ile cystern piwa trzeba codziennie dostarczyć do każdego z miast. Dysponuje także zestawieniem odległości pomiędzy miastami. Koszt transportu jednej cysterny na odległość 1 km wynosi 1 talar. Dzienny koszt transportu to suma, jaką trzeba wyłożyć, by do każdego miasta przetransportować z browaru tyle cystern, ile wynosi zapotrzebowanie w danym mieście. Jego wielkość zależy od miejsca położenia browaru. Inwestor chce zminimalizować koszty transportu piwa.

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `BRO.IN` *liczbę miast, odległości między nimi oraz dzienne zapotrzebowania na piwo,*
- *obliczy minimalny dzienny koszt transportu piwa,*
- *zapisze wynik w pliku tekstowym* `BRO.OUT`*.*

Wejście

W pierwszym wierszu pliku tekstowego `BRO.IN` *jest zapisana jedna liczba całkowita n równa liczbie miast,* $5 \leqslant n \leqslant 10\,000$*. W każdym z kolejnych n wierszy zapisana jest para nieujemnych liczb całkowitych oddzielonych pojedynczym odstępem. Liczby* z_i, d_i *zapisane w* $(i+1)$*-ym wierszu, to, odpowiednio, zapotrzebowanie na piwo w mieście nr i oraz odległość (w kilometrach) od miasta nr i do następnego miasta na autostradzie. Kolejne miasta na autostradzie mają kolejne numery, po mieście nr n leży miasto nr 1. Całkowita długość autostrady nie przekracza 1 000 000 km. Zapotrzebowanie na piwo w żadnym mieście nie przekracza 1 000 cystern.*

Wyjście

Twój program powinien zapisać w pierwszym i jedynym wierszu pliku tekstowego `BRO.OUT`*, dokładnie jedną liczbę całkowitą równą minimalnemu dziennemu kosztowi transportu piwa.*

Przykład

Dla pliku wejściowego `BRO.IN`*:*

```
6
```

```
1 2
2 3
1 2
5 2
1 10
2 3
```

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

```
41
```

ROZWIĄZANIE

Oczywiste rozwiązanie polega na obliczeniu kosztu transportu z każdego miasta do każdego innego i dodaniu otrzymanych liczb. Dla każdego miasta należy obliczyć w tym celu $n-1$ odległości, co powoduje, że złożoność tego algorytmu wynosi $O(n^2)$. Istnieje algorytm szybszy, działający w czasie $O(n)$.

Warto zauważyć, że jeśli miasta są połączone siecią dróg będącą drzewem, to właściwa lokalizacja browaru nie zależy od odległości między miastami, ale tylko od zapotrzebowań w miastach. Oczywiście łączny koszt transportu zależy od odległości. Przypadek, gdy graf dróg jest drzewem (zresztą bardzo prostym), był już tematem jednego z zadań olimpijskich (Mudstok Bis, II Olimpiada Informatyczna, zobacz [2]). Jeśli graf dróg zawiera cykle, odpowiedź jest znacznie bardziej skomplikowana. Dla jednego cyklu istnieje algorytm liniowy, przy czym nie tylko łączny koszt, ale i lokalizacja browaru zależy teraz również od odległości.

Trudność polega na tym, że w drzewie istnieje tylko jedna droga prowadząca z danego miasta do każdego innego, natomiast w przypadku miast położonych na okręgu zawsze są dwie drogi: można obiegać okrąg w kierunku zgodnym z ruchem wskazówek zegara lub w kierunku przeciwnym. W przypadku każdej pary miast musimy zdecydować, który kierunek jest bardziej opłacalny. Przyjmijmy dla ustalenia uwagi, że miasta są numerowane w kierunku zgodnym z ruchem wskazówek zegara. Wczytane dane umieszczamy w dwóch tablicach: $z[i]$ jest zapotrzebowaniem na piwo w mieście i, $d[i]$ jest odległością od miasta i do następnego.

W algorytmie optymalnym najpierw obliczamy koszt transportu z miasta o numerze 1 (w programie wzorcowym dla wygody rozpoczęto numerację miast od zera; łatwiej wtedy obliczyć numer następnego i poprzedniego miasta na okręgu za pomocą funkcji **mod**). Jednocześnie obliczamy kilka wielkości pomocniczych:

- indeks l najdalszego miasta, do którego dowozimy piwo kierując się „w lewo”, tzn. przeciwnie do ruchu wskazówek zegara;
- indeks r najdalszego miasta, do którego dowozimy piwo kierując się „w prawo”, tzn. zgodnie z ruchem wskazówek zegara;
- łączne zapotrzebowanie z_l na piwo w miastach, do których je dowozimy jadąc „w lewo”;
- łączne zapotrzebowanie z_r na piwo w miastach, do których je dowozimy jadąc „w prawo”;

- odległość d_l z pierwszego miasta do miasta o numerze l, gdy jedziemy „w lewo";
- odległość d_r z pierwszego miasta do miasta o numerze r, gdy jedziemy „w prawo";
- łączny koszt transportu c — ten koszt zapamiętujemy jako dotychczas minimalny.

W kolejnych krokach algorytmu koszt transportu piwa z następnych miast będzie obliczany jako korekta obliczonego poprzednio kosztu. Przypuśćmy, że mamy obliczone te wszystkie dane dla miasta o numerze $i-1$. Wtedy dla miasta o numerze i dokonujemy prostej korekty. Najpierw znajdujemy odległość d między miastami o numerach $i-1$ oraz i. Następnie obliczamy, o ile zmieni się łączny koszt transportu piwa, jeśli będziemy je dowozić z miasta i po **tych samych** drogach: a więc do kosztu dodajemy $(z_l + z[i-1]) \cdot d$ i odejmujemy $z_r \cdot d$. Mianowicie wszystkie cysterny wysyłane w lewo przejadą odległość o d większą i dojdą nowe cysterny, które muszą dojechać do miasta o numerze $i-1$, a wszystkie cysterny jadące w prawo (w tym również te, które jechały do miasta i) przejadą o d mniej. Modyfikujemy również z_l (dodając $z[i-1]$), z_r (odejmując $z[i]$), d_l (dodając d) i d_r (odejmując d). Teraz poprawiamy łączny koszt, zwiększając liczbę cystern wysyłanych w prawo i zmniejszając liczbę cystern wysyłanych w lewo.

Patrzymy na odległość $d[r]$ między miastem r i następnym leżącym na prawo (czyli zgodnie z ruchem wskazówek zegara) miastem l. Jeśli $d_l > d_r + d[r]$, to do miasta o numerze l opłaca się teraz jechać w prawo. A więc dokonujemy kolejnej korekty: zwiększamy d_r o $d[r]$ i z_r o $z[r+1]$, zmniejszamy d_l o $d[l]$ i z_l o $z[l]$, zwiększamy o 1 numery miast r i l (pamiętając, że za $n+1$ bierzemy 1) i wreszcie odpowiednio modyfikujemy łączny koszt transportu. Teraz znów sprawdzamy, czy w podobny sposób nie przesunąć granicy między kierunkami w lewo i w prawo do następnego miasta. Postępujemy tak dotąd, aż dalsze zmiany nie przyniosą korzyści. W ten sposób zakończyliśmy korekty i mamy obliczony łączny koszt transportu piwa z miasta o numerze i. Porównujemy go z dotychczasowym minimalnym zapamiętując, jeśli jest mniejszy, i przechodzimy do kolejnego miasta.

Jaka będzie złożoność obliczeniowa tego algorytmu? Dla każdego miasta i dokonywaliśmy najpierw pewnej ustalonej liczby zmian, a następnie w pętli dokonywaliśmy być może wielu korekt. Zauważmy jednak, że każda z tych korekt dotyczyła zawsze nowej granicy między kierunkiem lewym i prawym, przy czym ta granica przesuwała się zawsze tylko w prawo. Zatem łącznie dokonaliśmy tylko n takich korekt. Stąd wynika, że czas działania tego algorytmu będzie proporcjonalny do n.

Dla liczby $n = 10000$ różnica w czasie działania obu algorytmów (pierwszego o złożoności $O(n^2)$ i drugiego o złożoności $O(n)$) jest znaczna i duże testy pozwalały na odrzucenie programów działających w czasie $\Theta(n^2)$. Łączny koszt transportu mógł też przekroczyć zakres liczb typu *longint* i niektóre testy wykrywały ten błąd.

TESTY

Do sprawdzania rozwiązań zawodników użyto 17 testów. Testy 1 i 2, 3 i 4, 5 i 6, 14 i 15 były grupowane.

- BRO0.IN — przykładowy test z treści zadania,
- BRO1.IN — $n = 10$, mały test poprawnościowy,
- BRO2.IN — $n = 15$, mały test poprawnościowy,
- BRO3.IN — $n = 20$, test sprawdzający błąd przekroczenia zakresu,
- BRO4.IN — $n = 20$, pary mają postać (i,i),
- BRO5.IN — $n = 30$, mały test poprawnościowy,
- BRO6.IN — $n = 40$, mały test poprawnościowy,
- BRO7.IN — $n = 500$, na przemian pary (1,1), (100,1),
- BRO8.IN — $n = 1000$, na przemian pary (1,1), (100,1000),
- BRO9.IN — $n = 3000$, trzy grupy zawierające po 1000 miast oddalone o 300000 km,
- BRO10.IN — $n = 4000$, pary postaci ((i+1001) mod 1000, 100),
- BRO11.IN — $n = 5000$, pary postaci ($3000 - i$ mod 1000, 100+(i mod 100)),
- BRO12.IN — $n = 7000$, pary mają postać ((i+1000) mod 1000, 100),
- BRO13.IN — $n = 7500$, dane losowe,
- BRO14.IN — duży prosty test dla $n = 10000$, wszystkie pary mają postać (1,1),
- BRO15.IN — mały test poprawnościowy dla $n = 20$,
- BRO16.IN — $n = 10000$, dane losowe.

Wojciech Rytter
autor zadania, opracowanie

Piotr Sankowski
program wzorcowy

Wirusy

Komisja Badania Wirusów Binarnych wykryła, że pewne ciągi zer i jedynek są kodami wirusów. Komisja wyodrębniła zbiór wszystkich kodów wirusów. Ciąg zer i jedynek nazywamy **bezpiecznym***, gdy żaden jego segment (tj. ciąg kolejnych wyrazów) nie jest kodem wirusa. Komisja dąży do ustalenia, czy istnieje nieskończony, bezpieczny ciąg zer i jedynek.*

Przykład

Dla zbioru kodów $\{011, 11, 00000\}$ *nieskończonym, bezpiecznym ciągiem jest 010101 ....*
Dla zbioru kodów $\{01, 11, 00000\}$ *nie istnieje nieskończony, bezpieczny ciąg zer i jedynek.*

Zadanie

Napisz program, który:

- *wczyta kody wirusów z pliku tekstowego* `WIR.IN`*,*
- *stwierdzi, czy istnieje nieskończony, bezpieczny ciąg zer i jedynek,*
- *zapisze wynik do pliku tekstowego* `WIR.OUT`*.*

Wejście

W pierwszym wierszu pliku wejściowego `WIR.IN` *znajduje się jedna liczba całkowita* n *będąca liczbą wszystkich kodów wirusów.*

W każdym z kolejnych n *wierszy znajduje się jedno niepuste słowo złożone ze znaków* `0` *i* `1` *— kod wirusa. Sumaryczna długość wszystkich słów nie przekracza 30000.*

Wyjście

W pierwszym i jedynym wierszu pliku wyjściowego `WIR.OUT` *powinno znajdować się słowo:*

- `TAK` *— jeżeli istnieje nieskończony, bezpieczny ciąg zer i jedynek,*
- `NIE` *— w przeciwnym przypadku.*

Przykład

Dla pliku wejściowego `WIR.IN`*:*

```
3
01
11
00000
```

poprawną odpowiedzią jest plik tekstowy `WIR.OUT`*:*
`NIE`

ROZWIĄZANIE

Algorytm teorio-grafowy

W rozwiązaniu zadania najważniejszą rolę odgrywają sufiksy i prefiksy wirusów. Jeśli słowo α można zapisać jako złożenie trzech słów $\alpha = u \cdot v \cdot w$, to u nazywamy prefiksem, a w sufiksem α. Na przykład prefiksami słowa *rytter* są $\{\epsilon, r, ry, ryt, rytt, rytte, rytter\}$, a sufiksami $\{\epsilon, r, er, ter, tter, ytter, rytter\}$, gdzie ϵ oznacza słowo puste o długości zero.

Dla słowa α i zbioru słów-wirusów W, przez $max\_pref(\alpha)$ oznaczmy najdłuższy sufiks α, który jest jednocześnie prefiksem pewnego wirusa z W. Jeśli chcemy sprawdzić, czy dany ciąg binarny nie zawiera wirusa, to wystarczy przeglądać ten ciąg znak po znaku i pamiętać jedynie ostatni **istotny** fragment. Tym istotnym fragmentem jest maksymalny prefiks wirusa, zatem po wczytaniu ciągu $a_1a_2a_3 \ldots a_k$ pamiętamy $max\_pref(a_1a_2a_3 \ldots a_k)$. Słowo $a_1a_2a_3 \ldots a_n$ jest *bezpieczne*, gdy każdy z pośrednich istotnych fragmentów jest bezpieczny, tzn. gdy
$max\_pref(a_1)$, $max\_pref(a_1a_2)$, $max\_pref(a_1a_2a_3)$, $\ldots$, $max\_pref(a_1a_2a_3 \ldots a_n)$ są bezpieczne (nie są wirusami).
Motywuje to wprowadzenie pomocniczego grafu G, którego ścieżki odpowiadają dokładnie wszystkim możliwym bezpiecznym słowom i którego węzłami są wszystkie bezpieczne prefiksy wirusów. Od węzła x prowadzimy skierowaną krawędź do y, gdy dla pewnej litery a (gdzie $a = 0$ lub $a = 1$) po dopisaniu jej do x otrzymamy słowo takie, że maksymalnym jego sufiksem, który jest prefiksem pewnego wirusa, jest y. Oznaczmy y przez $(\sigma(x, a))$. Można to zapisać bardziej formalnie:

- $\sigma(x, a) = max\_pref(xa)$, gdy xa jest bezpieczny,
- $\sigma(x, a) =$ **NIL**, gdy xa nie jest bezpieczny (**NIL** oznacza brak krawędzi).

Rysunek 2 przedstawia graf G dla przykładowego zbioru wirusów

$$W = \{000,\ 0110,\ 100101, 1010, 111\}.$$

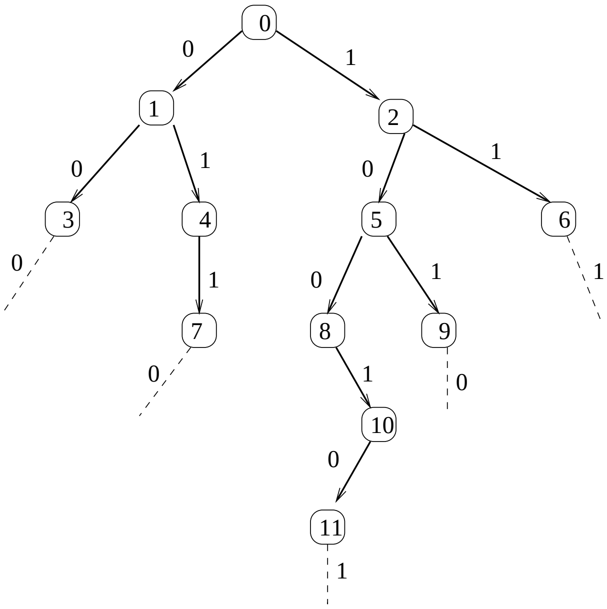
Grafy tego typu odpowiadają w *teorii obliczeń* tzw. *automatom skończonym.* Zbiór W jest bezpieczny wtedy i tylko wtedy, gdy w grafie G istnieje cykl skierowany osiągalny z węzła startowego odpowiadającego słowu pustemu. Każda ścieżka w tym grafie zaczynająca się od węzła startowego wyznacza ciąg bezpieczny.

Konstrukcja drzewa bezpiecznych prefiksów wirusów

Pierwszym krokiem w konstrukcji grafu G jest zbudowanie drzewa właściwych prefiksów wirusów, tzn. takich prefiksów słów ze zbioru W, które nie są równe żadnemu wirusowi. Drzewo dla przykładowego zbioru W jest przedstawione na rysunku 1.

Uwaga: Załóżmy początkowo, że żaden wzorzec nie jest właściwym podsłowem innego wzorca.

Rys. 1 Początkowa "aproksymacja" grafu G: drzewo T bezpiecznych prefiksów wirusów. Linie przerywane oznaczają przejścia zabronione. Brakujące niezabronione przejścia są uzupełnione w grafie G.



Wierzchołki są ponumerowane, jednakże każdy węzeł utożsamiamy z pewnym prefiksem wzorca. Załóżmy, że $syn(x, a)$ oznacza węzeł xa, jeśli xa jest bezpiecznym prefiksem wzorca. Jeśli $xa \in W$, to $syn(x, a) = N$, gdzie N jest specjalną wartością oznaczającą, że krawędzi etykietownej symbolem a z węzła x nigdy być nie może. W pozostałych przypadkach $syn(x, a) = \mathbf{NIL}$.

Właściwa konstrukcja grafu G

Algorytm korzysta istotnie z drzewa prefiksów T zbudowanego wcześniej i opisanego funkcją syn.

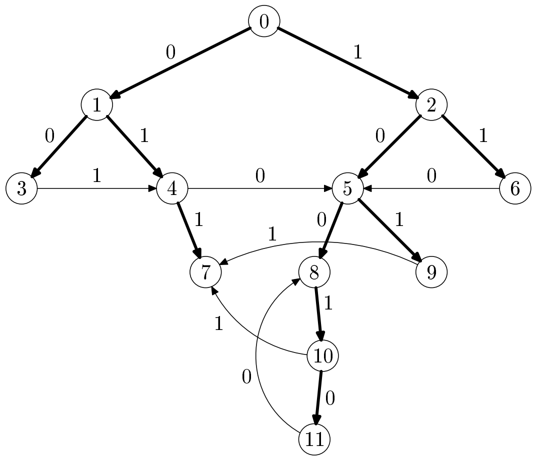
Algorytm
Inicjujemy $\sigma(\epsilon, a) = \epsilon$, $\sigma(\alpha, a) = NIL$, dla a=0,1 i $\alpha \neq \epsilon$ (słowa α odpowiadają węzłom w drzewie).
Przechodząc drzewo T wszerz, dla każdego węzła $x \neq root$ wykonaj:

niech y oraz a będą takie, że $x = syn(y, a)$;
if $\sigma(y, a) \neq NIL$ **then begin**
  $temp := \sigma(y, a)$; $\sigma(y, a) := x$;
  for b:=0 **to** 1 **do begin**

```
        if syn(x,b)=N ∨ syn(temp,b)=N then
            σ(x,b):=NIL
        else if syn(temp,b) ≠ NIL then
            σ(x,b):=syn(temp,b)
        else σ(x,b):=σ(temp,b)
    end
end
```

Na zakończenie algorytmu należy ograniczyć graf G do jego podgrafu indukowanego przez wierzchołki osiągalne z wierzchołka startowego.
Uwaga: Założenie, że żaden wirus nie jest podsłowem innego wirusa było użyteczne przy wprowadzeniu pomocniczego drzewa prefiksów. Algorytm konstrukcji grafu automatycznie wykrywa takie sytuacje, tak więc założenie to nie jest istotne.

Rys. 2 Graf G dla przykładowego zbioru wirusów. Pogrubione krawędzie (przejścia) są pozostałością z drzewa prefiksów T, pozostałe krawędzie powstały w wyniku działania algorytmu konstrukcji G.



Sprawdzanie acykliczności

Aby stwierdzić, czy w grafie jest cykl należy sprawdzić, czy wierzchołki grafu można posortować topologicznie. Można zastosować dwie alternatywne metody:

- sortowanie topologiczne przy pomocy DFS,
- sortowanie topologiczne poprzez usuwanie wierzchołków ze stopniem wejściowym równym zero.

Nie opisujemy dokładnie tych metod ponieważ sprawdzanie acykliczności grafu wystąpiło już w wielu poprzednich zadaniach olimpijskich.
Dla naszego przykładowego zbioru wirusów nieskończona ścieżka "idąca" po węzłach grafu G.

$$0,\ 2,\ 5,\ (8,\ 10,\ 11)^{\infty}$$

odpowiada nieskończonemu bezpiecznemu słowu:

$$100(100)^{\infty}$$

Algorytm alternatywny

Wprowadzamy dwie operacje na zbiorze W. Niech $x, y \in W$.

Usuwanie y. Jeśli x jest podsłowem y, to y usuwamy z W.

Skracanie y. Jeśli $y = za$, gdzie $a = 0 \ \vee \ a = 1$, oraz $x = u\bar{a}$ i u jest sufiksem z, to $y{:=}z$.

Symbol $\bar{a}$ oznacza $1 - a$.
Na przykład jeśli $x = 0010$, $y = 001010011$, to skracając y dostajemy $y = 00101001$. Operację skracania możemy zastosować dlatego, że jeśli w pewnym *nieskończonym* ciągu wystąpi skrócone y, to albo w tym ciągu wystąpi samo y, albo x.
Niech W' będzie zbiorem powstałym z W za pomocą jednej z tych operacji, wtedy:

$$W \text{ jest bezpieczny} \Leftrightarrow W' \text{ jest bezpieczny.}$$

Łatwo zauważyć, że W jest niebezpieczny wtedy i tylko wtedy, gdy po pewnej liczbie operacji otrzymamy zbiór zawierający oba pojedyncze symbole jako słowa. Natomiast, jeśli w wyniku dowolnego ciągu operacji otrzymamy zbiór zawierający słowo o długości większej niż jeden, to W jest bezpieczny.

Można wówczas skonstruować dowolnie długi bezpieczny ciąg postępując w następujący sposób: zaczynamy od ciągu pustego i dokładamy po jednym symbolu, do chwili gdy otrzymujemy ciąg niebezpieczny (gdy sufiks należy do w); zamieniamy ostatnio dodany symbol na przeciwny i kontynuujemy naszą konstrukcję.

Nasz algorytm polega na stosowaniu w dowolnej kolejności jednej z powyższych operacji, aż otrzymany zbiór zawierający obie litery $0, 1$, albo zbiór zawierający dłuższe słowo, który to zbiór nie da się dalej zmienić za pomocą jednej z naszych operacji. W pierwszym przypadku W jest niebezpieczny. Dla naszego przykładowego zbioru wirusów W z rysunku 1 od razu widać, że jest on bezpieczny, ponieważ żadna z dwóch operacji się do niego nie stosuje.

Przykład

Rozważmy przykład zbioru, który nie jest bezpieczny. W poniższym ciągu przekształceń podkreślamy słowo biorące udział w przekształceniu, słowo to jest usuwane lub skracane o jeden symbol od końca.

$$\{000, \underline{010}, 11\} \to \{\underline{000}, 01, 11\} \to \{00, \underline{01}, 11\} \to \{\underline{00}, 0, 11\} \to \{0, \underline{11}\} \to \{0, 1\}.$$

Zatem $\{000, 010, 11\}$ jest niebezpieczny, gdyż końcowy zbiór $\{0, 1\}$ jest niebezpieczny.

TESTY

Do sprawdzania rozwiązań zawodników użyto 10 testów:

- WIR1-3.IN — małe testy poprawnościowe,
- WIR4.IN — średni test poprawnościowy,
- WIR5.IN — test wydajnościowy sprawdzający metodę wyszukiwania cyklu, tylko bardzo długie cykle,
- WIR6.IN — mały test wydajnościowy, jeden cykl w automacie,
- WIR7.IN — duży test wydajnościowy, duża liczba możliwych nieskończonych słów,
- WIR8.IN — przypadek pesymistyczny dla iloczynu automatów, dużo krótkich słów,
- WIR9.IN — duży test wydajnościowy, mała liczba możliwych nieskończonych słów,
- WIR10.IN — średni test wydajnościowy, tylko jeden cykl w automacie.

Marcin Kubica
autor zadania, opracowanie

Marek Pawlicki
program wzorcowy

Narciarze

Drużyna narciarska organizuje trening na Bajtogórze. Na północnym stoku góry znajduje się jeden wyciąg. Wszystkie trasy prowadzą od górnej do dolnej stacji wyciągu. W trakcie treningu członkowie drużyny będą razem startować z górnej stacji wyciągu i spotykać się przy dolnej stacji. Poza tymi dwoma punktami trasy, zawodników nie mogą się przecinać, ani stykać. Wszystkie trasy muszą cały czas prowadzić w dół.

Mapa tras narciarskich składa się z sieci polan połączonych przecinkami. Każda polana leży na innej wysokości. Dwie polany mogą być bezpośrednio połączone co najwyżej jedną przecinką. Zjeżdżając od górnej do dolnej stacji wyciągu można tak wybrać drogę, żeby odwiedzić dowolną polanę (choć być może nie wszystkie w jednym zjeździe). Trasy narciarskie mogą się przecinać tylko na polanach i nie prowadzą przez tunele, ani estakady.

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `NAR.IN` *mapę tras narciarskich,*
- *wyznaczy maksymalną liczbę zawodników, którzy mogą brać udział w treningu,*
- *zapisze wynik do pliku tekstowego* `NAR.OUT`*.*

Wejście

W pierwszym wierszu pliku wejściowego `NAR.IN` *znajduje się jedna liczba całkowita n równa liczbie polan, $2 \leqslant n \leqslant 5\,000$.*

W każdym z kolejnych $n-1$ wierszy znajduje się ciąg liczb całkowitych pooddzielanych pojedynczymi odstępami. Liczby w $(i+1)$-szym wierszu pliku określają, do których polan prowadzą w dół przecinki od polany nr i. Pierwsza liczba k w wierszu określa liczbę tych polan, a kolejne k liczb to ich numery, które są uporządkowane wg ułożenia prowadzących do nich przecinek w kierunku ze wschodu na zachód. Polany są ponumerowane liczbami od 1 do n. Górna stacja wyciągu znajduje się na polanie numer 1, a dolna na polanie numer n.

Wyjście

W pierwszym i jedynym wierszu pliku wyjściowego `NAR.OUT` *powinna znajdować się dokładnie jedna liczba całkowita — maksymalna liczba narciarzy mogących wziąć udział w treningu.*

Przykład

Dla pliku wejściowego `NAR.IN`*:*

```
15
5 3 5 9 2 4
```

```
1 9
2 7 5
2 6 8
1 7
1 10
2 14 11
2 10 12
2 13 10
3 13 15 12
2 14 15
1 15
1 15
1 15
```

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

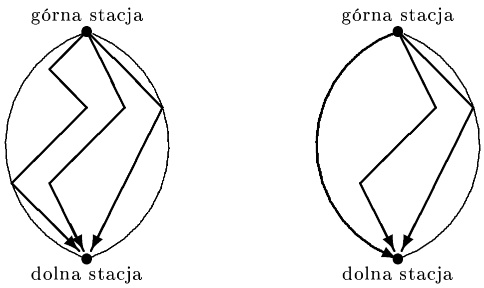
```
3
```

ROZWIĄZANIE

Zadanie to możemy sformułować w języku teorii grafów. Mamy dany acykliczny, planarny graf skierowany — wierzchołki tego grafu to polany i stacje wyciągu, a krawędzie to przecinki. W grafie tym mamy wyróżnione dwa wierzchołki reprezentujące górną i dolną stację wyciągu. Z górnej stacji wyciągu osiągalne są wszystkie wierzchołki w grafie. Podobnie, z każdego wierzchołka w grafie osiągalna jest dolna stacja wyciągu. Ponadto stacje wyciągu (jako najwyżej i najniżej położone wierzchołki) znajdują się na obwodzie grafu. Należy wyznaczyć maksymalną liczbę k takich ścieżek w grafie, które prowadzą od górnej do dolnej stacji wyciągu i poza stacjami wyciągu nie mają (parami) żadnych wspólnych wierzchołków.

Załóżmy chwilowo, że w grafie nie ma krawędzi łączącej bezpośrednio górną i dolną stację wyciągu. Jeżeli taka krawędź jest, to możemy ją usunąć, obliczyć wynik, po czym zwiększyć go o 1.

Zadanie to możemy rozwiązać za pomocą algorytmu zachłannego. Wyobraźmy sobie, że wybraliśmy k ścieżek spełniających warunki zadania. Oznaczmy te ścieżki, od lewej do prawej (ze wschodu na zachód), przez $s_1, \ldots, s_k$. Zauważmy, że bez łamania warunków zadania możemy przesunąć skrajnie lewą (wschodnią) ścieżkę s_1 w lewo tak, aby prowadziła przez wierzchołki leżące na lewym pół-obwodzie grafu.



Rys. 1

Tak więc, bez zmniejszenia ogólności, możemy przyjąć, że s_1 biegnie przez wierzchołki leżące na lewym pół-obwodzie grafu (zobacz rysunek 1).

Podobnie możemy poprzesuwać pozostałe ścieżki. Zauważmy, że wśród wszystkich możliwych ścieżek, które prowadzą od górnej do dolnej stacji wyciągu i nie mają (za wyjątkiem stacji wyciągu) wspólnych wierzchołków z s_1, istnieje skrajnie lewa ścieżka. Bez naruszenia warunków zadania możemy przyjąć, że s_2 jest właśnie taką ścieżką. Analogicznie ustalamy kolejne ścieżki. Przyjmujemy, że s_i jest skrajnie lewą spośród wszystkich możliwych ścieżek, które prowadzą od górnej do dolnej stacji wyciągu i nie mają (za wyjątkiem stacji wyciągu) wspólnych wierzchołków z żadną ze ścieżek $s_1, \ldots, s_{i-1}$.

Wyłania się następujący schemat algorytmu:

(1) $i := 1$;

(2) powtarzaj, dopóki można, konstrukcję kolejnych ścieżek:

  (a) wyznacz s_i jako skrajnie lewą ścieżkę prowadzącą od górnej do dolnej stacji wyciągu i (dla $i > 1$) nie przechodzącą (za wyjątkiem stacji wyciągu) przez wierzchołki należące do ścieżek $s_1, \ldots, s_{i-1}$,

  (b) jeśli konstrukcja ścieżki się udała, to zwiększ i o 1;

(3) jeżeli w zadanym grafie była krawędź prowadząca bezpośrednio z górnej do dolnej stacji wyciągu, to zwiększ i o 1;

(4) $(i - 1)$ jest równe szukanej liczbie ścieżek.

Pozostał do rozwiązania problem jak wyznaczyć skrajnie lewą ścieżkę w odpowiednim podgrafie zadanego grafu. Ścieżkę taką konstruujemy począwszy od górnej stacji wyciągu, korzystając z procedury przeszukiwania grafu w głąb (zobacz [13] lub [10]). Ważne jest, aby w trakcie przeszukiwania przeglądać krawędzie wychodzące z danego wierzchołka w kolejności od lewej do prawej (ze wschodu na zachód), czyli zgodnie z kolejnością, w jakiej zostały one podane w pliku wejściowym. W momencie osiągnięcia dolnej stacji wyciągu przerywamy przeszukiwanie — szukana ścieżka znajduje się wówczas na stosie. W trakcie przeszukiwania grafu kolorujemy odwiedzone wierzchołki. Zauważmy, że pokolorowane zostają wierzchołki tworzące skonstruowaną ścieżkę oraz niektóre wierzchołki leżące na lewo od tej ścieżki. Konstruując kolejne ścieżki omijamy wierzchołki już pokolorowane (za wyjątkiem stacji wyciągu).

