MINISTERSTWO EDUKACJI NARODOWEJ
INSTYTUT INFORMATYKI UNIWERSYTETU WROCŁAWSKIEGO
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

IV OLIMPIADA INFORMATYCZNA
1996/1997

WARSZAWA, 1997

MINISTERSTWO EDUKACJI NARODOWEJ
INSTYTUT INFORMATYKI UNIWERSYTETU WROCŁAWSKIEGO
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

IV OLIMPIADA INFORMATYCZNA
1996/1997

WARSZAWA, 1997

Autorzy tetsktów:

dr Bogdan S. Chlebus
dr Piotr Chrząstowski-Wachtel
Jacek Chrząszcz
prof. dr hab. Wojciech Guzicki
Grzegorz Jakacki
Albert Krzymowski
mgr Marcin Kubica
Marek Pawlicki
prof. dr hab. Wojciech Rytter
prof. dr hab. inż. Stanisław Waligórski
Komitet Główny Olimpiady Informatycznej

Autorzy programów na dyskietce:

Tomasz Błaszczyk
Jacek Chrząszcz
mgr Marcin Engel
Grzegorz Jakacki
Albert Krzymowski
mgr Marcin Kubica
Marcin Mucha
Marek Pawlicki
Krzysztof Sobusiak

Opracowanie i redakcja:

dr Piotr Chrząstowski-Wachtel
dr Krzysztof Diks
Grzegorz Jakacki

Skład:

Grzegorz Jakacki

ISBN 83–906301–3–3

Spis treści

Wstęp

W kraju zawody informatyczne organizowane są od wielu lat. Z początku zawody te miały raczej zasięg lokalny i były organizowane badź nieformalnie, bądź przez organizacje wspierające szkoły. W 1993 roku powstała Olimpiada Informatyczna, powołana przez Instytut Informatyki Uniwersytetu Wrocławskiego.

W roku szkolnym 1996/97 odbyła się IV Olimpiada Informatyczna. Przekazujemy czytelnikom relację z jej przebiegu, a także opracowania zadań wykorzystanych w trakcie Olimpiady. Tradycyjnie, w poprzednich publikacjach nt. Olimpiady Informatycznej zamieszczaliśmy relację z przebiegu Olimpiady Międzynarodowej. Niestety, w tym roku Olimpiada Międzynarodowa odbędzie się w grudniu (w Kapsztadzie, RPA), przeto tradycji nie możemy uczynić zadość.

W opracowaniu zawarliśmy oficjalne dokumenty Komitetu Głównego: „Regulamin olimpiady informatycznej" oraz „Zasady organizacji zawodów w 1996/1997 roku". Dokumenty te specyfikują wymogi stawiane rozwiązaniom zawodników, formę przeprowadzania zawodów, a także sposób przyznawania nagród i wyróżnień.

Uczestników, przyszłych uczestników i nauczycieli informatyki zapewne najbardziej zainteresują zamieszczone w publikacji opracowania zadań. Na każde opracowania składa się opis algorytmu oraz program, zamieszczony na załączonej do książki dyskietce. Autorami opracowań są pomysłodawcy zadań bądź członkowie Jury, którzy przygotowywali rozwiązania wzorcowe.

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

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

Do książki dołączona jest dyskietka zawierająca programy (w językach Pascal lub C) będące rozwiązaniami wzorcowymi zadań IV Olimpiady Informatycznej oraz testy, którymi posłużyło się Jury przy ocenie poprawności rozwiązań zawodników.

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

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.

Materiały dotyczące Olimpiad Informatycznych można nabyć w Ośrodku Edukacji Informatycznej i Zastosowań Komputerów (ul. Raszyńska 8/10, 02–026 Warszawa, tel. (+22)22–40–19) i w Instytucie Informatyki Uniwersytetu Wrocławskiego (ul. Przesmyckiego 20, 51–151 Wrocław, tel. (+71)25–12–71). Pojedyncze egzemplarze posiadają również wojewódzcy koordynatorzy edukacji informatycznej oraz nauczyciele informatyki w szkołach.

Sprawozdanie z przebiegu IV Olimpiady Informatycznej 1996/1997

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

Olimpiada Informatyczna jest trójstopniowa. Integralną częścią rozwiązania każdego zadania zawodów I, II i III stopnia był 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 średnich dla młodzieży (z wyjątkiem szkół policealnych). W Olimpiadzie mogą również uczestniczyć — za zgodą Komitetu Głównego — uczniowie szkół podstawowych.

10 października 1996 r. rozesłano plakaty zawierające zasady organizacji zawodów I stopnia oraz zestaw 4 zadań konkursowych do 3355 szkół i zespołów szkół oraz do wszystkich kuratoriów i koordynatorów edukacji informatycznej. Zawody I stopnia rozpoczęły się dnia 21 października 1996 roku. Ostatecznym terminem nadsyłania prac konkursowych był 18 listopada 1996 roku.

Zawody II i III stopnia polegały na indywidualnym rozwiązywaniu zadań w ciągu dwóch sesji przeprowadzonych w różnych dniach w warunkach kontrolowanej samodzielności. Zawody te były poprzedzone jednodniowymi sesjami próbnymi. Zawody II stopnia odbyły się w trzech okręgach w dniach 10–12.02.1997 r., natomiast zawody III stopnia odbyły się w Polsko-Japońskiej Wyższej Szkole Technik Komputerowych w Warszawie w dniach 23–27.03.1997 r.

Uroczystość zakończenia IV Olimpiady Informatycznej odbyła się w dniu 27.03.97 r. w Auli Polsko-Japońskiej Wyższej Szkoły Technik Komputerowych w Warszawie.

SKŁAD OSOBOWY KOMITETÓW OLIMPIADY INFORMATYCZNEJ

Komitet Główny

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

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

sekretarz naukowy:
dr Andrzej Walat (OEIiZK)*

kierownik organizacyjny:
Tadeusz Kuran (OEIiZK)

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

sekretarz Komitetu Głównego:
Monika Kozłowska

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

Komitet Główny odbył 9 posiedzeń, a Prezydium — 6 zebrań. 31 stycznia 1997 r. przeprowadzono jednodniowe seminarium przygotowujące przeprowadzenie w 3 okręgach zawodów II stopnia.

Komitet Okręgowy w Warszawie

przewodniczący:
dr Krzysztof Diks (Uniwersytet Warszawski)

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

---

* Ośrodek Edukacji Informatycznej i Zastosowań Komputerów w Warszawie

mgr Wojciech Plandowski (Uniwersytet Warszawski)

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

Komitet Okręgowy we Wrocławiu

przewodniczący:
prof. dr hab. Maciej M. Sysło (Uniwersytet Wrocławski)
z-ca przewodniczącego:
dr Krzysztof Loryś (Uniwersytet Wrocławski)
sekretarz:
mgr inż. Maria Woźniak (Uniwersytet Wrocławski)
członkowie:
dr Witold Karczewski (Uniwersytet Wrocławski)
mgr Przemysława Kanarek (Uniwersytet Wrocławski)
mgr Jacek Jagiełło (Uniwersytet Wrocławski)
mgr Paweł Keller (Uniwersytet Wrocławski)

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

Komitet Okręgowy w Toruniu

przewodniczący:
prof. dr hab. Józef Słomiński (Uniwersytet Mikołaja Kopernika w Toruniu)
z-ca przewodniczącego:
dr Mirosława Skowrońska (Uniwersytet Mikołaja Kopernika w Toruniu)
sekretarz:
dr Bolesław Wojdyło (Uniwersytet Mikołaja Kopernika w Toruniu)
członkowie:
dr Krzysztof Skowronek (V Liceum Ogólnokształcące w Toruniu)
mgr Anna Kwiatkowska (IV Liceum Ogólnokształcące w Toruniu)

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

JURY OLIMPIADY INFORMATYCZNEJ

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

Tomasz Błaszczyk
Jacek Chrząszcz
mgr Marcin Engel
Grzegorz Jakacki
Marcin Jurdziński
Albert Krzymowski
mgr Marcin Kubica
Marcin Mucha
Marek Pawlicki
Krzysztof Sobusiak

ZAWODY I STOPNIA

W IV Olimpiadzie Informatycznej wzięło udział 952-ch zawodników. Decyzją Komitetu Głównego Olimpiady do zawodów zostało dopuszczonych 10 uczniów ze szkół podstawowych:

- S.P. nr 54 im. Bohaterów Olszynki Grochowskiej z Warszawy: Łukasz Sawicki, Paweł Skotarek, Andrzej Żabkin, Andrzej Musiał, Michał Góral, Jacek Kujawiński,
- S.P. nr 19 z Białegostoku: Piotr Kurek,
- S.P. nr 8 im. K. Makuszyńskiego z Gdyni: Dominik Wojtczak,
- S.P. nr 112 z Krakowa: Łukasz Pobereżnik,
- S.P. nr 4 im. Urszuli Kochanowskiej w Nowym Sączu: Tomasz Duszka.

KG podjął decyzję o zdyskwalifikowaniu pracy zawodnika oznaczonego numerem 558, która napisana była w języku programowania niezgodnym z regulaminem Olimpiady (Visual Basic). Zdecydowano także nie uwzględniać rozwiązania zadania *Skok w bok* w pracy zawodnika 569 napisanego w języku Basic.

KG podjął decyzję o zdyskwalifikowaniu jednego zawodnika, który w momencie rozpoczynania Olimpiady był uczniem Policealnego Studium Informatycznego. Decyzja ta zapadła po zakończeniu II etapu, gdyż dopiero wówczas Komitet dowiedział się o tym przekroczeniu regulaminu.

Jury podjęło decyzję o sprawdzeniu zadań przysłanych przez 40 zawodników w kopiach zapasowych:

38 rozwiązań zadania	*XOR*
35	*Liczba wyborów symetrycznych*
34	*Tanie podróże*
27	*Skok w bok*
134 rozwiązania łącznie	

Przysłano 16 dyskietek, które były całkowicie lub częściowo nieczytelne. Z 15 dyskietek udało się odzyskać większość zapisów. Dyskietki jednego zawodnika nie udało się odczytać i zawodnik ten nie został sklasyfikowany. Nie wiadomo także jak się nazywał, gdyż w przesyłce nie było żadnej dokumentacji ani danych.

Wirusy usunięto z 50-ciu dyskietek.

Ostatecznie w IV Olimpiadzie Informatycznej sklasyfikowano 950 zawodników, którzy nadesłali łącznie na zawody I stopnia:

910	rozwiązań zadania	*XOR*
829		*Liczba wyborów symetrycznych*
750		*Tanie podróże*
649		*Skok w bok*
3138	rozwiązań łącznie	

Z rozwiązaniami:

czterech zadań nadeszły	522 prace
trzech zadań nadeszło	251 prac
dwóch zadań nadeszło	120 prac
jednego zadania nadeszło	57 prac

Zawodnicy reprezentowali 48 województw. Nie było zawodników tylko z województwa ostrołęckiego. Województwo leszczyńskie reprezentował jeden zawodnik. Najliczniej reprezentowane były następujące województwa:

st.warszawskie	131 zawodników
katowickie	77
gdańskie	54
krakowskie	48
rzeszowskie	45
poznańskie	44
szczecińskie	37
wrocławskie	34
łódzkie	30
bydgoskie	29
kieleckie	29
tarnowskie	27
bielskie	23
zielonogórskie	22
lubelskie	21
nowosądeckie	20
opolskie	20
częstochowskie	19

W zawodach I stopnia najliczniej reprezentowane były szkoły:

VIII L.O. im. A. Mickiewicza w Poznaniu	20 uczniów
XIV L.O. im. S. Staszica w Warszawie	14
XXVII L.O. im. T. Czackiego w Warszawie	14
III L.O. im. Marynarki Wojennej RP w Gdyni	12

V L.O. im. A. Witkowskiego w Krakowie	10
V L.O. im. J. Poniatowskiego w Warszawie	10
I L.O. im. B. Nowodworskiego w Krakowie	9
L.O. im. Króla W. Jagiełły w Dębicy	8
I L.O. im. S. Dubois w Koszalinie	7
I L.O. im. St. Konarskiego w Mielcu	7
S.P. nr 54 im. Bohaterów Olszynki Grochowskiej w Warszawie	7
XL L.O. im S. Żeromskiego w Warszawie	7
III L.O. im. A. Mickiewicza we Wrocławiu	7
I L.O. im. Ks. A. Czartoryskiego w Puławach	6
I L.O. im. St. Konarskiego w Rzeszowie	6
VI L.O. im. J. i J. Śniadeckich w Bydgoszczy	6
IV L.O. im. M. Kopernika w Rzeszowie	6
IX L.O. im. Bohaterów Monte Casino w Szczecinie	6
XII L.O. w Szczecinie	6
XXVIII L.O. im. J. Kochanowskiego w Warszawie	6
XIV L.O. im. Polonii Belgijskiej we Wrocławiu	6
V L.O. w Bielsku Białej	5
I L.O. im. Powstańców Śląskich w Rybniku	5
I L.O. im. M. Kopernika w Gdańsku	5
I L.O. im. St. Żeromskiego w Jeleniej Górze	5
VIII L.O. im. M. Skłodowskiej-Curie w Katowicach	5
VI L.O. im. J. Słowackiego w Kielcach	5
I L.O. im. M. Kopernikaw Opolu	5
II L.O. im. Królowej Jadwigi w Siedlcach	5
IV L.O. im T. Kościuszki w Toruniu	5
I Społeczne L.O. w Warszawie	5
LI L.O. im. T. Kościuszki w Warszawie	5

Ogólnie najliczniej reprezentowane były miasta:

Warszawa	125 uczniów	Katowice	15 uczniów
Kraków	46	Gdańsk	14
Poznań	40	Białystok	12
Szczecin	33	Opole	12
Gdynia	31	Ostrowiec Św.	12
Bydgoszcz	23	Tarnów	12
Wrocław	23	Zielona Góra	11
Rzeszów	21	Dębica	9
Łódź	19	Koszalin	9
Mielec	17	Olsztyn	9
Częstochowa	15		

809 zawodników podało klasę, do jakiej chodzi:

do klasy I	szkoły średniej	47 zawodników
do klasy II		176
do klasy III		255
do klasy IV		290
do klasy V		31
do klasy VIII	szkoły podstawowej	8
do klasy VI		2

Zawodnicy najczęściej używali następujących języków programowania:

Pascal firmy Borland	781 prac
C/C++ firmy Borland	162 prace

Ponadto pojawiły się:

Watcom C/C++	3 prace
High Speed Pascal	1 praca
MS Quick Pascal	1
MS Quick C	1
GNU C 2.0	1

Komputerowe wspomaganie umożliwiło podstawowe sprawdzenie prac zawodników kompletem testów w ciągu kilkunastu godzin.

Proces sprawdzania był utrudniony — choć w mniejszym stopniu niż w poprzednich olimpiadach — występowaniem takich niedokładności jak nie przestrzeganie przez zawodników reguł dotyczących nazywania plików i budowania zestawów danych wynikowych, podanych wyraźnie w treści zadań.

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

	XOR		*Liczba wyborów symetrycznych*		*Tanie podróże*		*Skok w bok*	
	liczba zawodn.	czyli	liczba zawodn.	czyli	liczba zawodn.	czyli	liczba zawodn.	czyli
100 pkt.	52	5,7%	16	1,9%	110	14,7%	68	10,5%
99–75 pkt.	37	4,1%	266	32,1%	54	7,2%	101	15,6%
74–50 pkt.	243	26,7%	277	33,4%	85	11,3%	115	17,7%
49–1 pkt.	487	53,5%	230	27,8%	376	50,1%	327	50,4%
0 pkt.	91	10,0%	40	4,8%	125	16,7%	38	5,8%

W sumie za wszystkie 4 zadania:

SUMA	liczba zawodników	czyli
400 pkt.	3	0,3%
399–300 pkt.	75	7,9%
299–200 pkt.	188	19,8%
199–1 pkt.	650	68,4%
0 pkt.	34	3,6%

Prezydium Komitetu Głównego przy udziale Jury rozpatrzyło 4 reklamacje, które w 3 przypadkach wniosły zmiany do punktacji zawodów I stopnia.

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

ZAWODY II STOPNIA

Do zawodów II stopnia zakwalifikowano 115 zawodników, którzy osiągnęli w zawodach I stopnia wynik nie mniejszy niż 276 pkt. Zawody II stopnia odbyły się w dniach 10–12 lutego 1997 r. w trzech okręgach:

- w Toruniu, 19-tu zawodników z następujących województw:
    - bydgoskiego (8),
    - gdańskiego (7),
    - olsztyńskiego (2),
    - toruńskiego (2).
- we Wrocławiu, 40-tu zawodników z następujących województw:
    - częstochowskiego (3),
    - jeleniogórskiego (2),
    - kaliskiego (2),
    - katowickiego (3),
    - konińskiego (1),
    - koszalińskiego (4),
    - krakowskiego (7),
    - łódzkiego (4),
    - opolskiego (2),
    - piotrkowskiego (1),
    - poznańskiego (2),

  - szczecińskiego (2),
  - wałbrzyskiego (1),
  - wrocławskiego (4),
  - zielonogórskiego (2).
- w Warszawie, 56-ciu zawodników z następujących województw:
  - białostockiego (1),
  - bielskiego (1),
  - bydgoskiego (1),
  - chełmskiego (1),
  - gdańskiego (4),
  - kieleckiego (3),
  - krośnieńskiego (3),
  - lubelskiego (1),
  - łomżyńskiego (2),
  - nowosądeckiego (3),
  - płockiego (1),
  - radomskiego (1),
  - rzeszowskiego (9),
  - siedleckiego (1),
  - st.warszawskiego (22),
  - suwalskiego (1),
  - tarnowskiego (1).

Najliczniej w zawodach II stopnia reprezentowane były szkoły:

XXVII L.O. im. T. Czackiego w Warszawie	8 uczniów
VI L.O. im. J. i J. Śniadeckich w Bydgoszczy	5
III L.O. im. Marynarki Wojennej RP w Gdyni	5
XIV L.O. im. S. Staszica w Warszawie	4
XIV L.O. im. Polonii Belgijskiej Wrocław	4
I L.O. im. St. Dubois w Koszalinie	4
V L.O. im. A. Witkowskiego w Krakowie	4
V L.O. im. A. Mickiewicza w Częstochowie	3
I L.O. im. M. Kopernika w Gdańsku	3
VI L.O. im. J. Słowackiego w Kielcach	3
V L.O. im. Ks. J. Poniatowskiego w Warszawie	3
I L.O. im. B. Nowodworskiego w Krakowie	3
I L.O. im. St. Konarskiego w Rzeszowie	3

Ogólnie najliczniej reprezentowane były miasta:

Warszawa	22 zawodników
Bydgoszcz	9
Kraków	7
Rzeszów	7
Gdynia	6

10 lutego odbyła się sesja próbna, na której zawodnicy rozwiązywali nie liczące się do ogólnej klasyfikacji zadanie *Lotniska*. W dniach konkursowych zawodnicy rozwiązywali zadania: *Addon*, *Genotypy*, *Rekurencyjna mrówka* i *Paliwo* oceniane każde maksymalnie po 100 punktów. Decyzją Komitetu Głównego nie przyznawano punktów uznaniowych za oryginalne rozwiązania.

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

	Addon		*Genotypy*		*Rekurencyjna mrówka*		*Paliwo*	
	liczba zawodn.	czyli	liczba zawodn.	czyli	liczba zawodn.	czyli	liczba zawodn.	czyli
100 pkt.	20	17,4%	0	0,0%	1	0,9%	24	20,9%
99–75 pkt.	20	17,4%	3	2,6%	1	0,9%	8	7,0%
74–50 pkt.	22	19,1%	5	4,3%	3	2,6%	15	13,0%
49–1 pkt.	13	11,3%	47	40,9%	48	40,8%	63	54,8%
0 pkt.	40	34,8%	60	52,2%	62	54,8%	5	4,3%

W sumie za wszystkie 4 zadania uzyskano najwyższy wynik wynoszący 390 pkt. Poniższa tabela przedstawia liczby zawodników, którzy łącznie uzyskali określone liczby punktów zarówno w zestawieniu ilościowym jak i procentowym:

SUMA	liczba zawodników	czyli
400 pkt.	0	0,0%
399–300 pkt.	2	1,8%
299–200 pkt.	12	10,4%
199–1 pkt.	94	81,7%
0 pkt.	7	6,1%

W każdym okręgu na uroczystym zakończeniu zawodów II stopnia zawodnicy otrzymali upominki książkowe. Po dokonaniu oceny przesłano im listy z wynikami zawodów i dyskietkami zawierającymi ich rozwiązania oraz testy, na podstawie których oceniano prace.

ZAWODY III STOPNIA

Zawody III stopnia odbyły się w Polsko-Japońskiej Wyższej Szkole Technik Komputerowych w Warszawie w dniach od 23 do 27 marca 1997 r.

W zawodach III stopnia wzięło udział 44 najlepszych uczestników zawodów II stopnia, którzy uzyskali wynik nie mniejszy niż 124 pkt. Zawodnicy reprezentowali nastepujące województwa:

st.warszawskie	11 zawodników
rzeszowskie	4
gdańskie	3
wrocławskie	3
bydgoskie	2
łódzkie	2
krakowskie	2
krośnieńskie	2
toruńskie	2
białostockie	1 zawodnik
bielskie	1
chełmskie	1
kaliskie	1
katowickie	1
koszalińskie	1
olsztyńskie	1
piotrkowskie	1
poznańskie	1
szczecińskie	1
tarnowskie	1
wałbrzyskie	1
zielonogórskie	1

Niżej wymienione szkoły miały w finale więcej niż jednego zawodnika:

XIV L.O. im. St. Staszica w Warszawie	4 zawodników
XXVII L.O. im. T. Czackiego w Warszawie	3
XIV L.O. im. Polonii Belgijskiej we Wrocławiu	3
VI L.O. im. J. i J. Śniadeckich w Bydgoszczy	2
III L.O. im Marynarki Wojennej RP w Gdyni	2
I L.O. im. B. Nowodworskiego w Krakowie	2
IV L.O. im. M. Kopernika w Rzeszowie	2
IV L.O. im. T. Kościuszki w Toruniu	2

23 marca odbyła się sesja próbna, na której zawodnicy rozwiązywali nie liczące się do ogólnej klasyfikacji zadanie *Wiarygodność świadków*. Każdego dnia konkursowego zawodnicy mogli uzyskać łącznie 100 punktów. Liczby punktów za poszczególne zadania były zróżnicowane. Dodatkowo zawodnicy uzyskiwali punkty za poprawną składnię rozwiązań, ściśle zgodną z wymaganiami postawionymi w zadaniach. Nazwy zadań i liczby punktów za poszczególne zadania podają tabele:

- dzień pierwszy

	Kajaki	*Trójkąty jednobarwne*	*Liczba zbiorów n-k-specjalnych*
maksymalna liczba punktów	20	30	50
w tym premia za poprawną składnię	1	1	2

- dzień drugi

	Rezerwacja sal wykładowych	*Alibaba*
maksymalna liczba punktów	40	60
w tym premia za poprawną składnię	2	3

W zawodach trzeciego stopnia sprawdzano rozwiązania zawodników za pomocą nowego programu sprawdzającego napisanego przez pana Marka Pawlickiego. Dzięki temu programowi można było przeprowadzić pełne sprawdzenie rozwiązań z udziałem zawodników bezpośrednio po zakończeniu zawodów każdego dnia. W treści sprawdzania możliwe było przeglądanie pliku wejściowego, pliku wyjściowego i pliku z wzorcem po każdym teście, tak jak dzieje się to na Olimpiadach Międzynarodowych.

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

- *Liczba zbiorów n-k-specjalnych*

	liczba zawodników	czyli
50 pkt.	7	15,9%
49–37 pkt.	5	11,4%
36–25 pkt.	16	36,4%
24–1 pkt.	14	31,8%
0 pkt.	2	4,5%

- *Trójkąty jednobarwne*

	liczba zawodników	czyli
30 pkt.	2	4,5%
29–21 pkt.	1	2,3%
20–15 pkt.	3	6,8%
14–1 pkt.	29	65,9%
0 pkt.	9	20,5%

- *Kajaki*

	liczba zawodników	czyli
20 pkt.	22	50,0%
19–15 pkt.	7	15,9%
14–10 pkt.	1	2,3%
9–1 pkt.	12	27,3%
0 pkt.	2	4,5%

- *Alibaba*

	liczba zawodników	czyli
60 pkt.	19	43,2%
59–45 pkt.	9	20,4%
44–30 pkt.	0	0,0%
29–1 pkt.	8	18,2%
0 pkt.	8	18,2%

- *Rezerwacja sal wykładowych*

	liczba zawodników	czyli
40 pkt.	20	45,4%
39–30 pkt.	5	11,4%
29–20 pkt.	8	18,2%
19–1 pkt.	10	22,7%
0 pkt.	1	2,3%

W sumie za wszystkie 5 zadań:

SUMA	liczba zawodników	czyli
200 pkt.	1	2,3%
199–150 pkt.	12	27,3%
149–100 pkt.	17	38,6%
99–1 pkt.	14	31,8%
0 pkt.	0	0,0%

W dniu 27 marca 1997 roku w auli Polsko-Japońskiej Wyższej Szkoły Technik Komputerowych w Warszawie ogłoszono wyniki Olimpiady Informatycznej 1996/97 i rozdano nagrody ufundowane przez: Ministerstwo Edukacji Narodowej, Ogólnopolską Fundację Edukacji Komputerowej, Optimus SA, Wydawnictwa Naukowe PWN i Microsoft. Poniżej zestawiono listę wszystkich laureatów:

(1) **Tomasz Waleń**, laureat I miejsca, 200 pkt.
(Komputer, OFEK)*

(2) **Andrzej Gąsienica-Samek**, laureat I miejsca, 197 pkt.
(Komputer, MEN)

(3) **Piotr Sankowski**, laureat II miejsca, 188pkt.
(Komputer, Optimus)

(4) **Piotr Zieliński**, laureat II miejsca, 188 pkt.
(Komputer, Olimpiada Informatyczna)

(5) **Eryk Kopczyński**, laureat II miejsca, 184 pkt.
(Drukarka, OFEK)

(6) **Adam Borowski**, laureat II miejsca, 178 pkt.
(Drukarka, Olimpiada Informatyczna)

(7) **Marcin Stefaniak**, laureat III miejsca, 161 pkt.
(Zestaw multimedialny, Olimpiada Informatyczna)

(8) **Paweł Kwiatkowski**, laureat III miejsca, 160 pkt.
(Zestaw multimedialny, Olimpiada Informatyczna)

(9) **Paweł Lenk**, laureat III miejsca, 159 pkt.
(Zestaw multimedialny, Olimpiada Informatyczna)

(10) **Grzegorz Owsiany**, laureat III miejsca, 159 pkt.
(Zestaw multimedialny, Olimpiada Informatyczna)

Wyróżniono też następujących finalistów:

(11) **Maciej Żenczykowski**, finalista z wyróżnieniem, 154 pkt.
(Oprogramowanie multimedialne, Optimus)

(12) **Jerzy Bałamut**, finalista z wyróżnieniem, 151 pkt.
(Oprogramowanie multimedialne, Optimus)

(13) **Jakub Dzierzbicki**, finalista z wyróżnieniem, 150 pkt.
(Oprogramowanie multimedialne, Optimus)

(14) **Marcin Bieńkowski**, finalista z wyróżnieniem, 149 pkt.
(Oprogramowanie multimedialne, Optimus)

(15) **Marek Rycharski**, finalista z wyróżnieniem, 147 pkt.
(Oprogramowanie multimedialne, Optimus)

(16) **Marek Futrega**, finalista z wyróżnieniem, 145 pkt.
(Oprogramowanie multimedialne, Optimus)

(17) **Łukasz Kolczyński**, finalista z wyróżnieniem, 143 pkt.
(Oprogramowanie multimedialne, Optimus)

---

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

Wszyscy laureaci i finaliści otrzymali książki ufundowane przez PWN i drobne upominki ufundowane przez firmę Microsoft.