Powyższy algorytm znajdowania skrajnie lewej ścieżki można zaimplementować w następujący sposób.

```
const
  MaxN = 5 000; { Maksymalna liczba polan (wierzchołków grafu) }
type
  { Lista sąsiedztwa jednego wierzchołka. }
  { Element 0-wy zawiera liczbę wierzchołków na liście. }
  TListaSasiedztwa = array [0..MaxN] of Integer;
  PListaSasiedztwa = ↑TListaSasiedztwa;
var
  { Graf reprezentowany jako listy sąsiedztwa wierzchołków. }
```

```
  T: array [1..MaxN] of PListaSasiedztwa;
  { Tablica do kolorowania odwiedzonych wierzchołków. }
  Odwiedzony: array [1..MaxN] of Boolean;
  N: Integer; { Liczba wierzchołków. }
function Wglab (w: Integer): Boolean;
{ Realizuje przeszukanie grafu T w głąb począwszy od wierzchołka w.
  Zwraca true w przypadku dotarcia do dolnej stacji (sukcesu). }
var
  i: Integer;
  s: Boolean;
begin
  if w = N then
    { Osiągnięto dolną stację wyciągu. }
    Wglab := true
  else begin
    Odwiedzony[w] := true; s := false; i := 1;
    { Przeszukuj aż do pierwszego sukcesu. }
    while not s and (i ⩽ T[w]↑[0]) do begin
      if not Odwiedzony [T[w]↑[i]] then
        s := Wglab (T[w]↑[i]);
      Inc (i)
    end;
    Wglab := s
  end
end;
```

Szukaną liczbę ścieżek możemy wyznaczyć w następujący sposób:

```
function ZnajdzRozwiazanie: Integer;
{ Znajduje rozwiązanie metodą poszukiwania najbardziej
  skrajnej ścieżki. }
var
  i: Integer;
  function UsunTraseBezposrednia: Boolean;
  { Usuwa ewentualną bezpośrednią krawędź 1→N,
    zwraca true, gdy takie połączenie istniało. }
  var
    i, j, lnast: Integer;
    zn: Boolean;
  begin
    i := 1; zn := false; lnast := T[1]↑[0];
    while (i ⩽ lnast) and not zn do
      if T[1]↑[i] = N then
        zn := true
      else
        Inc (i);
    if zn then begin
```

```
      for j := i to lnast - 1 do T[1]↑[j] := T[1]↑[j+1];
      Dec (T[1]↑[0])
    end
    UsunTraseBezposrednia := zn
  end
begin { ZnajdzRozwiazanie. }
  for i := 1 to N do Odwiedzony [i] := false;
  if UsunTraseBezposrednia then i := 1 else i := 0;
  while Wglab (1) do Inc (i);
  ZnajdzRozwiazanie := i
end
```

Z twierdzenia Eulera wiadomo (zobacz [10]), że w każdym spójnym grafie planarnym liczba krawędzi m jest takiego rzędu jak liczba wierzchołków n. Dokładniej, dla $n \geqslant 3$ mamy $n - 1 \leqslant m \leqslant 3n - 6$. Jeżeli więc nie będziemy alokowali w pamięci całych tablic sąsiedztwa, a jedynie ich wypełnione fragmenty, to uzyskamy złożoność pamięciową algorytmu rzędu n.

Sprawdzenie, czy w grafie jest krawędź łącząca bezpośrednio górną i dolną krawędź wyciągu wymaga przejrzenia wszystkich krawędzi wychodzących z górnej stacji wyciągu. Tak więc złożoność czasowa tej operacji jest rzędu n. Zauważmy, że w trakcie przeglądania grafu po każdej krawędzi przechodzimy co najwyżej raz. Stąd, złożoność czasowa przeszukiwania grafu jest rzędu n. Tak więc, złożoność czasowa całego rozwiązania jest rzędu n.

TESTY

Do sprawdzenia rozwiązań zawodników użyto 8-miu testów. Testy te można podzielić na trzy grupy:

- NAR1–3.IN — proste testy sprawdzające poprawność rozwiązania,
- NAR4–6.IN — bardziej skomplikowane testy sprawdzające poprawność rozwiązania,
- NAR7–8.IN — testy badające złożoność rozwiązania.

W poniższej tabelce podano wielkości i wyniki dla poszczególnych testów.

Test	n	m	i
1	2	1	1
2	15	26	2
3	20	37	4
4	900	1 740	2
5	2 502	2 901	5
6	4 006	9 001	1 001
7	4 600	10 005	3
8	4 913	9 524	16

Adam Borowski — autor zadania
Marcin Sawicki — opracowanie
Tomasz Waleń — program wzorcowy

Paski

Paski *to gra dwuosobowa. Rekwizytami potrzebnymi do gry są paski w trzech kolorach: czerwonym, zielonym i niebieskim. Wszystkie paski czerwone mają wymiary* $c \times 1$*, zielone* $z \times 1$*, a niebieskie* $n \times 1$*, gdzie* c*,* z *i* n *są liczbami naturalnymi. Gracze dysponują nieograniczoną pulą pasków każdego koloru (jeśli pewnych pasków zabraknie, to zawsze można dodrukować).*

Plansza do gry jest prostokątem o wymiarach $p \times 1$ *i składa się z* p *pól o wymiarach* 1×1*.*

Gracze wykonują ruchy na przemian. Ruch polega na ułożeniu na planszy paska dowolnego koloru. Obowiązują przy tym następujące zasady:

- *pasek nie może wystawać poza planszę,*
- *nie wolno przykrywać (nawet częściowo) pasków ułożonych wcześniej,*
- *końce paska muszą pokrywać się z brzegami pól planszy*

Przegrywa gracz, który jako pierwszy nie może wykonać ruchu zgodnie z zasadami gry. **Pierwszy gracz** *to gracz, który wykonuje pierwszy ruch w grze. Mówimy, że pierwszy gracz ma strategię wygrywającą, jeżeli niezależnie od posunięć drugiego gracza zawsze może wygrać.*

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `PAS.IN` *wymiary pasków i co najmniej jednej planszy,*
- *dla każdej planszy stwierdzi, czy pierwszy gracz posiada strategię wygrywającą,*
- *zapisze wyniki w pliku tekstowym* `PAS.OUT`*.*

Wejście

Pierwszy wiersz zbioru wejściowego `PAS.IN` *zawiera trzy liczby naturalne* c*,* z *i* n*,* $1 \leqslant c, n, z \leqslant 1000$*, równe długościom pasków, odpowiednio, czerwonych, zielonych i niebieskich. Liczby w wierszu są poodzielane pojedynczymi znakami odstępu.*

Drugi wiersz pliku `PAS.IN` *zawiera jedną liczbę* m*,* $1 \leqslant m \leqslant 1000$*, równą liczbie różnych plansz do rozpatrzenia.*

Wiersze od 3 to $m + 2$ *zawierają po jednej liczbie* p*,* $1 \leqslant p \leqslant 1000$*. Liczba w wierszu* $i + 2$ *jest długością* i*-tej planszy.*

Wyjście

Plik wyjściowy `PAS.OUT` *powinien zawierać* m *wierszy. W* i*-tym wierszu pliku powinna być zapisana jedna liczba:*

- *1, jeżeli pierwszy gracz ma strategię wygrywającą na i-tej planszy,*
- *2, w przeciwnym przypadku.*

PRZYKŁAD

Dla pliku wejściowego `PAS.IN`*:*

```
1 5 1
3
1
5
6
```

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

```
1
1
2
```

ROZWIĄZANIE

W teorii gier istnieje pojęcie gry zamkniętej. Gry, które polegają na kolejnych jawnych posunięciach dwu graczy (np. gra w paski, a także szachy, warcaby, kółko i krzyżyk, wilk i owce itd.) są przykładami gier zamkniętych. Znane twierdzenie mówi, że wszystkie takie gry są niesprawiedliwe lub czcze, tzn. że zawsze albo jeden z graczy ma strategię wygrywającą, albo obaj gracze mogą co najwyżej wymusić remis lub nieskończoną rozgrywkę.

Gra w paski jest ponadto grą kategoryczną, czyli zawsze kończy się wygraną jednego z graczy. Jest zatem niesprawiedliwa: któryś z graczy musi mieć strategię prowadzącą do zwycięstwa.

Będąc młodym informatykiem, z pewnością zdajesz sobie sprawę, Szanowny Czytelniku, że teoretycznie można znajdować strategie wygrywające poprzez przeszukiwanie drzewa gry, i że chyba dla żadnej rozsądnej gry nie jest to wykonalne w praktyce. Można polemizować, czy paski są rozsądną grą, w każdym razie przeszukiwanie całego drzewa jest w tym wypadku zbyt pracochłonne.

Ruch w grze w paski można jednak postrzegać jako rozbicie planszy na mniejsze kawałki. Podsuwa nam to pomysł skorzystania z metod programowania dynamicznego. Jeśli umielibyśmy powiedzieć coś o tych „mniejszych kawałkach”, to być może umielibyśmy również sklasyfikować jakoś wyjściową planszę?

Chwila namysłu pozwala stwierdzić, że nie wystarczy informacja, który z graczy ma strategię wygrywającą na każdym z kawałków planszy, by ustalić, kto ją ma na całej planszy. Potrzebujemy zatem subtelniejszego rozróżnienia pomiędzy planszami, jednak takiego, z którego łatwo wynika, kto na danej planszy zwycięży. Jakiego? O tym już za chwilę...

Skąd ten XOR?

Szanowny Czytelniku! Być może poznałeś już program wzorcowy i wiesz, że w magiczny sposób korzysta on z właściwości operacji XOR. W tym artykule postaram się przekonać Cię, że w rozwiązaniu tym nie ma aż tak wiele magii. Użyję języka teorii grup, co w wielu miejscach uprości opis. Mimo wszystko, zajmie to jednak trochę czasu, a skoro Ty oglądałeś już rozwiązanie wzorcowe, to pewnie nie należysz do cierpliwych. Dlatego już teraz przedstawię główną myśl rozwiązania. Oto ona:

W grze w paski, jeśli w jakiejś chwili plansza daje się podzielić na dwa identyczne fragmenty, to wiadomo, że gracz, na którego przypada kolejka, musi przegrać: drugi gracz będzie powtarzał jego ruchy, tyle że symetrycznie, na drugim z identycznych fragmentów planszy. Jeśli pierwszy gracz może jeszcze wykonać ruch, to również i drugi może po nim wykonać ruch symetryczny. A jeśli pierwszy nie może, to przegrał!

Ta cecha gry w paski jest w pełni analogiczna do znanej właściwości operacji XOR, polegającej na tym, że dla dowolnej liczby k mamy k XOR $k = 0$. Liczba k odpowiada jednemu z dwu identycznych egzemplarzy planszy, zaś 0 oznacza porażkę pierwszego gracza.

Oczywiście jest to tylko intuicja. Dlatego zachęcam nawet Ciebie, Niecierpliwy Czytelniku, do dalszej lektury.

Terminologia

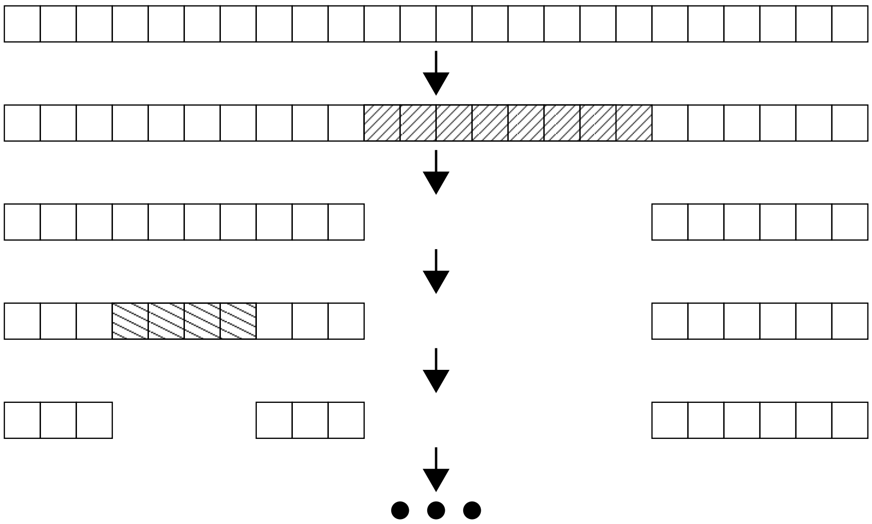
Na początku gry plansza jest pojedynczym prostokątem zbudowanym z $p \times 1$ kwadratowych pól. Kolejne ruchy polegają na zakrywaniu fragmentów planszy. Pola zakryte nie biorą dalej udziału w grze. Natomiast pola wolne po każdym ruchu tworzą pewien układ spójnych prostokątów. Przykładowo, na rysunku 1 widzimy początek pewnej rozgrywki na planszy długości 24. Po pierwszym ruchu wolne pola tworzą układ dwu prostokątów o długościach 10 i 6, a po następnym mamy trzy prostokąty o długościach 3, 3 i 6.

Definicja. Spójny prostokąt, zbudowany z wolnych pól, do którego skrajów przylegają pola zajęte lub brzegi planszy (czyli nie dający się powiększyć o kolejne wolne pola bez naruszania spójności), nazywać będziemy *obszarem*. Każdy układ obszarów (również początkowy, czyli zbudowany z jednego obszaru $p \times 1$) będziemy nazywać *planszą*.

Zwróćmy uwagę, że z punktu widzenia strategii rozgrywki (w szczególności tego, kto wygra), nie ma znaczenia kolejność, w jakiej na danej planszy występują poszczególne obszary. Czyli plansze 3, 3, 6 oraz 3, 6, 3 możemy uważać za jednakowe. Dlatego odpowiednim pojęciem matematycznym do reprezentowania plansz będzie *multizbiór*, którego elementami będą liczby całkowite dodatnie — długości poszczególnych obszarów.

Definicja. *Multizbiór* jest to zbiór par (x, n), gdzie x jest *elementem* multizbioru, zaś $n = 1, 2, 3, \ldots$ jest liczbą całkowitą dodatnią, określającą w ilu egzemplarzach dany element należy do multizbioru. Oczywiście dla każdego x do multizbioru może należeć co najwyżej jedna para postaci (x, n). Operacje sumy i części wspólnej są dla multizbiorów zdefiniowane w sposób naturalny.

Rys. 1 W trakcie rozgrywki plansza rozpada się na coraz mniejsze obszary



Na przykład, jeśli $A = \{(a,3),(b,1),(c,1)\}$, zaś $B = \{(a,2),(b,3),(d,5)\}$, to $A \cup B = \{(a,5),(b,4),(c,1),(d,5)\}$ oraz $A \cap B = \{(a,2),(b,1)\}$.

Plansze przedstawimy jako multizbiory długości tworzących je obszarów. Przykładowo, przedstawieniem planszy 3, 3, 6, jak również planszy 3, 6, 3, jest ten sam multizbiór $\{(3,2),(6,1)\}$. W dalszych rozważaniach potrzebna nam będzie operacja łączenia dwu plansz, czyli budowania planszy, składającej się ze wszystkich obszarów, wchodzących w skład obu plansz. Operacji łączenia plansz dokładnie odpowiada suma multizbiorów.

Odrobina abstrakcji

Rozważamy grę rozpoczynającą się od planszy $p \times 1$ i prowadzoną przy użyciu pasków o długościach c, z i n. W każdym stanie gry plansza będzie zatem zbudowana z obszarów o długościach $\leqslant p$. Oczywiście, nie każda plansza zbudowana z takich obszarów może się pojawić w naszej rozgrywce, w szczególności suma długości obszarów nigdy nie przekroczy p.

Na nasze potrzeby wygodnie jednak będzie o tym zapomnieć i zbadać zbiór X wszystkich plansz, zbudowanych ze skończonej liczby obszarów, z których każdy ma długość $\leqslant p$. Zatem X jest rodziną skończonych multizbiorów, których elementy są liczbami od 1 do p. Dzięki temu, w zbiorze X wykonalne jest działanie łączenia plansz (czyli dodawania multizbiorów) — to znaczy, że plansza uzyskana przez połączenie dwu elementów zbioru X również należy do tego zbioru. Zwróćmy uwagę, że w zbiorze X znajduje się również plansza pusta, czyli pusty multizbiór (odpowiada ona całkowitemu wypełnieniu planszy paskami). Oznaczmy ją przez $\emptyset$. Dla dowolnego $x \in X$ mamy oczywiście $x \cup \emptyset = x$, czyli $\emptyset$ jest elementem neutralnym dla działania łączenia plansz. Nadto łączenie plansz jest, jak każdy widzi, przemienne i (jak sama nazwa wskazuje) łączne.

Definicja. Zbiór Y z działaniem dwuargumentowym $+$ nazywamy *grupą przemienną* albo *abelową*, jeśli spełnia on następujące warunki:

- Łączność: dla dowolnych $y_1, y_2, y_3 \in Y$ mamy $y_1 + (y_2 + y_3) = (y_1 + y_2) + y_3$.
- Przemienność: dla dowolnych $y_1, y_2 \in Y$ zachodzi $y_1 + y_2 = y_2 + y_1$.
- Istnieje element $e \in Y$ taki, że dla każdego $y \in Y$ zachodzi $y + e = e + y = y$. Element e nazywamy *elementem neutralnym* grupy.
- Dla każdego elementu $y \in Y$ istnieje taki $y' \in Y$, że $y + y' = e$. Przyjmuje się oznaczenie $-y = y'$. Mówimy, że $-y$ jest *elementem odwrotnym* do y.

Pojęcie grupy przemiennej powinno być dobrze znane każdemu: najprostsze przykłady to liczby całkowite (lub wymierne, lub rzeczywiste) z dodawaniem (w tym kontekście mówi się raczej o elemencie przeciwnym, a nie odwrotnym), liczby wymierne (lub rzeczywiste) bez zera z mnożeniem, wektory na płaszczyźnie z dodawaniem wektorów, itd.

Widzimy, że zbiór X z działaniem $\cup$ spełnia wszystkie warunki oprócz odwracania. Przypomnijmy sobie jednak, że na planszy, dającej się podzielić na dwa identyczne fragmenty gracz, na którego przypada kolejka, musi przegrać, tak samo jak na planszy $\emptyset$. Z punktu widzenia strategii można więc uważać, że plansza postaci $x \cup x$ jest równoważna planszy $\emptyset$. Postaramy się zatem tak przerobić zbiór X z działaniem $\cup$, by każdy jego element był odwrotny sam do siebie. Z grubsza biorąc, utożsamimy elementy postaci $x \cup x$ z $\emptyset$. Znów będzie nam potrzebna

Definicja. Relacja dwuargumentowa $\sim$ na zbiorze Y jest *relacją równoważności*, jeśli jest

- *zwrotna*, tzn. dla każdego $y \in Y$ zachodzi $y \sim y$,
- *symetryczna*, tzn. dla każdych $y_1, y_2 \in Y$ jeśli $y_1 \sim y_2$, to również $y_2 \sim y_1$,
- *przechodnia*, czyli jeśli dla $y_1, y_2, y_3 \in Y$ zachodzi $y_1 \sim y_2$ oraz $y_2 \sim y_3$, to także $y_1 \sim y_3$.

Relacje równoważności są matematycznym sposobem na utożsamienie pewnych elementów zbioru: każda relacja równoważności na zbiorze Y jednoznacznie zadaje jego *podział*, czyli rodzinę $\mathcal{A}$ podzbiorów zbioru Y taką, że $y_1 \sim y_2$ wtedy i tylko wtedy, gdy dla pewnego elementu B rodziny $\mathcal{A}$ zachodzi jednocześnie $y_1 \in B$ oraz $y_2 \in B$. Taką rodzinę oznacza się przez ${}^{Y}/_{\sim} = \mathcal{A}$. Każdy element $y \in Y$ należy do dokładnie jednego elementu $B \in {}^{Y}/_{\sim}$; ten jedyny element oznaczamy $[y] = B$ i nazywamy *klasą abstrakcji* elementu y. Oczywiście $[y_1] = [y_2]$ wtedy i tylko wtedy, gdy $y_1 \sim y_2$. Czyli $[y] \subseteq Y$ jest zbiorem tych i tylko tych elementów zbioru Y, które właśnie utożsamiliśmy z y za pomocą relacji $\sim$.

Dobrym pomysłem okazuje się następujące określenie naszej relacji: $x_1 \equiv x_2$ wtedy i tylko wtedy, gdy na planszy $x_1 \cup x_2$ gracz rozpoczynający musi przegrać (tzn. jego przeciwnik posiada strategię wygrywającą). Wiemy zatem, że zawsze $x \equiv x$. Poza tym jest oczywiście wiele innych par x_1, x_2 takich, że $x_1 \equiv x_2$ — jak się niedługo okaże, działa to na naszą korzyść. Przypominam, że rozmiary pasków c, z, n są cały czas ustalone. Dla innych rozmiarów pasków możemy otrzymać zupełnie inną relację!

Z drugiej strony, uważny Czytelnik niedługo już zorientuje się, że utożsamianie plansz, na których gracz rozpoczynający ma strategię wygrywającą, byłoby dużo gorszym pomysłem. Poza bardzo prostymi przypadkami, musielibyśmy wówczas w trosce o przechodniość naszej relacji konsekwentnie posklejać ze sobą wszystko.

Zadanie dla czytelnika. Proszę sprawdzić, że $\equiv$ jest istotnie relacją równoważności (spełnia definicję). Wskazówka: aby udowodnić przechodniość, należy, znając strategię wygrywającą drugiego gracza dla plansz $x_1 \cup x_2$ oraz $x_2 \cup x_3$, opisać sposób (algorytm) gry dla drugiego gracza, który zagwarantuje mu zwycięstwo na planszy $x_1 \cup x_3$.

Zagadka tylko dla wtajemniczonych: czy jeśli zdefiniowalibyśmy $\simeq$ jako najmniejszą relację równoważności na X, która utożsamia wszystkie plansze postaci $x \cup x$ z $\emptyset$, to czy $\equiv = \simeq$? A gdyby dodatkowo zarządać, by $\simeq$ było najmniejszą kongruencją na X ze względu na $\cup$?

No dobrze, powiecie, utożsamiliśmy jakieś elementy, ale przecież działanie $\cup$ było określone na X, a my mamy teraz inny zbiór ${}^{X}/_{\equiv}$. A jednak wszystko jest pod kontrolą! Przypomnijmy tylko jedno pojęcie, z którym każdy spotkał się (lub spotka) w szkole:

Definicja. Relacja równoważności $\sim$ jest *kongruencją* na zbiorze Y ze względu na działanie $+$, jeśli dla dowolnych $y_1, y_1', y_2, y_2' \in Y$ jeśli tylko $y_1 \sim y_1'$ oraz $y_2 \sim y_2'$, to zachodzi również $y_1 + y_2 \sim y_1' + y_2'$.

Przypadek „szkolny” to relacja przystawania modulo n, która utożsamia liczby całkowite $a, b \in \mathbf{Z}$ wtedy i tylko wtedy, gdy $a - b$ dzieli się przez n. Jest to relacja równoważności, co więcej jest to kongruencja ze względu na dodawanie, odejmowanie i mnożenie.

Kongruencja pozwala wykonywać działania w zbiorze klas abstrakcji ${}^{Y}/_{\sim}$: aby np. dodać dwie klasy, bierzemy dowolną parę ich elementów i dodajemy, a następnie znajdujemy klasę abstrakcji sumy. To, że relacja jest kongruencją, nie oznacza nic więcej, niż że tak zdefiniowane działanie jest dobrze określone w zbiorze ${}^{Y}/_{\sim}$, tzn. że różne wybory reprezentantów prowadzą w wyniku do tej samej klasy abstrakcji.

Na całe szczęście, $\equiv$ jest kongruencją w X ze względu na $\cup$. Proszę to sprawdzić wprost z definicji, dowód będzie podobny do dowodu faktu, że relacja $\equiv$ jest przechodnia.

Mamy zatem

Fakt. ${}^{X}/_{\equiv}$ z działaniem zadanym przez $\cup$ jest grupą przemienną, z elementem neutralnym $[\emptyset]$ (klasa abstrakcji zbioru pustego). Działanie w tej grupie również będziemy oznaczać $\cup$, zaś element neutralny po prostu przez $0 = [\emptyset]$.

Istotnie, podzielenie X przez $\equiv$ nie mogło popsuć łączności, ani przemienności $\cup$, ani neutralności $\emptyset$ (która przeniosła się na całą klasę abstrakcji $[\emptyset]$), i oczywiście uzyskaliśmy elementy odwrotne: każdy element jest odwrotny sam do siebie.

Do czego była nam potrzebna ta „odrobina” abstrakcji?

Potrzeba wyjaśnić, czemu miał służyć ten długi ciąg definicji. Celów było kilka. Po pierwsze, możemy teraz zwięźle sformułować nasz cel: algorytm ma stwierdzać, czy dla danej planszy x, składającej się z jednego obszaru długości p, zachodzi $x \equiv \emptyset$, czyli czy $[x] = 0$ (gdyż zgodnie z naszą definicją, $x \equiv \emptyset$ wtedy i tylko wtedy, gdy drugi gracz ma strategię wygrywającą na planszy $x \cup \emptyset = x$).

Po drugie, bez określenia naszej grupy $\langle {}^{X}/_{\equiv}, \cup, 0\rangle$ wiele faktów, które w tej chwili są dla nas najzupełniej oczywiste, należałoby za każdym razem dowodzić „ręcznie" w sposób podobny do dowodu przechodniości relacji $\equiv$ (a więc byłyby one pozostawione jako zadania dla Ciebie, Szanowny Czytelniku). Przykład takiego faktu: jeśli na planszy x strategię wygrywającą ma drugi gracz (w szczególności, jeśli np. $x = x' \cup x'$), to na planszy $x_1 \cup x$ strategię wygrywającą ma ten sam gracz, co na planszy x_1. Uzasadnienie: $[x] = 0$, więc $[x_1] \cup [x] = [x_1]$. Własności grup pozwalają nam wycisnąć wszystko co się da z przeprowadzonych już przez Ciebie dowodów, że $\equiv$ jest relacją równoważności i kongruencją ze względu na $\cup$.

Po trzecie i najważniejsze, jak się za chwilę okaże, klasy abstrakcji naszej relacji dają się efektywnie obliczać metodą programowania dynamicznego.

Trochę porządków

Na drodze do rozwiązania czeka nas jeszcze jedno niespodziewane odkrycie: klasy plansz (czyli klasy abstrakcji relacji $\equiv$) dają się uporządkować!

Zauważmy najpierw, że dla dowolnej planszy x, jeśli $[x] \neq 0$, to z planszy x można jednym ruchem dojść do pewnej planszy $x_0 \in 0 = [\emptyset]$. Innymi słowy, z każdej planszy x wygrywającej dla gracza, na którego przypada kolejka, można zawsze jednym ruchem przejść do jakiejś planszy x_0 przegrywającej dla gracza, którego kolejka wypadnie po tym ruchu (a więc który będzie musiał położyć pierwszy pasek na planszy x_0). Jest to ten właśnie ruch, który graczowi rozpoczynającemu grę na planszy x gwarantuje zwycięstwo. Oczywiście, na planszy x być może da się wykonać wiele innych ruchów, które mogą prowadzić do plansz z innych klas. Ważne jest jednak istnienie co najmniej jednego wygrywającego posunięcia dla każdej spośród plansz z klasy $[x] \neq 0$.

Wykorzystajmy konsekwentnie tę obserwację i określmy relację dwuargumentową $\succ$ na zbiorze ${}^{X}/_{\equiv}$ jak następuje: dla klas $B_1, B_2 \in {}^{X}/_{\equiv}$ zachodzi $B_1 \succ B_2$ wtedy i tylko wtedy, gdy z każdej planszy $x_1 \in B_1$ można jednym ruchem dojść do pewnej planszy $x_2 \in B_2$.

Na razie wiemy tylko, że dla $[x] = B \neq 0$ jest $B \succ 0$. Nie wiemy ani czy może zajść $0 \succ B$ dla jakiegoś B, ani czy może być $B_1 \succ B_2$ dla jakichkolwiek $B_1, B_2 \neq 0$.

Fakt 1. (najważniejszy) Jeśli $B_1, B_2 \in {}^{X}/_{\equiv}$ i $B_1 \neq B_2$, to albo $B_1 \succ B_2$, albo $B_2 \succ B_1$.

Dowód. Niech nie zachodzi ani $B_1 \succ B_2$, ani $B_2 \succ B_1$. Czyli istnieje plansza $x_1 \in B_1$, z której jednym ruchem nie da się dojść do żadnej spośród plansz z klasy B_2. Analogicznie, istnieje pewna plansza $x_2 \in B_2$, na której żadne posunięcie nie prowadzi do klasy B_1. Rozważmy planszę $x_1 \cup x_2$. Jeśli gracz, na którego przypada kolejność, wykona na tej połączonej planszy ruch w obrębie części x_1, to przejdzie do pewnej planszy $x_1' \cup x_2$, gdzie $x_1' \notin B_2$. A zatem $[x_1'] \neq [x_2]$. Do tej nierówności dodajmy stronami $[x_2]$, otrzymamy $[x_1'] \cup [x_2] \neq [x_2] \cup [x_2]$, czyli $[x_1' \cup x_2] \neq 0$. Jest to zatem ruch prowadzący do porażki. Gracz przegra również, jeśli położy pasek w obrębie planszy x_2. Czyli plansza $x_1 \cup x_2$ jest dla niego przegrywająca, to znaczy $[x_1 \cup x_2] = 0$, skąd $B_1 = B_2$, co kończy dowód.

Fakt 2. Dla żadnego $B \in {}^X/_\equiv$ nie zachodzi $B \succ B$.

Dowód. Jeśli by tak było, to startując od dowolnej planszy $x_1 \in B$ moglibyśmy wykonać ruch, prowadzący do pewnej planszy $x_2 \in B$, następnie jednym ruchem przeszlibyśmy do planszy $x_3 \in B$ i tak w nieskończoność. A nie jest to możliwe, bo po każdym ruchu liczba wolnych pól zmniejsza się co najmniej o długość najkrótszego z pasków c, z, n!

Fakt 3. Dla żadnych $B_1, B_2 \in {}^X/_\equiv$ nie zachodzi jednocześnie $B_1 \succ B_2$ i $B_2 \succ B_1$.

Dowód. Podobnie jak poprzednio, prowadziłoby to do istnienia nieskończonej rozgrywki, w której na zmianę przechodzilibyśmy z planszy z klasy B_1 do planszy z klasy B_2 i z powrotem.

Fakt 4. Jeśli $B_1 \succ B_2$ oraz $B_2 \succ B_3$, to $B_1 \succ B_3$.

Dowód. Ten fakt jest chyba dość zaskakujący, gdyż oznacza, że z planszy z klasy B_1 do planszy z klasy B_3 można zawsze przejść *jednym* ruchem, a nie dwoma (choć oczywiście dwoma też się da). Uzasadnienie jest jednak proste: nie może oczywiście być $B_1 = B_3$ (wykluczyliśmy to przed chwilą). Musi być zatem $B_3 \succ B_1$ lub $B_1 \succ B_3$. Jednak w tym pierwszym przypadku znów znaleźlibyśmy nieskończoną rozgrywkę, prowadzącą kolejno z klasy B_1 do B_2, z B_2 do B_3, a z B_3 na powrót do B_1 i tak w kółko. Pozostaje jedyna możliwość $B_1 \succ B_3$.

Fakt 5. Klas plansz w X jest skończenie wiele.

Dowód. (Pamiętamy, że sam zbiór X był nieskończony.) Niech x_m oznacza planszę, zbudowaną z jednego obszaru długości m (czyli $x_m = \{(m,1)\}$), gdzie $1 \leqslant m \leqslant p$. Wówczas oczywiście każda plansza $x \in X$ daje się przedstawić jako suma

$$x = \underbrace{x_1 \cup x_1 \cup \ldots \cup x_1}_{n_1} \cup \underbrace{x_2 \cup \ldots \cup x_2}_{n_2} \cup \ldots \cup \underbrace{x_p \cup \ldots \cup x_p}_{n_p}$$

Oczywiście dopuszczamy, by dla pewnych k było $n_k = 0$. Planszę x reprezentuje multizbiór $\{(1,n_1),\ldots,(p,n_p)\}$, trzeba tylko pominąć te pary (k,n_k), dla których $n_k = 0$.

Aby znaleźć klasę planszy x zauważmy, że $\underbrace{[x_m] \cup \ldots \cup [x_m]}_{n_m}$ jest równe $[x_m]$ dla n_m nieparzystego, zaś 0 dla n_m parzystego. Suma ta upraszcza się zatem do co najwyżej jednego składnika. W związku z tym, mamy

$$[x] = B_1 \cup B_2 \cup \ldots \cup B_p$$

gdzie odpowiednio $B_m = [x_m]$ lub $B_m = 0$, zależnie od parzystości liczby n_m. Sum tej postaci jest zaś tylko skończona liczba, dokładnie 2^p.

Zwróćmy uwagę, że klas plansz może nie być dokładnie 2^p, tylko mniej, gdyż może się np. zdarzyć, że dla pewnych $m' \neq m''$ jest $[x_{m'}] = [x_{m''}]$. Np. jeśli $c, z, n \geqslant 3$, to oczywiście $0 = [x_1] = [x_2]$.

Wniosek z faktów 1–5: Klasy plansz można ponumerować kolejnymi liczbami naturalnymi tak, by klasa 0 miała numer 0 i by dla dowolnych dwóch klas B', B'' numer klasy B' był większy od numeru klasy B'' wtedy i tylko wtedy, gdy $B' \succ B''$.

Na podstawie udowodnionych faktów wniosek ten powinien wydawać się oczywisty. Takie własności może po prostu mieć tylko naturalny porządek na liczbach naturalnych $0 < 1 < 2 \ldots < l-1 < l$. Jako zadanie proponuję uważnie uzasadnić ten wniosek np. przez indukcję lub posiłkując się pojęciami "dagu" (acyklicznego grafu skierowanego) i sortowania topologicznego. Klasę o numerze j będziemy od teraz oznaczać B^j. W szczególności, $0 = B^0$.

Programowanie dynamiczne

Najwyższa pora przejść do opisu algorytmu. Przypominam, że naszym celem jest obliczanie klas poszczególnych plansz.

Na podstawie wyciągniętego przed chwilą wniosku wiemy, że z każdej planszy z klasy B^j można jednym ruchem przejść do każdej z klas $B^0, B^1, \ldots, B^{j-1}$.

Wiemy też, że nie zachodzi $B_j \succ B_j$, czyli z pewnej planszy z klasy B_j nie da się wykonać ruchu tak, by pozostać w klasie B_j. Prawdą jest jednak mocniejsze stwierdzenie: nigdy nie da się wykonać ruchu tak, by pozostać w tej samej klasie. Dowód jest bardzo prosty: gdyby tak było dla pewnej planszy $x \in B$, to na planszy $x \cup x$ gracz pierwszy wykonałby to posunięcie na jednej z połówek, doprowadzając do sytuacji $x' \cup x$, gdzie $x, x' \in B$, czyli $[x] = [x'] = B$, skąd $[x' \cup x] = 0$. Z sytuacji dla siebie przegrywającej $x \cup x$ przeszedłby zatem do sytuacji, w której przegrać musi jego oponent, co oczywiście jest sprzeczne z naszymi definicjami i ze zdrowym rozsądkiem.

Widzimy zatem co następuje: numer planszy x to jedyna taka liczba j, że z planszy x da się jednym posunięciem przejść do każdej z klas $B^0, B^1, \ldots, B^{j-1}$, i nie da się żadnym posunięciem przejść do żadnej spośród plansz, należących do klasy B^j.

Ta obserwacja poprowadzi nas już prosto do algorytmu, który indukcyjnie wyznacza numery klas plansz. Dla porządku przytoczę jeszcze tylko jedną definicję:

Definicja. Jeśli Y z działaniem $+$ jest grupą, a $Y' \subseteq Y$ takim podzbiorem, który również jest grupą ze względu na to samo działanie, to mówimy, że Y' jest *podgrupą* Y.

Przykładowo, liczby całkowite podzielne przez 3 stanowią podgrupę grupy liczb całkowitych z dodawaniem.

Oznaczmy przez $X_m \subseteq X$ zbiór wszystkich plansz, zbudowanych z obszarów, z których żaden nie jest dłuższy niż m. Oczywiście $X = X_p$.

Niech $\mathcal{X}_m$ oznacza zbiór tych wszystkich klas plansz (elementów grupy ${}^X/_{\equiv}$), które mają reprezentanta z X_m. Jest on zamknięty ze względu na działanie $\cup$ i oczywiście jest podgrupą grupy ${}^X/_{\equiv}$. Mamy zatem $0 = \mathcal{X}_0 \subseteq \mathcal{X}_1 \subseteq \cdots \subseteq \mathcal{X}_p = {}^X/_{\equiv}$. Zauważmy też, że każdy ze zbiorów $\mathcal{X}_m$ składa się z klas o kolejnych numerach, od B^0 do pewnego B^N. Jest tak, gdyż jeśli N jest największą liczbą taką, że $B^N \in \mathcal{X}_m$, to pewna plansza $x \in B^N$ składa się z obszarów nie dłuższych niż m. Można z niej jednym ruchem dojść do każdej z klas $B^0, B^1, \ldots, B^{N-1}$, a ruch w grze w paski nie może spowodować, by którykolwiek z obszarów wydłużył się. To dowodzi, że każda z tych klas ma reprezentanta w X_m, czyli należy do $\mathcal{X}_m$.

Podam teraz algorytm, który po każdym kroku dla kolejnych liczb m potrafi

- wyznaczyć numer klasy każdej planszy ze zbioru X_m
- wykonywać działanie $\cup$ w grupie $\mathcal{X}_m$ (to znaczy, wyznaczać numer klasy $B^i \cup B^j$ dla $B^i, B^j \in \mathcal{X}_m$)

Pierwszy krok algorytmu jest trywialny: wiemy, że plansza pusta ma numer 0, a grupa $\mathcal{X}_0 = 0$ jest grupą trywialną.

Załóżmy więc, że potrafimy wyznaczać klasy dla plansz z X_{m-1} oraz że dla takich klas potrafimy wykonywać działanie $\cup$. Weźmy planszę x_m zbudowaną z jednego obszaru długości m. Przejrzyjmy wszystkie możliwe ruchy, jakie możemy wykonać na tej planszy. Każdy z tych ruchów prowadzi do planszy zbudowanej z 1 lub 2 obszarów, z których każdy ma długość $\leqslant m-1$. Zatem zgodnie z założeniem potrafimy wyznaczyć numery klas wszystkich tych plansz (jeśli jakiś ruch rozspaja naszą planszę, to numer klasy wyznaczymy, obliczając $\cup$ klas obu obszarów, na które rozpadła się nasza plansza). Mamy wówczas pewność, że klasa, do której należy x_m, jest najmniejszą liczbą naturalną, jaka nie pojawiła się wśród otrzymanych numerów klas plansz. Jeśli np. możliwe ruchy prowadzą z x_m do plansz z klas o numerach $0, 1, 2, 3, 6, 9, 13$, to znaczy, że $[x_m] = B^4$.

Są tu dwie możliwości. Albo $[x_m] \in \mathcal{X}_{m-1}$, skąd wynika, że $\mathcal{X}_m = \mathcal{X}_{m-1}$ i nic więcej w tym kroku algorytmu nie musimy robić, albo z planszy x_m można dojść do każdej z klas, należących do $\mathcal{X}_{m-1}$, czyli klasa $[x_m]$ nie jest nam jeszcze znana. Ten drugi przypadek musimy rozważyć dokładniej.

Wiemy, że $\mathcal{X}_{m-1} = \{B^0, \ldots, B^{M-1}\}$ dla pewnego M. Stąd natychmiastowy wniosek, że $[x_m] = B^M \in \mathcal{X}_m \setminus \mathcal{X}_{m-1}$.

Rozważmy teraz klasy postaci $[x_m] \cup B^j$, gdzie $0 \leqslant j \leqslant M-1$.

- Wszystkie one są elementami $\mathcal{X}_m$.
- Żadna z nich nie należy do $\mathcal{X}_{m-1}$, gdyż jeśli $[x_m] \cup B^j = B^k \in \mathcal{X}_{m-1}$, to oczywiście mielibyśmy również $[x_m] = [x_m] \cup B^j \cup B^j = B^k \cup B^j \in \mathcal{X}^{m-1}$ (suma elementów B^j i B^k grupy $\mathcal{X}^{m-1}$ również należy do tej grupy).
- Klasa każdej planszy $x \in X_m$, jeśli nie należy do grupy $\mathcal{X}_{m-1}$, to może być przedstawiona w tej postaci. Plansza $x \in X_m$ jest bowiem połączeniem pewnej (parzystej bądź nieparzystej) liczby obszarów długości m oraz planszy, należącej do X_{m-1}, czyli $x = x_m \cup x_m \cup \ldots \cup x_m \cup x'$, gdzie $x' \in X_{m-1}$. Stąd istotnie $[x] = [x_m] \cup [x']$ lub $[x] = [x'] \in \mathcal{X}_{m-1}$, gdzie $[x'] = B^j$ dla pewnego $j < M$.
- Dla różnych wartości j klasy $[x_m] \cup B^j$ są między sobą różne, bo do nierówności $B^j \neq B^{j'}$ można stronami dodać $[x_m]$.

W takim razie mamy

$$\mathcal{X}_m = \{B^0, B^1, \ldots, B^{M-1},\ [x_m] \cup B^0, [x_m] \cup B^1, \ldots, [x_m] \cup B^{M-1}\}$$

Co więcej, skoro umieliśmy wykonywać działanie $\cup$ w grupie $\mathcal{X}_{m-1}$, to pamiętając, że $[x_m] \cup [x_m] = 0$, umiemy je wykonać także w $\mathcal{X}_m$!

Teraz już nietrudno zgadnąć, że numerem klasy $[x_m] \cup B^j$ będzie $M + j$. Aby się o tym przekonać, wybierzmy dowolnego reprezentanta $x \in B^j$ i rozważmy planszę $x_m \cup x$. Na tej planszy możemy wykonać ruch albo w obrębie części x_m, albo x.

Wykonując ruch w części x możemy przejść do dowolnie wybranej spośród klas $[x_m] \cup B^i$, gdzie $0 \leqslant i < j$. Być może uda nam się również przejść do którejś z klas $[x_m] \cup B^k$ dla $k > j$, ale dla każdego takiego k znajdziemy reprezentanta $x \in B^j$, dla którego nie będzie to możliwe. Wszystko odbywa się analogicznie jak na samej planszy $x \in B^j$, bez dołączonej części x_m.

Natomiast kładąc pasek w części x_m, możemy przejść do dowolnej klasy $B' \in \mathcal{X}_{m-1}$. Wystarczy wziąć $B'' = B' \cup B^j$. Wiemy, że $B'' \in \mathcal{X}_{m-1}$, więc istnieje taki ruch na planszy x_m, który prowadzi do klasy B''. Wykonując ten ruch na planszy $x_m \cup x$, istotnie otrzymamy w wyniku planszę klasy $B'' \cup B^j = (B' \cup B^j) \cup B^j = B'$. Widzimy też, że ruch w obrębie x_m nie doprowadzi nas do żadnej klasy spoza $\mathcal{X}_{m-1}$.

Podsumowując, widzimy wyraźnie, że:

$$B^{M-1} \prec [x_m] = [x_m] \cup B^0 \prec [x_m] \cup B^1 \prec [x_m] \cup B^2 \prec \cdots \prec [x_m] \cup B^{M-1}.$$

Ponieważ wiemy skądinąd, że $\mathcal{X}_m = \{B^0, \ldots, B^{2M-1}\}$, to musi rzeczywiście zachodzić $[x_m] \cup B^j = B^{M+j}$, dla każdego $0 \leqslant j \leqslant M-1$.

W ten sposób z nawiązką wypełniamy wszystkie warunki indukcyjne, dotyczące naszego algorytmu.

Pozostaje jeszcze jeden drobiazg: widzimy, że po pierwszym kroku znamy tylko jedną planszę (pustą), a po każdym następnym liczba znanych nam plansz (tzn. liczba elementów grupy $\mathcal{X}_m$) albo nie zmienia się, albo się podwaja. Będzie zatem zawsze potęgą dwójki. W szczególności, w całym powyższym rozumowaniu M było potęgą dwójki. Po tej uwadze nie powinno być dla Czytelnika najmniejszym problemem udowodnienie przez indukcję, że działaniu $\cup$ w grupie klas plansz odpowiada dokładnie operacja XOR na numerach tych klas.

W ten sposób wszystkie tajemnice gry w paski zostały ujawnione.

TESTY

Na potrzeby testowania rozwiązań przygotowano 15 testów. Każdy z nich zawierał p przypadków plansz do rozpatrzenia — było to zawsze kolejno p plansz o długościach od 1 do p.

W poniższej tabeli zestawiono wartości c, z, n oraz p dla poszczególnych testów. Ponadto, k oznacza największą liczbę taką, że co najmniej jedna z plansz x_m przy $m \leqslant p$ należy do klasy B^k. Dalej, W_1 oznacza liczbę tych plansz spośród $x_1, \ldots, x_p$, które są wygrywające dla gracza rozpoczynającego (czyli należą do klasy B^j dla $j > 0$), zaś W_2 — liczbę plansz przegrywających dla rozpoczynającego.

Nr	**c**	**z**	**n**	**p**	**k**	$\mathbf{W_1}$	$\mathbf{W_2}$
0	1	5	1	6	1	3	3
1	2	2	2	10	3	7	3
2	3	3	3	20	4	16	4
3	4	4	4	50	5	43	7
4	2	18	26	70	15	63	7
5	1	5	6	905	256	903	2
6	2	3	4	100	16	99	1
7	3	5	5	100	12	94	6
8	2	2	111	200	17	181	19
9	1000	1000	1000	1000	1	1	999
10	2	38	226	1000	150	984	16
11	3	221	233	1000	103	977	23
12	2	16	60	1000	65	985	15
13	3	29	91	1000	139	987	13
14	3	51	51	1000	64	980	20

Testy 4 i 5 oraz 6, 7, 8 i 9 zostały zgrupowane.

PODSUMOWANIE

Zadanie „Paski” było prawdopodobnie jednym z najtrudniejszych zadań w historii Olimpiady. Mimo że było zadaniem z pierwszego etapu, a więc można było poświęcić na nie wiele czasu i posiłkować się przy pracy literaturą, to jednak poprawnie rozwiązał je tylko jeden zawodnik.

Przedstawione w niniejszym artykule rozumowanie jest oparte na dowodzie poprawności programu wzorcowego. Autorem tego programu i dowodu jest Tomasz Waleń.

Podstawowe pojęcia teorii gier są przystępnie objaśnione w pierwszym rozdziale znakomitej książki [HS]. Serdecznie zachęcam do lektury!

Czytelnikom zainteresowanym tematem polecam również pozycję [JC], a w szczególności zawarte tam informacje o grze Nim. Gra ta okazuje się mieć wiele wspólnego z grą w paski. W niej także do znalezienia strategii wygrywającej przydaje się operacja *XOR*.

BIBLIOGRAFIA

[HS] Hugo Steinhaus *„Kalejdoskop matematyczny”*, WSiP Warszawa 1989
[JC] John Conway *„On Numbers and Games”*, Academic Press, Inc. New York 1976

Zawody II stopnia

opracowania zadań

Marcin Kubica
autor zadania, opracowanie

Marcin Mucha
program wzorcowy

Podpisy

W Urzędzie Ochrony Bajtocji (UOB) zatrudnieni są urzędnicy oraz dowódcy. W archiwum znajdują się teczki z aktami wszystkich urzędników. W każdej teczce znajduje się podpis urzędnika oraz podpisy pracowników (urzędników lub dowódców), którzy poręczają za jego lojalność. Każdy nowoprzyjmowany urzędnik musi uzyskać przynajmniej jedno poręczenie. Z biegiem czasu lista poręczycieli może się powiększać. UOB dowiedział się ostatnio, że do grona dowódców przeniknął szpieg wrogiej Mikromięklandii. Kolejni szpiedzy byli wprowadzani do UOB na stanowiska urzędnicze dzięki poręczeniu szpiega-dowódcy i/lub innych wprowadzonych szpiegów. Tacy szpiedzy mają poręczenia wyłącznie od pracowników będących szpiegami.

Wiarygodność urzędnika można podważyć, jeżeli pośrednio nie ma on poręczenia żadnego dowódcy, który nie jest szpiegiem, tzn. nie istnieje taki ciąg pracowników UOB $p_1, p_2, \ldots, p_k$, *że* p_1 *jest dowódcą nie będącym szpiegiem,* p_k *jest danym urzędnikiem i (dla* $i = 1, \ldots, k-1$*)* p_i *poręczył za* p_{i+1}.

Jeżeli założenie o pewnym dowódcy, że jest szpiegiem spowodowałoby, że wiarygodność urzędnika zostałaby podważona, to urzędnik ten jest podejrzany o szpiegostwo. Dowództwo UOB chciałoby zobaczyć listę takich urzędników, i to jak najszybciej!

Przykład

Dowódcy: *Anna, Grzegorz.*
Urzędnicy: *Bolesław (poręczyła Anna), Celina (poręczył Bolesław), Dorota (poręczyli Bolesław i Celina), Eugeniusz (poręczyli Anna i Grzegorz), Felicja (poręczył Eugeniusz), Halina (poręczyli Grzegorz i Ireneusz), Ireneusz (poręczyli Grzegorz i Halina).*
Podejrzani: *Bolesław, Celina, Dorota, Halina, Ireneusz.*

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `POD.IN` *liczbę dowódców i urzędników w UOB oraz informacje o poręczeniach,*
- *wyznaczy listę urzędników podejrzanych o szpiegostwo,*
- *wyniki zapisze do pliku tekstowego* `POD.OUT`.

Wejście

W pierwszym wierszu pliku tekstowego `POD.IN` *zapisana jest dokładnie jedna dodatnia liczba całkowita* n *(*$1 \leqslant n \leqslant 500$*) będąca liczbą pracowników UOB. Pracownicy są ponumerowani od 1 do* n*. W kolejnych* n *wierszach zapisane są opisy poręczeń. W* $i+1$*-ym wierszu pliku znajduje się opis poręczeń udzielonych pracownikowi nr* i*. Jest to ciąg liczb całkowitych* *poodzielanych pojedynczymi odstępami. Pierwsza liczba w tym ciągu, 0 $\leqslant m_i$, jest równa liczbie poręczeń udzielonych pracownikowi nr i. Kolejne m_i liczb to numery pracowników, którzy poręczyli za prawdomówność pracownika nr i. (Tak więc liczba wyrazów ciągu w i + 1-ym wierszu wynosi m_i + 1.) Dowódcy to Ci pracownicy, za których nikt nie poręczył.*

Wyjście

Twój program powinien:

- *w kolejnych wierszach pliku tekstowego* `POD.OUT` *zapisać w rosnącej kolejności, po jednej liczbie w każdym wierszu, ciąg dodatnich liczb całkowitych będących numerami urzędników podejrzanych o szpiegostwo — jeżeli tacy urzędnicy są,*
- *w pierwszym i jedynym wierszu pliku wyjściowego* `POD.OUT` *zapisać dokładnie jedno słowo* `BRAK` *— jeżeli takich urzędników nie ma.*

Przykład

Dla pliku wejściowego `POD.IN`*:*

```
9
0
1 1
1 2
2 2 3
2 1 7
1 5
0
2 7 9
2 7 8
```

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

```
2
3
4
8
9
```

ROZWIĄZANIE

Zadanie to można w naturalny sposób wyrazić w języku teorii grafów. Mamy dany graf skierowany. Wierzchołki tego grafu reprezentują pracowników UOB, a krawędzie reprezentują poręczenia. Krawędź $v \to u$ reprezentuje poręczenie udzielone przez pracownika v urzędnikowi u. Dowódcy są reprezentowani przez te wierzchołki, do których nie prowadzą żadne krawędzie.

Niech u będzie wierzchołkiem reprezentującym urzędnika. Jak sprawdzić czy u jest podejrzany o szpiegostwo? Zależy to od tego, ilu dowódców pośrednio poręczyło za jego lojalność. Temu, że pewien dowódca d pośrednio poręczył za lojalność urzędnika u odpowiada ścieżka prowadząca z d do u. Tak więc, jeśli istnieją przynajmniej dwa wierzchołki reprezentujące dowódców, z których możemy poprowadzić ścieżki do u, to u nie jest podejrzany o szpiegostwo — gdyby jeden z tych dowódców okazał się szpiegiem, to i tak drugi poręczył pośrednio za lojalność u. Jeżeli natomiast istnieje co najwyżej jeden wierzchołek d reprezentujący dowódcę, z którego możemy poprowadzić ścieżkę kończącą się w u, to u jest podejrzany o szpiegostwo — gdyby d okazał się szpiegiem, to wiarygodność u zostałaby podważona.

Zadanie to można rozwiązać wykorzystując przeszukiwanie grafu wszerz lub w głąb (zobacz [10] lub [13]). W najprostszej wersji możemy dla każdego urzędnika wyznaczyć liczbę dowódców, którzy pośrednio poręczyli za niego, przeszukując graf "wstecz". Podobnie, możemy dla każdego dowódcy wyznaczyć urzędników, za których on poręczył, przeszukując graf zgodnie z kierunkiem krawędzi. Rozwiązania te są jednak nieefektywne. Ich złożoność czasowa jest $O(nm)$, gdzie n jest liczbą wierzchołków, a m jest liczbą krawędzi w grafie.

Zauważmy jednak, że nie musimy wyznaczać wszystkich dowódców, którzy pośrednio poręczyli za danego urzędnika. Jeśli znajdziemy dwóch takich dowódców, to nie musimy wyznaczać kolejnych — dany urzędnik i tak nie jest podejrzany o szpiegostwo. Również wszyscy ci, za których ten urzędnik poręczył też nie są podejrzani o szpiegostwo. Ponadto nie jest istotne, którzy dowódcy poręczyli pośrednio za danego urzędnika, a jedynie ilu.

Możemy więc każdemu urzędnikowi przyporządkować licznik określający ilu dowódców pośrednio poręczyło za niego: "żaden", "jeden", "dwóch lub więcej". Jeśli licznik jest równy jeden, to dodatkowo pamiętamy który dowódca poręczył pośrednio za urzędnika — informację tę będziemy wykorzystywać zamiast kolorowania wierzchołków w trakcie przeszukiwania grafu. Początkowo przyjmujemy, że żaden dowódca nie poręczył pośrednio za żadnego urzędnika. Następnie, dla każdego dowódcy przeszukujemy graf począwszy od niego i zwiększamy liczniki odwiedzonych urzędników. Jeśli licznik któregoś urzędnika jest równy jeden i poręczył za niego pośrednio ten dowódca, od którego zaczęliśmy przeszukiwanie grafu, to urzędnik ten był już w tym przeszukiwaniu odwiedzony. Jeśli licznik któregoś urzędnika osiągnie wartość "dwóch lub więcej', to pomijamy go przy dalszych przeszukiwaniach grafu. W ten sposób każdy wierzchołek zostanie odwiedzony co najwyżej dwa razy. W rezultacie złożoność czasowa takiego rozwiązania jest $O(n+m)$, gdzie n jest liczbą wierzchołków, a m jest liczbą krawędzi w grafie. Złożoność pamięciowa jest takiego rzędu jak wielkość grafu i wynosi $O(n+m)$. Rozwiązanie to może być zaimplementowane w następujący sposób:

```
const
  N_MAX = 500; { Maksymalna liczba wierzchołków. }
  ST_UNKNOWN = 0;            { Brak poręczeń. }
  { Jedno poręczenie = nr poręczającego dowódcy. }
  ST_CLEAN   = N_MAX + 1; { Co najmniej dwa poręczenia. }
type
  t_array = array [1..N_MAX] of word;
  p_array = ↑t_array;
  t_official = record
    status: word; { Licznik poręczeń. }
    cpt: boolean; { Czy jest dowódcą? }
    vouchees_no: word; { Liczba poręczanych urzędników. }
    vouchees: p_array; { Poręczani urzędnicy. }
```

```
  end
var
  officials: array [1..N_MAX] of t_official; { Graf. }
  n: word; { Liczba wierzchołków. }
procedure find_suspects;
{ Wyznacz wartości liczników poręczeń dla urzędników. }
var
  i: word;
  procedure go_down (no: word);
  { Przeszukiwanie grafu w głąb, począwszy od wierzchołka nr no. }
  var
    i: word;
    captain: word;
  begin
    captain := officials[no].status;
    for i := 1 to officials[no].vouchees_no do
      with officials[officials[no].vouchees↑[i]] do begin
        if (status = ST_UNKNOWN) then begin
          { Pierwszy raz wchodzimy do wierzchołka. }
          { Przepisujemy status poprzedniego wierzchołka. }
          status := captain;
          go_down (officials[no].vouchees↑[i])
        end else if ((status < ST_CLEAN) and
                    (status ≠ captain)) then begin
          { Znaleźliśmy drugiego dowódcę. }
          status := ST_CLEAN;
          go_down (officials[no].vouchees↑[i])
        end
      end
  end { go_down }
begin
   for i := 1 to n do officials[i].status := ST_UNKNOWN;
   for i := 1 to n do if officials[i].cpt then begin
     { Dowódca }
     officials[i].status := i;
     go_down (i);
     officials[i].status := ST_CLEAN
   end
end
```

TESTY

Do sprawdzania rozwiązań zawodników użyto 14 testów.

Większość testów ma postać: KULE(n, k, p1, p2), gdzie:

(1) cały graf n-wierzchołkowy jest podzielony na k równolicznych podgrafów

(2) każdy z podgrafów jest losowy, przy czym prawdopodobieństwo, że między danymi dwoma wierzchołkami występuje krawędź wynosi $p1$

(3) każdy podgraf zawiera dokładnie jednego dowódcę

(4) każdy podgraf zawiera jeden wyróżniony wierzchołek, który nie jest dowódcą

(5) na wierzchołkach wyróżnionych znów mamy strukturę grafu, tym razem z prawdopodobieństwem wystąpienia krawędzi równym $p2$

(6) jeśli $k = 1$, to dodatkowo doczepiamy do grafu gadżety, które powodują, że nie wszyscy urzędnicy są podejrzani (nie chcemy trywializować testu)

- POD0.IN — test z treści zadania
- POD1.IN — mały test poprawnościowy,
- POD2.IN — KULE(100, 1, 0.5),
- POD3.IN — KULE(500, 1, 0.5),
- POD4.IN — KULE(500, 1, 0.8),
- POD5.IN — KULE(500, 1, 1.0) - duży graf pełny z gadżetami,
- POD6.IN — KULE(50, 5, 0.5, 0.2),
- POD7.IN — KULE(200, 10, 0.8, 0.1),
- POD8.IN — KULE(500, 5, 0.9, 0.2),
- POD9.IN — KULE(500, 10, 1.0, 0.1),
- POD10.IN — KULE(500, 5, 1.0, 0.3),
- POD11.IN — KULE(500, 5, 0.8, 0.3),
- POD12.IN — KULE(500, 2, 0.9, 0.8),
- POD13.IN — KULE(480, 3, 0.8, 0.5).

Grzegorz Jakacki
autor zadania

Krzysztof Sobusiak
opracowanie, program wzorcowy

Automorfizmy

Turniejem *nazywamy graf skierowany, w którym:*

- *dla dowolnych dwóch różnych wierzchołków u i v istnieje dokładnie jedna krawędź pomiędzy tymi wierzchołkami (tzn. albo $u \to v$, albo $v \to u$),*
- *nie istnieją pętle (tzn. dla dowolnego wierzchołka u nie ma krawędzi $u \to u$).*

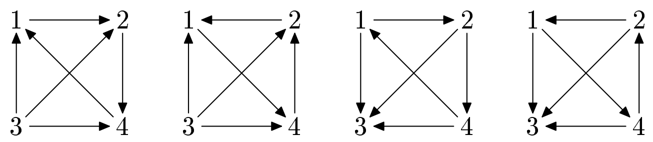
Oznaczmy przez p dowolną permutację zbioru wierzchołków turnieju. (Permutacją skończonego zbioru X nazywamy każdą różnowartościową funkcję z X w X.) Permutację p nazywamy **automorfizmem***, jeżeli dla dowolnych dwóch różnych wierzchołków u i v zwrot krawędzi pomiędzy u i v jest taki sam, jak pomiędzy $p(u)$ i $p(v)$ (tzn. $u \to v$ jest krawędzią w turnieju wtedy i tylko wtedy, gdy $p(u) \to p(v)$ jest krawędzią w tym turnieju). Dla zadanej permutacji p interesuje nas, dla ilu turniejów jest ona automorfizmem.*

Przykład

Weźmy zbiór wierzchołków oznaczonych liczbami $1, \ldots, 4$ oraz permutację p:

$$p(1) = 2 \qquad p(2) = 4 \qquad p(3) = 3 \qquad p(4) = 1.$$

Istnieją tylko cztery turnieje, dla których ta permutacja jest automorfizmem:



Zadanie

Napisz program, który:

- *z pliku tekstowego* `AUT.IN` *wczyta opis permutacji n-elementowego zbioru wierzchołków,*
- *obliczy liczbę t różnych n-wierzchołkowych turniejów, dla których ta permutacja jest automorfizmem,*
- *zapisze w pliku tekstowym* `AUT.OUT` **resztę z dzielenia** t **przez** *1 000.*

WEJŚCIE

W pierwszym wierszu pliku tekstowego `AUT.IN` *znajduje się jedna liczba naturalna n, $1 \leqslant n \leqslant 10\ 000$, będąca liczbą wierzchołków.*

W kolejnych n wierszach znajduje się opis permutacji p. Zakładamy, że wierzchołki są ponumerowane liczbami od 1 do n. W wierszu $(k+1)$-szym znajduje się wartość permutacji p dla wierzchołka nr k (tzn. wartość $p(k)$).

WYJŚCIE

W pierwszym i jedynym wierszu pliku tekstowego `AUT.OUT` *powinna znaleźć się jedna liczba całkowita będąca resztą z dzielenia przez 1 000 liczby różnych n-wierzchołkowych turniejów, dla których permutacja p jest automorfizmem.*

PRZYKŁAD

Dla pliku wejściowego `AUT.IN`*:*

```
4
2
4
3
1
```

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

```
4
```

ROZWIĄZANIE

Rozważmy zbiór A wszystkich nieuporządkowanych par liczb ze zbioru $\{1, 2, \ldots, n\}$ — każdą parę liczb będziemy utożsamiać z parą wierzchołków o odpowiednich numerach. Pokażemy, że:

- albo nie istnieje żaden turniej, dla którego dana permutacja p jest automorfizmem,
- albo wyznacza ona podział zbioru A na rozłączne podzbiory $A_1, A_2, \ldots, A_s$, zwane dalej *gromadami*, dające w sumie zbiór A i takie, że:
  - zwrot krawędzi między dowolną parą wierzchołków z danej gromady determinuje zwrot krawędzi między pozostałymi parami wierzchołków z tej gromady,
  - zwroty krawędzi między parami wierzchołków należącymi do różnych gromad można wybrać niezależnie.

Szukana liczba t turniejów, dla których permutacja p jest automorfizmem będzie równa 2^s, ponieważ zwrot krawędzi pomiędzy parami wierzchołków z jednej gromady można ustalić na 2 sposoby. Pary należące do tej samej gromady nazwiemy parami *zależnymi*.

Cyklem permutacji p będziemy nazywać k-elementowy ciąg różnych liczb $(c_0, c_1, \ldots, c_{k-1})$ taki, że $c_i = p(c_{i-1})$, dla $1 \leqslant i < k$ i $c_0 = p(c_{k-1})$. Łatwo pokazać, że każdą permutację można przedstawić w postaci zbioru rozłącznych cykli.

Dla przykładu permutacja p:

$$p(1) = 2 \qquad p(2) = 5 \qquad p(3) = 6 \qquad p(4) = 1 \qquad p(5) = 4 \qquad p(6) = 3$$

składa się z dwóch cykli:

$$1 \longrightarrow 2 \longrightarrow 5 \longrightarrow 4 \qquad\qquad 3 \longrightarrow 6$$

Liczbę s gromad wyznaczonych przez daną permutację p można szybko znaleźć, znając rozbicie permutacji na cykle.

Weźmy cykl C o długości k oraz należące do niego elementy u i v. Połóżmy biały żeton na u i czerwony żeton na v. Jeśli oba żetony jednocześnie przesuniemy po cyklu o jeden element, to dostaniemy parę zależną od $\{u, v\}$. Aby wyznaczyć wszystkie takie pary powtarzamy ten ruch tak długo, aż oba żetony z powrotem znajdą się na pozycjach startowych u i v.

Zauważmy, że jeśli długość cyklu k jest liczbą parzystą i v jest odległe o $\frac{k}{2}$ od u na cyklu C, to po $k/2$ krokach czerwony żeton znajdzie się na u, a biały — na v. To oznacza sprzeczność — $u \to v$ jest krawędzią w turnieju wtedy i tylko wtedy, gdy $v \to u$ jest krawędzią w tym turnieju. W takim przypadku nie istnieje żaden turniej, dla którego permutacja p byłaby automorfizmem — ostateczną odpowiedzią jest liczba 0. Dalej będziemy zakładać, że wszystkie cykle są nieparzystej długości.

Jeśli długość cyklu jest liczbą nieparzystą, to niezależnie od tego jak wybraliśmy u i v, do początkowego ustawienia żetonów powrócimy po k krokach. Liczba k jest zatem licznością dowolnej gromady złożonej z par elementów cyklu C. Zbiór różnych par elementów cyklu C ma $k(k-1)/2$ elementów, zatem rozpada się na $(k-1)/2$ gromad.

Niech teraz u będzie elementem cyklu C o długości k, a v — elementem cyklu D długości l. Połóżmy znowu biały żeton na u, a czerwony na v. Aby wyznaczyć pary zależne od $\{u, v\}$, podobnie jak poprzednio synchronicznie przesuwamy żetony, tym razem po rozłącznych cyklach. Po ilu krokach wrócimy do sytuacji początkowej? Żeton biały wraca na u po k krokach, żeton czerwony wraca na v po l krokach, zatem musimy wykonać $NWW(k, l)$ kroków. Wszystkich par o jednym elemencie z C i drugim z D jest kl, a wszystkie gromady utworzone z takich par są liczności $NWW(k, l)$, więc mamy $kl/NWW(k, l) = NWD(k, l)$ takich gromad.