Ogłoszono komunikat o powołaniu reprezentacji Polski na Bałtycką Olimpiadę Informatyczną w Wilnie w składzie:

(1) **Tomasz Waleń**

(2) **Andrzej Gąsienica-Samek**

(3) **Piotr Sankowski**

(4) **Piotr Zieliński**

(5) **Eryk Kopczyński**

(6) **AdamBorowski**

(7) **MarcinStefaniak**

(8) **Paweł Kwiatkowski**

Zawodnikami rezerwowymi zostali:

(9) **Paweł Lenk**

(10) **Grzegorz Owsiany**

Powołano także reprezentację Polski na Olimpiadę Informatyczną Centralnej Europy w Nowym Sączu w składzie:

- I drużyna (delegacja oficjalna)
  (1) **Tomasz Waleń**
  (2) **Andrzej Gąsienica-Samek**
  (3) **Piotr Sankowski**
  (4) **Piotr Zieliński**
  (5) **Eryk Kopczyński** (zawodnik rezerwowy)
  (6) **Adam Borowski** (zawodnik rezerwowy)
- II drużyna (poza konkursem)
  (1) **Eryk Kopczyński**
  (2) **Marcin Stefaniak**
  (3) **Maciej Żenczykowski**
  (4) **Jerzy Bałamut**
  (5) **Adam Koprowski** (zawodnik rezerwowy)
  (6) **Krzysztof Onak** (zawodnik rezerwowy)

Wyznaczono także reprezentację Polski na Międzynarodową Olimpiadę Informatyczną, która odbędzie się w grudniu br. w Kapsztadzie (Republika Południowej Afryki):

(1) **Tomasz Waleń**

(2) **Andrzej Gąsienica-Samek**

(3) **Piotr Sankowski**

(4) **Piotr Zieliński**

Zawodnikami rezerwowymi zostali:

(5) **Eryk Kopczyński**

(6) **Adam Borowski**

Zawodnicy zakwalifikowani do zawodów III stopnia Olimpiady są zwolnieni z egzaminu dojrzałości z przedmiotu informatyka na mocy 9 ust. 1 pkt 2 Zarządzenia Ministra Edukacji Narodowej z dnia 14 września 1992 r. Sekretariat olimpiady wystawił łącznie 44 zaświadczenia o zakwalifikowaniu do zawodów III stopnia celem przedłożenia dyrekcji szkoły. Laureaci i finaliści mogą być zwolnieni z egzaminów wstępnych do wielu szkół wyższych na mocy uchwał senatów uczelni podjętych na wniosek MEN zgodnie z art. 141 ust. 1 ustawy z dnia 12 września 1990 r. o szkolnictwie wyższym (Dz.U. nr 65, poz. 385). Sekretariat wystawił łącznie 10 zaświadczeń o uzyskaniu tytułu laureata i 34 zaświadczeń o uzyskaniu tytułu finalisty IV Olimpiady Informatycznej celem przedłożenia władzom szkół wyższych.

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

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

- **Małgorzata Rostkowska** (XIV L.O. im. S. Staszica w Warszawie), uczniowie:
  - Andrzej Gąsienica-Samek (laureat I miejsca),
  - Eryk Kopczyński (laureat II miejsca),
  - Jerzy Szczepkowski (finalista).
- **Krzysztof Stefański** (VIII L.O. im. A. Mickiewicza w Poznaniu), uczeń:
  - Piotr Zieliński (laureat II miejsca).
- **Małgorzata Szatybełko** (I L.O. im. M. Skłodowskiej-Curie w Starogardzie Gdańskim), uczeń:
  - Adam Borowski (laureat II miejsca)

- **Anna Kwiatkowska** (IV L.O. im. T. Kościuszki w Toruniu), uczniowie:
  - Paweł Kwiatkowski (laureat III miejsca),
  - Marek Rycharski (finalista z wyróżnieniem).
- **Joanna Śmierzchalska** (III L.O. im. Marynarki Wojennej RP w Gdyni), uczeń:
  - Marcin Stefaniak (laureat III miejsca).
- **Marek Maryniak** (IV L.O. im. St. Staszica w Sosnowcu), uczeń:
  - Paweł Lenk (laureat III miejsca).
- **Rafał Wysocki** (XXVII L.O. im. T. Czackiego w Warszawie), uczniowie:
  - Łukasz Kolczyński (finalista z wyróżnieniem),
  - Przemysław Jaroszewski (finalista).
- **Iwona Waszkiewicz** (VI L.O. im. J. i J. Śniadeckich w Bydgoszczy), uczniowie:
  - Łukasz Anforowicz (finalista),
  - Wojciech Meler (finalista).

Komitet postanowił, biorąc pod uwagę zasługi pana **Krzysztofa Lorysia** w przygotowaniu jednego laureata i trzech finalistów (Tomasz Waleń — laureat I miejsca, Marcin Bieńkowski — finalista z wyróżnieniem, Stanisław Pańko — finalista, Jacek Suliga — finalista), jak i fakt, że jest on członkiem Komitetu Głównego Olimpiady, skierować do niego tylko list gratulacyjny.

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

Warszawa, dn. 16 maja 1997 roku

Regulamin Olimpiady Informatycznej

§1 WSTĘP

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

§2 CELE OLIMPIADY INFORMATYCZNEJ

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

(2) Rozszerzanie współdziałania nauczycieli akademickich z nauczycielami szkół w kształceniu młodzieży uzdolnionej.

(3) Stymulowanie aktywności poznawczej młodzieży informatycznie uzdolnionej.

(4) Kształtowanie umiejętności samodzielnego zdobywania i rozszerzania wiedzy informatycznej.

(5) Stwarzanie młodzieży możliwości szlachetnego współzawodnictwa w rozwijaniu swoich uzdolnień, a nauczycielom — warunków twórczej pracy z młodzieżą.

(6) Wyłanianie reprezentacji Rzeczypospolitej Polskiej na Międzynarodową Olimpiadę Informatyczną.

§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) Liczbę i treść zadań na każdy stopień zawodów ustala Komitet Główny, wybierając je drogą głosowania spośród zgłoszonych projektów.

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

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

(8) Liczbę uczestników zakwalifikowanych 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.

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

(12) Zawody II i III stopnia polegają na samodzielnym rozwiązaniu przez uczestników Olimpiady zakwalifikowanych do tych zawodów zadań przygotowanych dla danego stopnia w ciągu dwóch sesji przeprowadzanych w różnych dniach w warunkach kontrolowanej samodzielności.

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

§4 KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

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

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

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

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

(5) Komitet Główny powołuje Komitety Okręgowe Olimpiady w miarę powiększania się liczby uczestników zawodów i powstawania odpowiednich warunków organizacyjnych.

(6) Komitet Główny może powołać Komitet Honorowy spośród sponsorów przyczyniających się do rozwoju Olimpiady.

(7) Komitet Główny Olimpiady Informatycznej:

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

   (b) ustala treść zadań na wszystkie stopnie Olimpiady,

   (c) powołuje jury Olimpiady, które jest odpowiedzialne za sprawdzenie zadań oraz mianuje kierownika Jury

   (d) udziela wyjaśnień w sprawach dotyczących Olimpiady,

   (e) ustala listy uczestników zawodów II i III stopnia,

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

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

   (h) ustala kryteria wyłaniania uczestników uprawnionych do startu w Międzynarodowej Olimpiadzie Informatycznej i publikuje je w *Zasadach organizacji zawodów* oraz ustala ostateczną listę reprezentacji,

   (i) zatwierdza liczbę etatów biura Olimpiady na wniosek kierownika organizacyjnego, który odpowiada za sprawne działanie biura.

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

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

(10) Decyzje Komitetu we wszystkich sprawach dotyczących zadań i uczestników są ostateczne.

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

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

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

(14) Przewodniczący:

- (a) czuwa nad całokształtem prac Komitetu,
- (b) zwołuje posiedzenia Komitetu,
- (c) przewodniczy tym posiedzeniom,
- (d) reprezentuje Komitet na zewnątrz,
- (e) czuwa nad prawidłowością wydatków związanych z organizacją i przeprowadzeniem Olimpiady oraz zgodnością działalności Komitetu z przepisami.

(15) Komitet prowadzi archiwum akt Olimpiady przechowując w nim:

- (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 mianowanych przez Komitet Główny na okres swojej kadencji.

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

(3) Zadaniem Komitetów Okręgowych jest organizacja zawodów II stopnia oraz popularyzacja Olimpiady.

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

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

§7 UPRAWNIENIA I NAGRODY

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

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

(3) Laureaci zawodów III stopnia, a także finaliści są zwolnieni w części lub w całości z egzaminów wstępnych do szkół wyższych — na mocy uchwał senatów poszczególnych uczelni, podjętych zgodnie z przepisami ustawy z dnia 12 września 1990 r. o szkolnictwie wyższym (Dz.U. nr 65 poz. 385) — o ile te uchwały nie stanowią inaczej.

(4) Zaświadczenia o uzyskanych uprawnieniach wydają uczestnikom Komitet Główny i komitety okręgowe. 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 nagrody pieniężne z funduszu Olimpiady.

§8 FINANSOWANIE OLIMPIADY

(1) Komitet Główny będzie się ubiegał o dotację z budżetu państwa, składając wniosek w tej sprawie do Ministra Edukacji Narodowej i przedstawiając przewidywany plan finansowy organizacji Olimpiady na dany rok. Komitet będzie także zabiegał o pozyskanie dotacji z innych organizacji wspierających.

§9 PRZEPISY KOŃCOWE

(1) Koordynatorzy Edukacji Informatycznej i dyrektorzy szkół mają obowiązek dopilnowania, aby wszystkie wytyczne oraz informacje dotyczące Olimpiady zostały podane do wiadomości uczniów.

(2) Wyniki zawodów I stopnia Olimpiady są tajne do czasu ustalenia listy uczestników zawodów II stopnia. Wyniki zawodów II stopnia są tajne do czasu ustalenia listy uczestników zawodów III stopnia Olimpiady.

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

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

Warszawa, 20 września 1996 roku

Zasady organizacji zawodów w roku szkolnym 1996/97

Podstawowym aktem prawnym dotyczącym Olimpiady Jest Regulamin Olimpiady Informatycznej, którego pełny tekst znajduje się w kuratoriach oświaty oraz komitetach olimpiady. Poniższe zasady są uzupełnieniem tego regulaminu, zawierającym szczegółowe postanowienia Komitetu Głównego Olimpiady Informatycznej o jej organizacji w roku szkolnym 1996/97.

§1 WSTĘP

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

§2 ORGANIZACJA OLIMPIADY

(1) Olimpiadę przeprowadza Komitet Główny Olimpiady Informatycznej.

(2) Olimpiada informatyczna jest trójstopniowa.

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

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

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

(6) Zawody II i III stopnia polegają na indywidualnym rozwiązywaniu zadań w ciągu dwóch sesji przeprowadzanych w różnych dniach w warunkach kontrolowanej samodzielności.

(7) Do zawodów II stopnia zostanie zakwalifikowanych 100 uczestników, których rozwiązania zadań I stopnia zostaną ocenione najwyżej; do zawodów III stopnia — 40 uczestników, których rozwiązania zadań II stopnia zostaną ocenione najwyżej.

Komitet Główny może zmienić podane liczby zakwalifikowanych uczestników o co najwyżej 15%.

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

§3 WYMAGANIA DOTYCZĄCE ROZWIĄZAŃ ZADAŃ ZAWODÓW I STOPNIA

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

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

w nieprzekraczalnym terminie nadania do 18 listopada 1996 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 uruchomienia na komputerze IBM PC i zawierającą:

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

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

(5) Wszystkie nadsyłane teksty powinny być drukowane (lub czytelnie pisane) jednostronnie 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 i numer telefonu szkoły oraz klasę, do której uczęszcza,
- nazwę i numer wersji użytego języka programowania,
- opis konfiguracji komputera, na którym rozwiązywał zadania.

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

Pascal	PAS
C	C
C++	CPP

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

§4 UPRAWNIENIA I NAGRODY

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

(2) Uczestnicy Olimpiady, którzy zostali zakwalifikowani do zawodów III stopnia, są zwolnieni z egzaminu dojrzałości (zgodnie z zarządzeniem Ministra Edukacji Narodowej z dn. 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 szkół wyższych na mocy uchwał senatów poszczególnych uczelni, podjętych zgodnie z przepisami ustawy z dnia 12 września 1990 roku o szkolnictwie wyższym (Dz. U. nr 65, poz. 385), o ile te uchwały nie stanowią inaczej.

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

(5) Komitet Główny ustala składa reprezentacji Polski na IX Międzynarodową Olimpiadę Informatyczną w 1997 roku na podstawie wyników zawodów III stopnia i regulaminu tej Olimpiady. Szczegółowe zasady zostaną podane po otrzymaniu formalnego zaproszenia na IX Międzynarodową Olimpiadę Informatyczną.

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

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

§5 PRZEPISY KOŃCOWE

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

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

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

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

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

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

(3) Twój program powinien czytać dane z pliku i zapisywać wyniki do pliku o podanych w treści zadania nazwach.

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

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

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

Zawody I stopnia

opracowania zadań

Wojciech Rytter **Grzegorz Jakacki**
treść zadania, opracowanie program wzorcowy

Liczba wyborów symetrycznych

Dane są dwa ciągi słów (x_i) *oraz* (y_i)*, gdzie:* $1 \leqslant i \leqslant n$ *oraz* $1 \leqslant n \leqslant 30$*. Kolejno dla każdego* i *od 1 do* n *wybieramy jedno z dwóch słów:* x_i *albo* y_i *i wybrane słowa składamy (kolejne wybrane słowo dopisujemy do poprzednich z prawej strony).*
Wybór *składa się z* n *decyzji, czy w kolejnym kroku wybrać odpowiedni* i*-ty wyraz z pierwszego, czy z drugiego ciągu słów. Bardziej formalnie: wybór jest* n *wyrazowym ciągiem liczb 1 lub 2. Różne wybory mogą dać w wyniku to samo słowo.*
Wybór nazywamy **symetrycznym***, gdy jego wynikiem jest palindrom — tzn. takie słowo, które nie zmienia się, gdy je czytamy od strony prawej do lewej.*

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego LIC.IN liczbę* n *i dwa* n*-wyrazowe ciągi słów* (x_i) *i* (y_i)*,*
- *oblicza liczbę wszystkich wyborów symetrycznych dla tych dwóch ciągów słów,*
- *zapisuje wynik w pliku tekstowym LIC.OUT.*

Wejście

W pierwszym wierszu pliku tekstowego LIC.IN jest zapisana jedna liczba całkowita dodatnia $n \leqslant 30$*.*
W następnych n *wierszach są zapisane kolejne słowa ciągu* (x_i) *— każde w osobnym wierszu.*
W następnych n *wierszach są zapisane — w taki sam sposób — kolejne słowa ciągu* (y_i)*.*
Wszystkie słowa składają się wyłącznie z małych liter alfabetu angielskiego od a do z i mają długość nie mniejszą niż 1 i nie większą niż 400.