Możemy już policzyć liczbę gromad, na które rozpada się zbiór wszystkich par. Musimy najpierw wyznaczyć długości wszystkich cykli permutacji p — tę operację można łatwo zaimplementować, by działała w czasie $O(n)$ — a następnie posumować liczby gromad po wszystkich parach cykli, stosując wyżej wyprowadzone wzory. Ponieważ permutacja może mieć rzędu n cykli, nasz algorytm miałby złożoność czasową nie lepszą niż $O(n^2)$. Możemy jednak poprawić ten wynik poprzez zbiorowe traktowanie cykli o identycznych długościach. Niech c_k oznacza liczbę cykli o długości k. Wtedy sumaryczna liczba gromad utworzonych przez pary o elementach pochodzących

- z tego samego cyklu o długości k wynosi $\frac{c_k \cdot (k-1)}{2}$,
- z różnych cykli o długości k wynosi $\frac{c_k(c_k-1)}{2} \cdot k$,
- z cykli o różnych długościach k i l wynosi $c_k c_l \cdot NWD(k,l)$.

Ile może być różnych długości cykli? Aby było ich najwięcej, musimy wziąć po jednym cyklu o długości $1, 3, 5, \ldots$, przy czym suma długości cykli nie może być większa od n. Ponieważ $1 + 3 + 5 + \ldots + \lceil 2\sqrt{n} \rceil \geqslant n$, różnych długości cykli jest co najwyżej $O(\sqrt{n})$. Do rozpatrzenia mamy zatem $O(n)$ par grup cykli różnych długości. Policzenie $NWD(p,q)$ algorytmem Euklidesa zajmuje czas $O(\log p + \log q)$ czasu (jego opis Czytelnik znajdzie w [13]), zatem otrzymujemy algorytm o złożoności czasowej $O(n \log n)$.

To nie koniec zadania, gdyż pozostaje nam jeszcze policzyć resztę z dzielenia $t = 2^s$ przez 1000. Kolejne potęgi dwójki będziemy naliczać iteracyjnie. Łatwo zauważyć, że przy wykonywaniu kolejnych mnożeń przez 2 wystarczy przechowywać jedynie resztę z dzielenia aktualnego wyniku przez 1000. Obserwacja ta jednak nie wystarcza, ponieważ s może być rzędu n^2. Zauważmy jednak, że po wystarczająco dużej liczbie kroków (ale na pewno mniejszej niż 1000) trafimy w końcu na wartość, którą otrzymaliśmy już wcześniej. Niech i będzie właśnie takim najmniejszym wykładnikiem, że $2^i \bmod 1000 = 2^{\mathrm{m}} \bmod 1000$ dla pewnego $m < i$. Wtedy dla każdego $j > i$ mamy

$$2^j \bmod 1000 = (2^i \bmod 1000) \cdot 2^{(j-i) \bmod (i-m)} \bmod 1000$$

Wystarczy zatem namnożyć dwójkę jeszcze $(j - i) \bmod (\mathrm{i} - \mathrm{m})$ razy. Aby zrealizować ten algorytm potrzebna nam będzie 1000-elementowa tablica, w której r-tym elementem będzie wykładnik i, dla którego otrzymaliśmy resztę r.

Algorytm ten wykona co najwyżej 2000 mnożeń, więc jego złożoność czasowa to $O(1)$ (jest stała)!

TESTY

Do sprawdzania rozwiązań zawodników użyto 12 testów. Testy 2 i 3 oraz 11 i 12 były zgrupowane. Permutacje z wszystkich testów, za wyjątkiem testu 2, zawierały wyłącznie cykle nieparzystych długości. Krótki opis testów zawarty jest poniżej:

- AUT1.IN — prosty test poprawnościowy,
- AUT2.IN — prosty test poprawnościowy z cyklem parzystej długości,
- AUT3.IN — prosty test do zgrupowania z 2,
- AUT4.IN — poprawnościowy test losowy, permutacja 50-elementowa o 4 cyklach,
- AUT5.IN — test poprawnościowy, permutacja 501-elementowa o jednym cyklu,
- AUT6.IN — wydajnościowy test losowy, permutacja 10000-elementowa o 100 cyklach,

- AUT7.IN — wydajnościowy test losowy, permutacja o 100 cyklach długości 1, 3, 5, ..., 199,
- AUT8.IN — wydajnościowy test losowy, permutacja 5000-elementowa o 4500 cyklach,
- AUT9.IN — wydajnościowy test losowy, permutacja 10000-elementowa o 5000 cyklach,
- AUT10.IN — wydajnościowy test losowy, permutacja 10000-elementowa o 8000 cyklach,
- AUT11.IN — 10000-elementowa permutacja identycznościowa,
- AUT12.IN — prosty test do zgrupowania z 11.

Wojciech Rytter
autor zadania, opracowanie

Tomasz Waleń
program wzorcowy

Trójramienny dźwig

Trójramienny dźwig ustawia kontenery na wagonach kolejowych. Wagony są ponumerowane kolejno $1, 2, \ldots$. Na każdym wagonie można postawić co najwyżej jeden kontener. W jednym ruchu dźwig pobiera ze składowiska trzy kontenery i ustawia je na wagonach o numerach i, $i+p$ oraz $i+p+q$, albo na wagonach o numerach i, $i+q$ oraz $i+p+q$ (dla pewnych stałych $p, q \geqslant 1$). Dźwig trzeba zaprogramować tak, żeby załadował kontenerami pierwsze n wagonów pociągu (pociąg ma $n+p+q$ wagonów). Program składa się z ciągu instrukcji. Każda z instrukcji opisuje jeden ruch dźwigu. Instrukcja programu ma postać trójki (x, y, z), gdzie $1 \leqslant x < y < z \leqslant n+p+q$, i określa numery wagonów, na które dźwig ma ustawić kontenery. Jeżeli po wykonaniu programu na każdym spośród n pierwszych wagonów pociągu znajduje się dokładnie jeden kontener, to mówimy, że program jest **poprawny**.

Zadanie

Napisz program, który:

- *z pliku tekstowego* `TRO.IN` *wczyta charakterystykę dźwigu (liczby p i q) oraz liczbę wagonów do załadowania (n),*
- *wygeneruje poprawny program dla dźwigu,*
- *zapisze go do pliku tekstowego* `TRO.OUT`.

Wejście

W pierwszym i jedynym wierszu pliku tekstowego `TRO.IN` *znajdują się dokładnie trzy dodatnie liczby całkowite pooddzielane pojedynczymi odstępami. Są to odpowiednio: liczby p i q określające parametry dźwigu oraz liczba n, będąca liczbą początkowych wagonów pociągu do załadowania. Liczby te spełniają nierówności $1 \leqslant n \leqslant 300\,000$, $2 \leqslant p+q \leqslant 60\,000$.*

Wyjście

W pierwszym wierszu pliku tekstowego `TRO.OUT` *powinna znajdować się dokładnie jedna liczba całkowita m będąca liczbą instrukcji w wygenerowanym programie. W każdym z kolejnych m wierszy powinny znajdować się dokładnie trzy liczby naturalne x, y, z pooddzielane pojedynczymi odstępami, $1 \leqslant x < y < z \leqslant n+p+q$, $x \leqslant n$, $y \in \{x+p, x+q\}$, $z = x+p+q$. Są to numery wagonów, na które dźwig ma położyć kontenery w kolejnym ruchu.*

Przykład

Dla pliku wejściowego `TRO.IN`*:*

```
2 3 10
```

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

```
4
1 3 6
2 4 7
5 8 10
9 11 14
```

ROZWIĄZANIE

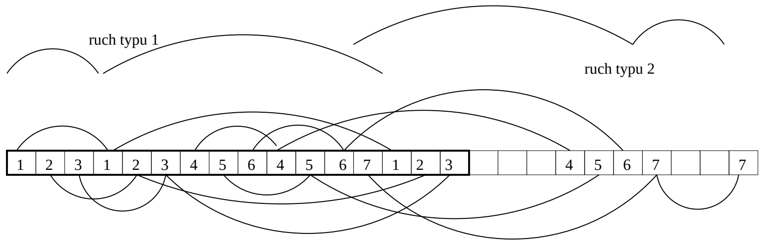
Przyjmijmy, że $p \leqslant q$. Ruch typu $i, i+p, i+p+q$ nazwijmy ruchem typu 1, a $i, i+q, i+p+q$ ruchem typu 2, gdzie i jest najmniejszym numerem wolnego wagonu.

Opis algorytmu

Rozwiązanie polega na wykonywaniu tak długo jak to możliwe, ruchów typu 1, a gdy to nie jest możliwe ruchu typu 2. Okazuje się, że takie rozwiązanie zawsze generuje prawidłowy program.

Złożoność czasowa takiego algorytmu wynosi $O(n+p+q)$, pamięciowa — $O(p+q)$.

Rys. 1 Przykładowa historia działania dźwigu trójramiennego. W każde pole wpisany jest kolejny numer ruchu, w którym to pole zostaje zajęte. Jedynie 7-my ruch jest typu 2.



Na przykład, jeśli charakterystyką dźwigu jest $(3, 10)$ oraz $n = 16$, to jeden z możliwych programów dźwigu jest przedstawiony na rysunku 1.

Dowód poprawności

Załóżmy, że dla pewnego wagonu $i = s$ po raz pierwszy nie można wykonać żadnego z ruchów. Łatwo zauważyć, że wagon $i+p+q$ jest wolny. Również s jest wolny, natomiast $s+p$, $s+q$ są zajęte, ponieważ nie można wykonać ruchu. Wagon $s+q$ został zajęty w czasie wykonywania ruchu typu 2 dla pozycji $j = s-p$. Ale z tego wynika, że w momencie wykonywania ruchu dla $i = j$ wolne były $i, j+p, j+p+q$, zatem można było wykonać ruch typu 1, który ma priorytet i zająć pozycje $j, s, s+q$. Pozycja s nie może być zatem wolna — sprzeczność.

Obserwacja

Można się zastanowić nad podobnym problem dla dźwigu czteroramiennego. Jest to trudne zadanie, ponieważ najpierw dla danego n trzeba sprawdzić, czy w ogóle istnieje odpowiedni program dźwigu — nie każdy ciąg kolejnych n liczb da się wtedy "pokryć". Na przykład nie ma pokrycia dla $n = 3$ i dźwigu o charakterystyce (1, 2, 1), tzn. takiego który zapełnia wagony $i, i + 1, i + 3, i + 4$.

TESTY

Do sprawdzania rozwiązań zawodników użyto 24 testów, których charakterystyki znajdują się poniżej.

Test	p	q	n
0	2	3	10
1	2	3	5
2	2	3	12
3	2	3	20
4	5	2	10
5	2	5	20
6	2	5	30
7	10	20	30
8	20	10	5
9	20	10	30
10	20	20	20
11	20	20	100
12	50	67	100
13	68	37	200
14	123	1234	1000
15	1234	123	2000
16	1000	2000	10000
17	2000	1000	20000
18	10000	10001	20000
19	10001	10000	30000
20	17777	20000	50000
21	20000	17777	60000
22	25000	35000	200000
23	35000	25000	300000

Następujące testy zostały zgrupowane: (1,2 i 3), (4,5 i 6), (7,8 i 9), (10 i 11), (12 i 13), (14 i 15), (16 i 17), (18 i 19), (20 i 21), (22 i 23).

Wojciech Guzicki
autor zadania, opracowanie

Marcin Mucha
program wzorcowy

Kod

Drzewo binarne *może byc puste, albo składać sie z wierzchołka, do którego przyczepione są dwa drzewa, tzw. lewe i prawe poddrzewo. W każdym wierzchołku zapisana jest jedna litera alfabetu angielskiego. Wierzchołek drzewa, który nie znajduje się w żadnym poddrzewie, nazywamy* **korzeniem**. *Mówimy, że drzewo jest* **binarnym drzewem poszukiwań (BST)**, *jeżeli dla każdego wierzchołka spełniony jest warunek, mówiący, że wszystkie litery z lewego poddrzewa wierzchołka występują w alfabecie wcześniej, niż litera zapisana w wierzchołku, natomiast wszystkie litery z prawego poddrzewa — później.* **Kodem drzewa BST** *nazywamy:*

- *ciąg pusty (0-elementowy), gdy drzewo jest puste,*
- *ciąg liter zaczynający się od litery zapisanej w korzeniu drzewa, po którym następuje kod lewego poddrzewa, a następnie kod prawego poddrzewa.*

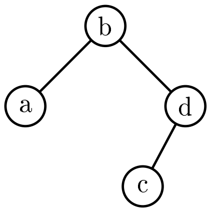
Rozważmy wszystkie k-wierzchołkowe drzewa BST, w wierzchołkach których umieszczono początkowe k liter alfabetu angielskiego. Wyobraźmy sobie listę kodów tych drzew, wypisanych w kolejności alfabetycznej. **(n,k)-kodem** *nazywamy n-ty kod na tej liście.*

Przykład

Istnieje dokładnie 14 kodów 4-wierzchołkowych binarnych drzew poszukiwań, konkretnie (w kolejności alfabetycznej):

`abcd abdc acbd adbc adcb bacd badc cabd cbad dabc dacb dbac dcab dcba`

Napis `badc` *jest (7,4)-kodem i odpowiada mu następujące drzewo BST:*



Zadanie

Napisz program, który:

- *z pliku tekstowego* `KOD.IN` *wczyta liczby n i k,*
- *wyznaczy (n, k)-kod,*
- *zapisze go do pliku tekstowego* `KOD.OUT`.

Wejście

W pierwszym i jedynym wierszu pliku tekstowego `KOD.IN` *zapisane są dokładnie dwie dodatnie liczby całkowite n i k, oddzielone pojedynczym znakiem odstępu, $1 \leqslant k \leqslant 19$. Liczba n nie przekracza liczby kodów drzew BST o k wierzchołkach.*

Wyjście

W pierwszym i jedynym wierszu pliku tekstowego `KOD.OUT` *powinno znajdować się słowo złożone z małych liter alfabetu angielskiego będące (n, k)-kodem.*

Przykład

Dla pliku wejściowego `KOD.IN`*:*
```
11 4
```
poprawną odpowiedzią jest plik tekstowy `KOD.OUT`*:*
```
dacb
```

ROZWIĄZANIE

Zastanówmy się najpierw, ile jest drzew binarnych o zadanej liczbie wierzchołków. Oznaczmy przez C_k liczbę drzew mających k wierzchołków. Oczywiście $C_0 = 1$: tylko drzewo puste ma zero wierzchołków. Również jest tylko jedno drzewo mające jeden wierzchołek, zatem $C_1 = 1$. Niech teraz k będzie liczbą naturalną większą od jedności. Drzewo mające k wierzchołków składa się z korzenia, do którego przyczepione są dwa poddrzewa mające łącznie $k-1$ wierzchołków. Mamy więc k możliwości:

- Przypadek 0. Lewe poddrzewo ma 0 wierzchołków, prawe ma $k-1$ wierzchołków (takich drzew jest oczywiście $C_0 \cdot C_{k-1}$).
- Przypadek 1. Lewe poddrzewo ma 1 wierzchołek, prawe ma ich $k-2$ (takich drzew jest $C_1 \cdot C_{k-2}$);

  ...
- Przypadek $k-1$. Lewe poddrzewo ma $k-1$ wierzchołków, prawe ma 0 wierzchołków (takich drzew jest $C_{k-1} \cdot C_0$).

Stąd otrzymujemy następujący wzór rekurencyjny na liczbę drzew:

$$C_k = C_0 \cdot C_{k-1} + C_1 \cdot C_{k-2} + \ldots + C_{k-1} \cdot C_0 = \sum_{i=0}^{k-1} C_i \cdot C_{k-i-1}.$$

Liczby C_k nazywamy liczbami Catalana. Występują one w wielu innych zadaniach kombinatorycznych. Na przykład:

- Liczba funkcji niemalejących

$$f : \{1,2,\ldots,n\} \to \{1,2,\ldots,n\},$$

  tzn. takich, że jeśli $k \leqslant l$, to $f(k) \leqslant f(l)$ dla dowolnych $k,l \in \{1,2,\ldots,n\}$ i spełniających warunek $f(k) \leqslant k$, dla każdego $k \in \{1,2,\ldots,n\}$, jest równa C_n.

- Liczba różnych triangulacji (czyli sposobów podziału przekątnymi na trójkąty) wielokąta wypukłego mającego n boków jest równa C_{n-2}.

- Przypuśćmy, że mamy dane działanie dwuargumentowe $\circ$ i rozważamy wyrażenie postaci $x_1 \circ x_2 \circ \ldots \circ x_n$. W tym wyrażeniu możemy rozstawić nawiasy (tak, by zawsze wykonywać działanie na dwóch argumentach) na wiele sposobów. Na przykład, dla $n = 4$ mamy następujące sposoby:

$$(x_1 \circ x_2) \circ (x_3 \circ x_4), \quad ((x_1 \circ x_2) \circ x_3) \circ x_4, \quad x_1 \circ (x_2 \circ (x_3 \circ x_4)),$$
$$(x_1 \circ (x_2 \circ x_3)) \circ x_4, \quad x_1 \circ ((x_2 \circ x_3) \circ x_4.$$

  Liczba różnych sposobów rozstawienia nawiasów jest równa C_{n-1}. Inaczej mówiąc, jeśli działanie $\circ$ jest niełączne (przykładem takiego działania jest dzielenie liczb rzeczywistych dodatnich), to wykonując to działanie na wszystkie możliwe sposoby, z zachowaniem kolejności $x_1, x_2, \ldots, x_n$, możemy otrzymać co najwyżej C_{n-1} różnych wyników.

- Liczby Catalana występują również jako rozwiązania następującego zadania: w kolejce do kina stoi $2n$ osób, n z nich ma tylko 5 złotych, każda z pozostałych ma tylko banknot dziesięciozłotowy. Bilet kosztuje 5 złotych. W kasie nie ma pieniędzy, więc może się okazać, że w którymś momencie kasjerka nie będzie mogła wydać reszty kolejnej osobie (na przykład, jeśli pierwsza osoba ma 5 złotych, a dwie następne po 10 złotych, to trzecia osoba nie otrzyma reszty i kolejka się zatrzyma). Okazuje się, że liczba sposobów ustawienia osób w kolejce tak, by wszyscy mogli kupić bilety, jest równa C_n.

Czytelnikowi zainteresowanemu kombinatoryką polecamy książki *W. Lipski, W. Marek – Analiza kombinatoryczna* i *V. Bryant – Aspekty kombinatoryki*, z których można dowiedzieć się więcej o liczbach Catalana.

Podany wyżej wzór rekurencyjny pozwala obliczyć kolejne liczby Catalana. Początkowe liczby Catalana są równe odpowiednio:

$$\begin{aligned} C_0 &= 1, \\ C_1 &= 1, \\ C_2 &= 2, \\ C_3 &= 5, \\ C_4 &= 14, \\ C_5 &= 42, \\ C_6 &= 132, \\ C_7 &= 429, \\ C_8 &= 1430, \\ C_9 &= 4862, \\ C_{10} &= 16796. \end{aligned}$$

Znany jest też wzór ogólny na k-tą liczbę Catalana. Mianowicie

$$C_k = \frac{1}{k+1} \cdot \binom{2k}{k}.$$

W programie wzorcowym liczby Catalana są obliczane za pomocą wzoru rekurencyjnego.

Teraz opiszemy algorytm znajdowania (n, k)-kodu drzewa binarnego. Najpierw wyjaśnimy istotę działania tego algorytmu na przykładzie. Przyjmijmy $k = 10$ i $n = 8751$. Wiemy już, że istnieje 16796 drzew binarnych mających 10 wierzchołków. Ta liczba jest sumą następujących 10 liczb:

1. liczby $C_0 \cdot C_9$, równej 4862; jest to liczba tych drzew, w których lewe poddrzewo ma 0 wierzchołków, a prawe 9 wierzchołków; są to dokładnie te drzewa binarne, które mają w korzeniu umieszczoną literę `a`;
2. liczby $C_1 \cdot C_8$, równej 1430; jest to liczba tych drzew, w których lewe poddrzewo ma 1 wierzchołek, a prawe 8 wierzchołków; są to drzewa, które mają w korzeniu umieszczoną literę `b`;
3. liczby $C_2 \cdot C_7$, równej 858; jest to liczba tych drzew, w których lewe poddrzewo ma 2 wierzchołki, a prawe 7 wierzchołków; są to drzewa, które mają w korzeniu umieszczoną literę `c`;

   ...

10. liczby $C_9 \cdot C_0$, równej 4862; jest to liczba tych drzew, w których lewe poddrzewo ma 9 wierzchołki, a prawe 0 wierzchołków; są to drzewa, które mają w korzeniu umieszczoną literę `j`.

Kody wszystkich drzew z grupy pierwszej poprzedzają oczywiście w porządku alfabetycznym kody drzew z grupy drugiej, te z kolei poprzedzają kody drzew w grupy trzeciej itd. Zauważmy teraz, że łączna liczba kodów zaczynających się od liter `a`, `b`, `c`, `d`, `e` jest równa 8398 (czyli mniej niż 8751), a jeśli dopuścimy jeszcze literę `f`, to liczba kodów wyniesie 8986 (a więc więcej niż 8751). Stąd wynika, że pierwszą literą kodu jest `f`. Po niej nastąpią litery od `a` do `e` (w nieznanej jeszcze kolejności) i na końcu litery od `g` do `j` (również w nieznanej kolejności). W ten sposób ustaliliśmy pierwszą literę kodu drzewa. Następnie będziemy chcieli ustalić kody obu poddrzew. W tym celu obliczymy kolejne numery tych kodów i rekurencyjnie znajdziemy same kody.

Każdemu z C_5 kodów złożonych z liter od `a` do `e` na początku odpowiada C_4 kodów złożonych z liter od `g` do `j` na końcu. Mamy znaleźć 353-ci z kolei kod rozpoczynający się literą `f` (gdyż $8751 - 8398 = 353$). Każdemu kodowi lewego poddrzewa odpowiada 14 kodów prawych poddrzew. Ale $352 = 25 \cdot 14 + 2$. Stąd wynika, że odcinek początkowy naszego kodu (po literze `f`) ma być dwudziestym szóstym z kolei kodem składającym się z liter od `a` do `e`, a odcinek końcowy ma być trzecim z kolei kodem składającym się z liter od `g` do `j`. Jest tak dlatego, że pierwszym dwudziestu pięciu kodom lewych poddrzew odpowiada tylko $25 \cdot 14$ czyli 350 kodów całych drzew. Nasz kod ma więc być trzecim kodem z następnej, tzn. dwudziestej szóstej grupy. Powstaje pytanie, dlaczego dzielimy 352, a nie 353. Mianowicie mamy proste wzory ogólne na numer grupy (czyli numer kodu lewego poddrzewa) i numer w tej grupie (czyli numer kodu prawego poddrzewa):

> przypuśćmy, że nasz kod ma być m-tym kodem zaczynającym się od pewnej znanej już litery c (w naszym przypadku $c =$ `f`); znamy więc wielkości obu poddrzew: niech lewe składa się z l wierzchołków, a prawe z p wierzchołków. Wtedy lewe poddrzewo ma numer $(m-1)$ **div** C_p+1, a prawe $(m-1)$ **mod** C_p+1.

Kontynuując obliczenia w naszym przykładzie zauważymy, że odcinek początkowy kodu zaczyna się literą `d`, po której następuje trzeci kod składający się z liter `a`, `b`, `c` i jedyny kod składający się z litery `e`. Odcinek końcowy zaczyna się literą `g`, po której następuje trzeci kod składający się z liter `h`, `i`, `j`. Ostatecznie otrzymamy kod `fdbacegihj`.

Teraz już nietrudno napisać odpowiedni algorytm rekurencyjny. Główna funkcja wyznaczająca (n,k)-kod ma w nim postać następującą. Argumentami funkcji są: litera cp, od której zaczynają się litery umieszczone w wierzchołkach drzewa (będzie to potrzebne w przypadku prawego poddrzewa, w którym występują litery nie od początku alfabetu), numer n drzewa i liczba wierzchołków k. W tablicy $Catalan$ umieszczone są oczywiście liczby Catalana, obliczone wcześniej za pomocą wzoru rekurencyjnego. Na początku funkcja zwraca wartości w przypadkach najprostszych: kod pusty dla drzewa pustego i kod jednoliterowy cp dla drzewa składającego się z jednego wierzchołka. Dla drzew większych obliczamy w pętli sumę iloczynów liczb Catalana dotąd, aż przekroczy ona n. Zauważmy, że wartości zmiennych l i p są wtedy równe dokładnie liczbom wierzchołków w lewym i prawym poddrzewie. Korzeń otrzymujemy znajdując literę stojącą w alfabecie o l miejsc dalej od cp. Potem zgodnie z powyższymi wzorami znajdujemy numery lewego i prawego poddrzewa i dwukrotnie wywołujemy procedurę rekurencyjnie. Otrzymane kody drzew łączymy wreszcie ze znalezionym wcześniej korzeniem. Oto treść tej procedury:

```
function Kod (cp : char; n : longint; k : integer) : String;
   { cp : litera początkowa kodu drzewa;
     n  : numer kodu drzewa wśród drzew mających wierzchołki
          oznaczone literami począwszy od cp zakładamy, że liczba
          n nie przekracza liczby Catalana C[k];
     k  : liczba wierzchołków drzewa}
var
   l,p : integer; {liczba wierzchołków w lewym i prawym poddrzewie}
   Suma : longint; {suma iloczynów liczb Catalana}
   SumaPoprzednia : longint;
   m : longint;
   n1,n2 : longint; {numery lewego i prawego poddrzewa}
   znak : char;
begin
   if (k=0) then
      Kod:=''
   else if (k=1) then
      Kod:=cp
   else begin
      Suma:=0;
      l:=-1;
      p:=k;
      repeat
         l:=l+1;
         p:=p-1;
         SumaPoprzednia:=Suma;
         Suma:=Suma+Catalan[l]*Catalan[p];
      until (Suma>=n);
      znak:=chr(ord(cp)+1);
      m:=n-SumaPoprzednia;
      n1:=((m-1) div Catalan[p])+1;
      n2:=((m-1) mod Catalan[p])+1;
      Kod:=znak+Kod(cp,n1,l)+Kod(succ(znak),n2,p)
   end
end;
```

Ten najprostszy program zamieszczony jest w pliku o nazwie `kod0.pas`. Nie jest on jednak optymalny. Można zauważyć, że pewne obliczenia są wykonywane wielokrotnie, na przykład sumowanie iloczynów liczb Catalana. Te iloczyny i ich sumy można obliczyć wcześniej i umieścić w tablicy. Tak napisany program znajduje się w pliku `kod.pas`.

Ograniczenie $k = 19$ wynika z wielkości liczb Catalana. Liczba $C_{19} = 1767263190$ mieści się jeszcze w zakresie liczb typu *longint*.

TESTY

Do sprawdzania rozwiązań zawodników użyto 17 testów:

- KOD0.IN — test z treści zadania,
- KOD1.IN — $n = 3$, $k = 3$,
- KOD2.IN — $n = 42$, $k = 5$,
- KOD3.IN — $n = 1$, $k = 6$,
- KOD4.IN — $n = 2000$, $k = 9$,
- KOD5.IN — $n = 50000$, $k = 11$,
- KOD6.IN — $n = 200000$, $k = 13$,
- KOD7.IN — $n = 1000000$, $k = 15$,
- KOD8.IN — $n = 20000000$, $k = 17$,
- KOD9.IN — $n = 100000000$, $k = 18$,
- KOD10.IN — $n = 400000000$, $k = 19$,
- KOD11.IN — $n = 800000000$, $k = 19$,
- KOD12.IN — $n = 100$, $k = 19$,
- KOD13.IN — $n = 1767263190$, $k = 19$,
- KOD14.IN — $n = 1111111111$, $k = 19$,
- KOD15.IN — $n = 1234567890$, $k = 19$,
- KOD16.IN — $n = 608197699$, $k = 19$.

Marcin Kubica
autor zadania, opracowanie

Krzysztof Sobusiak
program wzorcowy

Labirynt studni

Wewnątrz Bajtogóry znajduje się mityczny labirynt studni. Wejście do labiryntu znajduje się na szczycie góry. Labirynt składa się z wielu komnat. Każda z nich jest w jednym z trzech kolorów: czerwonym, zielonym lub niebieskim. Dwie komnaty tego samego koloru wyglądają identycznie i są nieodróżnialne. W każdej komnacie znajdują się trzy studnie oznaczone numerami 1, 2 i 3. Między komnatami można poruszać się tylko w jeden sposób — wskakując do jednej ze studni spada się (niekoniecznie pionowo) do jednej z komnat położonych niżej. Z komnaty, w której znajduje się wejście do labiryntu można przedostać się w ten sposób do każdej z pozostałych komnat. Wszystkie drogi przez labirynt prowadzą do smoczej jamy znajdującej się na samym jego dnie. Każdemu przejściu przez labirynt odpowiada ciąg numerów studni wybieranych w kolejno odwiedzanych komnatach. Ciąg ten nazywa się **planem podróży**.

W smoczej jamie żyje Bajtosmok. Legenda głosi, że ten, kto przedstawi Bajtosmokowi dokładny plan labiryntu, otrzyma wielkie skarby. Wszystkich innych Bajtosmok potężnym kopnięciem wyprasza z wnętrza góry.

Śmiałek o imieniu Bajtazar wielokrotnie przemierzał labirynt i opracował jego mapę. Jednak Bajtosmok orzekł, że co prawda wszystkie komnaty znajdują się na mapie, ale wiele komnat jest na niej powtórzonych.

— Ja też — powiedział poklepując Bajtazara po ramieniu — projektując labirynt narysowałem podobny rysunek, ale szybko stwierdziłem, że komnat można zrobić o wiele mniej, a gość poruszający się według dowolnego planu podróży i tak będzie oglądał takie same sekwencje komnat. Pomyślałem trochę i maksymalnie zredukowałem projekt.

Zadanie

Napisz program, który:

- *wczyta mapę Bajtazara z pliku tekstowego* `LAB.IN`*,*
- *obliczy prawdziwą liczbę komnat w labiryncie,*
- *zapisze wynik do pliku tekstowego* `LAB.OUT`*.*

Wejście

W pierwszym wierszu pliku tekstowego `LAB.IN` *zapisana jest jedna liczba całkowita n, $2 \leqslant n \leqslant 6\,000$, będąca liczbą komnat (łącznie ze smoczą jamą). Komnaty są ponumerowane od 1 do n tak, że komnaty o większych numerach znajdują się niżej (komnata, w której znajduje się wejście do labiryntu ma numer 1, a smocza jama ma numer n). W kolejnych $n-1$ wierszach pliku są opisane komnaty (poza smoczą jamą) oraz studnie prowadzące od nich w dół. W każdym z tych wierszy znajduje się litera, po niej jeden znak odstępu, a następnie trzy liczby całkowite oddzielone pojedynczymi odstępami. Litera oznacza kolor komnaty (*`C` *— czerwony,* `Z` *— zielony,* `N` *— niebieski), a i-ta liczba (dla $i = 1, 2, 3$) jest numerem komnaty, do której prowadzi i-ta studnia.*

Wyjście

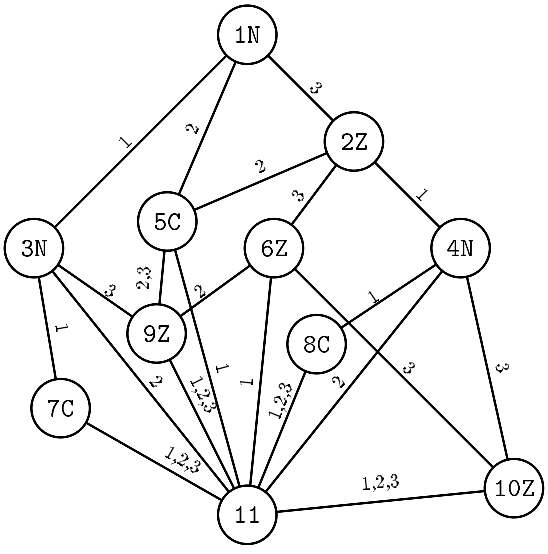
W pierwszym i jedynym wierszu pliku tekstowego `LAB.OUT` *powinna znaleźć się dokładnie jedna liczba całkowita będąca minimalną liczbą komnat w labiryncie (łącznie ze smoczą jamą), przy której podróżnik poruszający się według dowolnego planu obserwuje taką samą sekwencję komnat, jak dla labiryntu opisanego w pliku wejściowym.*

Przykład

Dla pliku wejściowego `LAB.IN`*:*

```
11
N 3 5 2
Z 4 5 6
N 7 11 9
N 8 11 10
C 11 9 9
Z 11 9 10
C 11 11 11
C 11 11 11
Z 11 11 11
Z 11 11 11
```

opisującego następujący labirynt:



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

```
8
```

ROZWIĄZANIE

Sformułujmy to zadanie w terminologii grafowej. Plan labiryntu to multigraf* skierowany bez cykli — jego wierzchołki reprezentują komnaty, a krawędzie reprezentują studnie. Każdy wierzchołek (z wyjątkiem smoczej jamy) jest pokolorowany jednym z trzech kolorów: czerwonym, zielonym lub niebieskim. Z każdego wierzchołka wychodzą po trzy krawędzie etykietowane liczbami 1, 2 i 3, z wyjątkiem smoczej jamy, z której nie wychodzą żadne krawędzie. Dla każdego wierzchołka na planie istnieje ścieżka prowadząca do niego z wierzchołka reprezentującego wejście do labiryntu. Podobnie, dla każdego wierzchołka istnieje ścieżka prowadząca z niego do smoczej jamy.

Każdej ścieżce prowadzącej z wejścia do labiryntu do smoczej jamy odpowiada plan podróży, będący ciągiem etykiet krawędzi tworzących daną ścieżkę, oraz ciąg kolorów wierzchołków tworzących tę ścieżkę. Równocześnie plan podróży wyznacza jednoznacznie odpowiadającą mu ścieżkę łączącą wejście do labiryntu ze smoczą jamą. Możemy powiedzieć, że dwa plany labiryntów są sobie równoważne, jeżeli:

- mają one takie same zbiory możliwych planów podróży, oraz
- dla każdego z tych planów podróży, odpowiadające im w obu multigrafach ścieżki łączące wejścia do labiryntów ze smoczymi jamami, charakteryzują się takimi samymi sekwencjami kolorów wierzchołków.

Zadanie polega na znalezieniu planu labiryntu o najmniejszej liczbie wierzchołków, równoważnego zadanemu planowi. Zastanówmy się, jak mógłby wyglądać taki minimalny plan.

Przez głębokość wierzchołka oznaczamy długość najdłuższej ścieżki prowadzącej z tego wierzchołka do smoczej jamy. Przyjmujemy, że smocza jama ma głębokość 0. Zbiór wszystkich wierzchołków o danej głębokości nazywamy warstwą.

Powiedzieliśmy, że każdej ścieżce łączącej wejście do labiryntu ze smoczą jamą odpowiada pewien plan podróży i jednocześnie ten plan podróży jednoznacznie wyznacza tę ścieżkę. Fakt ten można uogólnić. Każdej ścieżce możemy przyporządkować jej "plan" będący ciągiem etykiet krawędzi tworzących tę ścieżkę. Równocześnie taki plan, wraz z początkowym wierzchołkiem, wyznacza jednoznacznie tę ścieżkę.

Charakterystyką ścieżki będziemy nazywać parę złożoną z planu ścieżki oraz ciągu kolorów wierzchołków tworzących tę ścieżkę. Dla każdego wierzchołka definiujemy jego charakterystykę, jako zbiór charakterystyk wszystkich ścieżek prowadzących z tego wierzchołka do smoczej jamy.

Zauważmy, że wierzchołki mające taką samą charakterystykę należą do tej samej warstwy. Ponadto, jeśli z wierzchołków o takiej samej charakterystyce przejdziemy równocześnie wzdłuż krawędzi o takiej samej etykiecie, to dojdziemy do wierzchołków o takiej samej charakterystyce. Tak więc, w szukanym planie nie może być dwóch wierzchołków o takiej samej charakterystyce — gdyby były, to moglibyśmy skleić ze sobą dwa wierzchołki o takiej samej charakterystyce i minimalnej głębokości, uzyskując plan labiryntu równoważny danemu, ale o mniejszej liczbie wierzchołków. Ponadto, dla dowolnego wierzchołka v w danym planie labiryntu i dla dowolnej ścieżki

* Multigraf różni się tym od grafu, że może mieć wiele krawędzi łączących tę samą parę wierzchołków.

prowadzącej od wejścia do labiryntu do v, w szukanym planie istnieje ścieżka o takiej samej charakterystyce, prowadząca od wejścia do labiryntu do wierzchołka o takiej samej charakterystyce, jak charakterystyka v. Wynika stąd, że szukany plan labiryntu możemy uzyskać sklejając wierzchołki o takich samych charakterystykach.

Sklejając wierzchołki nie musimy wyznaczać ich charakterystyk. Możemy skorzystać z następującego prostego faktu. Niech v i w będą dwoma wierzchołkami o głębokości h. Jeżeli wszystkie wierzchołki o głębokościach mniejszych od h mają różne charakterystyki, to v i w mają takie same charakterystyki wtedy i tylko wtedy, gdy krawędzie o takich samych etykietach, wychodzące z v i w prowadzą do takich samych wierzchołków. Fakt ten wynika bezpośrednio z definicji charakterystyki. Tak więc, najlepiej sklejać wierzchołki o tych samych charakterystykach z kolejnych warstw o rosnących głębokościach.

Nasze rozwiązanie składa się z dwóch faz: podzielenia wierzchołków na warstwy i sklejania wierzchołków z kolejnych warstw. Jak jednak podzielić wierzchołki na warstwy? Możemy posłużyć się tu algorytmem podobnym do sortowania topologicznego. Warstwy wyznaczamy w kolejności rosnących głębokości. W momencie, gdy wstawiamy jakiś wierzchołek do warstwy, to "przeglądamy" wszystkie krawędzie wchodzące do tego wierzchołka. Jeżeli wszystkie trzy krawędzie wychodzące z któregokolwiek wierzchołka zostaną przejrzane, to możemy wstawić ten wierzchołek do warstwy o głębokości o jeden większej od głębokości wierzchołka, do którego prowadzi ostatnia przejrzana krawędź.

Z każdym wierzchołkiem możemy mieć związany licznik, który początkowo jest równy 3. Wówczas "przejrzenie" krawędzi polega na zmniejszeniu o 1 licznika związanego z wierzchołkiem, z którego dana krawędź wychodzi. Gdy licznik osiągnie 0, to wszystkie krawędzie wychodzące z danego wierzchołka zostały przejrzane.

Do zaimplementowania opisanego algorytmu może być przydatna następująca struktura danych. Dla każdego wierzchołka pamiętamy jego kolor, krawędzie wychodzące z tego wierzchołka, krawędzie wchodzące do tego wierzchołka oraz pewne dodatkowe informacje potrzebne w trakcie sklejania wierzchołków, które omówimy dalej.

```
const MAXN = 6000; { Maksymalna liczba wierzchołków. }
type
  PElem = ↑TElem; { Listy nieujemnych liczb całkowitych. }
  TElem = record
    l: Word;
    nast: PElem
  end
  TKolejka = record { Kolejki nieujemnych liczb całkowitych. }
    pocz, kon: PElem
  end
  TListy = array [0..MAXN] of PElem; { Tablica list. }
  TKolejki = array [1..MAXN] of TKolejka; { Tablica kolejek. }
var
  n: Word; { Liczba wierzchołków. }
  V: array [1..MAXN] of record { Wierzchołki. }
    kolor: Byte; { Kolor. }
```

```
      numer: Word; { Numeracja używana przy sklejaniu wierzchołków. }
      dol: array [1..3] of Word { Krawędzie wychodzące z wierzchołków. }
    end
    Gora: ↑TListy; { Krawędzie prowadzące do wierzchołków. }
    Warstwy: ↑TListy; { Warstwy o kolejnych głębokościach. }
```

Zwróćmy uwagę, że ze względu na ograniczoną wielkość pojedynczej tablicy, dane o wierzchołkach są pamiętane w trzech tablicach.

Funkcja dzieląca wierzchołki na warstwy wygląda następująco:

```
procedure Wstaw(var L: PElem; i: Word);
{ Wstawia liczbę i na początek listy L. }
  ⋮
function WyznaczWarstwy: Word;
{ Wyznacza warstwy, w wyniku daje liczbę warstw. }
var
  i, gleb: Word;
  P, Q: PElem;
  T: array [0..MAXN] of Byte; { Liczby nieodwiedzonych krawędzi. }
begin
  for i := 0 to n do begin
    Warstwy↑[i] := nil;
    T[i] := 3
  end;
  Wstaw(Warstwy↑[0], n);
  gleb := 0; { Głębokość aktualnie przetwarzanej warstwy. }
  while Warstwy↑[gleb] ≠ nil do begin
    { Wyznacz kolejne warstwy. }
    P := Warstwy↑[gleb];
    while P ≠ nil do begin
      { Przejrzyj wszystkie wierzchołki warstwy. }
      i := P↑.l;
      Q := Gora↑[i];
      while Q ≠ nil do begin
        { Przejrzyj wszystkie krawędzie wchodzące do danego }
        { wierzchołka. }
        Dec(T[Q↑.l]);
        if T[Q↑.l] = 0 then
          Wstaw(Warstwy↑[gleb+1], Q↑.l);
          { Wierzchołek, z którego prowadzi krawędź, }
          { należy do następnej warstwy. }
        Q := Q↑.nast
      end;
      P := P↑.nast
    end;
    Inc(gleb)
```

```
    end;
    WyznaczWarstwy := gleb-1
  end;
```

Każdy wierzchołek jest raz wstawiany do swojej warstwy i raz przeglądany. Każda krawędź jest przeglądana dokładnie raz. Tak więc złożoność czasowa pierwszej fazy algorytmu jest rzędu n, gdzie n jest liczbą wierzchołków.

Druga faza algorytmu polega na sklejaniu w kolejnych warstwach wierzchołków o takich samych charakterystykach. Każdemu wierzchołkowi możemy przyporządkować "kod" postaci $\langle k, v_1, v_2, v_3 \rangle$, gdzie k to kolor wierzchołka, a v_1, v_2, v_3, to wierzchołki, do których prowadzą krawędzie wychodzące z danego wierzchołka. Przetwarzając wierzchołki z kolejnej warstwy sklejamy wierzchołki o takich samych kodach. Jak jednak wyznaczyć, które wierzchołki mają takie same kody? Możemy skorzystać ze struktury słownikowej (dowolnego rodzaju, o dostępie w czasie logarytmicznym) — w momencie, gdy wstawiamy do struktury kod, który już w nim jest, to odpowiednie wierzchołki należy skleić. W ten sposób wyznaczenie wierzchołków do sklejenia wymaga czasu $O(n \log n)$, gdzie n jest liczbą wierzchołków. Jeżeli zamiast słownika o dostępie w czasie logarytmicznym użyjemy tablicy haszującej, to możemy uzyskać *oczekiwany* czas $O(n)$.

Można jednak wyznaczyć wierzchołki do sklejenia w *pesymistycznym* czasie $\Theta(n)$. Oznaczmy przez h głębokość wierzchołka z wejściem do labiryntu, przez n_i oznaczmy liczbę wierzchołków w warstwie głębokości i. Zauważmy, że $n_0 = n_h = 1$ oraz $\Sigma_{i=0}^{h} n_i = n$. W i-tym kroku drugiej fazy algorytmu sklejamy w i-tej warstwie wierzchołki o takich samych kodach. Z i-tej warstwy wychodzi $3n_i$ krawędzi prowadzących do co najwyżej $3n_i$ wierzchołków. Wierzchołki te możemy wyznaczyć i ponumerować w czasie rzędu n_i. W kodach wierzchołków z i-tej warstwy wierzchołki, do których prowadzą krawędzie możemy reprezentować za pomocą numerów w tej numeracji. Wówczas możemy posortować kody leksykograficznie w czasie rzędu n_i.

Oto fragment programu realizujący sortowanie kodów wierzchołków i-tej warstwy. Zakładamy, że początkowo wszystkie wierzchołki mają zainicjowane pola `numer` na 0, oraz że "kubły" są zainicjowane na puste kolejki.

```
  var num, i, j : Word;
    P, Q, W: PElem;
    Kubly : ↑TKolejki; { Kubełki do sortowania leksykograficznego. }
    procedure Przerzuc(var K: TKolejka; var L: PElem);
    { Przerzuca pierwszy element listy L na koniec kolejki K. }
      ⋮
    function ScalKubly(maxnum: Word): PElem;
    { Łączy kolejki z kubełków o numerach [1..maxnum] w jedną listę. }
      ⋮
  begin
    num := 0;
    W := Warstwy[i];
    P := W;
    while P ≠ nil do begin
```

```
      { Ponumeruj wszystkie wierzchołki, do których prowadzą }
      { krawędzie z i-tej warstwy. }
      for j := 1 to 3 do with V[V[P↑.l].Dol[j]] do
        if numer = 0 then begin
          Inc(num);
          numer := num
        end
      P := P↑.nast
    end
    { Sortuj leksykograficznie po numerach wierzchołków, }
    { do których prowadzą krawędzie z i-tej warstwy. }
    for j := 3 downto 1 do begin
      while W ≠ nil do
        Przerzuc(Kubly↑[V[V[W↑.l].Dol[j]].numer], W);
      W := ScalKubly(num)
    end
    { Sortuj po kolorach wierzchołków. }
    while W ≠ nil do
      Przerzuc(Kubly↑[V[W↑.l].kolor], W);
    Warstwy[i] := ScalKubly(3);
  end
```

Po posortowaniu, takie same kody będą sąsiadować ze sobą — wystarczy więc przejrzeć je w kolejności posortowania i skleić wierzchołki o takich samych kodach.

Sklejając kilka wierzchołków, do wybranego wierzchołka doklejamy pozostałe. Doklejając jeden wierzchołek do drugiego musimy poprawić krawędzie prowadzące do pierwszego z nich — korzystamy tu z list wierzchołków, z których krawędzie prowadzą do danego wierzchołka. Możemy natomiast zaniedbać poprawianie list, na których występują sklejane wierzchołki, gdyż sklejamy wierzchołki w warstwach o coraz większych głębokościach i listy te nie będą już nam potrzebne. Oznaczmy przez w_i liczbę krawędzi prowadzących do i-tej warstwy. Koszt czasowy sklejania wierzchołków z warstwy o głębokości i wynosi $\Theta(n_i + w_i)$. Poniższy fragment kodu skleja odpowiednie wierzchołki.

```
function CzyZgodne(a, b: Word): Boolean;
{ Sprawdza, czy a i b mają te same kody. }
var j: Word;
begin
  CzyZgodne := False;
  if V[a].kolor ≠ V[b].kolor { Muszą mieć ten sam kolor. }
    then exit;
  for j := 1 to 3 do
{ Krawędzie muszą prowadzić do tych samych wierzchołków. }
    if V[a].Dol[j] ≠ V[b].Dol[j] then exit;
  CzyZgodne := True
end
procedure LikwidujKomnate(a, b: Word);
```

```
  { Dokleja wierzchołek b do a. }
  var
    P: PElem; j: Word;
  begin
    P := Gora↑[b];
    while P ≠ nil do begin
      with V[P↑.l] do
        for j := 1 to 3 do
          if Dol[j] = b then Dol[j] := a;
      P := P↑.nast
    end
  end
  var
    ile, a, b: Word;
    P, Q, W: PElem;
  begin
    W := Warstwy[i]; P := W; Q := P↑.nast;
    ile := 1; {Liczba różnych wierzchołków warstwy.}
    while Q ≠ nil do begin {Przejdź posortowaną listę.}
      a := P↑.l; b := Q↑.l; {Wierzchołki do ew. sklejenia.}
      if CzyZgodne(a,b) then
        LikwidujKomnate(a,b)
      else begin {Wierzchołek istotnie różny od dotychczasowych.}
        Inc(ile);
        P := Q;
      end
      Q := Q↑.nast
    end
    P := W;
    while P ≠ nil do begin {Likwiduj numerację.}
      for j := 1 to 3 do
        V[V[P↑.l].Dol[j]].numer := 0;
      P := P↑.nast
    end
    {ile = liczba różnych wierzchołków warstwy.}
  end
```

Pełną treść przedstawionego rozwiązania można znaleźć na załączonej dyskietce.

Zanalizujmy koszt działania opisanego algorytmu. Z każdym wierzchołkiem i każdą krawędzią jest związana stała ilość informacji. Liczba krawędzi jest liniowa ze względu na liczbę wierzchołków. Tak więc złożoność pamięciowa opisanego algorytmu wynosi $\Theta(n)$, gdzie n jest liczbą wierzchołków.

Podział wierzchołków na warstwy wymaga czasu $\Theta(n)$. Posortowanie wierzchołków i-tej warstwy według ich kodów wymaga czasu $\Theta(n_i)$. Sklejenie wierzchołków z

i-tej warstwy o tej samej charakterystyce wymaga czasu $\Theta(n_i + w_i)$. Tak więc łączny czas działania opisanego algorytmu wynosi

$$\Theta\left(n + \Sigma_{i=1}^{h-1} n_i + \Sigma_{i=1}^{h-1}(n_i + w_i)\right)$$

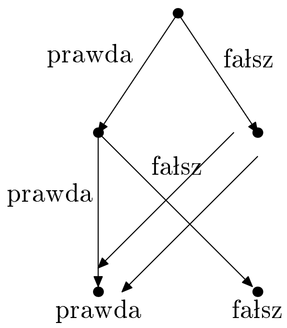
Zauważmy jednak, że $\Sigma_{i=1}^{h-1} n_i = n - 2$ oraz, że $\Sigma_{i=1}^{h-1} w_i < 3n$. Stąd czas działania opisanego algorytmu wynosi $\Theta(n)$.

Binarne diagramy decyzyjne

Opisywane zadanie ma związek z tzw. binarnymi diagramami decyzyjnymi (ang. *binary decision diagrams*, w skrócie BDD) — stosowaną w praktyce strukturą danych. Struktura ta służy do zwięzłego reprezentowania formuł rachunku zdań. Przyjmijmy, że w formułach może się pojawiać k różnych zmiennych zdaniowych: $x_1, \ldots, x_k$. Podobnie jak w przypadku labiryntu Bajtosmoka, BDD ma postać multigrafu skierowanego. Wierzchołki tego multigrafu są podzielone na $k+1$ warstw, o głębokościach 0–k. Warstwę o głębokości 0 tworzą dokładnie dwa wierzchołki, reprezentujące prawdę i fałsz — tak jakby były dwie smocze jamy. Z każdego wierzchołka należącego do pozostałych warstw wychodzą dokładnie po dwie krawędzie, jedna z etykietą "prawda" i jedna z etykietą "fałsz" (zamiast trzech krawędzi ponumerowanych 1, 2 i 3). Krawędzie wychodzące z danego wierzchołka muszą prowadzić do wierzchołków należących do warstwy o głębokości o 1 mniejszej. Warstwę o głębokości k tworzy dokładnie jeden wierzchołek — odpowiednik wejścia do labiryntu. Zauważmy, że wszystkie ścieżki prowadzące z tego wierzchołka do wierzchołków reprezentujących prawdę i fałsz mają długość $k + 1$.

W jaki sposób możemy traktować BDD jak formułę rachunku zdań? Dla zadanego wartościowania zmiennych zdaniowych (tzn. przypisania zmiennym zdaniowym wartości prawda/fałsz) formuła rachunku zdań jest albo prawdziwa, albo fałszywa. Aby odczytać, czy dla zadanego wartościowania zmiennych zdaniowych BDD jest prawdziwe, czy fałszywe musimy przejść przez BDD zgodnie z "planem podróży" wyznaczonym przez wartościowanie zmiennych. Przejście zaczynamy od jedynego wierzchołka należącego do warstwy o głębokości k. Między wierzchołkami przechodzimy wzdłuż krawędzi — będąc w wierzchołku należącym do warstwy o głębokości h wybieramy tę krawędź, która ma etykietę równą wartości zmiennej x_h. Wartość BDD dla danego wartościowania jest taka, jak etykieta wierzchołka, do którego docieramy na końcu przejścia.

Poniższy rysunek przedstawia BDD reprezentujące formułę $\neg x_2 \vee (x_1 \wedge x_2)$.



Rys. 1

Zauważmy, że dla każdej formuły rachunku zdań istnieje BDD reprezentujące daną formułę. BDD takie możemy łatwo zbudować. Weźmy pełne drzewo binarne wysokości $k+1$. Krawędzie wychodzące z wierzchołków na lewo etykietujemy prawdą, a wychodzące na prawo fałszem. Zauważmy, że każdemu wartościowaniu zmiennych zdaniowych odpowiada dokładnie jeden liść — ten, do którego prowadzi z korzenia drzewa przejście zgodne z wartościowaniem. Liście drzewa etykietujemy prawdą i fałszem, zgodnie z tym jaka jest wartość formuły dla wartościowania zmiennych odpowiadającego danemu liściowi. Jeżeli skleimy liście mające takie same etykiety (i ew. dodamy jeden wierzchołek, jeżeli wszystkie liście mają takie same etykiety), to uzyskujemy BDD reprezentujące daną formułę.

Zauważmy, że formuła rachunku zdań jest tautologią wtw. gdy w reprezentującym ją BDD żadna krawędź nie prowadzi do wierzchołka reprezentującego fałsz.

Omawiane zadanie odpowiada przekształceniu BDD na równoważne o minimalnej liczbie wierzchołków. Takie BDD stosuje się jako formę zwięzłego reprezentowania formuł rachunku zdań. Warto nadmienić, że dwie formuły rachunku zdań są sobie równoważne wtedy i tylko wtedy, gdy minimalne BDD reprezentujące te formuły są takie same.

TESTY

Do sprawdzania rozwiązań zawodników użyto 14-tu testów. Oto ich krótka charakterystyka:

- LAB1.IN — prosty test badający poprawność rozwiązań, $n = 10$,
- LAB2.IN — test losowy badający poprawność rozwiązań, $n = 50$,
- LAB3.IN — test losowy badający poprawność rozwiązań, $n = 200$,
- LAB4.IN — w tym teście plan labiryntu składa się z 335 warstw, a każda warstwa z 3 wierzchołków: czerwonego, zielonego i niebieskiego, (z wyjątkiem najpłytszej i najgłębszej warstwy, które mają po jednym wierzchołku); z każdego wierzchołka prowadzą krawędzie łączące go z wszystkimi wierzchołkami o głębokości o 1 mniejszej; żadne wierzchołki nie są sklejane ze sobą,

- LAB5.IN — prosty test badający poprawność rozwiązań, $n = 12$,
- LAB6.IN — w tym teście plan labiryntu ma kształt pełnego drzewa wysokości 8, plus smocza jama; wszystkie wierzchołki z jednej warstwy są tego samego koloru, a kolejne warstwy mają kolory dobierane na przemian; w każdej warstwie wszystkie wierzchołki są sklejane ze sobą,
- LAB7.IN — losowy test badający efektywność rozwiązań, wszystkie wierzchołki są tego samego koloru, $n = 6\,000$, 558 sklejeń wierzchołków,
- LAB8.IN — losowy test badający efektywność rozwiązań, wszystkie wierzchołki są tego samego koloru, $n = 6\,000$, 51 sklejeń wierzchołków,
- LAB9.IN — losowy test badający efektywność rozwiązań, wszystkie wierzchołki są tego samego koloru, $n = 6\,000$, żadne wierzchołki nie są sklejane ze sobą,
- LAB10.IN — prosty test badający poprawność rozwiązań, $n = 12$,
- LAB11.IN — test badający efektywność rozwiązań, wszystkie wierzchołki są tego samego koloru, $n = 6\,000$, dla każdego $i \leqslant n - 3$ z wierzchołka nr i prowadzą krawędzie do wierzchołków o numerach $i + 1$, $i + 2$ i $i + 3$, z wierzchołka nr $n - 2$ prowadzi jedna krawędź do wierzchołka nr $n - 1$ i dwie krawędzie do wierzchołka nr n, a z wierzchołka nr $n - 1$ wszystkie krawędzie prowadzą do wierzchołka nr n; żadne wierzchołki nie są sklejane ze sobą,
- LAB12.IN — prosty test badający poprawność rozwiązań, $n = 12$,
- LAB13.IN — test badający efektywność rozwiązań, wszystkie wierzchołki są tego samego koloru, $n = 6\,000$, każda warstwa składa się z dokładnie jednego wierzchołka, krawędzie wychodzące z każdego wierzchołka prowadzą do (jedynego) wierzchołka o głębokości o jeden mniejszej; żadne wierzchołki nie są sklejane ze sobą,
- LAB14.IN — prosty test badający poprawność rozwiązań, $n = 12$.

W testach nr 4, 9, 11 i 13 nie było żadnych sklejeń wierzchołków. Testy te zostały zgrupowane z prostymi testami badającymi poprawność rozwiązań (4 z 5, 9 z 10, 11 z 12 oraz 13 z 14) — oceniane rozwiązanie otrzymywało punkty za testy z danej grupy tylko wtedy, gdy uzyskiwało punkty za obydwa testy tworzące grupę. Takie grupowanie testów pozwalało wychwycić rozwiązania stwierdzające zawsze, że żadne wierzchołki nie ulegają sklejeniu.

Poniższa tabelka przedstawia wielkości poszczególnych testów oraz wyniki.

Nr testu	n	Wynik
1	10	6
2	50	33
3	200	106
4	1 001	1 001
5	12	8
6	3 281	9
7	6 000	5 442
8	6 000	5 949
9	6 000	6 000
10	12	7
11	6 000	6 000
12	12	8
13	6 000	6 000
14	12	8

Zawody III stopnia

opracowania zadań

Piotr Chrząstowski
autor zadania, opracowanie

Marcin Sawicki
program wzorcowy

Lollobrygida

W fabryce poduszkowców do budowy torów testowych używa się standardowych bloków o różnych wysokościach, ustawianych jeden za drugim. W idealnie zbudowanym torze, zwanym **lollobrygidą**, *nigdy nie występują obok siebie dwa bloki jednakowej wysokości, nigdy też trzy kolejne bloki nie mają kolejno coraz większych, albo coraz mniejszych wysokości.*

Mówiąc bardziej formalnie, niech $h_1, \ldots, h_n$ *oznacza ciąg wysokości kolejnych bloków należących do toru. Jeśli dla każdego* $1 \leqslant i \leqslant n-2$ *zachodzi:*

- $h_i < h_{i+1}$ *i* $h_{i+1} > h_{i+2}$ *lub*
- $h_i > h_{i+1}$ *i* $h_{i+1} < h_{i+2}$,

to taki tor można nazwać **lollobrygidą**.

Przykład

Z zestawu 5 bloków o wysokościach 3, 3, 3, 5, 2 nie da się zbudować lollobrygidy, gdyż albo musiałyby stać w niej obok siebie dwa bloki wysokości 3, albo musi się w niej pojawić jedna z niedozwolonych sekwencji 2, 3, 5 lub 5, 3, 2.
A oto przykład lollobrygidy, poprawnie zbudowanej z innego zestawu bloków: 3, 2, 5, 2, 3, 1. Z tego zestawu można też zbudować inne lollobrygidy.

Zadanie

Napisz program, który wczyta z pliku tekstowego `LOL.IN` *liczbę zestawów danych i dla każdego zestawu:*

- *wczyta liczbę bloków oraz wysokości poszczególnych bloków,*
- *stwierdzi, czy z podanego zestawu można zbudować lollobrygidę,*
- *zapisze wynik w pliku tekstowym* `LOL.OUT`.

Wejście

W pierwszym wierszu pliku tekstowego `LOL.IN` *znajduje się liczba całkowita* d, $1 \leqslant d \leqslant 100$, *równa liczbie zestawów danych. W następnym wierszu pliku* `LOL.IN` *zaczyna się pierwszy zestaw danych.*

W pierwszym wierszu każdego zestawu danych znajduje się liczba całkowita n, $3 \leqslant n \leqslant 1\,000\,000$. *Jest to liczba bloków w tym zestawie.*

W kolejnych n *wierszach znajdują się wysokości bloków. Każdy z tych wierszy zawiera jedną liczbę całkowitą* h *równą wysokości odpowiedniego bloku,* $1 \leqslant h \leqslant 10^9$.

Kolejne zestawy danych następują bezpośrednio po sobie.

Wyjście

Plik tekstowy `LOL.OUT` *powinien zawierać dokładnie d wierszy, po jednym dla każdego zestawu danych. W i-tym wierszu pliku* `LOL.OUT` *powinien być zapisany jeden wyraz:*

- *TAK, jeżeli z i-tego zestawu bloków można zbudować lollobrygidę,*
- *NIE, w przeciwnym przypadku.*

Przykład

Dla pliku wejściowego `LOL.IN`*:*

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

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

```
NIE
TAK
```

ROZWIĄZANIE

Zadanie o lollobrygidzie w dużej części sprowadzało się do znanego i opisywanego w literaturze zadania o wyborze lidera. Zadanie to formułuje się następująco: Stwierdzić, czy w ciągu $a_1, \ldots, a_n$ istnieje wartość, która występuje w nim więcej niż $\frac{n}{2}$ razy. Jeżeli taka wartość występuje, to nazywamy ją liderem.

Okazuje się bowiem, że zachodzi następujące twierdzenie:

Twierdzenie. Z elementów $a_1, \ldots, a_n$ można ułożyć lollobrygidę wtedy i tylko wtedy, gdy zachodzi jeden z dwóch przypadków:

(1) W ciągu $a_1, \ldots, a_n$ nie występuje lider.

(2) Długość n jest nieparzysta i w ciągu $a_1, \ldots, a_n$ istnieje lider występujący w nim dokładnie $\frac{n+1}{2}$ razy, a pozostałe elementy ciągu $a_1, \ldots, a_n$ są albo wszystkie większe, albo wszystkie mniejsze od lidera.

Dowód. Najpierw pokażemy, że jeżeli istnieje lider nie spełniający warunku (2), to lollobrygidy ułożyć się nie da, a potem, że jeśli ciąg spełnia którykolwiek z warunków (1) lub (2), to lollobrygidę ułożyć się da.

Załóżmy więc, że istnieje lider nie spełniający warunku (2). Jeżeli n jest parzyste, to lider występuje co najmniej $\frac{n}{2}+1$ razy, zatem przy każdym ułożeniu wyrazów ciągu gdzieś dwaj liderzy muszą ze sobą sąsiadować. Jeśli bowiem podzielimy ciąg na dwuelementowe bloki (a jest ich dokładnie $\frac{n}{2}$), to z zasady szufladkowej Dirichleta wynika, że przynajmniej w jednym bloku znajdą się dwaj liderzy, więc podany ciąg nie będzie lollobrygidą.

Jeżeli n jest nieparzyste i lider występuje co najmniej $\frac{n+1}{2}+1$ razy, to dzieląc ciąg $a_1,\ldots,a_n$ na bloki dwuelementowe, otrzymamy takich bloków $\frac{n-1}{2}$ i zostanie jeszcze jeden blok jednoelementowy. Tak jak poprzednio, każda próba umieszczenia w takich blokach $\frac{n+1}{2}+1$ liderów musi się zakończyć tym, że dwaj liderzy znajdą się w jednym bloku obok siebie. Jeśli zaś n jest nieparzyste i lider występuje dokładnie $\frac{n+1}{2}$ razy, ale nie jest najmniejszą, ani największą wartością, to co prawda można wszystkich liderów rozmieścić w $\frac{n-1}{2}+1$ różnych blokach, ale żeby tor był lollobrygidą musimy umieścić ich wszystkich na nieparzystych pozycjach. Wtedy jednak na pewno wystąpi sytuacja, w której kolejne trzy wartości będą albo rosnące, albo malejące. Załóżmy bowiem dla symetrii, że $a_2 > a_1$ (a_1 jest wartością lidera). Wtedy $a_4 > a_3 = a_1$ (inaczej ciąg a_2, a_3, a_4 byłby malejący), $a_6 > a_5 = a_1$, itd. Okazałoby się więc, że a_1 jest najmniejszą wartością wbrew założeniu. Analogicznie przeprowadzamy rozumowanie, jeśli $a_2 < a_1$.

Pokazaliśmy więc pierwszą część twierdzenia. Pozostaje wykazać, że jeśli lidera nie ma lub jeśli jest lider, ale spełniający warunek (2), to lollobrygidę da się ułożyć z podanych wartości. Proste jest pokazanie, że jeśli lider spełnia warunek (2), to lollobrygidę daje się ułożyć. Wystarczy na wszystkich nieparzystych pozycjach ulokować lidera, a pozostałe elementy umieścić dowolnie na parzystych pozycjach. Warunek lollobrygidy w oczywisty sposób jest spełniony.

Zostało zatem do wykazania, że jeśli lidera nie ma, to lollobrygidę zawsze się da ułożyć. Dowód będzie indukcyjny ze względu na długość ciągu n.

Wzmocnimy najpierw nieco tezę — jest to częsty chwyt przy dowodach indukcyjnych; korzystamy przecież wtedy z mocniejszego założenia indukcyjnego. Pokażemy zatem, że jeśli lidera w ciągu $a_1,\ldots,a_n$ nie ma, to istnieje dla niego lollobrygida $b_1,\ldots,b_n$, i to taka, że $b_1 \neq b_n$.

Bazę indukcji pokażemy dla wartości $n = 2$. To nie szkodzi, że w sformułowaniu zadania $n \geqslant 3$. Jeżeli uda się nam zacząć indukcję od mniejszej wartości — tym lepiej. Twierdzenie staje się trochę ogólniejsze, a zazwyczaj dla mniejszych wartości n łatwiej jest udowodnić bazę indukcji. Faktycznie, załóżmy, że w ciągu a_1, a_2 nie ma lidera. Wobec tego $a_1 \neq a_2$, zatem ciąg a_1, a_2 jest lollobrygidą: nie ma w nim ani dwóch sąsiadujących, równych sobie elementów, jak również żadne trzy sąsiednie elementy nie są posortowane — po prostu nie istnieją.