Wyjście

W pliku tekstowym LIC.OUT należy zapisać jedną liczbę całkowitą nieujemną, a mianowicie liczbę wszystkich wyborów symetrycznych.

Przykład

Dla pliku LIC.IN:

```
5
ab
a
a
ab
a
a
```

```
baaaa
a
a
ba
```

poprawnym rozwiązaniem jest plik LIC.OUT:

```
12
```

Twój program powinien szukać pliku LIC.IN w katalogu bieżącym i tworzyć plik LIC.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę LIC.???, gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonalnej powinien być zapisany w pliku LIC.EXE

UWAGA DO TREŚCI ZADANIA

W treści zadania znalazła się pewna niedokładność. Ostatnie zdanie punktu „Wejście" miałoby sens bliższy zamierzonemu przez autora zadania, gdyby brzmiało:

> *Wszystkie słowa składają się wyłącznie z małych liter alfabetu angielskiego od a do z i mają <u>łączną</u> długość nie mniejszą niż 1 i nie większą niż 400.*

ROZWIĄZANIE

Zadanie jest dosyć proste, łatwo rozwiązuje się je przez sprowadzenie do problemu obliczenia liczby wszystkich ścieżek prowadzących od wierzchołka początkowego **start** do wierzchołka końcowego **koniec** w pewnym grafie acyklicznym G. Jak się okaże, bezpośrednia konstrukcja grafu G w programie nie jest konieczna.

Rozwiązywanie zadania w sposób „naiwny" przez zbadanie wszystkich możliwych wyborów prowadzi do programu działającego w przypadku pesymistycznym zbyt długo, aby można było taki program zaakceptować (liczba możliwych wyborów jest wykładnicza).

UWAGA: Jeśli zastąpimy liczenie wszystkich wyborów przez liczenie wszystkich różnych słów symetrycznych, które możemy otrzymać, to zadanie staje się dużo trudniejsze. Nie jest znany żaden wielomianowy algorytm dla takiej wersji zadania.

Na wejściu dostajemy ciąg słów $x_1, \ldots, x_n$, $y_1, \ldots, y_n$. Załóżmy, że N jest sumą długości wszystkich słów. Słowa pamiętamy w tablicy T, jako jedno długie słowo $x_1 y_1 x_2 y_2 \ldots x_n y_n$. Dodatkowo przechowujemy informacje o tym, gdzie w tablicy T są końce i początki słów x_i, y_i.

Chociaż nasze zadanie ma charakter tekstowy, to sprowadzimy je do problemu teoriografowego.

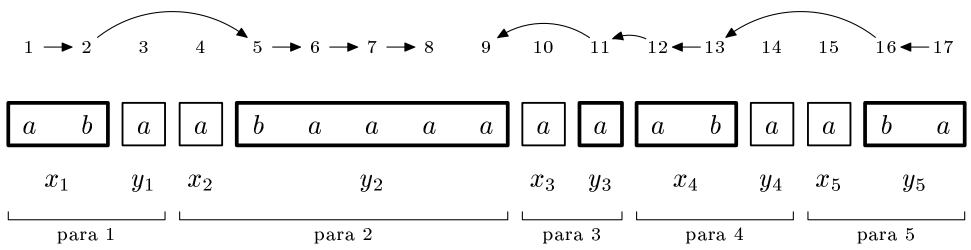
Wyobraźmy sobie, że mamy dwa wskaźniki przesuwające się po tablicy T. Jeden z nich „wędruje" od strony lewej do prawej, a drugi od prawej do lewej. Wskaźniki służą do wybierania po jednym słowie z każdej pary i przesuwają się w sposób zsynchronizowany, tak że tyle samo liter jest pobieranych zarówno z lewej, jak i z prawej strony. Przesuwając wskaźniki sprawdzamy jednocześnie symetrię tekstu. W momencie, gdy wskaźniki się spotkają wiemy, że znaleźliśmy słowo symetryczne i dokonaliśmy wyboru z każdej pary x_k, y_k. Wskaźniki przesuwamy tylko wtedy, gdy kolejne litery są identyczne (rozszerzamy symetrię od zewnątrz do środka).

Rozważmy skierowany acykliczny graf $G = (V, E)$, którego budowę omówimy na przykładzie z treści zadania. W tym celu posłużymy się rys. 1. Zbiór V składa się ze wszystkich par (i, j), gdzie $1 \leqslant i < j \leqslant N$, zwanych **wierzchołkami wewnętrznymi**, oraz dodatkowych wierzchołków *start* i *koniec*.

Każdy wewnętrzny wierzchołek jest parą (i, j) interpretowaną jako

$$\text{(pozycja lewego wskaźnika, pozycja prawego wskaźnika)}$$

Rys. 1 Załóżmy, że z każdej pary słów wybieramy słowo w pogrubionej ramce. Jest to wybór symetryczny. Odpowiada to przykładowej ścieżce w grafie G przedstawionej pod rysunkiem.



$$\mathit{start} \rightsquigarrow (1, 17) \rightsquigarrow (2, 16) \rightsquigarrow (5, 13) \rightsquigarrow (6, 12) \rightsquigarrow (7, 11) \rightsquigarrow (8, 9) \rightsquigarrow \mathit{koniec}$$

Tablica T zawiera słowo $ab|a|a|baaaa|a|a|ab|a|a|ba$, gdzie symbol '|' jedynie pokazuje granice między poszczególnymi słowami, ale w rzeczywistości nie ma go w tablicy. Zatem tablica ta ma długość $N = 17$.

Lewy wskaźnik przesuwa się zgodnie ze strzałkami $\rightarrow$ i $\curvearrowright$, a prawy zgodnie z $\leftarrow$ i $\curvearrowleft$. W grafie G mamy krawędź $(1, 17) \rightsquigarrow (2, 16)$ gdyż 2-gi i 16-ty symbol są identyczne, a przy tym jesteśmy wewnątrz pojedynczych słów. Natomiast krawędź $(2, 16) \rightsquigarrow (5, 13)$ bierze się stąd, że podobnie jak poprzednio, odpowiednie symbole są identyczne oraz pozycje 2 i 16 są końcami pojedynczych słów (czytając od lewej do prawej — pozycja 2 — i od prawej do lewej — pozycja 16), natomiast pozycje 5 i 13 są początkami pojedynczych słów z następnych par w kolejności.

Łatwo zauważyć następujący fakt:

Fakt 1: Liczba wyborów symetrycznych jest równa liczbie wszystkich ścieżek prowadzących od wierzchołka *start* do wierzchołka *koniec* w grafie acyklicznym G.

Liczbę ścieżek liczymy następująco. Przechodzimy graf w porządku topologicznym, to znaczy takim porządku, że dla każdego wierzchołka wszystkie jego poprzedniki są odwiedzane wcześniej niż on sam. Liczymy liczbę ścieżek od wierzchołka *start* do każdego wierzchołka v, jako sumę liczb ścieżek dla jego bezpośrednich poprzedników. Wierzchołek $v = start$ ma jedną ścieżkę od siebie samego do siebie samego.

W programie oprócz tablicy T mamy tablice *nr_slowa* i *skok*, które służą do wskazywania początków i końców pojedynczych słów x_k, y_k w tablicy T oraz pomagają wyznaczać krawędzie grafu G. Sam graf jest przechowywany tylko pośrednio — dla każdego wierzchołka, do którego istnieje ścieżka z wierzchołka *start* pamięta się obliczaną przez program liczbę wszystkich takich ścieżek oraz liczbę poprzedników wierzchołka (wykorzystywaną przy przechodzeniu grafu w porządku topologicznym). Istotnym problemem technicznym jest duży rozmiar tablicy, w której przechowuje się powyższe wartości (tablica G w programie). Wykorzystując fakt, że jest to tablica trójkątna (używamy tylko komórek po jednej stronie przekątnej tablicy), można zmniejszyć rozmiar potrzebnej pamięci. Nie konstruuje się krawędzi grafu. Fakt istnienia krawędzi każdorazowo wyznacza się na podstawie zawartości tablic T, *nr_slowa* i *skok*.

TESTY

Do sprawdzania rozwiązań zawodników użyto 12-tu testów LIC0.IN–LIC11.IN.

- LIC0.IN — test z treści zadania;
- LIC1.IN–LIC8.IN — testy sprawdzające poprawność przyjętej metody rozwiązania;
- LIC9.IN–LIC11.IN — testy sprawnościowe. Ich celem było sprawdzenie szybkości działania algorytmu.

Bogdan S. Chlebus — treść zadania, opracowanie

Krzysztof Sobusiak — program wzorcowy

Skok w bok

Plansza do gry w „Skok w bok” jest nieskończoną taśmą pól, nieograniczoną zarówno w lewo jak i w prawo. Na polach planszy stoją pionki. Ich liczba jest skończona. Na jednym polu może stać wiele pionków jednocześnie. Zakładamy, że pierwsze od lewej pole, na którym jest co najmniej jeden pionek, ma numer 0. Pola na prawo od niego są oznaczone kolejno liczbami naturalnymi 1, 2, 3 itd., a pola w lewo liczbami ujemnymi: -1, -2, -3 itd. Ustawienie pionków na taśmie, które będziemy także nazywać **konfiguracją**, *można opisać w ten sposób, że dla każdego pola, na którym jest co najmniej jeden pionek, podaje się numer pola i liczbę pionków na tym polu.*

Są dwa rodzaje ruchów zmieniających konfigurację: skok w prawo i skok w lewo.

Skok w prawo *polega na zabraniu po jednym pionku z wybranych dwóch sąsiednich pól o numerach p i $p+1$ i dodaniu jednego pionka na polu $p+2$.*

Skok w lewo: *zabieramy jeden pionek z pola $p+2$, a dodajemy po jednym na polach p i $p+1$.*

Mówimy, że konfiguracja jest **końcowa,** *jeśli na dowolnych dwóch sąsiednich polach znajduje się co najwyżej jeden pionek.*

Dla każdej konfiguracji istnieje dokładnie jedna konfiguracja końcowa, którą można z niej otrzymać w wyniku skończonej liczby ruchów w prawo lub w lewo.

Zadanie

Ułóż program, który:

- *wczytuje opis konfiguracji początkowej z pliku tekstowego SKO.IN;*
- *znajduje konfigurację końcową, do jakiej można doprowadzić daną konfigurację początkową i zapisuje wynik w pliku tekstowym SKO.OUT.*

Wejście

W pierwszym wierszu pliku SKO.IN jest zapisana jedna liczba całkowita dodatnia n. Jest to liczba niepustych pól danej konfiguracji początkowej. $1 \leqslant n \leqslant 10000$ (dziesięć tysięcy).

W każdym z kolejnych n wierszy znajduje się opis jednego niepustego pola konfiguracji początkowej w postaci pary liczb całkowitych oddzielonych odstępem. Pierwsza liczba to numer pola, a druga — to liczba pionków na tym polu. Te opisy są uporządkowane rosnąco względem numerów pól. Największy numer pola nie przekracza 10000 (dziesięć tysięcy), a liczba pionków na żadnym polu nie przekracza 10^8 (sto milionów).

Wyjście

W pierwszym wierszu pliku tekstowego SKO.OUT należy zapisać konfigurację końcową, do której można przekształcić daną konfigurację początkową — numery niepustych pól konfiguracji końcowej. Numery te powinne być uporządkowane rosnąco. Liczby w wierszu powinny być pooddzielane pojedynczym odstępem.

PRZYKŁAD

Dla pliku tekstowego SKO.IN:

```
2
0 5
3 3
```

poprawnym rozwiązaniem jest plik SKO.OUT:

```
-4 -1 1 3 5
```

ROZWIĄZANIE

Opis rozwiązania zadania rozpoczniemy od kilku prostych uwag. Chociaż ruchy pionków w grze są opisane w terminach pojedynczych pionków (dla każdego pola gdzie występują zmiany liczby pionków), to efektywne rozwiązanie wymaga symulowania ruchów wieloma pionkami jednocześnie.

Zauważmy, że jeżeli na każdym niepustym polu leży co najwyżej jeden pionek, to taką konfigurację można przekształcić do końcowej wykonując tylko ruchy w prawo (proste ćwiczenie). Jeżeli natomiast wszystkie pionki leżą na jednym polu, to ruch w lewo jest konieczny. Stąd wynika, że algorytm musi przeplatać ciągi ruchów w prawo z ciągami ruchów w lewo.

Spodziewamy się, że zmiany konfiguracji prowadzące do zmniejszania się całkowitej liczby pionków mogą prowadzić do konfiguracji końcowych lub im bliskich. Ruch w prawo zmniejsza liczbę pionków, natomiast ruch w lewo zwiększa. Przed każdym wykonaniem ruchu w lewo musimy więc wykazać ostrożność, aby nie spowodować nadmiernego wzrostu liczby pionków, nawet przy lokalnym uproszczeniu konfiguracji, które w ogólnym rozliczeniu okazać się może pozorne.

JEDNOZNACZNOŚĆ KONFIGURACJI KOŃCOWEJ

Przypominamy pojęcie **liczb Fibonacciego** — tworzą one ciąg nieujemnych liczb całkowitych $\langle F_i \rangle$, dla $i \geq 0$, określony rekurencyjnie w następujący sposób:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_{i+2} &= F_i + F_{i+1} \end{aligned}$$

Określamy **wyrażenia** jako skończone sumy liczb Fibonacciego, zakładając, że te liczby zapisane są zawsze w kolejności niemalejących wskaźników. Rozważamy następujące operacje na takich wyrażeniach:

- F_{i+2} zastępujemy przez sumę $F_i + F_{i+1}$;
- sumę $F_i + F_{i+1}$ zastępujemy przez F_{i+2}.

Takie przekształcanie wyrażeń nie zmienia ich wartości.

Na związek między sumami liczb Fibonacciego a konfiguracjami pionków na planszy wskazuje naturalne podobieństwo między operacjami wykonania skoku w lewo lub w prawo, a operacjami wymiany podwyrażeń $F_i + F_{i+1}$ i F_{i+2}. Jeżeli od jednego wyrażenia można przejść do drugiego przez wykonanie skończonego ciągu takich operacji, to nazwiemy je **podobnymi**. Jeżeli od jednej konfiguracji można przejść do innej przez wykonanie skończonego ciągu ruchów, to także takie konfiguracje nazwiemy **podobnymi**. Chcielibyśmy konfiguracjom przypisać wyrażenia w taki sposób, żeby podobne konfiguracje odpowiadały podobnym wyrażeniom. Tu napotykamy na pewne trudności. Konfiguracje są jednoznacznie określone przez względne położenie pionków na planszy. Pola planszy są ponumerowane tylko w celu wygodnego opisu konfiguracji, natomiast liczby Fibonacciego są określone przez swoje wskaźniki. Innymi słowy: gdy konfigurację przesuniemy o jedno pole w prawo to będzie ona „taka sama", natomiast gdy w sumie liczb Fibonacciego zwiększymy każdy wskaźnik, to dostaniemy większą liczbę (poza przypadkiem wyrazeń F_1 i F_2). W opisie konfiguracji mogą pojawić się pola o numerach ujemnych, natomiast liczby Fibonacciego mają wskaźniki nieujemne. Największą trudnością logiczną jest jednak to, że wykonując odpowiednio wiele ruchów w lewo możemy utworzyć dowolnie wiele pionków na planszy!

Aby uniknąć takich trudności, najprościej rozważyć **grę pomocniczą**, bardzo podobną do właściwej gry „Skok w bok". Przyjmijmy, że plansza jest ciągiem pól nieskończonym tylko w jedną, prawą stronę. Pola są ponumerowane kolejnymi liczbami całkowitymi dodatnimi, począwszy od skrajnie lewego pola, które ma numer 1. Ruchy w prawo można wykonywać z każdego pola, natomiast ruchy w lewo tylko z pól o numerach nie mniejszych niż 3. Dodatkowo wprowadzamy regułę, że pionki między polami 1 oraz 2 można dowolnie przesuwać. Konfiguracji przyporządkowujemy liczbę $\sum_{i\geqslant 1} a_i \cdot F_i$, gdzie a_i jest liczbą pionków na polu i (suma jest skończona, ponieważ tylko skończenie wiele a_i jest niezerowych). Jest jasne, że takie przyporządkowanie sum liczb Fibonacciego konfiguracjom dla gry pomocniczej jest dobre w tym sensie, że konfiguracjom podobnym odpowiadają podobne wyrażenia.

Wiadomo że każdą liczbę całkowitą dodatnią n można jednoznacznie przedstawić w postaci sumy $\sum_{i>1} b_i \cdot F_i$, gdzie $b_i = 0$ poza skończoną liczbą indeksów i, oraz jeżeli $b_i \neq 0$, to $b_i = 1$ i $b_{i-1} = b_{i+1} = 0$ (patrz [13]). Takie sumy to wyrażenia, w których każda liczba Fibonacciego F_i, dla $i > 0$, występuje co najwyżej jeden raz i dwie liczby o sąsiednich indeksach nie występują jednocześnie. Stąd od razu wynika, że dla każdej konfiguracji istnieje dokładnie jedna podobna do niej konfiguracja końcowa w grze pomocniczej.

Jednoznaczność końcowej konfiguracji dla gry „Skok w bok" także zachodzi. Można to pokazać, odwołując się do powyższej własności liczb Fibonacciego (o jednoznaczności przedstawienia liczb naturalnych w postaci odpowiedniego wyrażenia). Szczegóły pozostawiamy jako ćwiczenie. Jako wskazówkę proponujemy takie ponumerowanie pól planszy aby pierwsze od lewej niepuste pole miało numer o tak dużej wartości, żeby w konfiguracji końcowej występowały tylko pola o dodatnich numerach.

ROZŁADOWYWANIE

Pole z co najmniej trzema pionkami nazywamy **wieżą**. Ciąg ruchów, który zmniejsza liczbę pionków danej wieży, jednocześnie zmniejszając łączną liczbę pionków na planszy, nazywamy **rozładowaniem wieży**. Poniżej opisujemy jeden z takich sposobów.

Notacja

$$\ldots, a_1, a_2, \ldots, a_n, \ldots$$

oznacza konfigurację, w której na i-tym polu spośród n kolejnych, znajduje się a_i pionków. Wykonanie ruchu i przejście do następnej konfiguracji zapisujemy jako dwie konfiguracje przedzielone strzałką. Na przykład ruch w prawo możemy zapisać jako

$$\ldots, x+1, y+1, z \ldots \quad \Longrightarrow \quad \ldots, x, y, z+1, \ldots$$

Natomiast ruch w lewo zapisujemy jako

$$\ldots, x, y, z+1 \ldots \quad \Longrightarrow \quad \ldots, x+1, y+1, z, \ldots$$

Rozważmy następujący ciąg trzech kolejnych ruchów:

$$\begin{aligned} \ldots, a, b, c+3, d, e, \ldots \quad &\Longrightarrow \quad \ldots, a+1, b+1, c+2, d, e \ldots \quad \Longrightarrow \\ &\Longrightarrow \quad \ldots, a+1, b, c+1, d+1, e \ldots \quad \Longrightarrow \\ &\Longrightarrow \quad \ldots, a+1, b, c, d, e+1 \ldots \end{aligned}$$

Nazwiemy go **pojedynczym rozładowaniem**. Taka operacja usuwa trzy pionki z pola i umieszcza dwa na innych polach. Jeżeli $3k$ jest maksymalną wielokrotnością 3 nie większą niż liczba pionków na danym polu, to możemy zastosować jednocześnie operację pojedynczego rozładowania do każdej z k trójek, co nazwiemy też **rozładowaniem**. Taka operacja usuwa $3k$ pionków z rozważanego pola i umieszcza po k pionków na dwóch innych polach (razem $2k$).

ALGORYTM OBLICZANIA KONFIGURACJI KOŃCOWEJ

Konfiguracja reprezentowana jest jako skończona lista kolejnych pól. Możemy ograniczyć się tylko do niepustych pól, ale to zależy od szczegółów implementacji. Z polem wiążemy informację o liczbie pionków. Algorytm obliczania konfiguracji końcowej składa się z dwóch faz.

Faza 1

while istnieje pole zawierające co najmniej 3 pionki **do**
   przejrzyj listę niepustych pól w kolejności od lewej do prawej
   i rozładuj każde zawierające co najmniej 3 pionki

Po każdym rozładowaniu cofamy się o dwa pola, żeby sprawdzić czy nie pojawią się tam co najmniej trzy pionki, ale takich pól nie rozładowujemy w tej iteracji pętli.

Niech *pion* będzie funkcją zwracającą liczbę pionków w polu.

Faza 2

pole := numer skrajnie prawego niepustego pola;
pole := $pole - 1$;
while konfiguracja nie jest końcowa **do**
case
  $pion(pole) > 0$ i $pion(pole + 1) > 0$:
    wykonaj ruch w prawo z *pole*;
    pole := $pole + 2$;
  $pion(pole) = 2$ i $pion(pole - 1) > 0$:
    pole := $pole - 1$;
  $pion(pole) = 2$ i $pion(pole + 1) = 0$:
    wykonaj ruch w lewo z *pole*;
    wykonaj ruch w prawo z $pole - 1$;
    pole := $pole + 1$;
  $pion(pole) = 3$:
    rozładuj *pole*;
    pole := $pole + 2$;
  pole jest numerem skrajnie lewego niepustego pola:
    zakończ fazę 2.;
  w pozostałych przypadkach:
    pole := numer następnego niepustego pola z lewej;

Przyjmujemy, że warunki w konstrukcji **case** są sprawdzane po kolei, a po znalezieniu pierwszego spełnionego warunku wykonujemy stojącą przy nim instrukcję (i na tym wykonanie **case** się kończy).

POPRAWNOŚĆ ALGORYTMU

Faza 1. kiedyś się zakończy, bowiem każde rozładowanie pola zmniejsza łączną liczbę pionków na planszy. Faza 2. zaczyna się w sytuacji gdy na każdym polu leżą co najwyżej dwa pionki. Idziemy od strony prawej do lewej starając się, by na prawo była już sytuacja taka, jak w konfiguracji końcowej. Poprawność fazy 2 wynika z zachowania następujących niezmienników po każdej iteracji instrukcji **case**:

- co najwyżej jedno pole zawiera trzy pionki;
- na prawo od *pole* nie ma pola zawierającego więcej niż dwa pionki;
- jeżeli *pole* i $pole + 1$ są niepuste, to na prawo od *pole* nie ma już innych takich par sąsiednich pól.

Formalny dowód przebiega przez indukcję i polega na rozważeniu wszystkich możliwych przypadków.

Niezmienniki z instrukcji **case** pokazują, że nie nastąpi „spiętrzenie" pionków. Jednocześnie zauważmy, że w każdych czterech kolejnych iteracjach instrukcji **case** albo następuje zmniejszenie łącznej liczby pionków albo *pole* przyjmuje wartość mniejszą od wszystkich dotychczasowych.

IMPLEMENTACJA ALGORYTMU

Listę pól pamiętamy w tablicy liczb całkowitych; elementami tablicy są liczby pionków stojących na polach. Tablica ma rozmiar taki, jak podany rozmiar planszy, powiększony z obu końców tak, by pomieścić pola potrzebne do trzymania wież powstałych przy rozładowywaniu wysokich wież znajdujących sie na skraju planszy w początkowej konfiguracji. Rozmiar powiększenia wyraża się logarytmem z maksymalnej dopuszczalnej na wejściu wysokości wieży pionków, ponieważ każde rozładowanie zmniejsza wysokość rozładowywanej wieży o jedną trzecią. W fazie 1 wieże, które pozostają jeszcze do rozładowania trzymamy w kolejce, implementowanej w tablicy w taki sposób, że wieże są pobierane w kolejności ich wstawiania. Program realizujący tę implementację znajduje się w pliku SKO.PAS.

INNE ROZWIĄZANIA

Można rozważać inne rozwiązania. Omówimy algorytm oparty na następującej zasadzie: wykonujemy ruchy w prawo, dopóki to jest możliwe, a gdy nie jest, to wykonujemy ruch w lewo. W takich sytuacjach ruch oznacza skok wieloma pionkami jednocześnie. Skok w prawo najlepiej wykonać maksymalną możliwą liczbą pionków jednocześnie. Przy skoku w lewo dobrze jest tak rozłożyć pionki, aby zapewnić sobie potem długi ciąg ruchów w prawo. Można to osiagnąć stosują zasadę „złotego podziału”. Niech $\phi = (1+\sqrt{5})/2$. Przypuśćmy, że mamy konfigurację

$$\ldots, x, 0, w, 0 \ldots$$

którą nazywamy **izolowaną wieżą**, o ile $w > 1$. Chcielibyśmy wykonać ruch w lewo:

$$\ldots, x, 0, w, 0 \ldots \implies \ldots, x + a \cdot w, a \cdot w, (1-a) \cdot w, 0, \ldots$$

a następnie w prawo:

$$\ldots, a \cdot w, a \cdot w, (1-a) \cdot w, 0, \ldots \implies \ldots, a \cdot w, 0, (1-2a) \cdot w, a \cdot w, \ldots$$

i kontynuować ruchy w prawo. Najdłuższy możliwy ciąg ruchów w prawo otrzymamy, gdy iloraz wysokości dwóch sąsiednich wież zmienia się jak najmniej, to znaczy gdy dwa ułamki

$$\frac{a}{1-a} \quad \text{oraz} \quad \frac{1-2a}{a}$$

są jak najbliższe co do wartości. Rozwiązując równanie otrzymane przez przyrównanie do siebie ułamków dostajemy, że $a = 1 + \phi$. Szybka implementacja takiego algorytmu wygląda następująco. Zaczynając z prawej strony przesuwamy się w lewo szukając pierwszej możliwości skoku w prawo lub izolowanej wieży. Jeżeli skok w prawo jest możliwy, wykonujemy serię takich skoków w prawo „do oporu". Jeżeli napotkamy na izolowaną wieżę, wykonujemy ruch w lewo zgodnie z zasadą złotego podziału, po czym kontynuujemy ruchy w prawo.

Omówimy także krótko pułapki tego zadania. Typowe błędy to:

- przyjęcie, że każda konfiguracja zawsze zmieści się w tablicy pamiętającej konfigurację początkową,
- niewłaściwe przeplatanie skoków w lewo ze skokami w prawo, tak, że doprowadza to do zapętlenia.

Rozwiązania poprawne ale bardzo czasochłonne też nie są dobre. Typowy błąd to wykonywanie skoków tylko jednym pionkiem na raz, co prowadzi do algorytmu o dużej złożoności czasowej. Ciekawszy przykład nieefektywnego rozwiązania to algorytm skonstruowany podobnie do metody złotego podziału, ale rozkładający wieże na trzy prawie równe części, co przy odpowiednio dobranych danych prowadzi do bardzo złego zachowania (ćwiczenie dla czytelnika).

TESTY

Testy do tego zadania znajdują się w plikach SKO0.IN–SKO12.IN. Można je podzielić na testy poprawności i testy złożoności. Testy poprawności:

- SKO0.IN — test z treści zadania;
- SKO1.IN — prosty test wymagający jedynie skoków w prawo;
- SKO2.IN — prosty test wymagający skoków w prawo i w lewo;
- SKO3.IN — konfiguracja końcowa tożsama wejściowej — bez skoków;
- SKO4.IN — prosty test z rozwiązaniem zawierającym pola o numerach ujemnych;
- SKO5.IN — dwie sąsiadujące ze sobą wieże Fibonacciego o wysokościach F_{37} i F_{38}.

Testy złożoności:

- SKO6.IN — pojedyncza wieża wysokości 10^8;
- SKO7.IN — dwie wieże wysokości 10^8 na polach K_0 i K_{10000};
- SKO8.IN — ciąg 10000 małych wież $3, 2, 2, \ldots, 2, 2, 3$;
- SKO9.IN — test losowy: ciąg 100 wież wysokości $\leqslant 10^2$;
- SKO10.IN — test losowy: ciąg 1000 wież wysokości $\leqslant 10^4$;
- SKO11.IN — test losowy: ciąg 10000 wież wysokości $\leqslant 10^8$;
- SKO12.IN — konfiguracja rzadka: 120 wież o wysokości 10^8 w odstępach co 80 pól.