Załóżmy teraz, że teza zachodzi dla każdego ciągu $a_1,\ldots,a_k$, $k < n$. Rozbijmy dowód na dwa przypadki: n nieparzyste i n parzyste.

Najpierw n **nieparzyste**. Niech m będzie modą ciągu $a_1,\ldots,a_n$. (Moda ciągu $a_1,\ldots,a_n$ jest to taka wartość, która w ciągu powtarza się najczęściej. Jeśli takich wartości jest więcej niż jedna, to każda z tych wartości jest modą. Lider oczywiście zawsze jest modą, ale nie na odwrót.) Usuńmy z ciągu $a_1,\ldots,a_n$ jeden element — modę. W powstałym $(n-1)$-elementowym ciągu nie ma lidera, bo przy nieparzystym

n, co najwyżej $\frac{n-1}{2}$ elementów ciągu mogło być równych modzie nie będącej liderem i żadna inna wartość nie mogła wystąpić w nim więcej razy.

Taki ciąg po usunięciu jednego elementu będącego modą spełnia założenie indukcyjne: jest krótszy od n i nie zawiera lidera, zatem można z niego utworzyć lollobrygidę $b_1, \ldots, b_{n-1}$ taką, że $b_1 \neq b_{n-1}$. Załóżmy, że $b_1 < b_2$, a co za tym idzie, $b_{n-2} < b_{n-1}$ ($n-1$ jest parzyste, a kolejne wyrazy są na przemian od siebie większe i mniejsze). W przypadku odwrotnym, kiedy $b_1 > b_2$ i $b_{n-2} > b_{n-1}$, rozumowanie będzie analogiczne. Jeśli teraz $m > b_1$, to wstawiając m przed ciąg $b_1, \ldots, b_{n-1}$ otrzymujemy lollobrygidę, jeśli nie, to sprawdzamy, czy $m < b_{n-1}$. Jeśli tak, to lollobrygidę otrzymamy wstawiając m na koniec. Pozostał więc przypadek $b_1 \geqslant m \geqslant b_{n-1}$, przy czym jedna z tych nierówności musi być ostra. Wstawienie m z tej strony, z której nierówność jest ostra, między skrajny element, a jego sąsiada, spowoduje, że otrzymamy lollobrygidę o n elementach. To kończy dowód dla n nieparzystego.

Jeśli n jest **parzyste**, to mamy trochę inną sytuację, bo usunięcie mody może doprowadzić nas do sytuacji, w której pojawi się lider w ciągu $n-1$-elementowym i nie będzie można skorzystać z założenia indukcyjnego. Poradzimy sobie z tym przypadkiem zauważając, że tak może być jedynie wtedy, gdy w ciągu występują tylko dwie wartości i każda z nich jest modą. Wtedy nie musimy stosować indukcji — wystarczy ustawić jedne wartości na nieparzystych, a drugie na parzystych miejscach i mamy lollobrygidę. W każdym innym przypadku po usunięciu z ciągu, w którym nie było lidera, jednego elementu o wartości mody, lidera nadal nie będzie. Możemy wtedy stosować założenie indukcyjne. Załóżmy więc, że ciąg $b_1, \ldots, b_{n-1}$ jest lollobrygidą taką, że $b_1 \neq b_{n-1}$. Bez utraty ogólności możemy założyć, że $b_1 < b_2$, a co za tym idzie, $b_{n-2} > b_{n-1}$, gdyż $n-1$ jest nieparzyste. Przypadek przeciwny ma analogiczny dowód.

Jeżeli teraz $m > b_1$ lub $m > b_{n-1}$, to wstawiając m na początek lub odpowiednio na koniec, uzyskujemy lollobrygidę. Pozostaje zatem do rozważenia przypadek, gdy m nie przekracza żadnego ze skrajnych elementów. Ponieważ są one różne, więc m od jednego z nich musi być ostro mniejsze. Dajmy na to, że $m < b_1$. Wtedy wstawiając m między b_1 a b_2, otrzymujemy lollobrygidę. Analogicznie postępujemy, gdy $m < b_{n-1}$.

Wyczerpaliśmy tym samym wszystkie przypadki, za każdym razem pokazując, że lollobrygida jest możliwa do ułożenia. Twierdzenie zostało tym samym udowodnione.

Uff! Mamy już w ręku ważny wynik, który ułatwi rozwiązanie zadania w efektywny sposób. Pozostaje więc sprawdzić, czy lider istnieje i jeżeli nie, to odpowiedź jest TAK, a jeśli lider istnieje, to wystarczy upewnić się, czy nie spełnia warunku (2). Jeżeli spełnia, to odpowiedź jest TAK, a jeśli nie spełnia, to odpowiedź jest NIE.

Omówmy tu 4 różne algorytmy stwierdzania istnienia lidera.

Pierwszy algorytm, być może najprostszy do wymyślenia, polega na posortowaniu danych i znalezieniu w posortowanym ciągu najdłuższej sekwencji tych samych elementów. Koszt dobrego sortowania jest proporcjonalny do $n \log n$, koszt znalezienia najdłuższej stałej sekwencji jest proporcjonalny do n. Łącznie mamy więc algorytm o złożoności rzędu $n \log n$. Nieźle, ale nie optymalnie.

Drugi algorytm bazuje na spostrzeżeniu, że jeśli w ciągu jest lider, to jest on równy medianie ciągu, czyli elementowi, który w posortowanym ciągu wystąpiłby pośrodku (dla n parzystego jako medianę definiuje się dowolny z dwóch elementów środkowych). Algorytm zatem rozbija się na dwie części: wyznaczenie mediany i sprawdzenie, czy mediana jest liderem. Ta druga część jest bardzo prosta. Wyznaczenie mediany jest bardziej kłopotliwe, o ile nie chcemy korzystać z sortowania.

Medianę można wyznaczyć w czasie pesymistycznym proporcjonalnym do n, ale nie jest to proste. Zadanie to realizuje na przykład algorytm Bluma, Floyda, Pratta, Rivesta, Tarjana, zwany algorytmem piątek lub algorytmem pięciu — każda z nazw jest uzasadniona. Algorytm ten opisany jest np. w książce [10]. Problem polega na tym, że koszt tego algorytmu, choć liniowy, obarczony jest dużą stałą i dla potrzeb naszego zadania algorytm ten raczej się nie nadaje. Przy podanych ograniczeniach na rozmiar danych uzyskanie za pomocą tego algorytmu lepszego czasowo rozwiązania, niż przez sortowanie, wydaje się trudne, a być może jest wręcz niemożliwe. Pamiętajmy bowiem, że choć teoretycznie złożoność jednego algorytmu może być lepsza od złożoności drugiego, to wcale nie znaczy, że zawsze należy stosować ten pierwszy. Na przykład dla krótkich ciągów danych (rzędu kilku-kilkunastu) nie należy sortować quicksortem, tylko którymś z algorytmów o złożoności kwadratowej, choćby przez proste wstawianie.

Z praktycznego punktu widzenia lepiej byłoby tu zastosować opisany również w [10] algorytm Hoare'a wyznaczania mediany. Jego idea jest podobna do pomysłu sortowania quicksortem. Losuje się bowiem jedną z wartości ciągu, a następnie rozrzuca pozostałe wartości na 3 grupy: elementów mniejszych, równych i większych od tej wartości, po czym szuka się elementu o odpowiednim numerze w jednej z tych grup, tą samą zresztą metodą. Ten algorytm ma kwadratową złożoność pesymistyczną (gdy zawsze będziemy mieli pecha i wybierali element skrajny jako wartość, względem której rozrzucamy), ale średnio działa w czasie liniowym. W praktyce doskonale nadaje się do znajdowania mediany i jest dość prosty do zakodowania — własność bardzo ważna w trakcie zawodów.

Trzeci algorytm, trochę niepewny, bo nie dający dobrych rezultatów ze stuprocentową pewnością, to algorytm probabilistyczny polegający po prostu na kilkukrotnym wylosowaniu dowolnej wartości i sprawdzeniu, czy jest ona liderem. Zauważmy, że jeśli lider istnieje, to z prawdopodobieństwem większym niż $\frac{1}{2}$ wylosujemy właśnie jego. Jeżeli na przykład po dziesięciu losowaniach nie natrafimy na lidera, to uznajemy, że lidera nie ma i dajemy odpowiedź TAK. Prawdopodobieństwo błędu jest mniejsze niż $\frac{1}{1000}$. W praktyce tego typu rozwiązania są stosowane i na olimpiadzie czasami bardzo skuteczne. Tutaj wadą jest konieczność wielokrotnego przeglądania całego ciągu wejściowego. Im większą chcemy mieć pewność, tym więcej to kosztuje. Testy były tak przygotowane, żeby z dużym prawdopodobieństwem spowodować podanie złej odpowiedzi, albo przekroczenie limitu czasowego, co najmniej w jednym z zestawów. Ale podobnie jak zawodnicy, którzy użyli tego algorytmu do rozwiązania naszego zadania, nie mogli być pewni, czy otrzymają dobre wyniki, tak i organizatorzy, sprawdzając zadanie, nie mogli być pewni, czy wychwycą niepewne rozwiązanie.

Czwarty algorytm, wzorcowy, i bardzo elegancki, którego poprawność jest wysoce nieoczywista. Algorytm ten opisany jest w książce [23]. Ze względu na prostotę algorytmu podamy tutaj jego pełny kod pascalowy.

```
{ Zakładamy, że elementy ciągu podane są w tablicy A[1..n].
  Algorytm stwierdza, czy w tablicy A znajduje się lider. }
ile:=1; l:=A[1]; { l - kandydat na lidera }
for i:=2 to n do
```

```
if A[i]=l then ile:=ile+1
else if ile = 1  then l:=A[i]
                 else ile:=ile-1;
{ Po tej pętli, jeśli w A jest lider, to jest on równy l.
  Sprawdzamy więc, czy l jest liderem. }
ile:=0;
for i:=1 to n do if A[i]=l then ile:=ile+1;
if ile > n div 2 then { lider jest }
                 else { lidera nie ma }
```

Uwaga: aby użyć tego algorytmu do rozwiązania naszego zadania, w drugiej pętli należy dodać fragment sprawdzający warunek (2) z udowodnionego twierdzenia. Fragment ten pominęliśmy, aby nie popsuć przejrzystości tekstu.

Nie od razu widać, dlaczego ten algorytm miałby dawać poprawne odpowiedzi. Najlepiej się o tym przekonać próbując skonstruować kontrprzykład. Po kilku próbach okaże się to niemożliwe. Pełny dowód poprawności algorytmu jednak nie jest banalny. Przy okazji warto zauważyć, że o ile poprzednie algorytmy potrzebowały tablicy do zapamiętania wszystkich wartości ciągu, to tutaj możemy z tablicy zrezygnować i wczytywać na bieżąco wartości z pliku. Dwukrotne wczytanie wystarcza, żeby stwierdzić istnienie lidera. Algorytm ten ma zatem koszt czasowy liniowy, a pamięciowy stały.

Pozostaje jeszcze wyjaśnić, dlaczego tor o podanych własnościach nazwaliśmy lollobrygidą. Narciarze zapewne wiedzą, że taką nazwę nosi jedna z bardzo muldziastych tras narciarskich w Szczyrku. Dlaczego jednak trasa ta została nazwana nazwiskiem słynnej włoskiej aktorki, pozostawmy dociekliwości czytelników.

TESTY

Testy i czasy odcięcia zostały tak dobrane, aby możliwie jak najdokładniej oddzielić algorytmy liniowe od algorytmów o czasie działania rzędu $n \log n$ i wyeliminować programy błędne (np. zgadujące odpowiedź).

Każdy test składa się z 30 przypadków. Pierwsze 4 testy, to testy poprawnościowe, zawierające same proste przypadki ($n \leqslant 22$). Następne 6 testów, to testy wydajnościowe. Zawierają po kilka przypadków dużych danych ($n = 99999\frac{1}{2} \pm \frac{1}{2}$), pozostałe przypadki są podobne co do wielkości do pierwszych czterech testów.
Oto opisy poszczególnych testów:

- LOL1.IN — małe dane, n parzyste.
- LOL2.IN — małe dane, n parzyste, jeśli odpowiedzią ma być „TAK”, to żadne $\frac{n}{2}$ elementów się nie powtarza (tzn. nie ma „prawie lidera”), wówczas konstrukcja lollobrygidy może być dla niektórych algorytmów nieco trudniejsza.
- LOL3.IN — małe dane, n nieparzyste, nie ma lidera, o ile odpowiedzią ma być „TAK”.

- LOL4.IN — małe dane, n nieparzyste, w każdym przypadku istnieje lider, czyli jeśli odpowiedzią jest „TAK”, to wszystkie inne elementy są od niego albo mniejsze, albo większe.

W testach wydajnościowych występują przemieszane wszystkie 4 powyższe przypadki parzystości i istnienia, lub nie, lidera.

- LOL5.IN — 3 przypadki duże, 27 małych, dużo różnych wartości, kolejność elementów losowa.
- LOL6.IN — 3 przypadki duże, 27 małych, dużo różnych wartości, kolejność elementów rosnąca. Chodzi o wyeliminowanie pewnych nieefektywnych implementacji algorytmu Hoare'a lub Quicksortu, które w takim przypadku zachowują się kwadratowo.
- LOL7.IN — 3 przypadki duże, 27 małych, dużo różnych wartości, kolejność elementów malejąca.
- LOL8.IN — 3 przypadki duże, 27 małych, kolejność rosnąca, zarówno przypadki z dużą ilością różnych wartości, jak i z kilkunastoma wartościami powtarzającymi się wielokrotnie oraz z zaledwie trzema różnymi wartościami.
- LOL9.IN — jak wyżej, kolejność malejąca.
- LOL10.IN — 8 dużych przypadków, 22 małe, każdy z przypadków składa się jedynie z trzech różnych wartości, albo przemieszanych, albo posortowanych rosnąco lub malejąco.

Czasy odcięcia były dobrane tak, aby więcej niż dwukrotnie przewyższały czas działania wzorcowego rozwiązania, ale dla dużych testów powodowały odcięcie przy korzystaniu z procedur sortujących. Małe testy doczepione zostały do dużych aby wyeliminować algorytmy zgadujące odpowiedzi, bowiem jedynie rozwiązanie kompletu 30 przypadków zaliczało test.

Marcin Mucha
autor zadania, opracowanie

Paweł Wolff
program wzorcowy

Jajka

Wiadomo, że jajko zrzucone z wystarczająco dużej wysokości brzydko się rozbija. Dawniej wystarczyła wysokość jednego piętra, ale genetycznie podrasowane kury znoszą jajka nie tłukące się nawet po zrzuceniu z wysokości 100 000 000 pięter. Badania nad wytrzymałością jajek prowadzi się wykorzystując drapacze chmur. Opracowano specjalną skalę wytrzymałości jajek: jajko ma wytrzymałość k pięter, jeśli zrzucone z k-tego piętra nie rozbija się, ale zrzucone z $(k + 1)$-ego już tak. W przypadku gdy drapacz chmur, którym dysponujemy, ma n pięter, przyjmujemy, że jajko rozbija się, gdy zrzucimy je z $(n + 1)$-ego piętra. Przyjmujemy też, że każde jajko zrzucone z piętra o numerze 0 nie rozbija się.

Kierownik laboratorium postanowił wprowadzić oszczędności w procesie badawczym. Ograniczył on liczbę jajek, które wolno rozbić w trakcie eksperymentu mającego na celu ustalenie wytrzymałości jajek danego gatunku. Dodatkowo należy zminimalizować liczbę zrzutów jajek. Oznacza to, że mając do dyspozycji pewną liczbę jajek danego gatunku i drapacz chmur należy, w jak najmniejszej liczbie prób stwierdzić, jaka jest wytrzymałość jajek danego gatunku.

Zadanie

Twoim zadaniem jest napisanie modułu zawierającego trzy procedury (funkcje w przypadku języka C/C++):

- `nowy_eksperyment` — *ta procedura będzie wywoływana na początku każdego nowego eksperymentu (może być ich więcej niż jeden);*
- `daj_pytanie` — *ta procedura służy do zadawania pytania, czy jajko wytrzyma zrzucenie z określonego piętra, czy też rozbije się; w tym celu należy w opisanej dalej zmiennej globalnej* `pietro` *umieścić numer piętra, z którego ma być zrzucone jajko,*
- `analizuj_odpowiedz` — *ta procedura powinna odczytać wartość zmiennej globalnej* `odpowiedz`*, która zawiera odpowiedź na ostatnio zadane pytanie (opis tej zmiennej znajduje się w kolejnym paragrafie) i dokonać analizy tej odpowiedzi — jeśli w wyniku tej analizy zostanie określona wytrzymałość jajka, to procedura ta powinna ten fakt zasygnalizować poprzez nadanie odpowiednich wartości zmiennym globalnym* `x` *oraz* `wiem` *(szczegóły znajdują się w kolejnym paragrafie).*

Program nadzorujący przebieg eksperymentów wywoła napisaną przez Ciebie procedurę `nowy_eksperyment` *na początku każdego nowego eksperymentu, po czym cyklicznie będzie wykonywał następujące czynności:*

- *wywołanie procedury* `daj_pytanie`*,*
- *odpowiadanie na pytanie,*
- *wywołanie procedury* `analizuj_odpowiedz`*,*

aż do momentu, gdy Twój program stwierdzi, że zna wytrzymałość jajek gatunku używanego w danym eksperymencie (tzn. procedura `analizuj_odpowiedz` *umieści odpowiednią wartość w zmiennej globalnej* `wiem`*).*

Uwaga: *Nie zakładaj, że program nadzorujący przebieg eksperymentów faktycznie ustala pewną wytrzymałość jajka przed rozpoczęciem danego eksperymentu. Może on dobierać ją w trakcie trwania eksperymentu w taki sposób, aby pasowała do wszystkich wcześniej udzielonych odpowiedzi oraz aby zmusić Twój program do zadania jak największej liczby pytań. Tak więc, powinieneś dążyć do tego, aby liczba pytań, jakie Twój program będzie musiał zadać w najgorszym przypadku, była jak najmniejsza.*

Komunikacja

Komunikacja pomiędzy napisanym przez Ciebie modułem, a programem nadzorującym przebieg eksperymentów, odbywa się poprzez zmienne globalne.

Liczba pięter drapacza zapisana będzie w zmiennej globalnej `wysokosc` *typu* `Longint` *(w przypadku języka C/C++ jest to typ* `long int`*). Będzie to dodatnia liczba całkowita, nie większa niż 100 000 000.*

Maksymalna liczba jajek, które można stłuc w trakcie trwania eksperymentu zapisana będzie w zmiennej globalnej `jajka` *typu* `Integer` *(w przypadku języka C/C++ jest to typ* `int`*). Będzie to dodatnia liczba całkowita, nie większa niż 1000.*

Zadanie pytania, czy jajko wytrzyma zrzucenie z k-tego piętra, w procedurze `daj_pytanie` *polega na przypisaniu zmiennej globalnej* `pietro` *typu* `Longint` *(w przypadku języka C/C++ jest to typ* `long int`*) liczby k.*

Odpowiedź na zadane pytanie jest umieszczana w zmiennej globalnej `odpowiedz` *typu* `Boolean` *(w przypadku języka C/C++ jest to typ* `int`*). Twierdzącej odpowiedzi na zadane pytanie (tzn. jajko wytrzyma) odpowiada wartość* `TAK`*, a przeczącej (tzn. jajko rozbije się) odpowiada* `NIE`*, gdzie* `TAK` *i* `NIE` *są stałymi o wartościach, odpowiednio,* `true` *i* `false` *(w przypadku języka C/C++ są to makra o wartościach, odpowiednio,* `1` *i* `0`*).*

W przypadku znalezienia wytrzymałości jajka Twój program powinien w procedurze `analizuj_odpowiedz` *zapisać do zmiennej globalnej* `wiem` *typu* `Boolean` *(w przypadku języka C/C++ jest to typ* `int`*)* `TAK` *oraz w zmiennej globalnej* `x` *typu* `Longint` *(w przypadku języka C/C++ jest to typ* `long int`*) znalezioną wytrzymałość jajka.*

Katalogi i pliki

Programujący w Pascalu powinni przygotowywać swoje rozwiązanie w katalogu `JAJPAS`*, natomiast programujący w C i C++ w katalogu* `JAJC`*.*

W katalogu `JAJPAS` *(odpowiednio* `JAJC`*) znajdziesz następujące pliki:*

- `JAJmod.pas` *(odpowiednio* `JAJmod.h` *i* `JAJmod.c`*) — moduł zawierający definicje stałych* `TAK` *i* `NIE` *oraz deklaracje zmiennych globalnych,*
- `JAJ.pas` *— szkielet modułu znajdującego wytrzymałość jajka; powinieneś uzupełnić ten plik definicjami funkcji* `nowy_eksperyment`, `daj_pytanie` *i* `analizuj_odpowiedz` *(dla piszących w C lub C++ nagłówki powyższych funkcji znajdują się w pliku* `JAJ.h`*, musisz napisać plik* `JAJ.c` *lub* `JAJ.cpp` *z definicjami tych funkcji),*
- `test.pas` *(odpowiednio* `test.c`*) — przykładowy program nadzorujący przebieg eksperymentów i korzystający z Twojego modułu.*

Wyjście

Wynikiem Twojej pracy powinien być zapisany na dyskietce tylko jeden plik: `JAJ.pas`*,* `JAJ.c` *lub* `JAJ.cpp`*.*

ROZWIĄZANIE

Na początku eksperymentu wiemy, że wytrzymałość jajek danego gatunku mieści się w przedziale $[0, \texttt{wysokosc}]$. Jeśli na pewnym etapie eksperymentu wiemy, że wytrzymałość mieści się w przedziale $[a, b]$, to czego dowiemy się po zrzuceniu jajka z piętra o numerze p? Oczywiście będziemy wybierać $a < p \leqslant b$, gdyż dla innych wartości p jest jasne, czy jajko się rozbije, czy nie. W przypadku, gdy jajko się rozbije będziemy wiedzieli, że jego wytrzymałość zawiera się w przedziale $[a, p-1]$. Jeżeli natomiast jajko wytrzyma próbę, oznacza to, że jego wytrzymałość mieści się w przedziale $[p, b]$. Oczywiście eksperyment możemy zakończyć tylko wtedy, gdy długość przedziału, w jakim mieści się wytrzymałość jajek danego gatunku, wynosi 0. Zastanówmy się teraz, jak duży przedział wytrzymałości możemy sprawdzić mając do dyspozycji k jajek i n prób (tę maksymalną długość przedziału oznaczmy przez $T(n, k)$). Oczywiście mamy:

$$T(n, 0) = T(0, k) = 0, \text{ dla } n, k \geqslant 0,$$

bo nie mając żadnych jajek lub prób, niewiele możemy zdziałać.

Jeżeli chcemy znaleźć wytrzymałość jajek pewnego gatunku w n próbach mając do dyspozycji k jajek i wiemy, że ich wytrzymałość mieści się w przedziale $[a, b]$, to musimy zrzucić jajko z piętra o numerze nie większym niż $a + T(n-1, k-1) + 1$ (w przeciwnym razie, w przypadku rozbicia jajka, pozostaje nam $n-1$ prób i $k-1$ jajek, lecz wytrzymałość mieści się w przedziale, którego długość jest większa niż $T(n-1, k-1)$) oraz nie mniejszym niż $b - T(n-1, k)$ (w przeciwnym razie, w przypadku, gdy jajko wytrzyma próbę, pozostaje nam $n-1$ prób i k jajek, lecz wytrzymałość mieści się w przedziale o długości większej niż $T(n-1, k)$). Aby oba te warunki mogły być spełnione, długość przedziału $[a, b]$ nie może być większa od $T(n-1, k-1) + 1 + T(n-1, k)$. W takim przypadku, jajko można zrzucić z piętra o numerze $min(a + T(n-1, k-1) + 1, b)$, a zatem:

$$T(n, k) = T(n-1, k) + T(n-1, k-1) + 1, \text{ dla } n, k \geqslant 1.$$

Ta formuła bardzo przypomina poniższą:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

To sugeruje nam zbadanie związku liczb $T(n, k)$ ze współczynnikami dwumianowymi $\binom{n}{k}$. Łatwo sprawdzić (troszkę trudniej zgadnąć), że:

$$T(n, k) = \sum_{i=1}^{k} \binom{n}{i}, \text{ dla } n, k \geqslant 0.$$

Porównując zapisy różnych $T(n, k)$ łatwo otrzymujemy:

(1) $T(n+1, k) = 2T(n, k) + 1 - \binom{n}{k}$, dla $n, k \geqslant 0$;

(2) $T(n-1, k) = \frac{1}{2}(T(n, k) + \binom{n-1}{k} - 1)$, dla $n \geqslant 1, k \geqslant 0$;

(3) $T(n,k-1) = T(n,k) - \binom{n}{k}$, dla $n \geqslant 0, k \geqslant 1$.

Przy pomocy wzoru (1) możemy wyznaczyć najmniejsze n takie, że $T(n, \texttt{jajka}) \geqslant \texttt{wysokosc}$. Jest to, jak łatwo zauważyć, minimalna liczba prób jaka wystarcza do wyznaczenia wytrzymałości jajek w danym eksperymencie. Czas wyznaczenia liczby n jest oczywiście liniowy ze względu na tę liczbę. Przy pomocy wzorów (2), (3) możemy z kolei wyznaczyć w czasie stałym numer piętra, z którego należy zrzucić jajko (szczegóły można znaleźć w treści programu).

Jak liczyć symbole Newtona

Rozwiązanie wzorcowe wydaje się proste do zaprogramowania. Jest w nim jednak pewien haczyk. Na przykład, w jaki sposób obliczać $\binom{n}{k}$ znając $\binom{n-1}{k}$? Nie możemy po prostu pomnożyć przez n i podzielić przez $n-k$, bo przekroczymy zakres. Można jednak poradzić sobie z tym problemem stosując częściowe dzielenie. Wiemy, że wynik naszych obliczeń będzie liczbą całkowitą. Można więc w ułamku $\frac{n\binom{n-1}{k}}{n-k}$ skrócić najpierw lewą część ($\frac{n}{n-k}$), a potem prawą (szczegóły można znaleźć w treści programu).

Inne rozwiązania

Czy można uniknąć zgadywania tajemniczego wzoru na $T(n,k)$? Chyba tak. Możemy przecież wyliczać $T(n,k)$ z formuły rekurencyjnej:

$$T(n,k) = T(n-1,k) + T(n-1,k-1) + 1, \text{ dla } n,k \geqslant 1,$$

tzn. obliczać $T(n,0), T(n,1), \ldots, T(n, \texttt{jajka})$ dla kolejnych n, aż $T(n, \texttt{jajka}) \geqslant \texttt{wysokosc}$.

Następnie postępować tak, jak w rozwiązaniu wzorcowym, przy czym jeśli mamy policzone wartości

$$T(n,0), \ldots, T(n, \texttt{jajka}),$$

to

$$T(n-1,0), \ldots, T(n-1, \texttt{jajka})$$

możemy uzyskać z wzoru:

$$T(n-1,k) = T(n,k) - T(n-1,k-1) - 1, \text{ dla } k > 0 \text{ i } T(n-1,0) = 0$$

(oczywiście można pamiętać wszystkie wyliczone wartości i wtedy korzystanie z powyższego wzoru nie jest konieczne).

Jaka jest złożoność takiego algorytmu? Oczywiście $O(\texttt{jajka} \cdot n)$, gdzie n jest najmniejszą taką liczbą, że $T(n, \texttt{jajka}) \geqslant \texttt{wysokosc}$. Pozostaje pytanie: jak duże może być n? Można znaleźć dość dobre oszacowanie korzystając z mądrych wzorów, wystarczy jednak prosty rachunek, aby się przekonać, że iloczyn $n \cdot \texttt{jajka}$ nigdy nie jest zbyt duży.

Jeśli `jajka` = 1, to oczywiście n = `wysokosc`. Ten przypadek można jednak rozpatrywać osobno. Wystarczy rzucać jajko z kolejnych pięter.

Jeśli `jajka` = 2, to mamy:

$$T(n,2) = \binom{n}{1} + \binom{n}{2} = n + \frac{n(n-1)}{2} = \frac{n(n+1)}{2} > \frac{n^2}{2},$$

a więc wystarczy $n = \sqrt{2wysokosc} \leqslant 10000\sqrt{2} < 15000$, co daje łącznie `jajka` $\cdot\, n = 30000$ operacji.

Dla `jajka` $\geqslant 3$ mamy:

$$T(n,k) \geqslant \binom{n}{1} + \binom{n}{2} + \binom{n}{3} = n + \frac{n(n-1)}{2} + \frac{n(n-1)(n-2)}{6} =$$

$$= n + \frac{n(n-1)(n+1)}{6} = \frac{n(n^2+5)}{6} > \frac{n^3}{6},$$

a więc wystarczy $n = \sqrt[3]{6wysokosc} \leqslant \sqrt[3]{6000000000} \leqslant 900$ i łącznie najwyżej $900\cdot$`jajka` operacji. To już jest dobra złożoność, ale oczywiście można przeprowadzić podobny prosty rachunek dla `jajka` $\geqslant 4$ i otrzymać jeszcze lepsze oszacowanie.

Opis działania programu arbitra

Arbiter oszukuje

Arbiter wcale nie ustala na początku eksperymentu wytrzymałości jajka. Wczytuje natomiast z pliku testowego przedział $[min..max]$ wytrzymałości dopuszczalnych. Oznacza to z grubsza tyle, że jego odpowiedzi mają się zgadzać z pewną wytrzymałością z tego przedziału. Przykładowo, jeśli $min = max$, to arbiter faktycznie ustala na początku eksperymentu wytrzymałość jajka

Arbiter jest złośliwy

Arbiter nie ma wyboru przy udzielaniu odpowiedzi na pytanie, czy jajko zrzucone z k-tego piętra rozbije się, gdy $k \leqslant min$ lub $k > max$. W przeciwnym razie, jeśli któraś z odpowiedzi wymusi na module zawodnika zadanie większej liczby pytań niż jest to konieczne, to taka odpowiedź jest udzielana. Jeśli zaś przy obu odpowiedziach moduł zawodnika zachowuje szansę na wyznaczenie wytrzymałości jajek w odpowiedniej liczbie prób, to udzielana jest odpowiedź wymuszająca na module zawodnika zadanie co najmniej minimalnej wystarczającej liczby pytań. Jeśli obie odpowiedzi wymuszają zadanie co najmniej minimalnej wystarczającej liczby pytań, to udzielana jest odpowiedź losowa. Jeśli żadna odpowiedź nie ma tej właściwości (może się tak zdażyć wtedy, gdy liczby min i max różnią się mało, a zawodnik "wstrzeli" się w przedział $[min, max]$), to udzielana jest odpowiedź `TAK`.

Oczywiście jeśli arbiter odpowiedział, że jajko zbije się po zrzuceniu z piętra $k < max$, to od tego momentu $max = k - 1$. Analogicznie arbiter musi zwiększyć min, jeśli odpowiedział, że jajko się nie bije i $k > min$.

Arbiter wie wszystko

No dobrze, ale w jaki sposób można stwierdzić, czy po udzieleniu konkretnej odpowiedzi zawodnik będzie miał szansę na zmieszczenie się w minimalnej wystarczającej liczbie pytań? Można to zrobić, jeśli zna się wartości $T(n, k)$. Arbiter wylicza te wartości w taki sam sposób, jak moduł wzorcowy.

Diabelska strategia arbitra

Jak już zaznaczyliśmy na wstępie, przypadek $min = max$ jest wyjątkowo łagodny dla zawodnika. Na drugim biegunie leży przypadek $min = 0$, $max = wysokosc$. W tej sytuacji arbiter nie jest w żaden sposób ograniczony i gładko wymusza na module zawodnika, aby ten zadał możliwie najwięcej pytań. Ten przypadek będziemy nazywać diabelską strategią arbitra.

Czy arbiter gra w kości?

W niektórych przypadkach arbiter udziela odpowiedzi losowych. Czy to oznacza, że ten sam deterministyczny moduł uruchomiony dwa razy może uzyskać dwie różne liczby punktów? To zdecydowanie nie byłoby dobrze. Dlatego arbiter z pliku testowego oprócz wartości min i max wczytuje ciąg 100 losowych bitów, z którego korzysta przy udzielaniu odpowiedzi.

TESTY

Do sprawdzania rozwiązań zawodników użyto 11 testów, których opisy znajdują się poniżej.