Stanisław Waligórski **Jacek Chrząszcz**
treść zadania program wzorcowy, opracowanie

Tanie podróże

Pasażerowie autokarów turystycznych, przewożących wycieczki transkontynentalną autostradą, spędzają w drodze wiele dni, zatem opłaty za noclegi stanowią poważną część kosztów podróży. Ze względu na bezpieczeństwo jazdy i wygodę pasażerów każdy autokar jedzie tylko w ciągu dnia i nie może przejechać więcej niż 800 km dziennie. Noc na trasie (poza jej początkiem i końcem) pasażerowie i kierowca spędzają w hotelach.
Dotychczas planowano przejazdy tak, by liczba noclegów na trasie była jak najmniejsza. Dążąc do obniżki kosztów przedsiębiorstwo przewozowe zdecydowało zbadać, czy opłaci się układanie planów podróży tak, by suma opłat za noclegi była możliwie najniższa, nawet gdyby to miało przedłużyć podróż. W tych obliczeniach można korzystać z ofert hoteli położonych przy autostradzie. W każdej ofercie jest podana odległość hotelu od początku trasy i cena jednego noclegu jednej osoby.
Podróż jest tylko w jedną stronę. Trasa nie ma rozgałęzień. Przez każdy punkt trasy autokar przejeżdża tylko jeden raz. Nigdzie na trasie nie ma dwóch hoteli w jednym punkcie, więc dla zidentyfikowania hotelu wystarczy podać jego odległość od początku trasy. Nie planuje się noclegu na początku ani na końcu trasy. Liczba osób w autobusie nie zmienia się i w każdym hotelu wszyscy (łącznie z kierowcą) płacą za nocleg jednakowo — zgodnie z ofertą. Pojemność hoteli jest na tyle duża, że nie istnieje problem braku miejsc. Zawsze można liczyć na to, że w dowolnej chwili w każdym hotelu będzie wystarczająco dużo wolnych miejsc, by przenocować wszystkich pasażerów autobusu.
Na każdym odcinku trasy o długości 800 km jest przynajmniej jeden hotel, co oznacza, że przejechanie trasy z zachowaniem podanych powyżej warunków jest możliwe.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego TAN.IN dane: długość trasy, liczbę hoteli oraz oferty hoteli;*
- *znajduje dwa plany podróży:*
  - **najtańszej**, *tzn. takiej, żeby suma opłat za hotele była najmniejsza, a jeśli takich rozwiązań jest wiele, wybiera jedno z najmniejszą liczbą noclegów,*
  - **najszybszej**, *tzn. takiej, żeby liczba noclegów była najmniejsza, a jeśli takich rozwiązań jest wiele, wybiera jedno z najmniejszą sumą opłat za noclegi;*
- *zapisuje wyniki, tj. dwa plany podróży — najtańszej i najszybszej, w pliku tekstowym TAN.OUT.*

Wejście

W pierwszym wierszu pliku tekstowego TAN.IN są zapisane dwie liczby całkowite dodatnie, oddzielone pojedynczym odstępem: długość trasy d w kilometrach oraz liczba hoteli h, gdzie $d \leqslant 16000$, $h \leqslant 1000$.

W kolejnych h wierszach są zapisane oferty hoteli — każda oferta w osobnym wierszu. Są one uporządkowane według rosnącej odległości hoteli od początku trasy. Każda oferta jest zapisana w postaci dwóch liczb całkowitych dodatnich, oddzielonych pojedynczym odstępem, pierwsza liczba — to odległość hotelu od początku trasy w kilometrach, a druga — to cena jednego noclegu w tym hotelu nie większa niż 1000.

Wyjście

W pierwszym wierszu pliku tekstowego TAN.OUT należy zapisać plan podróży najtańszej — odległość kolejnych miejsc noclegu od początku trasy.
W drugim wierszu należy — w taki sam sposób — zapisać plan podróży najszybszej.
Liczby w wierszu powinny być pooddzielane pojedynczym odstępem.

Przykład

Dla pliku TAN.IN

```
2000 7
100 54
120 70
400 17
700 38
1000 25
1200 18
1440 40
```

poprawnym rozwiązaniem jest plik TAN.OUT mający dwa identyczne wiersze:

```
400 1200
400 1200
```

Twój program powinien szukać pliku TAN.IN w katalogu bieżącym i tworzyć plik TAN.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę TAN.???, gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonalnej powinien być zapisany w pliku TAN.EXE

ROZWIĄZANIE

Przy omawianiu zadania będą używane następujące oznaczenia:

- D — długość trasy,
- H — liczba hoteli na trasie,
- M — maksymalny zasięg dzienny,
- o_i — odległość i-tego hotelu od początku trasy,
- c_i — cena noclegu w i-tym hotelu,

Hotele będą numerowane od 1 do H, początek trasy będzie traktowany jak hotel numer 0, a koniec — jak hotel numer $H+1$. Oczywiście $c_0 = c_{H+1} = 0$, $o_0 = 0$, a $o_{H+1} = D$.

Najlepszą metodą rozwiązania obu problemów zadania jest programowanie dynamiczne. Dla każdego hotelu i liczony jest następnik n_i oraz wartość k_i — para (liczba noclegów, pieniądze) — minimalny koszt dojazdu od niego do końca trasy. Dla hotelu końcowego następnik jest nieokreślony, a koszt $k_{H+1} = (0,0)$. Dla każdego hotelu i od H do 0, wartość k_i liczy się jako minimalny koszt dojazdu od pierwszego noclegu do końca, powiększony o koszt spędzenia nocy w hotelu i, co się wyraża wzorem:

$$k_i = \min_{j:\ i<j\leq H+1 \ \wedge\ o_j - o_i \leqslant M} k_j + (1, c_i)$$

przy czym minimum jest liczone względem współrzędnej odpowiadającej wersji zadania (liczba noclegów lub pieniądze). Następnik n_i to oczywiście ten hotel z rozpatrywanego zbioru, którego koszt był minimalny. Wartość $k_0 - (1,0)$ to wartość minimalnego kosztu przejazdu*, a ciąg $(n_0, n_{n_0}, \ldots)$ jest poszukiwanym rozwiązaniem. Wszystkie operacje na parach dokonujemy po współrzędnych.

W zależności od wersji szukanej trasy (najtańsza, najkrótsza) zmienia się tylko sposób obliczania funkcji minimum. Ponieważ kolejno liczone od końca wartości k_i są zapamiętywane, dla każdego hotelu wystarczy przejrzeć tylko te hotele, które są od niego oddalone nie więcej niż o 800 km, a jest ich nie więcej niż 800. Dzięki temu pesymistyczna złożoność czasowa tego algorytmu wynosi $O(HM)$.

Można by jeszcze pokusić się o zmniejszenie złożoności czasowej do $O(H \log M)$. Zbiór $\{j : i < j \leqslant H+1 \wedge o_j - o_i \leq M\}$, z którego wybiera się kolejne minima, modyfikuje się bardzo nieznacznie przy zmianie i, a więc gdyby go zaimplementować np. jako drzewo AVL, czasy wyboru minimum, usunięcia hoteli „wystających” poza 800 km, wstawienia nowego hotelu, byłyby rzędu $O(\log M)$. Ale ponieważ zgodnie z treścią zadania M jest stałe, a narzut na wykonywanie operacji na drzewie AVL byłby niemały, więc zmiana ta wcale nie musiałaby spowodować istotnego przyśpieszenia.

Minimalna złożoność pamięciowa tego problemu wynosi oczywiście $\Theta(H)$, gdyż dla każdego hotelu musimy zapamiętać przynajmniej jego odległość i cenę. Rozwiązanie opisane powyżej ma właśnie złożoność liniową. (Zobacz program TAN.PAS na dołączonej dyskietce.)

Rozwiązanie to można by zrealizować za pomocą procedur rekurencyjnych z zapamiętywaniem wyników poprzednich wywołań.

Do rozwiązania tego zadania można użyć wielu znanych algorytmów grafowych. Traktują one trasę z hotelami, jak graf i wyszukują w nim odpowiednie ścieżki. Wierzchołkami grafu są oczywiście wszystkie hotele oraz punkt początkowy i końcowy, a skierowanymi krawędziami połączone są tylko te hotele, których wzajemna odległość nie przekracza 800 km. Rozwiązaniem zadania będą dwie ścieżki łączące wierzchołek początkowy z końcowym: pierwsza — o najmniejszej długości, a druga — o najmniejszej sumie cen noclegów. Skuteczność programów zależy od użytego algorytmu szukania optymalnej ścieżki. Należy przy tym tak zaprojektować algorytm, by nie trzeba było liczyć i zapamiętywać wszystkich krawędzi, gdyż ich liczba może być bardzo duża!

* Trzeba odjąć nocleg w hotelu 0, który się nie liczy.

Na podstawie algorytmu wyszukiwania najkrótszych ścieżek, opisanego np. w książce [10], można skonstruować rozwiązanie równie skuteczne, co programowanie dynamiczne. Tutaj również liczony jest koszt dojazdu od danego hotelu do końca trasy, ale w odwrotny sposób.

Podobnie, jak w przedstawionym rozwiązaniu, z każdym* hotelem i związane są dwie wartości: tymczasowy koszt minimalny dojazdu do końca k_i — para (liczba noclegów, pieniądze), początkowo $(+\infty, +\infty)$ — oraz następnik n_i — początkowo nieokreślony. Następnie końcowi trasy — hotelowi numer $H+1$ — nadane są wartości początkowe (0,0). Potem dla każdego hotelu i od $H+1$ do 1, bada się wszystkie jego poprzedniki j, czy dotychczasowa wartość kosztu minimalnego k_j nie jest większa od kosztu jaki byłby potrzebny na przejazd, gdyby pierwszy nocleg był spędzony w hotelu i. Po dojechaniu do hotelu 1 wartość $k_0 - (1, 0)$, podobnie jak w rozwiązaniu wzorcowym, będzie minimalnym kosztem dojazdu od początku trasy do końca, a ciąg $(n_0, n_{n_0}, \ldots)$ — ciągiem numerów hoteli na optymalnej trasie.

Oczywiście wyłącznie od wyboru metody porównywania kosztów zależy, czy znalezione rozwiązanie będzie najtańszym czy najkrótszym. Koszt czasowy tego rozwiązania będzie wynosił $O(HM)$ i tym razem nie da się go w łatwy sposób poprawić. Koszt pamięciowy oczywiście jest również $O(H)$.

Metoda płonącego lasu

Ta metoda opiera się na odmiennej filozofii od rozwiązań poprzednich. Rozważamy tu największy zbiór hoteli, z których można dojechać do końca trasy za określoną pulę noclegów i pieniędzy. Następnie stopniowo powiększamy tę pulę, aż wierzchołek końcowy wpadnie do rozważanego zbioru.

Podobnie jak w dwóch poprzednich rozwiązaniach, dla każdego hotelu i określamy tymczasowy minimalny koszt (liczba noclegów, pieniądze) dojazdu od niego do końca trasy k_i, na początku równy $(+\infty, +\infty)$ dla każdego wierzchołka, oprócz wierzchołka końcowego, dla którego $k_{H+1} = (0, 0)$. Dla każdego wierzchołka i, n_i oznacza jego następnika (na początku nieokreślony). Określamy zbiór Q hoteli o ustalonym minimalnym koszcie dojazdu do końca, który na początku jest pusty.

Do zbioru Q dorzucamy ten wierzchołek $i \notin Q$, którego koszt k_i jest najmniejszy (jako pierwszy oczywiście zostanie dorzucony $H+1$). Dla wszystkich jego poprzedników j (hoteli odległych o nie więcej niż 800 km) weryfikuje się wartości k_j i n_j, sprawdzając, czy koszt dojazdu z wierzchołka j do końca z pierwszym noclegiem w dorzuconym właśnie wierzchołku i nie byłby niższy od dotychczasowej wartości. Powyższy krok powtarza się tak długo, aż wierzchołek początkowy 0 zostanie dorzucony do zbioru Q. Wówczas, podobnie jak dotąd, ciąg $(n_0, n_{n_0}, \ldots)$ to lista hoteli na ścieżce od 0 do $H+1$ o najmniejszym koszcie. Szczegółowy opis oraz dowód poprawności przedstawionego właśnie **algorytmu Dijkstry** można znaleźć w książce [10].

Powyższy algorytm stosuje się oczywiście do obu wersji, w zależności od sposobu porównywania par (noclegi, pieniądze). Jego pesymistyczna złożoność czasowa wynosi

* Fakt, że hotele są tu w kolejności jest istotny. W ogólności pierwszym krokiem algorytmu jest sortowanie topologiczne wierzchołków grafu.

$O(H^2M)$, ale dałoby się ją poprawić. Użycie bardziej skomplikowanych struktur danych do przechowywania zbioru Q jako kolejki priorytetowej zmniejszyłoby złożoność do $O(MH \log H)$. Złożoność pamięciowa jest rzędu H, czyli optymalna.

Przeszukiwanie grafu wszerz i w głąb

Jeśli graf nie jest fizycznie konstruowany, to algorytmy te odpowiadają rozwiązaniom, w którym w ogóle nie występuje pojęcie grafu. Przeszukiwanie w głąb odpowiada przeglądaniu wszystkich lub niektórych ścieżek, a przeszukiwanie wszerz — jakiejś wersji programowania dynamicznego

Następniki czy poprzedniki?

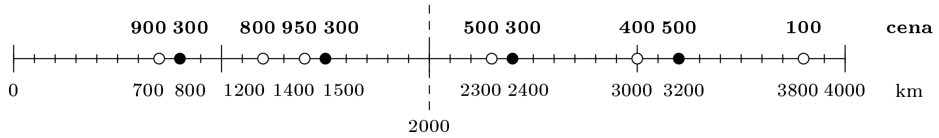
Podane dotychczas rozwiązania mogłyby zamiast następników i kosztów dojazdu od początku trasy liczyć dla każdego hotelu jego poprzednik oraz koszt dojazdu do końca trasy. Jednak musielibyśmy wtedy odwrócić obliczony ciąg poprzedników, czyli potrzebna byłaby dodatkowa pętla.

Badanie wszystkich możliwych rozkładów

Najbardziej oczywiste rozwiązanie polegało oczywiście na badaniu wszystkich możliwych rozkładów. Niestety liczba tych rozkładów może być tak wielka, że rozwiązanie takie jest nieakceptowalne, niezależnie czy implementacja będzie rekurencyjna czy iteracyjna. Pomimo ograniczenia liczby badanych rozkładów niezadowalające są również ulepszone wersje, np. branie pod uwagę rozkładów o niezbyt dużej liczbie noclegów (nie ma sensu nocować więcej niż $2 * D/M$ razy), lub wyłącznie takich rozkładów noclegów $(i_1, i_2, \ldots, i_n)$, w których dla dowolnego j mamy $o_{i_{j+1}} - o_{i_{j-1}} > 800$ (nie zatrzymujemy się, jeżeli tego samego dnia można dojechać do następnego planowanego miejsca postoju).

Podział na podzadania

Dobrą metodą rozwiązania wielu podobnych na pierwszy rzut oka zadań jest podział całości na części i rozwiązywanie każdej z nich oddzielnie, a następnie połączenie rozwiązań poszczególnych części. Jednak w tym zadaniu, metoda ta nie przynosiła oczekiwanych rezultatów. Na rysunku poniżej optymalne noclegi dla każdej z połówek są oznaczone na czarno, a optymalny rozkład dla całej trasy to noclegi czarne z części lewej i białe z prawej:



TESTY

Rozwiązania zawodników sprawdzano za pomocą trzech rodzajów testów: testów o niewielkiej liczbie hoteli (TAN0.IN–TAN5.IN), testów do badania przypadków granicznych (TAN6.IN–TAN8.IN) oraz testów trudnych z długimi trasami i dużą liczbą hoteli (TAN9.IN–TAN11.IN).

- TAN0.IN — przykład z treści zadania
- TAN1.IN — test pokazujący, że nie zawsze z rozwiązań częściowych da się w prosty sposób złożyć rozwiązanie całości.
- TAN2.IN — test mający na celu odrzucenie programów przeszukujących wszystkie ścieżki (ponad 10^{20}, gdy ilość ścieżek minimalnych dla tego testu wynosi 10^7);
- TAN3.IN — test na poprawność implementacji algorytmu dla dużych liczb;
- TAN4.IN — test poprawnościowy;
- TAN5.IN — test poprawnościowy; optymalny rozkład noclegów obejmuje same najdroższe hotele;
- TAN6.IN — test mający na celu odrzucenie rozwiązań budujących w pamięci graf hoteli (ponad 400 tys. połączeń).
- TAN7.IN — test poprawnościowy; na trasie o długości 700 km znajduje się 500 hoteli, w żadnym nie trzeba nocować;
- TAN8.IN — test poprawnościowy; jedyny hotel w środku trasy o długości 1600 km;
- TAN9.IN–TAN11.IN — testy wydajnościowe.

Tomasz Śmigielski **Marek Pawlicki**
treść zadania program wzorcowy, opracowanie

Bramki XOR

Każda **bramka XOR** *ma dwa wejścia i jedno wyjście, a jej działanie opisuje następująca tabelka:*

wejście 1	*wejście 2*	*wyjście*
0	*0*	*0*
0	*1*	*1*
1	*0*	*1*
1	*1*	*0*

Siecią XOR *nazywamy układ bramek XOR, mający n wejść i jedno wyjście, spełniający następujące warunki:*

(1) Każde wejście sieci XOR jest połączone z przynajmniej jednym wejściem bramki.

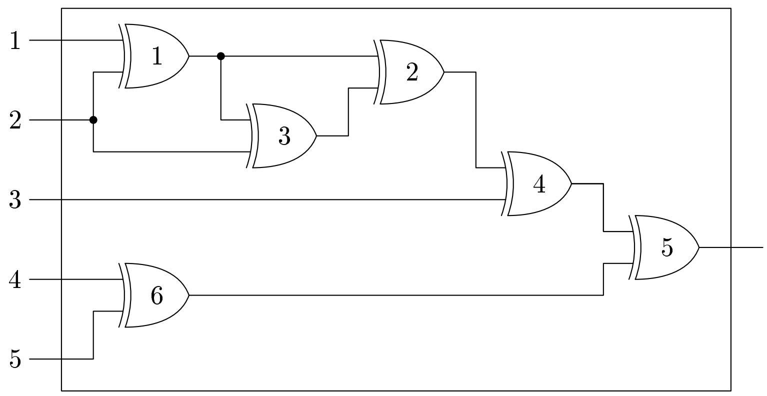
(2) Każde wejście każdej bramki jest połączone z jednym wejściem sieci albo z jednym wyjściem innej bramki.

(3) Wyjście dokładnie jednej bramki jest połączone z wyjściem sieci.

(4) Każde wyjście bramki w sieci jest połączone z przynajmniej jednym wejściem innej bramki albo z jedynym wyjściem sieci.

(5) Istnieje taka numeracja bramek, że do każdego wejścia dowolnej bramki jest podłączone wejście sieci albo wyjście bramki o mniejszym numerze.

Przykład



Przedstawiony na rysunku układ 6 bramek mający 5 wejść i 1 wyjście spełnia warunki 1–5, więc jest siecią XOR.

UWAGA: *Bramki na rysunku zostały ponumerowane dowolnie, ale istnieje numeracja spełniająca warunek określony w punkcie 5.*

Wszystkie wejścia sieci są ponumerowane od 1 do n. Stan wejść sieci XOR opisuje **słowo wejściowe** *utworzone z n cyfr dwójkowych 0 i 1 — przyjmujemy, że i-ta od lewej cyfra danego słowa wejściowego, to stan i-tego wejścia sieci. Dla dowolnego stanu wejść sieć daje na wyjściu 0 albo 1. Każde słowo wejściowe jest dwójkowym zapisem jakiejś liczby naturalnej, więc słowa te można uporządkować zgodnie z ich wartościami liczbowymi. Sieci XOR będziemy testowali podając na wejściu kolejne słowa z ustalonego zakresu i zliczając liczbę otrzymanych w wyniku jedynek.*

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego XOR.IN opis sieci XOR: liczbę wejść n, liczbę bramek m, numer bramki połączonej z wyjściem sieci oraz opisy połączeń, a następnie dwa n-bitowe słowa wejściowe — dolne i górne ograniczenie zakresu, w jakim będziemy testowali sieć,*
- *oblicza liczbę jedynek otrzymanych na wyjściu sieci dla słów wejściowych z danego zakresu,*
- *zapisuje wynik w pliku tekstowym XOR.OUT.*

Zakładamy, że $3 \leqslant n \leqslant 100$, $3 \leqslant m \leqslant 3000$ oraz, że bramki danej sieci są ponumerowane w dowolnym porządku liczbami od 1 do m.

Wejście

W pierwszym wierszu pliku tekstowego XOR.IN są zapisane trzy liczby całkowite dodatnie pooddzielane pojedynczym odstępem. Jest to liczba wejść n danej sieci XOR, liczba bramek m oraz numer bramki połączonej z wyjściem sieci.
W kolejnych m wierszach znajdują się opisy połączeń bramek sieci. W i-tym z tych wierszy, dla i od 1 do m, znajduje się opis połączeń dwóch wejść bramki o numerze i, który ma postać dwóch liczb całkowitych nie mniejszych niż $-n$ i nie większych niż m, oddzielonych pojedynczym odstępem. Jeśli odpowiednie wejście do bramki jest połączone z wejściem do sieci o numerze k, to opisem tego połączenia jest liczba ujemna $-k$, a jeśli wejście do bramki jest połączone z wyjściem innej bramki o numerze j, to opisem tego połączenia jest liczba dodatnia j.
W kolejnych 2 wierszach pliku tekstowego XOR.IN są zapisane dwa n-bitowe słowa $\underline{a}$ oraz $\underline{b}$. Jest to dolne i górne ograniczenie zakresu testowania sieci. Zakładamy, że w danym zakresie mieści się nie więcej niż sto tysięcy słów.

Wyjście

W pliku tekstowym XOR.OUT należy zapisać jedną liczbę całkowitą nieujemną — liczbę jedynek, jakie powinniśmy otrzymać na wyjściu poprawnie działającej danej sieci XOR dla słów wejściowych $\underline{s}$ z danego zakresu $\underline{a} \leqslant \underline{s} \leqslant \underline{b}$, gdzie nierówność $\leqslant$ należy rozumieć jako relację porządku zgodnego z wartościami liczbowymi słów dwójkowych.

Przykład

Dla pliku XOR.IN, zawierającego opis przedstawionej powyżej sieci XOR:

```
5 6 5
-1 -2
1 3
1 -2
2 -3
4 6
-4 -5
00111
01110
```

poprawnym rozwiązaniem jest następujący plik XOR.OUT:

```
5
```

Twój program powinien szukać pliku XOR.IN w katalogu bieżącym i tworzyć plik XOR.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę XOR.???, gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonywalnej powinien być zapisany w pliku XOR.EXE

ROZWIĄZANIE

Kluczem do efektywnego rozwiązania jest znalezienie zależności pomiędzy wartościami na wejściu sieci, a wynikiem na jej wyjściu. W tym celu przyjrzyjmy się bliżej działaniu bramki XOR. Niech $\oplus$ oznacza dodawanie liczb całkowitych modulo 2, tzn. takie, że jego wynikiem jest reszta, jaką daje suma ze zwykłego dodawania przy dzieleniu przez 2. Tak określone działanie jest przemienne i łączne, więc możemy go używać analogicznie do zwykłego dodawania. Stan na wyjściu bramki (lub inaczej: liczbę dwójkową obliczaną przez bramkę) można teraz wyrazić jako $a \oplus b$, gdzie a i b oznaczają wartości na jej wejściach. W ogólności, jeżeli przez $x_1, \ldots, x_n$ oznaczymy stany wejść sieci, to stan na wyjściu każdej z bramek zadanej sieci można wyrazić jako wartość funkcji postaci

$$f(x_1, \ldots, x_n) = a_1 x_1 \oplus a_2 x_2 \oplus \cdots \oplus a_n x_n \tag{1}$$

gdzie zapis $a_1 x_1$ oznacza zwykłe mnożenie, a $a_1, \ldots, a_n \in \{0, 1\}$ są współczynnikami zależnymi od położenia bramki w sieci. W szczególności stan na wyjściu bramki połączonej z wyjściem sieci jest także opisany powyższą zależnością, a jej funkcję nazywamy **wielomianem sieciowym**. Dla przykładowej sieci z treści zadania wielomiany dla odpowiednich bramek mają postać (f_i oznacza wielomian dla i-tej bramki):

$$\begin{aligned}
f_1(x_1, \ldots, x_5) &= x_1 \oplus x_2 \\
f_2(x_1, \ldots, x_5) &= x_2 \\
f_3(x_1, \ldots, x_5) &= x_1 \\
f_4(x_1, \ldots, x_5) &= x_2 \oplus x_3 \\
f_5(x_1, \ldots, x_5) &= x_2 \oplus x_3 \oplus x_4 \oplus x_5 \\
f_6(x_1, \ldots, x_5) &= x_4 \oplus x_5
\end{aligned}$$

Jak znaleźć wielomian sieciowy? Po przyjrzeniu się postaci wielomianu (1) łatwo zauważyć, że dla $x_i = 1$ oraz $x_j = 0$ dla wszystkich $j \neq i$ wartość na wyjściu sieci jest tożsama ze współczynnikiem a_i jej wielomianu. Aby wyznaczyć cały wielomian, wystarczy obliczyć, jakie wyniki daje sieć na wyjściu dla n słów wejściowych o postaci podanej wyżej dla $i = 1, \ldots, n$. Powyższe obliczenie można prosto zaimplementować w postaci procedury rekurencyjnej wyznaczającej wartość na wyjściu bramki na podstawie wartości na wyjściach bramek podłączonych do jej wejść lub wartości na wejściach sieci. Przyjmijmy następujące globalne deklaracje:

```
const
  MaxLiczbaBramek = 3000;
  MaxLiczbaWejsc = 100;
type
  { Stan binarny }
  TStan = 0..1;
  { Element oznaczający bramkę lub wejście sieci }
  TElement = record
    { Numery bramek podłączonych do wejśc; nieistotne w przypadku wejść sieci }
    Wejscie1, Wejscie2 : Integer;
    { Stan na wyjściu bramki lub stan wejścia sieci }
    Wyjscie : TStan;
    { Oznacza, czy wartość Wyjscie jest wyznaczona }
    WyjscieObliczone : Boolean
  end;
  { Sieć złożona z bramek i wejść }
  TSiec = array [-MaxLiczbaWejsc..MaxLiczbaBramek] of TElement;
  { Współczynniki wielomianu lub stan wejść }
  TWielomian = array [1..MaxLiczbaWejsc] of TStan;
var
  S : TSiec; { Cała sieć }
  N : Integer; { Liczba wejść }
  M : Integer; { Liczba bramek }
  WyjscieSieci : Integer; { Numer bramki podłączonej do wyjścia sieci }
```

Elementy $S[-N]..S[-1]$ opisują odpowiednie wejścia sieci, natomiast elementy $S[1]..S[M]$ przechowują dane o bramkach sieci. Poniższa procedura oblicza wartość na wyjściu bramki o numerze *nr* dla danego stanu wiejść $S[-N]..S[-1]$:

```
procedure ObliczBramke (nr : Integer);
var
  We1, We2 : Integer;
begin
  We1 := S[nr].Wejscie1; We2 := S[nr].Wejscie2;
  { Oblicz wartości wyjściowe bramek podłączonych do wejść,
    jeżeli nie są jeszcze wyznaczone }
  if not S[We1].WyjscieObliczone then ObliczBramke(We1);
  if not S[We2].WyjscieObliczone then ObliczBramke(We2);
```

```
  { Oblicz wartość na wyjściu }
  S[nr].Wyjscie := (S[We1].Wyjscie + S[We2].Wyjscie) mod 2;
  { Zaznacz, że wartość na wyjściu wyznaczona }
  S[nr].WyjscieObliczone := true
end;
```

Zauważmy, że dzięki pamiętaniu raz obliczonej wartości na wyjściu bramki, liczba kroków obliczeń wykonywanych przy wywołaniu *ObliczBramke*(*WyjscieSieci*) będzie proporcjonalna do liczby bramek m. Zauważmy także, że dzięki warunkowi 5. dla sieci XOR z treści zadania, takie wywołanie zakończy się sukcesem (nie będzie zapętlenia). Możemy teraz, wykorzystując procedurę `ObliczBramke`, łatwo wyznaczyć wielomian sieciowy:

```
var
   WielomianSieci : TWielomian; { Współczynniki wielomianu sieci }
  for i := 1 to N do begin
  { S[-N].Wyjscie..S[-1].Wyjscie := 0, S[-i].Wyjscie := 1 }
  { S[-N].WyjscieObliczone..S[-1].WyjscieObliczone := true }
  { S[1].WyjscieObliczone..S[M].WyjscieObliczone := false }
  { Oblicz wartość na wyjściu sieci... }
  ObliczBramke(WyjscieSieci);
  { ...która jest i-tym współczynnikiem }
  WielomianSieci[i] := S[WyjscieSieci].Wyjscie
end
```

Dobrze zaimplementowany powyższy fragment programu wykonuje liczbę operacji rzędu $m \cdot n$. Obliczenie liczby jedynek dla zadanego przedziału danych wejściowych wydaje się teraz proste:

```
var
  Dane, Koniec : TWielomian; { Słowa z pliku wejściowego }
  LiczbaJedynek : Longint; { Zlicza jedynki }
...
  LiczbaJedynek := 0;
  { Wczytaj wartości graniczne z pliku wejściowego na zmiene Dane i Koniec }
  LiczbaJedynek := LiczbaJedynek + WartoscWielomianu(WielomianSieci, Dane);
  while not RowneDane(Dane, Koniec) do begin
    KolejnyZestaw(Dane);
  LiczbaJedynek := LiczbaJedynek + WartoscWielomianu(Dane)
end;
```

Po uzupełnieniu implementacji procedur: *WartoscWielomianu* — obliczającej wartość wielomianu dla zadanego słowa, *RowneDane* — stwierdzającej, czy dwa słowa są równe, *KolejnyZestaw* — generującej kolejny zestaw wartości wejściowych zgodnie z porządkiem liczb dwójkowych oraz po zaimplementowaniu wczytania danych o sieci, otrzymujemy gotowy program, który można znaleźć na dołączonej dyskietce.