- JAJ1.IN — małe wysokości drapacza, mało jajek, uczciwy arbiter,
- JAJ2.IN — średnie wysokości drapacza, różna liczba jajek, uczciwy arbiter,
- JAJ3.IN — małe wysokości drapacza, mało jajek, średniej trudności strategia udzielania odpowiedzi przez arbitra,
- JAJ4.IN — średnie wysokości drapacza, różna liczba jajek, średniej trudności strategia udzielania odpowiedzi przez arbitra,
- JAJ5.IN — duże wysokości drapacza, różna liczba jajek, średniej trudności strategia udzielania odpowiedzi przez arbitra,
- JAJ6.IN — małe wysokości drapacza, mało jajek, strategia diabelska,
- JAJ7.IN — średnie wysokości drapacza, różna liczba jajek, strategia diabelska,

- JAJ8.IN — duże wysokości drapacza, różna liczba jajek, strategia diabelska,
- JAJ9.IN — duże wysokości drapacza, różna liczba jajek, strategia diabelska,
- JAJ10.IN — duże wysokości drapacza, różna liczba jajek, strategia diabelska.

Grzegorz Jakacki
autor zadania

Marcin Mucha
opracowanie, program wzorcowy

Po-łamana

W prostokątnym układzie współrzędnych każdy punkt o współrzędnych całkowitych nazywamy **p-punktem***.*

Dowolny odcinek równoległy do jednej z osi współrzędnych, o różnych końcach będących **p-punktami***, nazywamy* **p-odcinkiem***. Rozważamy odcinki domknięte, tzn. końce należą do odcinka. Łamaną zbudowaną z k* **p-odcinków***, z których każde kolejne dwa są prostopadłe, nazywamy* **po-łamaną** *stopnia k.*

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `POL.IN` *opisy pewnej liczby p-odcinków oraz współrzędne dwóch różnych p-punktów A i B,*
- *wyznaczy minimalny stopień po-łamanej łączącej te dwa punkty i nie przecinającej żadnego z danych p-odcinków lub stwierdzi, że taka po-łamana nie istnieje,*
- *zapisze wynik do pliku tekstoweg* `POL.OUT`*.*

Wejście

Opis p-punktu składa się z dwóch nieujemnych liczb całkowitych oddzielonych pojedynczym odstępem, będących odpowiednio współrzędnymi x i y tego p-punktu. Liczby te należą do przedziału [0..1 000 000 000]. W pierwszym wierszu pliku wejściowego `POL.IN` *znajduje się tylko opis p-punktu A. W drugim wierszu znajduje się tylko opis p-punktu B. W trzecim wierszu zapisana jest dokładnie jedna nieujemna liczba całkowita n będąca liczbą p-odcinków, $1 \leqslant n \leqslant 50$. W każdym z kolejnych n wierszy znajdują się opisy dokładnie dwóch p-punktów, oddzielone pojedynczym odstępem. Są to współrzędne końców jednego p-odcinka.*

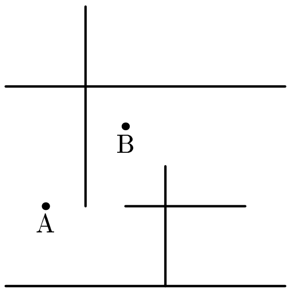
Wyjście

W pierwszym i jedynym wierszu pliku tekstowego `POL.OUT` *powinna znaleźć się jedna liczba będąca minimalnym stopniem po-łamanej łączącej punkty A i B oraz nie przecinającej żadnego z zadanych p-odcinków, lub słowo* `BRAK`*, jeśli po-łamana o powyższych własnościach nie istnieje.*

Przykład

Dla pliku wejściowego `POL.IN`*:*

```
1 2
3 4
5
0 0 7 0
0 5 7 5
2 2 2 7
4 0 4 3
3 2 6 2
```



poprawną odpowiedzią jest plik wyjściowy `POL.OUT`*:*

```
5
```

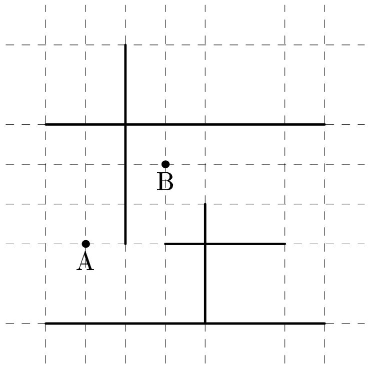
ROZWIĄZANIE WZORCOWE

Niech $x_1 < x_2 < \ldots$ będzie rosnąco uporządkowanym ciągiem (bez powtórzeń) współrzędnych x wszystkich końców odcinków z dodatkowo dodaną współrzędną x punktu B. Niech $y_1 < y_2 < \ldots$ będzie analogicznym ciągiem dla współrzędnej y. Rozważmy podział osi $X = (-\infty, \infty) = \bigcup_i X_i$, gdzie

$$X_1 = (-\infty, x_1), X_2 = \{x_1\}, X_3 = (x_1, x_2), X_4 = \{x_2\} \ldots$$

Analogicznie definiujemy podział osi Y. Podziały osi wyznaczają podział płaszczyzny na prostokąty otwarte, odcinki otwarte (być może nieograniczone) i punkty.

Rys. 1 Podział płaszczyzny dla przykładowego zestawu p-odcinków



Niech P_1 i P_2 będą dwoma elementami powyższego podziału. Jeśli z pewnego punktu $p \in P_1$ istnieje łamana o ustalonym stopniu i końcu w P_2, to z każdego innego punktu P_1 także. W szczególności za P_2 można przyjąć punkt B.

Oznacza to, że punkty w jednym elemencie podziału zachowują się identycznie, jeśli chodzi o istnienie łamanych o końcu w punkcie B. Rozwiązanie w zasadzie się narzuca. Trzeba skonstruować graf, w którym wierzchołkami są elementy podziału (ma on strukturę kraty, co ułatwia implementację), krawędzie zaś oznaczają możliwość przejścia z jednego elementu do innego. Następnie w tym grafie używamy procedury przeszukiwania wszerz (BFS), aby znaleźć łamaną minimalnego stopnia łączącą element zawierający punkt A z punktem B. Musimy jednak w procedurze BFS dokonać dwu modyfikacji:

(1) oddzielnie rozpatrywać wejścia pionowe i poziome, ponieważ ze względu na dalszą trasę nie są to sytuacje jednakowe,

(2) po dojściu do wierzchołka "pionowo" przechodzimy nie do jego najbliższych sąsiadów, ale do wszystkich wierzchołków, do których można z niego dojść poziomo (przypadek dojścia poziomego jest analogiczny) — w ten sposób algorytm ma złożoność przeszukiwania BFS, a pierwsze dojście do punktu B od razu daje odpowiedź na postawione w zadaniu pytanie.

TESTY

Do sprawdzenia rozwiązań zawodników użyto 12 testów:

- POL0.IN — test z treści zadania,
- POL1.IN — spirala o 10 krawędziach,
- POL2.IN — spirala o 20 krawędziach,
- POL3.IN — spirala o 30 krawędziach,
- POL4.IN — spirala o 40 krawędziach,
- POL5.IN — spirala o 45 krawędziach,
- POL6.IN — spirala o 50 krawędziach,
- POL7.IN — test poprawnościowy,
- POL8.IN — test wydajnościowy o 20 krawędziach,
- POL9.IN — test wydajnościowy o 30 krawędziach,
- POL10.IN — test wydajnościowy o 40 krawędziach,
- POL11.IN — test wydajnościowy o 50 krawędziach.

Tomasz Waleń — autor zadania, opracowanie
Marcin Mucha — program wzorcowy

Agenci

W związku z ostatnimi wpadkami swoich agentów, Urząd Ochrony Bajtocji postanowił usprawnić działalność.

Największym dotychczasowym problemem było bezpieczne urządzanie spotkań agentów. Twój program ma pomóc w rozwiązaniu tego problemu. Dla podanego opisu sieci dróg Bajtocji oraz początkowej pozycji dwóch agentów, powinien stwierdzać, czy możliwe jest bezpieczne spotkanie tych agentów.

Żeby spotkanie uznać za bezpieczne agenci muszą przestrzegać następujących reguł:

- *agenci poruszają się w dzień, natomiast spotkania odbywają się wieczorami,*
- *każdego dnia agent musi zmienić miejsce pobytu,*
- *agenci mogą poruszać się jedynie po drogach łączących miasta (niestety dodatkowym utrudnieniem jest fakt, iż w Bajtocji drogi są jednokierunkowe),*
- *agent nie może jednak poruszać się zbyt szybko (mogło by to wzbudzać niepotrzebne zainteresowanie) — jednego dnia nie może przemieścić się dalej niż do sąsiedniego miasta,*
- *odległość między dwoma miastami połączonymi drogą jest na tyle mała, że agent wyruszający z pierwszego miasta zawsze dotrze do drugiego miasta przed wieczorem,*
- *spotkanie uznaje się za odbyte w momencie, gdy dwaj agenci znajdą się tego samego wieczora w tym samym mieście.*

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `AGE.IN` *liczbę miast i opis sieci dróg Bajtocji, oraz pozycje początkowe dwóch agentów,*
- *stwierdzi, czy możliwe jest ich bezpieczne spotkanie, a jeśli tak, to po ilu dniach,*
- *zapisze wynik w pliku tekstowym* `AGE.OUT`.

Wejście

W pierwszym wierszu pliku tekstowego `AGE.IN` *znajdują się dwie liczby całkowite n i m, oddzielone pojedynczym odstępem, gdzie $1 \leqslant n \leqslant 250$, $0 \leqslant m \leqslant n \cdot (n-1)$.*

W drugim wierszu znajdują się dwie liczby całkowite a_1 i a_2 oddzielone pojedynczym odstępem, $1 \leqslant a_1, a_2 \leqslant n$ oraz $a_1 \neq a_2$, oznaczające, odpowiednio, początkowe pozycje agentów nr 1 i nr 2.

W m następnych wierszach znajdują się pary liczb naturalnych a i b oddzielone pojedynczymi odstępami, $1 \leqslant a, b \leqslant n$ oraz $a \neq b$, oznaczające istnienie drogi z miasta a do miasta b.

Wyjście

Plik tekstowy `AGE.OUT` *powinien zawierać dokładnie 1 wiersz zawierający:*

- *dokładnie jedną dodatnią liczbę całkowitą t, oznaczającą minimalny czas (w dniach) potrzebny do zorganizowania bezpiecznego spotkania dwóch agentów – jeżeli do takiego spotkania można doprowadzić,*
- *słowo* `NIE`*, gdy nie można doprowadzić do bezpiecznego spotkania.*

Przykład

Dla pliku wejściowego `AGE.IN`*:*

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

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

```
3
```

ROZWIĄZANIE

Sieć dróg Bajtocji reprezentujemy przez graf G o n wierzchołkach (miastach) i m krawędziach (drogach). Pierwszym, najbardziej narzucającym się rozwiązaniem, jest symulacja poruszania się obu agentów. Oznaczmy przez $s = (i, j)$ stan obliczeń oznaczający, iż w tym samym czasie pierwszy agent może znajdować się w mieście o numerze i, a drugi w mieście o numerze j. Teraz rozważmy graf G' zbudowany na stanach s, ma on n^2 wierzchołków i m^2 krawędzi. W przypadku posługiwania się grafem G' problem sprowadza się do znalezienia najkrótszej ścieżki z wyróżnionego wierzchołka $s_0 = (a_1, a_2)$ do dowolnego wierzchołka postaci (i, i). Ponieważ wszystkie krawędzie mają wagę 1, więc wystarczy przejść graf G' wszerz. Nie trzeba konstruować grafu G' wprost, można także, jak w poniższym przykładzie, dynamicznie generować jego krawędzie.

Schemat algorytmu (Q i $Q2$ oznaczają kolejki typu "pierwszy wchodzi-pierwszy wychodzi", czyli FIFO):

```
Q:={a_1,a_2};
{ początkowo, dla dowolnego i, j: v[i,j]=0 }
v[a_1,a_2]:=1;
```

```
t:=0;
while not Q.pusta do begin
  Q2:=pusta_kolejka;
  while not Q.pusta do begin
    (u,v):=Q.PobierzElement;
    if (u=v) then
        "Znaleziono rozwiązanie — t"
        STOP;
    for i∈ {k : (u,k) jest krawędzią w G} do
      for j∈ {k : (v,k) jest krawędzią w G} do
        if v[i,j]=0 then begin
          v[i,j]:=1;
          Q2.Dodaj(i,j);
        end
  end;
  t:=t+1;
  Q:=Q2;
end;
"Brak rozwiązania"
```

Teraz zastanówmy się jaką złożoność ma powyższy algorytm. Oznaczmy przez $K(i,j)$ koszt jaki trzeba ponieść aby wygenerować następniki stanu (i,j). Wówczas $K(i,j) = d_i \cdot d_j$, gdzie d_l oznacza liczbę krawędzi wychodzących z wierzchołka l. Ponieważ w pesymistycznym przypadku trzeba będzie rozpatrzyć wszystkie stany, całkowity koszt może wynosić:

$$n^2 + \sum_{i=1..n} \sum_{j=1..n} K(i,j) = n^2 + \sum_{i=1..n} \sum_{j=1..n} d_i \cdot d_j = n^2 + \sum_{i=1..n} \left(d_i \cdot \sum_{j=1..n} d_j \right) =$$

$$= n^2 + \sum_{i=1..n} (d_i \cdot m) = n^2 + m \cdot \sum_{i=1..n} d_i = n^2 + m^2$$

Łączny koszt to $O(n^2 + m^2)$.

Rozwiązanie wzorcowe polega na pewnym ulepszeniu poprzedniego rozwiązania. W poprzednim rozwiązaniu jedna faza algorytmu polegała na wykonaniu ruchu oboma agentami. Tym razem podzielimy tą fazę na dwie części: ruch pierwszym agentem i ruch drugim agentem. Co prawda takie rozwiązanie dwukrotnie powiększa przestrzeń stanów: $s = (i,j)$ (ten sam stan co w poprzednim rozwiązaniu), oraz dochodzą nowe stany $s' = (i',j')$, oznaczające, że pierwszy agent wykonał już swój ruch i znajduje się na pozycji i', natomiast drugi powinien teraz wykonać ruch, a na razie znajduje się na pozycji j'. Jednak dzięki tej metodzie znajdowanie następników stanów jest nieco mniej kosztowne, dla stanów $s = (i,j)$ potrzeba d_i operacji, natomiast dla stanów $s' = (i',j')$ potrzeba $d_{j'}$ operacji. Tak jak w poprzednim algorytmie, dotarcie do stanu $s = (i,i)$ oznacza, że agenci mogą zorganizować bezpieczne spotkanie.

Teraz już możemy naszkicować schemat algorytmu (Q i $Q2$ oznaczają kolejki typu "pierwszy wchodzi-pierwszy wychodzi", czyli FIFO):

```
Q:={a_1,a_2};
{ początkowo, dla dowolnego i, j: v[i,j]=0, oraz v'[i,j]=0 }
v[a_1,a_2]:=1;
t:=0;
while not Q.pusta do begin
  Q2:=pusta_kolejka;
  {ruch pierwszego agenta}
  while not Q.pusta do begin
    (u,v):=Q.PobierzElement;
    if (u=v) then
        "Znaleziono rozwiązanie — t"
        STOP;
    for i∈ {k : (u, k) jest krawędzią w G} do
      if v'[i,v]=0 then begin
        v'[i,v]:=1;
        Q2.Dodaj(i,v)
      end
  end;
  {ruch drugiego agenta}
  while not Q2.pusta do begin
    (u,v):=Q2.PobierzElement;
    for i∈ {k : (v, k) jest krawędzią w G} do
      if v[u,i]=0 then begin
        v[u,i]:=1;
        Q.Dodaj(i,v)
      end
  end;
  t:=t+1
end;
"Brak rozwiązania"
```

Analiza złożoności

Niech $K(i,j)$ oznacza koszt poniesiony na znalezienie następników stanu $s = (i,j)$. Tak więc $K(i,j) = d_i$. Analogicznie $K'(i,j)$ oznacza koszt dla stanów typu $s' = (i', j')$, a zatem $K'(i,j) = d_j$. W pesymistycznym przypadku łączny koszt całego algorytmu wynosi:

$$2n^2 + \sum_{i=1..n}\sum_{j=1..n} K(i,j) + \sum_{i=1..n}\sum_{j=1..n} K'(i,j) = 2n^2 + \sum_{i=1..n}\sum_{j=1..n} d_i + \sum_{i=1..n}\sum_{j=1..n} d_j =$$

$$= 2n^2 + \sum_{i=1..n} n \cdot d_i + \sum_{i=1..n} m = 2n^2 + n \cdot \sum_{i=1..n} d_i + n \cdot m = 2n^2 + 2nm$$

Więc to rozwiązanie ma złożoność $O(nm)$, co daje w pesymistycznym przypadku (dla grafów mających $O(n^2)$ krawędzi) $O(n^3)$ operacji.

Inne rozwiązania

Innym rozwiązaniem, mającym złożoność $O(dm)$, gdzie d oznacza minimalny czas potrzebny do zorganizowania bezpiecznego spotkania (lub n^2, jeśli takiego spotkania nie można zorganizować), jest obliczanie do jakich miast mogą dojść w i-tym kroku agenci nr 1 i 2. Jeśli te dwa zbiory mają część wspólną, oznacza to możliwość spotkania w i-tym dniu.

Niestety w niektórych przypadkach czas potrzebny do zoorganizowania spotkania może być bardzo duży, nawet rzędu n^2. Tak więc w pesymistycznych przypadkach to rozwiązanie jest równie wolne jak pierwsze.

(Zmienne $V1$ i $V2$ oznaczają zbiory)

```
V1:={a_1};
V2:={a_2};
dl:=0;
while (not V1.pusty and not V2.pusty and dl < n^2) do
begin
  dl:=dl+1;
  jeśli istnieje wierzchołek v należący do V1 i V2
      to "Znaleziono rozwiązanie";
   V1:=wierzchołki v takie, że istnieje wierzchołek u ∈ V1
        i (u,v) należy do G;
   V2:=wierzchołki v takie, że istnieje wierzchołek u ∈ V2
        i (u,v) należy do G
end;
"Brak rozwiązania"
```

TESTY

Do sprawdzania rozwiązań zawodników użyto 16 testów:

Testy oznaczone $K_2(b_1, b_2, n)$ oznaczają testy składające się z dwóch pełnych grafów skierowanych (o rozmiarach b_1, b_2) połączonych ścieżką długości n

Testy oznaczone $E(n_1, n_2, b_1, b_2)$ oznaczają testy składające się z dwóch pełnych grafów skierowanych (o rozmiarach b_1, b_2) połączone dwoma ścieżkami o wspólnym końcu, różniącymi się długością o 1. Na ścieżkach tych występują dodatkowo cykle długości n_1, n_2.

- AGE0.IN — test z treści zadania,
- AGE1.IN — mały graf z odpowiedzią negatywną,
- AGE2.IN — mały graf z odpowiedzią pozytywną,
- AGE3.IN — $K_2(30, 30, 10)$,
- AGE4.IN — $K_2(50, 50, 50)$,
- AGE5.IN — $K_2(100, 100, 50)$,
- AGE6.IN — $K_2(75, 75, 100)$,
- AGE7.IN — $K_2(50, 50, 150)$,
- AGE8.IN — $E(51, 61, 134, 0)$,
- AGE9.IN — $E(53, 59, 124, 10)$,
- AGE10.IN — $E(51, 59, 109, 20)$,
- AGE11.IN — $E(53, 64, 80, 30)$,
- AGE12.IN — duży graf dwudzielny, $n = 180$, z odpowiedzią negatywną,
- AGE13.IN — duży graf dwudzielny, $n = 250$, agenci znajdują się w tej samej części grafu,
- AGE14.IN — duży graf dwudzielny, $n = 150$, dołączone przejście gwarantujące rozwiązanie,
- AGE15.IN — duży graf dwudzielny, $n = 250$, z odpowiedzią negatywną.

Testy 11 i 12 oraz 14 i 15 zostały zgrupowane.

Adam Malinowski
autor zadania, opracowanie

Tomasz Waleń
program wzorcowy

Powtórzenia

Dany jest ciąg słów nad alfabetem $\{'a', \ldots, 'z'\}$. Należy znaleźć długość najdłuższego słowa występującego jako spójny fragment w każdym z danych słów.

Zadanie

Napisz program, który:

- *wczyta ciąg słów z pliku tekstowego* `POW.IN`*,*
- *obliczy długość najdłuższego słowa występującego jako spójny fragment w każdym z podanych słów,*
- *zapisze wynik w pliku tekstowym* `POW.OUT`*.*

Wejście

W pierwszym wierszu pliku tekstowego `POW.IN` *zapisano liczbę n, gdzie $1 \leqslant n \leqslant 5$, oznaczającą liczbę słów. W każdym z n kolejnych wierszy znajduje się jedno słowo utworzone z małych liter alfabetu angielskiego $'a', \ldots, 'z'$. Każde ze słów ma długość przynajmniej 1, ale nie większą niż 2 000.*

Wyjście

Plik tekstowy `POW.OUT` *powinien zawierać dokładnie jeden wiersz zawierający pojedynczą liczbę całkowitą równą długości najdłuższego słowa występującego jako spójny fragment w każdym z danych słów.*

Przykład

Dla pliku wejściowego `POW.IN`*:*

```
3
abcb
bca
acbc
```

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

```
2
```

ROZWIĄZANIE

Do rozwiązania zadania posłuży nam adaptacja struktury danych zwanej *słownikiem podsłów bazowych*. Słownik podsłów bazowych to z grubsza rzecz biorąc tablica *nazw* wszystkich podsłów danego słowa o długościach będących potęgami dwójki, przy czym takie same podsłowa mają te same nazwy, a różne podsłowa o tej samej długości — różne nazwy (nazwa to po prostu liczba całkowita z zakresu od 1 do długości słowa; $\texttt{num}[i,k]$ to nazwa podsłowa o długości 2^k zaczynającego się na pozycji i w danym słowie). Taka informacja umożliwia nam sprawdzenie w czasie stałym, czy dwa podsłowa tej samej długości są jednakowe, nawet jeśli ich długości nie są potęgami dwójki: nazwą identyfikującą jednoznacznie podsłowo długości r ($2^k < r < 2^{k+1}$) zaczynające się na pozycji i jest para $\langle \texttt{num}[i,k], \texttt{num}[i+r-2^k+1,k] \rangle$. Rozmiar tej struktury danych, to oczywiście $\Theta(d)$, gdzie d jest długością słowa, i można ją skonstruować w takim właśnie czasie *metodą wstępującą* (ang. bottom-up):

- Najpierw, na podstawie kodów poszczególnych znaków, nadajemy nazwy słowom jednoliterowym.
- Jeśli nadaliśmy już nazwy wszystkim podsłowom długości 2^k (czyli wypełniliśmy wiersz $\texttt{num}[k,*]$ w naszej tablicy), to teraz podsłowu długości 2^{k+1} zaczynającemu się na pozycji i, gdzie $1 \leqslant i \leqslant d - 2^{k+1} + 1$, nadajemy tymczasową nazwę $\langle \texttt{num}[i,k], \texttt{num}[i+2^k,k] \rangle$. Jest to dobra, unikalna nazwa — tyle, że nie jest liczbą z zakresu od 1 do d, lecz parą takich liczb.
- Zamieniamy teraz nazwy będące parami na nazwy liczbowe. W tym celu sortujemy wszystkie pary leksykograficznie (kubełkowo, zaczynając od mniej znaczącej współrzędnej, zob. [7], [13]) i wykonujemy przenumerowanie (takie same wektory będą po posortowaniu leżały koło siebie). Nazwy po przenumerowaniu umieszczamy w kolejnych kolumnach tablicy num (oczywiście wraz z każdym wektorem musimy przechowywać numer pozycji początku odpowiadającego mu podsłowa).

Czas wypełniania każdego wiersza jest liniowy, a liczba wierszy do wypełnienia to $\lceil \log d \rceil$. Szczegóły konstrukcji można znaleźć w książce [10].

W rozwiązaniu wzorcowym obliczamy *wspólny* słownik dla wszystkich zadanych słów, sklejonych w jedno długie słowo z separatorami (specjalnymi znakami występującymi tylko jednokrotnie) na granicach słów wejściowych. Separatory wprowadzamy po to, żeby uniknąć odrębnego analizowania podsłów „przechodzących przez granice" słów wejściowych. Żeby zaoszczędzić pamięć, przechowujemy tylko jeden, ostatnio obliczony wiersz tablicy num. Schemat postępowania wygląda następująco:

- Obliczamy nazwy podsłów o długościach będących kolejnymi potęgami dwójki. Robimy to dopóty, dopóki istnieje podsłowo długości 2^k występujące w każdym ze słów wejściowych.
- Kiedy już wiemy, że najdłuższe podsłowo wspólne dla wszystkich słów wejściowych ma długość mieszczącą się w zakresie $2^k, \ldots, 2^{k+1}-1$, to jej dokładną wartość wyznaczamy metodą bisekcji w k krokach. (Zachodzi tu potrzeba wyznaczenia nazw dla podsłów o długości p na podstawie nazw dla podsłów o długości r, gdzie $r < p < 2r$. Robimy to, posługując się opisaną na początku sztuczką, czyli jako nazwę tymczasową biorąc parę nazw zachodzących na siebie podsłów długości r pokrywających podsłowo długości p.)

Inne rozwiązania

Przedstawione tu rozwiązanie nie jest optymalne. Postawiony w zadaniu problem można rozwiązać w czasie *liniowym* względem sumy długości słów wejściowych. Wymaga to jednak użycia dość wyrafinowanych struktur danych (np. *drzew sufiksowych*), raczej trudnych do zaimplementowania podczas pięciogodzinnej sesji. W dodatku stopień komplikacji tych struktur mógłby spowodować na tyle duże stałe narzuty czasowe, że przy określonych w zadaniu ograniczeniach na rozmiar danych wejściowych, rozwiązanie asymptotycznie optymalne byłoby przy testowaniu trudne do odróżnienia lub wręcz gorsze, od znacznie prostszego rozwiązania opisanego powyżej.

TESTY

Do sprawdzenia rozwiązań zawodników użyto 14 testów, niekiedy łączonych w grupy.

- POW0.IN — test z treści zadania,
- POW1.IN — prosty test poprawnościowy,
- POW2.IN — test poprawnościowy, słowa składające się z róznych liter,
- POW3.IN — test poprawnościowy, trzy identyczne słowa,
- POW4.IN — prosty test poprawnościowy, 4 krótkie słowa losowe,
- POW5.IN — test poprawnościowy, trzy kolejne słowa Fibonacciego,
- POW6.IN — test poprawnościowy, 1 krótkie i 4 długie słowa,
- POW7.IN — test wydajnościowy, 3 słowa o długości 100,
- POW8.IN — test wydajnościowy, 5 słów o długości 400,
- POW9.IN — test wydajnościowy, 3 słowa o długości 400,
- POW10.IN — test wydajnościowy, 4 słowa o długości 1000,
- POW11.IN — test wydajnościowy, 2 słowa o długości 1200,
- POW12.IN — test wydajnościowy, 5 słów o długości 2000,
- POW13.IN — test wydajnościowy, 5 słów o długości 2000.

Tomasz Waleń
autor zadania, opracowanie

Marcin Sawicki
program wzorcowy

Promocja

Wielka bajtocka sieć supermarketów poprosiła Cię o napisanie programu symulującego koszty właśnie przygotowywanej promocji.

Przygotowywana promocja ma mieć następujące zasady:

- *klient, który chce wziąć udział w promocji, wpisuje na zapłaconym przez siebie rachunku swoje dane i wrzuca go do specjalnej urny,*
- *pod koniec każdego dnia promocji z urny wyciągane są dwa rachunki:*
    - *najpierw wybierany jest rachunek opiewający na największą kwotę,*
    - *następnie wybierany jest rachunek opiewający na najmniejszą kwotę;*

  klient, który zapłacił największy rachunek otrzymuje nagrodę pieniężną równą różnicy pomiędzy wysokością jego rachunku, a wysokością rachunku opiewającego na najmniejszą kwotę,
- *aby uniknąć kilkukrotnych nagród za jeden zakup, oba wybrane wg reguł z poprzedniego punktu rachunki nie wracają już do urny, ale wszystkie pozostałe rachunki dalej biorą udział w promocji.*

Obroty supermarketu są bardzo duże, możesz więc założyć, że pod koniec każdego dnia, przed wyciągnięciem rachunków opiewających na największą i najmniejszą kwotę, w urnie znajdują się co najmniej 2 rachunki.

Twoim zadaniem jest obliczenie na podstawie informacji o wysokościach rachunków wrzucanych do urny w poszczególnych dniach promocji, jaki będzie łączny koszt nagród w całej promocji.

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `PRO.IN` *listę wysokości rachunków wrzucanych do urny w poszczególnych dniach promocji,*
- *obliczy łączny koszt nagród wypłacanych w kolejnych dniach promocji,*
- *zapisze wynik w pliku tekstowym* `PRO.OUT`*.*

Wejście

W pierwszym wierszu pliku tekstowego `PRO.IN` *znajduje się jedna dodatnia liczba całkowita n, gdzie $1 \leqslant n \leqslant 5000$, oznaczająca czas trwania promocji w dniach.*

W każdym z kolejnych n wierszy znajduje się ciąg nieujemnych liczb całkowitych pooddzielanych pojedynczymi odstępami. Liczby w $(i+1)$-szym wierszu pliku określają wysokości rachunków wrzuconych do urny w i-tym dniu promocji. Pierwsza w wierszu *liczba k, $0 \leqslant k \leqslant 10^5$, jest liczbą rachunków z danego dnia, a kolejne k liczb to dodatnie liczby całkowite będące wysokościami poszczególnych rachunków, każda z tych liczb jest nie większa niż 10^6.*

Łączna liczba rachunków wrzuconych do urny podczas całej promocji nie przekracza 10^6.

Wyjście

Plik tekstowy `PRO.OUT` *powinien zawierać dokładnie jedną liczbę całkowitą równą łącznemu kosztowi nagród wypłacanych podczas całej promocji.*

Przykład

Dla pliku wejściowego `PRO.IN`*:*

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

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

```
19
```

ROZWIĄZANIE

Przy rozwiązaniu tego zadania bardzo pomocna będzie struktura danych umożliwiająca wykonywanie następujących operacji:

`ADD(x)` – dodanie elementu `x` do struktury;

`MIN` – podanie wartości minimalnego elementu w strukturze;

`MAX` – podanie wartości maksymalnego element w strukturze;

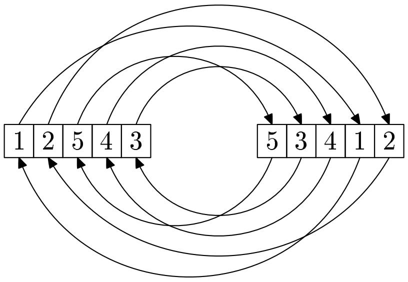
`ExtractMin` – usunięcie minimalnego elementu;

`ExtractMax` – usunięcie maksymalnego elementu.

Zwykły kopiec (zob. [13]) oferuje efektywną implementację trzech z tych operacji: `ADD`, `MIN`, `ExtractMin` (lub `ADD`, `MAX`, `ExtractMax`). Rozwiązaniem może być struktura składającą się z dwóch połączonych kopców, jeden z nich udostępnia operacje `ADD`, `MIN`, `ExtractMin`, natomiast drugi `ADD`, `MAX`, `ExtractMax`. W takim rozwiązaniu decydujemy się na pewną rozrzutność i każdy element należący do struktury będziemy pamiętać podwójnie: w pierwszym i drugim kopcu, dodatkowo z każdym elementem jest związany wskaźnik do bliźniaczego elementu w drugim kopcu. Operacje `MIN`, `MAX` odwołują się do odpowiednich kopców i wymagają stałego czasu. Operacja `ADD` wymaga dodania elementu do obu kopców, oraz uaktualnienia wskaźników, i tak jak w zwykłym kopcu wymaga czasu $O(logn)$. Operacje `ExtractMin`, `ExtractMax` sprowadzają się do wykonania odpowiedniej operacji na odpowiednim kopcu (tym który oferuje tę operację), oraz usunięciu elementu bliźniaczego z drugiego kopca (ponieważ pamiętamy wskaźnik do tego elementu, możemy wykonać i tę operację w czasie

$O(logn)$). Zatem całość również wymaga czasu $O(logn)$. Przy wszystkich modyfikacjach, należy zwracać baczną uwagę na uaktualnianie wskaźników między oboma kopcami.

Rys. 1 Przykład podwójnego kopca



Rozwiązanie 1

Mając do dyspozycji powyższą stukturę danych, schemat algorytmu jest już oczywisty. Dla każdego dnia dodajemy do struktury rachunki, a następnie wyjmujemy z niej jeden największy i jeden najmniejszy, oraz odpowiednio zwiększamy całkowity koszt promocji.

Niestety takie rozwiązanie, ze względu na bardzo dużą liczbę rachunków, które mogły pojawić się w promocji, nie jest akceptowalne ze względu na ogranicznia pamięciowe. Należy zastanowić się, jakie rachunki na pewno nie mają szans na wyjęcie z urny. Można spostrzec, że gdy mamy w strukturze więcej niż $2n$ rachunków, to szanse na wyciągnięcie z urny ma jedynie n największych i n najmniejszych, natomiast wszystkie pozostałe, "środkowe" rachunki, nie mają już takiej szansy i można je pominąć. To spostrzeżenie prowadzi już do ulepszonego rozwiązania, które będzie wymagać jedynie pamięci rzędu $O(n)$.

Rozwiązanie 2

- Dopóki liczba rachunków w strukturze S nie przekracza $2n$ stosuj rozwiązanie 1.
- Jeśli liczba rachunków osiągnie $2n$, przepisz n najmiejszych do struktury S_{min}, natomiast n największych do struktury S_{max}. Strukturę S można już usunąć, natomiast wszystkie dalsze operację będą wykonywane na S_{min} i S_{max}. Dodawanie nowego rachunku x wygląda następująco:
  - jeśli $x < S_{min}.MAX$, to usuwamy z S_{min} element $S_{min}.MAX$ (operacja $S_{min}.ExtractMax$) i dodajemy x do S_{min};

- jeśli $x > S_{max}.MIN$, to postępujemy symetrycznie jak w poprzednim przypadku (zamieniając MAX na MIN i S_{min} na S_{max});
- w przeciwnym przypadku możemy zapomnieć o tym rachunku.

Chcąc wydobyć rachunek o najmniejszej wartości wykonujemy operację $ExtractMin$ na S_{min} i analogicznie, chcąc otrzymać rachunek o największej wartości wykonujemy operację $ExtractMax$ na S_{max}.

TESTY

Do sprawdzania rozwiązań zawodników użyto 10 testów opisanych poniżej:

- PRO1-6.IN — proste testy poprawnościowe,
- PRO7.IN — trudny test wydajnościowy, $n = 5000$, $K = 100006$,
- PRO8.IN — prosty test, prawie każdego dnia dwa rachunki, $n = 5000$, $K = 210004$,
- PRO9.IN — duży test wydajnościowy, $n = 5000$, $K = 1000000$,
- PRO10.IN — duży, losowy test wydajnościowy, $n = 5000$, $K = 999800$.

XI Międzynarodowa Olimpiada Informatyczna IOI'99, Antayla-Belek Turcja, październik 1999

treści zadań

Krzysztof Diks, Marcin Kubica
przekład

Kwiaciarnia

Jesteś właścicielem kwiaciarni i przygotowujesz okno wystawowe. Dysponujesz F bukietami kwiatów – każdy innego rodzaju. Masz też do dyspozycji co najmniej tyle samo wazonów, ustawionych w rzędzie na parapecie okna. Wazony są przymocowane na stałe do parapetu i ponumerowane kolejno od 1 do V, gdzie V jest liczbą wazonów. Skrajnie lewy wazon ma numer 1, a skrajnie prawy ma numer V. Bukiety są jednoznacznie ponumerowane od 1 do F. Te numery są ważne z następującego powodu – określają one kolejność występowania bukietów w wazonach. Dla $i < j$ bukiet nr i musi zawsze znajdować się w wazonie położonym na lewo od wazonu, w którym znajduje się bukiet nr j. Np., jeśli azalie mają nr 1, begonie mają nr 2, a cyprysy mają nr 3, to ich bukiety muszą znajdować się w wazonach właśnie w takim porządku – wazon z azaliami musi być na lewo od wazonu z begoniami, a ten na lewo od wazonu z cyprysami. Jeśli mamy więcej wazonów niż bukietów, to nadmiarowe wazony pozostają puste. W każdym wazonie może znajdować się co najwyżej jeden bukiet kwiatów.

Wazony (tak jak bukiety kwiatów) mają swoje charakterystyki. Wkładając bukiet kwiatów do wazonu uzyskujemy określony efekt estetyczny, wyrażany liczbą całkowitą. W tabeli poniżej przedstawiono liczby wyrażające przykładowe efekty estetyczne. W przypadku gdy wazon jest pusty daje to efekt estetyczny równy 0.

Bukiety	**Wazony**				
	1	2	3	4	5
1 (azalie)	7	23	-5	-24	16
2 (begonie)	5	21	-4	-10	23
3 (cyprysy)	-21	5	-4	-20	20

Zgodnie z powyższą tabelą, azalie wyglądałyby wspaniale w wazonie nr 2, a strasznie w wazonie nr 4.

Dla osiągnięcia najlepszego efektu musisz umieścić bukiety w wazonach, zachowując podany porządek i maksymalizując sumę efektów estetycznych. Jeśli jest kilka takich rozmieszczeń, to każde z nich jest dopuszczalne. Musisz znaleźć jedno z nich.

ZAŁOŻENIA

- $1 \leqslant F \leqslant 100$, *gdzie F jest liczbą bukietów kwiatów. Bukiety są ponumerowane od 1 do F.*
- $F \leqslant V \leqslant 100$, *gdzie V jest liczbą wazonów.*
- $-50 \leqslant A_{ij} \leqslant 50$, *gdzie A_{ij} jest efektem estetycznym umieszczenia i-go bukietu w j-tym wazonie.*

Wejście

Nazwą pliku wejściowego jest: `flower.inp`.

- *Pierwszy wiersz zawiera dwie liczby: F, V.*
- *Kolejnych F wierszy ma następującą postać: każdy z nich zawiera V liczb całkowitych - liczba A_{ij} jest j-tą liczbą w $(i + 1)$-ym wierszu.*

Wyjście

Plik wyjściowy `flower.out` *musi być plikiem tekstowym i zawierać dwa wiersze:*

- *Pierwszy wiersz zawiera sumę efektów estetycznych twojego rozmieszczenia.*
- *Drugi wiersz musi zawierać to rozmieszczenie w postaci ciągu F liczb całkowitych, w którym k-ty element jest numerem wazonu zawierającego bukiet nr k.*

Przykład

`flower.inp`*:*

```
3 5
7 23 -5 -24 16
5 21 -4 10 23
-21 5 -4 -20 20
```

`flower.out`*:*

```
53
2 4 5
```

Ocena

Ograniczenie na czas działania programu wynosi 2 sekundy. Nie można otrzymać części punktów za pojedynczy test.

Krzysztof Diks, Marcin Kubica
przekład

Kody

Dany jest zbiór słów kodowych oraz tekst. Tekst zawiera komunikat złożony ze słów kodowych, umieszczonych w tekście w specjalny (być może niejednoznaczny) sposób.

Zarówno słowa kodowe, jak i tekst, są ciągami zbudowanymi tylko z wielkich i małych liter alfabetu angielskiego. Rozróżnienie wielkich i małych liter jest istotne. Długość słowa kodowego określa się w zwykły sposób. Na przykład słowo kodowe ALL ma długość 3.

Litery słowa kodowego nie musza występować w tekście kolejno po sobie. Np., słowo kodowe ALL zawsze występuje we fragmencie tekstu postaci AuLvL, gdzie u i v oznaczają dowolne (być może puste) ciągi kolejnych liter. O AuLvL mówimy, że jest pokryciem słowa ALL. W ogólności, pokryciem słowa kodowego nazywamy fragment tekstu, w którym pierwsza i ostatnia litera są takie same jak w słowie kodowym, a samo słowo możemy otrzymać przez usunięcie pewnych (być może żadnych) liter z tego fragmentu. Zauważmy, ze słowo kodowe może mieć wiele pokryć lub może ich nie mieć w ogóle. Podobnie, ten sam fragment tekstu może być pokryciem więcej niż jednego słowa kodowego.

Pokrycie opisujemy za pomocą pozycji jego początku (jego pierwszej litery) i końca (jego ostatniej litery) w tekście. (Pierwsza litera tekstu znajduje się na pozycji 1.) Mówimy, że pokrycia c_1 i c_2 nie zachodzą na siebie, gdy koniec c_2 znajduje się na wcześniejszej pozycji niż początek c_1, lub symetrycznie.

Aby odczytać ukryty w tekście komunikat masz znaleźć rozwiązanie. Rozwiązanie to zbiór elementów, z których każdy składa się ze słowa kodowego i jego pokrycia, spełniający następujące warunki:

(a) pokrycia nie zachodzą na siebie parami,

(b) długość żadnego pokrycia nie przekracza 1000,

(c) łączna długość wystąpień słów kodowych jest maksymalna (licząc każde wystąpienie słowa kodowego w elemencie).

Jeśli jest więcej niż jedno rozwiązanie, należy podać jedno z nich.

Założenia

- *$1 \leqslant N \leqslant 100$, gdzie N jest liczbą słów kodowych.*
- *Żadne słowo kodowe nie jest dłuższe niż 100.*
- *$1 \leqslant$ długość tekstu $\leqslant 1\,000\,000$.*

Mówimy, że pokrycie c słowa kodowego w jest prawostronnie minimalne jeśli żaden właściwy prefiks (tzn. jego początkowy fragment, krótszy od niego samego) pokrycia c nie jest pokryciem w. Np., `AAALAL` *jest prawostronnie minimalnym pokryciem słowa* `ALL`*, natomiast* `AAALALAL` *takim pokryciem nie jest. Tekst zawarty w danych wejściowych zawsze spełnia następujące warunki:*

(a) dla każdej pozycji w tekście liczba prawostronnie minimalnych pokryć, zawierających taką pozycję, nie przekracza 2500,

(b) liczba prawostronnie minimalnych pokryć nie przekracza 10,000.

Wejście

Dane wejściowe są zawarte w dwóch plikach tekstowych: `words.inp` *i* `text.inp`. *Plik* `words.inp` *zawiera listę słów kodowych, natomiast plik* `text.inp` *zawiera tekst wejściowy.*

- *Pierwszy wiersz pliku* `words.inp` *zawiera liczbę słów kodowych N. Każdy z kolejnych N wierszy zawiera po jednym słowie kodowym zapisanym jako ciąg liter, bez znaków odstępu między nimi. Słowa kodowe są ponumerowane od 1 do N zgodnie z porządkiem ich występowania w pliku* `words.inp`.
- *W pliku* `text.inp` *zapisano ciąg liter (zakończony znakami końca wiersza i końca pliku). Plik ten nie zawiera znaków odstępu.*

Zalecenia dla programujących w Pascalu:
Zaleca się deklarowanie plików wejściowych jako plików typu `text`, *w odróżnieniu od plików znakowych (*`file of char`*).*

Wyjście

Plikiem wyjściowym jest plik tekstowy `codes.out`.

- *W pierwszym wierszu należy zapisać łączną długość wystąpień słów kodowych obliczoną przez Twój program.*
- *Każdy z kolejnych wierszy powinien zawierać opis jednego elementu Twojego rozwiązania. Wiersz taki zawiera trzy liczby całkowite i, s, e, gdzie i jest numerem słowa kodowego mającego pokrycie zaczynające się na pozycji s i kończącego się na pozycji e. Kolejność opisów elementów może być dowolna.*

Przykład

`words.inp`:
```
4
RuN
RaBbit
HoBbit
StoP
```
`text.inp`:
```
StXRuYNvRuHoaBbvizXztNwRRuuNNP
```

`codes.out`:
```
12
2 9 21
1 4 7
1 24 28
```
(Uwaga: W tekście jest ukryty komunikat "`RuN RaBbit RuN`*". (Ukryty jest tam również komunikat "*`RuN HoBbit RuN`*"). Pamiętaj, żeby nie wypisywać treści komunikatu do pliku wyjściowego.)*

Ocena

Czas działania twojego programu nie może przekraczać 10 sekund. Nie można otrzymać części punktów za pojedynczy test.

Krzysztof Diks, Marcin Kubica
przekład

Podziemne miasto

Jesteś uwięziony w jednym z podziemnych miast Kapadocji. Błądząc w ciemnościach znalazłeś przypadkiem plan miasta. Niestety na planie nie jest zaznaczone miejsce, w którym jesteś. Badanie miasta pomoże Ci je znaleźć i na tym polega Twoje zadanie.

Plan jest prostokątna siatką jednostkowych kwadratów. Każdy kwadrat jest albo otwartym kawałkiem przestrzeni, krócej, jest otwarty i wtedy jest oznaczony literą `'O'`*, albo jest częścią muru i wtedy jest oznaczony literą* `'W'`*. Na planie jest zaznaczony kierunek północny. Na szczęście masz przy sobie kompas i możesz poprawnie zorientować swój plan. Na początku jesteś w kwadracie otwartym.*

Wszystko zaczyna się od wywołania bezargumentowej procedury (albo funkcji) start. Możesz badać miasto z pomocą procedur (lub funkcji) `look` *oraz* `move`*.*

Możesz zadawać pytania w postaci wywołania funkcji `look(dir)`*, gdzie* `dir` *oznacza kierunek, w którym spoglądasz, przedstawiony jako jedna z liter* `'N'`*,* `'S'`*,* `'E'` *oraz* `'W'`*, oznaczających odpowiednio północ, południe, wschód i zachód. Załóżmy teraz, iż wartością argumentu wywołania* `dir` *jest* `'N'`*. Odpowiedzią będzie litera* `'O'`*, jeśli kwadrat na północ od Ciebie jest otwarty, zaś* `'W'` *jeśli jest częścią muru. Podobnie można spoglądać w innych kierunkach i zbierać informacje o innych sąsiednich kwadratach.*

Możesz wejść na jeden z czterech sąsiednich kwadratów, wywołując `move(dir)`*, gdzie* `dir` *oznacza kierunek kroku wykonywanego w opisany wyżej sposób. Możesz przechodzić tylko na kwadraty otwarte. Próba wejścia na kwadrat będący częścią muru byłaby ciężkim błędem. Do każdego otwartego kwadratu w mieście można dojść z dowolnego innego otwartego kwadratu.*

Masz znaleźć, z pomocą najmniejszej możliwej liczby spojrzeń (wywołań procedury `look(dir)`*), położenie otwartego kwadratu, w którym znalazłeś plan. Zaraz po znalezieniu tego położenia musisz go przekazać z pomocą wywołania procedury* `finish(x,y)`*, gdzie* `x` *jest współrzędną poziomą (zachód-wschód), zaś* `y` *jest współrzędna pionowa (południe-północ) położenia.*

Założenia

- *$3 \leqslant U \leqslant 100$, gdzie U jest szerokością planu, to znaczy długością mierzoną liczbą kwadratów w kierunku poziomym (zachód-wschód).*
- *$3 \leqslant V \leqslant 100$, gdzie V jest wysokością planu, to znaczy długością mierzoną liczbą kwadratów w kierunku (południe-północ).*
- *Miasto jest otoczone murami, które są przedstawione na planie.*
- *Południowo-zachodni róg miasta ma współrzędne $(1,1)$, zaś północno-wschodni ma współrzędne (U,V).*

Wejście

Plikiem wejściowym jest plik tekstowy `under.inp`*.*

- *Pierwszy wiersz zawiera dwie liczby: U, V.*

- *Każdy z następnych V wierszy zawiera wiersz planu w kierunku poziomym. Każdy wiersz składa się z U znaków, a x-ty znak w $(V - y + 2)$-im wierszu pliku wejściowego podaje informacje o kwadracie planu, mającym współrzędne (x, y): jest to albo litera* `'W'` *oznaczająca mur, albo litera* `'O'` *oznaczająca otwarty kwadrat. Dane w tych wierszach nie są pooddzielane żadnymi odstępami.*

Wyjście

Nie generuje się żadnego pliku wyjściowego. Wynik znaleziony przez Twój program należy przekazać wywołując procedurę `finish(x,y)`*.*

Przykład

`under.inp`*:*

```
5 8
WWWWW
WWWOW
WWWOW
WOOOW
WOWOW
WOOWW
WWOOW
WWWWW
```

Możliwa interakcja, kończąca się poprawnym wywołaniem procedury `finish`*:*

```
Interaction:
Start()
look('N')
'W'
look('E')
'O'
move('E')
look('E')
'W'
finish(3,5)
```

Wytyczne dla programujących w Pascalu

Powinieneś mieć w swoim pliku źródłowym zapisane:

```
    uses undertpu;
```

Ten moduł zapewni Ci dostęp do:

```
procedure start;  należy wywołać w pierwszej kolejności
function look (dir:char):char;
procedure move (dir:char);
procedure finish (x,y:integer);  wywołać jako ostatnią
```

Wytyczne dla programujących w C/C++

Powinieneś mieć w swoim pliku źródłowym:

```
    #include under.h
```

Zapewni Ci to następujące deklaracje:

```
void start (void); /* należy wykonać w pierwszej kolejności*/
char look (char);
void move (char);
void finish (int,int); /* wykonać jako ostatnią */
```

Utwórz również projekt, nazwany `under`*, który powinien zawierać Twój program oraz bibliotekę służącą do interakcji, nazwana* `underobj.obj`*. Żeby utworzyć projekt, należy wybrać z menu* `project` *opcje* `open`*, a następnie za pomocą opcji* `add item` *dołączyć twój plik źródłowy (*`under.c` *lub* `under.cpp`*) oraz plik* `underobj.obj`*. W trakcie kompilacji korzystaj z modelu pamięci* `LARGE`*. (UWAGA: Jest to zmiana ustalenia podanego w Regulaminie Zawodów.)*

Ocena

Ograniczenie na czas działania programu wynosi 5 sekund.

Aby otrzymać maksymalną liczbę punktów A za test, liczba wywołań x funkcji `look` *nie może przekraczać ograniczenia M ustalonego przez program oceniający. Liczba M jest większa ($>$) od minimum. W szczególności, M nie zależy od tego, czy kolejność kierunków spoglądania jest zgodna, czy przeciwna do kierunku ruchu wskazówek zegara. Jeśli twój program wywoła funkcje* `look` *więcej niż ($>$) M razy, ale mniej niż ($<$) dwa razy M, to możesz otrzymać część punktów. W takim przypadku liczba punktów jest wyliczana zgodnie z następującą formułą (zaokrąglając do najbliższej liczby całkowitej):*

$$\begin{array}{ccc} A & \textit{gdy} & x \leqslant M \\ A(2M-x)/M & \textit{gdy} & M < x < 2M \\ 0 & \textit{gdy} & x \geqslant 2M \end{array}$$

Jeśli twój program zachowa się w sposób niedozwolony, to otrzymasz 0 punktów. Niedozwolone zachowania programu w tym zadaniu to:

- *wywołanie procedury lub funkcji bibliotecznej z niedozwolonym parametrem, np. ze znakiem nie oznaczającym kierunku,*
- *próba wejścia w ścianę,*
- *nie przestrzeganie wytycznych.*

Jak testować program

Utwórz plik tekstowy `place.txt` *zawierający miejsce znalezienia planu. Uruchom program. Obejrzyj wyniki znajdujące się w pliku* `result.txt`*.*

Plik `place.txt` *powinien zawierać dokładnie jeden wiersz, w którym zapisano współrzędne (poziomą i pionową) miejsca znalezienia planu. Musisz utworzyć własny plik wejściowy* `under.inp`*. Plik* `result.txt` *będzie składał się z dwóch wierszy. W pierwszym wierszu znajdą się argumenty x i y wywołanej przez Ciebie procedury* `finish (x,y)`*. Drugi wiersz będzie zawierał komunikat postaci* `''You used look nnn times''`*, co się tłumaczy na* `''Użyłeś look nnn razy''`*. Pamiętaj, że służy to tylko sprawdzeniu zgodności twojego programu z biblioteką. Nie ma to nic wspólnego z poprawnością Twojego rozwiązania.*

Krzysztof Diks, Marcin Kubica
przekład

Światła drogowe

W mieście Dingilville wprowadzono niezwykły sposób sterowania ruchem ulicznym. Są tam skrzyżowania i łączące je drogi. Pomiędzy dowolnymi dwoma skrzyżowaniami jest co najwyżej jedna droga. Żadna droga nie łączy skrzyżowania z nim samym. Czas przejazdu każdą drogą jest taki sam dla obu kierunków jazdy. Na każdym skrzyżowaniu jest jedno światło, które w każdym momencie jest albo niebieskie, albo purpurowe. Światło na każdym skrzyżowaniu zmienia się cyklicznie: niebieskie świeci przez pewien okres czasu, a potem purpurowe przez pewien okres czasu, itd. Wolno przejechać drogą łączącą dwa skrzyżowania wtedy i tylko wtedy, gdy w momencie ruszania światła na obu tych skrzyżowaniach mają taki sam kolor. Jeżeli pojazd przyjeżdża na skrzyżowanie dokładnie w chwili zmiany świateł na skrzyżowaniu(-ach), musisz przyjąć nowe kolory świateł. Pojazdy mogą czekać na skrzyżowaniach. Masz plan miasta pokazujący:

- *czasy przejazdu dla wszystkich dróg (liczby całkowite),*
- *czasy trwania kolorów świateł na każdym skrzyżowaniu (liczby całkowite),*
- *początkowy kolor światła i czas (liczba całkowita) pozostały do jego zmiany, dla każdego skrzyżowania.*

Masz znaleźć sposób przejazdu w najkrótszym czasie, od zadanego skrzyżowania początkowego do zadanego skrzyżowania końcowego, zaczynając w momencie rozpoczęcia ruchu. W przypadku, gdy istnieje wiele takich sposobów przejazdu, masz podać tylko jeden z nich.

Założenia

- $2 \leqslant N \leqslant 300$, *gdzie N jest liczbą skrzyżowań. Skrzyżowania są ponumerowane od 1 do N. Numery te identyfikują skrzyżowania.*
- $1 \leqslant M \leqslant 14\,000$, *gdzie M jest liczbą dróg.*
- $1 \leqslant l_{ij} \leqslant 100$, *gdzie l_{ij} jest czasem przejazdu drogą łączącą skrzyżowania i oraz j.*
- $1 \leqslant t_{ic} \leqslant 100$, *gdzie t_{ic} jest czasem trwania światła koloru c na skrzyżowaniu i. Wartością c może być B dla koloru niebieskiego i P dla purpurowego.*
- $1 \leqslant r_{ic} \leqslant t_{ic}$, *gdzie r_{ic} jest czasem pozostałym do zmiany początkowego koloru c światła na skrzyżowaniu i.*

Wejście

Plik wejściowy jest plikiem tekstowym o nazwie `lights.inp`.

- *Pierwszy wiersz zawiera dwie liczby: numer skrzyżowania początkowego i numer skrzyżowania końcowego.*

- *Drugi wiersz zawiera dwie liczby: N, M.*
- *Następne N wierszy zawiera informacje o N skrzyżowaniach. $(i + 2)$-gi wiersz pliku wejściowego zawiera informacje o skrzyżowaniu nr i: C_i, r_{ic}, t_{iB}, t_{iP}, gdzie C_i ma wartość 'B' lub 'P' oznaczającą początkowy kolor światła na skrzyżowaniu i.*
- *Dalsze M wierszy zawiera informacje o M drogach. Każdy wiersz ma postać: i, j, l_{ij}, gdzie i oraz j są numerami skrzyżowań, które dana droga łączy.*

Wyjście

Plik wyjściowy jest plikiem tekstowym o nazwie `lights.out`. *Jeśli istnieje sposób przejazdu:*

- *Pierwszy wiersz zawiera czas najkrótszego przejazdu od skrzyżowania początkowego do skrzyżowania końcowego.*
- *Drugi wiersz zawiera opis szukanego sposobu przejazdu - ciąg kolejnych skrzyżowań, przez które należy przejechać. W szczególności, pierwsza liczba w tym wierszu będzie numerem skrzyżowania początkowego, a ostatnia numerem skrzyżowania końcowego.*

Jeśli szukany sposób przejazdu nie istnieje:

- *Pojedynczy wiersz zawierający liczbę 0.*

Przykład

`lights.inp`*:*
```
1 4
4 5
B 2 16 99
P 6 32 13
P 2 87 4
P 38 96 49
1 2 4
1 3 40
2 3 75
2 4 76
3 4 77
```

`lights.out`*:*
```
127
1 2 4
```

Ocena

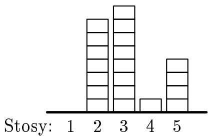
Ograniczenie na czas działania Twojego programu wynosi 2s. Nie można uzyskać części punktów za pojedynczy test.

Krzysztof Diks, Marcin Kubica
przekład

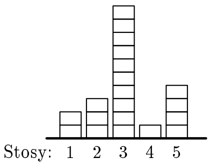
Spłaszczanie

W pewnej jednoosobowej grze ustawia się w rzędzie N stosów tak, że każdy z nich zawiera pewną liczbę (być może zero) krążków. Zobacz rysunek 1. Stosy są ponumerowane od 1 do N w ten sposób, że stosy 1 i N są na końcach. Ruch w tej grze polega na tym, że gracz wskazuje pewien stos, powiedzmy p, oraz podaje liczbę, powiedzmy m. Powoduje to przełożenie po m krążków ze stosu p na każdy z sąsiednich stosów. Zobacz przykład na rysunku 2. Stos p ma dwóch sąsiadów, $p-1$ oraz $p+1$, o ile $1 < p < N$, ma sąsiada 2, gdy $p = 1$, oraz sąsiada $N-1$, gdy $p = N$. Zauważ, że aby można było wykonać taki ruch, stos p musi zawierać co najmniej $2m$ krążków, jeśli ma dwóch sąsiadów i musi zawierać przynajmniej m krążków, jeśli ma tylko jednego sąsiada.

Celem gry jest "spłaszczenie" wszystkich stosów przez zrównanie liczb zawartych w nich krążków w możliwie najmniejszej liczbie ruchów. W przypadku gdy istnieje wiele rozwiązań, masz podać jedno z nich.



Rys. 1. Piec stosów z 0, 7, 8, 1 i 4 krążkami.



Rys. 2. Te same stosy po ruchu: p=2, m=2

Założenia

- *Gwarantuje się, że zawsze można spłaszczyć dane stosy w co najwyżej 10,000 ruchów.*
- $2 \leqslant N \leqslant 200$
- $0 \leqslant C_i \leqslant 2\,000$, *gdzie C_i to początkowa liczba krążków na stosie nr i ($1 \leqslant i \leqslant N$).*

Wejście

Plikiem wejściowym jest plik tekstowy `flat.inp`. *Zawiera on dwa wiersze.*

- *Pierwszy wiersz: N.*
- *Drugi wiersz zawiera N liczb całkowitych: i-ta z nich jest wartością C_i.*

Wyjście

Plikiem wyjściowym jest plik tekstowy `flat.out`.

- *Pierwszy wiersz: liczba ruchów. (Oznaczmy te liczbę przez M.)*

- *Każdy z kolejnych M wierszy zawiera po dwie liczby reprezentujące ruch: p i m.*

Ruchy muszą być zapisane w pliku wyjściowym w takiej samej kolejności, w jakiej są wykonywane. Tak więc pierwszy ruch powinien być zapisany w drugim wierszu pliku.

Przykład

`flat.inp`*:*

```
5
0 7 8 1 4
```

`flat.out`*:*

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

Ocena

Limit na czas działania dla Twojego programu wynosi 3 sekundy.

Aby uzyskać pełną liczbę punktów, A, liczba Twoich ruchów, x, nie może przekraczać pewnej liczby B ustalonej przez program oceniający. B nie musi być równe minimalnej liczbie ruchów. W rzeczywistości B jest ustalane na podstawie liczby ruchów pewnej bardzo prostej strategii (unikającej zbędnych ruchów) oraz średniej liczby krążków na stosie. W tym zadaniu można otrzymać część punktów za pojedynczy test. Punkty, które otrzymasz zostaną wyliczone jako zaokrąglenie do najbliższej liczby całkowitej wartości określonej następującą formułą:

$$\begin{array}{ccc} A & \textit{jeśli} & x \leqslant B \\ 2A(\frac{3}{2}B - x)/B & \textit{jeśli} & B < x < \frac{3}{2}B \\ 0 & \textit{jeśli} & x \geqslant \frac{3}{2}B \end{array}$$

Krzysztof Diks, Marcin Kubica
przekład

Pas ziemi

Mieszkańcy Dingiville próbują znaleźć lokalizację dla lotniska. Mają przed sobą mapę. Mapa jest prostokątną siatką jednostkowych kwadratów. Każdy kwadrat jest identyfikowany przez parę współrzędnych (x, y), gdzie x jest współrzędną poziomą (zachód-wschód), a y jest współrzędną pionową (południe-północ). Dla każdego kwadratu podana jest jego wysokość.

Twoim zadaniem jest znaleźć prostokątny obszar (zbudowany z jednostkowych kwadratów) o największej powierzchni (tj. liczbie zawartych w nim kwadratów) oraz taki, że:

(a) różnica wysokości między najwyższym i najniższym kwadratem obszaru nie przekracza podanego ograniczenia C, oraz

(b) szerokość obszaru (tzn. liczba kwadratów w kierunku zachód-wschód) wynosi co najwyżej 100.

W przypadku gdy jest wiele takich obszarów należy podać jeden z nich.

Założenia

- $1 \leqslant U \leqslant 700$, $1 \leqslant V \leqslant 700$, *gdzie U i V są wymiarami mapy. Dokładniej, U jest liczbą kwadratów w kierunku zachód-wschód, a V liczbą kwadratów w kierunku południe-północ.*
- $0 \leqslant C \leqslant 10$
- $-30\,000 \leqslant H_{xy} \leqslant 30\,000$, *gdzie liczba całkowita H_{xy} jest wysokością kwadratu o współrzędnych (x, y), $1 \leqslant x \leqslant U$, $1 \leqslant y \leqslant V$.*
- *Południowo-zachodni róg mapy ma współrzędne $(1, 1)$, a róg północno-wschodni ma współrzędne (U, V).*

Wejście

Plikiem wejściowym jest plik tekstowy o nazwie `land.inp`*.*

- *Pierwszy wiersz zawiera trzy liczby całkowite: U, V i C.*
- *Każdy z kolejnych V wierszy zawiera liczby całkowite H_{xy}, dla $x = 1, \ldots, U$. Dokładniej, H_{xy} jest x-tą liczbą w $(V - y + 2)$-gim wierszu.*

Wyjście

Wynikiem jest plik tekstowy `land.out` *złożony z jednego wiersza zawierającego cztery liczby całkowite: X_{min}, Y_{min}, X_{max}, Y_{max}. Opisują one znaleziony obszar — (X_{min}, Y_{min}) są współrzędnymi jego południowo-zachodniego rogu, a (X_{max}, Y_{max}) są współrzędnymi jego północno-wschodniego rogu.*

Przykład

`land.inp`:

```
10 15 4
41 40 41 38 39 39 40 42 40 40
39 40 43 40 36 37 35 39 42 42
44 41 39 40 38 40 41 38 35 37
38 38 33 39 36 37 32 36 38 40
39 40 39 39 39 40 40 41 43 41
39 40 41 38 39 38 39 39 39 42
36 39 39 39 39 40 39 41 40 41
31 37 36 41 41 40 39 41 40 40
40 40 40 42 41 40 39 39 39 39
42 40 44 40 38 40 39 39 37 41
41 41 40 39 39 40 41 40 39 40
47 45 49 43 43 41 41 40 39 42
42 41 41 39 40 39 42 40 42 42
41 44 49 43 46 41 42 41 42 42
45 40 42 42 46 42 44 40 42 41
```

41	40	41	38	39	39	40	42	40	40
39	40	43	40	36	37	35	39	42	42
44	41	39	40	38	40	41	38	35	37
38	38	33	39	36	37	32	36	38	40
39	40	39	39	39	40	40	41	43	41
39	40	41	38	39	38	39	39	39	42
36	39	39	39	39	40	39	41	40	41
31	37	36	41	41	40	39	41	40	40
40	40	40	42	41	40	39	39	39	39
42	40	44	40	38	40	39	39	37	41
41	41	40	39	39	40	41	40	39	40
47	45	49	43	43	41	41	40	39	42
42	41	41	39	40	39	42	40	42	42
41	44	49	43	46	41	42	41	42	42
45	40	42	42	46	42	44	40	42	41

$y(N)$

$x(E)$

Rys. 1. Przykładowe rozwiązanie dla danych z `land.inp`

`land.out`:

```
4 5 8 11
```

Ocena

Ograniczenie czasowe na czas działania Twojego programu wynosi 60 sekund. Nie można otrzymać części punktów za pojedynczy test.

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] *VI Olimpiada Informatyczna 1998/1999*, Warszawa 1999

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

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

[9] L. Banachowski, A. Kreczmar, W. Rytter *Analiza algorytmów i struktur danych*, WNT, Warszawa 1987

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Niniejsza publikacja stanowi wyczerpujące źródło informacji o zawodach VII Olimpiady Informatycznej, przeprowadzonych w roku szkolnym 1999/2000. Książka zawiera zarówno informacje dotyczące organizacji, regulaminu oraz wyników zawodów. Zawarto także opis rozwiązań wszystkich zadań konkursowych. Do książki dołączona jest dyskietka zawierająca wzorcowe rozwiązania i testy do wszystkich zadań Olimpiady.

Książka zawiera także informacje o XI Międzynarodowej Olimpiadzie Informatycznej i teksty zadań tej Olimpiady.

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

Olimpiada Informatyczna
jest organizowana przy współudziale

PROKOM
SOFTWARE SA

ISBN 83–906301–6–8