Złożoność czasowa przedstawionego rozwiązania jest rzędu $mn+ln$, gdzie l oznacza liczbę zestawów wejściowych, natomiast wymagania pamięciowe są rzędu $m+n$.

Możemy sobie wyobrazić, że procedura rekurencyjna podobna do *ObliczBramke* będzie wyznaczać nie wartość na wyjściu bramki lecz cały jej wielomian, na podstawie wielomianów bramek podłączonych do jej wejść lub wielomianów odpowiadającym wejściom sieci. Oczywiście wielomianem odpowiadającym i-temu wejściu sieci jest x_i. Złożoność czasowa rozwiązania opartego na tym pomyśle jest tego rzędu, co złożoność rozwiązania poprzedniego, lecz potrzebna pamięć jest rzędu mn na przechowywanie wielomianów.

Jeżeli wyeliminujemy pamiętanie wyników pośrednich (wyjście bramki lub wielomian), przy zastosowaniu procedury podobnej do *ObliczBramke* otrzymamy nieco prostszy algorytm, za to bardzo nieefektywny (dane dla jednej bramki wyliczane byłyby wielokrotnie).

Dane dla bramki (wyjście bramki lub wielomian) można wyznaczyć w inny sposób niż przez stosowanie rekurencji. W tym celu dla bramek zadanej sieci zdefiniujmy relację $<$ (mniejsza) w ten sposób, że bramka B_1 jest mniejsza od bramki B_2 ($B_1 < B_2$), jeżeli wyjście bramki B_1 podłączone jest do jednego z wejść bramki B_2. Zgodnie z warunkiem 5. dla sieci XOR z treści zadania, jej bramki można ustawić w ciąg tak, aby na prawo od danej bramki w ciągu nie występowały bramki od niej mniejsze. Łatwo zauważyć, że na początku ciągu będzie bramka, której oba wejścia są podłączone do wejść sieci, a na końcu ta, której wyjście jest podłączone do wyjścia sieci. Znalezienie takiego uporządkowania nazywa się **sortowaniem topologicznym**. Z odpowiednim algorytmem i jego omówieniem Czytelnik może się zapoznać np. w książce [21]. Gdy już znajdziemy takie uporządkowanie, wystarczy obliczyć kolejno wartości na wyjściach dla każdej bramki w ciągu. Złożoność czasowa rozwiązania opartego na tym pomyśle jest taka sama, jak rozwiązania pierwszego (!), a złożoność pamięciowa zależy od tego, czy obliczamy wielomian sieci przez symulację, czy przez wyznaczanie wielomianów dla każdej bramki.

Możemy wreszcie zrezygnować z wyznaczania wielomianu sieci i obliczać wartość na jej wyjściu, dla każdych danych wejściowych, stosując rekurencję z pamiętaniem wyników pośrednich lub bez, bądź też sortowanie topologiczne. Jednak złożoność czasowa takich rozwiązań będzie przynajmniej rzędu lm, a ponieważ m może być dużo większe niż n, przy dużych l rozwiązania takie będą znacznie wolniejsze niż rozwiązanie zaproponowane jako pierwsze.

TESTY

Do sprawdzania rozwiązań zawodników użyto 10 testów XOR0.IN–XOR9.IN. Testy można podzielić na dwie grupy: testy sprawdzające szczególne przypadki oraz testy sprawdzające szybkość działania zastosowanego algorytmu. Oto krótki ich opis:

- XOR0.IN — test z treści zadania;

- XOR1.IN — test sprawdzający zachowanie programu, gdy występują w sieci bramki o obu wejściach podłączonych do tego samego punktu;
- XOR2.IN — dla tego testu odpowiedzią jest 0 jedynek;
- XOR3.IN — dla tego testu odpowiedź przekracza zakres typu **Word**.
- XOR4.IN — prosty test sprawdzający poprawność użytej metody;

Testy XOR5.IN–XOR9.IN to testy złożnościowe o wzrastającym stopniu trudności.

Zawody II stopnia

opracowania zadań

Grzegorz Jakacki **Albert Krzymowski**
treść zadania, opracowanie program wzorcowy

Addon

Addon, nowo odkryty pierwiastek promieniotwórczy jest najwydajniejszym ze znanych paliw jądrowych. Trwają prace nad uruchomieniem energetycznego reaktora addonowego.
Projekt przewiduje, że komora paliwowa reaktora będzie miała postać pionowej rury. W komorze, jeden na drugim będą ustawione pręty paliwa, czyli walce z addonu. Pręty paliwowe będą produkowane w różnych długościach.
Cykl pracy reaktora rozpoczyna się od wstawienia paliwa do komór. Kolejnym krokiem jest zapłon paliwa. Niestety, wysokość słupa paliwa biorącego udział w reakcji nie może być dowolna, ponieważ tylko dla niektórych wysokości reakcja przebiega bezpiecznie. Wysokości te nazywamy **wysokościami stabilnymi***.*
Projektanci reaktora mają dwa zadania: ustalić wysokość komory paliwowej oraz dobrać zestaw długości, w jakich będą produkowane pręty addonu.
Mówimy, że zestaw długości jest **bezpieczny** *dla danej komory, jeżeli wysokość dowolnego słupa, jaki można ustawić w komorze z prętów o długościach z tego zestawu, jest stabilna.*
Mówimy, że zestaw długości jest **pełny** *dla danej komory, jeżeli z prętów o długościach z tego zestawu można ustawić każdy słup o stabilnej wysokości nie większej niż wysokość tej komory.*

Zadanie

Ułóż program, który:

- *wczytuje z pliku tekstowego ADD.IN zbiór wysokości stabilnych,*
- *oblicza maksymalną wysokość komory, dla której istnieje zestaw długości jednocześnie bezpieczny i pełny,*
- *znajduje, dla takiej komory, zestaw długości bezpieczny i pełny mający minimalną liczbę elementów,*
- *zapisuje wyniki w pliku tekstowym ADD.OUT*

Wejście

W pierwszym wierszu pliku tekstowego ADD.IN jest zapisana liczba naturalna $1 \leqslant n \leqslant 10000$*.* *Jest to liczba danych wysokości stabilnych.*
W każdym z n kolejnych wierszy jest zapisana jedna liczba całkowita dodatnia nie większa niż 1000. Są to dane wysokości stabilne zapisane w porządku rosnącym.

Wyjście

W pierwszym wierszu pliku ADD.OUT należy zapisać jedną liczbę — maksymalną wysokość komory.
W kolejnych wierszach — rosnący ciąg liczb (każdą w osobnym wierszu) stanowiący wyznaczony zestaw długości.

Przykład

Dla pliku ADD.IN

```
14
5
10
12
15
17
20
21
22
24
26
27
30
31
33
```

Poprawnym rozwiązaniem jest plik ADD.OUT

```
24
5
12
21
```

Twój program powinien szukać pliku ADD.IN w katalogu bieżącym i tworzyć plik ADD.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę ADD.???, gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonywalnej powinien być zapisany w pliku ADD.EXE.

ROZWIĄZANIE

Liczbą **generowalną** ze zbioru liczbowego A będziemy nazywać liczbę, która jest elementem A lub daje się przedstawić jako suma pewnych elementów A (być może z powtórzeniami). Np. 11 jest liczbą generowalną (lub „generuje się") ze zbioru $\{2, 5, 7, 12\}$. Zbiór wszystkich liczb generowalnych ze zbioru A będziemy oznaczać symbolem $\mathrm{Domk}(A)^*$ (taki zbiór albo jest pusty, albo nieskończony). Trudno nie zauważyć, że w wyniku ustawiania prętów addonu jeden na drugim otrzymujemy walec o wysokości równej sumie ich długości, zatem:

Fakt 1: Wysokości słupów addonu, jakie można ustawić korzystając z prętów o długościach ze zbioru P, tworzą zbiór $\mathrm{Domk}(P)$.

Rzut oka na własności operacji Domk pozwala zauważyć następujące fakty:

* Jest to tzw. **domknięcie** zbioru A ze względu na dodawanie.

Fakt 2: Jeżeli $A \subseteq B$, to $\mathrm{Domk}(A) \subseteq \mathrm{Domk}(B)$, czyli dołożenie elementów do A nie zmniejsza zbioru $\mathrm{Domk}(A)$.

Fakt 3: $A \subseteq \mathrm{Domk}(A)$.

Fakt 4: $\mathrm{Domk}(A) = \mathrm{Domk}(\mathrm{Domk}(A))$.

Fakt 5: Jeżeli $A \subseteq \mathrm{Domk}(B)$, to $\mathrm{Domk}(A) \subseteq \mathrm{Domk}(B)$.

Czytelnik z pewnością poradzi sobie z ich dowodami; dowody faktów 2 i 3 są proste, nieco pracy wymaga dowód faktu 4, fakt 5 wynika z faktów 2, 3 i 4. Zauważmy jeszcze, że:

Fakt 6: Zbiór wysokości stabilnych mniejszych od m jest pełny dla komory o wysokości m (choć nie musi być bezpieczny, jak np. $\{2, 4\}$ dla $m = 7$).

Pierwszym pojawiającym się problemem jest wyznaczenie maksymalnej wysokości komory, dla której istnieje bezpieczny i pełny zestaw prętów. Mając na uwadze jedynie wyznaczenie wysokości komory nie musimy (na razie) dbać o to, by zestaw długości prętów był minimalny (jeżeli istnieje jakikolwiek zestaw, to istnieje również minimalny). Oznaczmy przez WS zbiór **wysokości stabilnych**.

Twierdzenie 7: Niech d będzie najmniejszą liczbą nie należącą do zbioru wysokości stabilnych, ale generowalną z niego ($d = \min\{v : v \in \mathrm{Domk}(WS)$ oraz $v \notin WS\}$). Wtedy $w = d - 1$ jest maksymalną wysokością komory, dla której istnieje bezpieczny i pełny zbiór długości prętów.

Dowód: Dowód przeprowadzimy w dwóch fazach. Pokażemy, że dla komory o wysokości większej niż w nie istnieje bezpieczny i pełny zestaw długości prętów, a następnie pokażemy, że taki zestaw istnieje dla komory o wysokości w.

Przypuśćmy, że wysokość w nie jest maksymalna, czyli że istnieje komora o wysokości $w' > w$ spełniająca warunki zadania. Pokażemy, że z dowolnego pełnego zbioru długości prętów, dla komory o wysokości w', można ustawić słup wysokości d, czyli że dla takiej komory dowolny zbiór pełny nie będzie bezpieczny.

Dla dowolnego pełnego (dla komory o wysokości w') zbioru długości prętów P każda wysokość stabilna mniejsza lub równa w' musi generować się z P (z definicji pełności), czyli*

$$WS \cap [1..w'] \subseteq \mathrm{Domk}(P) \tag{1}$$

zatem wszystko, co da się wygenerować ze zbioru wysokości stabilnych nie większych od w', musi się także dać wygenerować ze zbioru P (fakt 5):

$$\mathrm{Domk}(WS \cap [0..w']) \subseteq \mathrm{Domk}(P) \tag{2}$$

Wiadomo też, że do ułożenia słupa wysokości d wystarczą pręty ze zbioru WS i to nie wszystkie, bo tylko te krótsze od d, czyli (pamiętamy, że $[1..(d-1)] = [1..w]$):

$$d \in \mathrm{Domk}(WS \cap [0..w]) \tag{3}$$

* Dowód łatwiej jest czytać, gdy pamięta się, że $WS \cap [1..w]$ oznacza zbiór tych wysokości stabilnych, które nie przekraczają wysokości komory w.

Poza tym, z tego, że $[1..w] \subseteq [1..w']$ i faktu 2 wiemy, że wszystko, co da się wygenerować ze zbioru wysokości stabilnych nie większych od w, da się także wygenerować ze zbioru wysokości stabilnych nie większych od w':

$$\mathrm{Domk}(WS \cap [1..w]) \subseteq \mathrm{Domk}(WS \cap [1..w']) \tag{4}$$

Z (3), (4) i (2) otrzymamy:

$$d \in \mathrm{Domk}(WS \cap [1..w]) \subseteq \mathrm{Domk}(WS \cap [1..w']) \subseteq \mathrm{Domk}(P)$$

czyli ostatecznie $d \in \mathrm{Domk}(P)$, co oznacza, że przy ustaleniu wysokości komory w' większej niż w, z prętów o długościach z dowolnego pełnego zbioru P można ustawić słup o wysokości d, który mieści się w tej komorze, choć wysokość d nie jest stabilna. Zatem dla komory o wysokości większej niż w żaden zbiór pełny nie jest bezpieczny.

Pokażemy teraz, że dla komory o wysokości w istnieje bezpieczny i pełny zbiór długości prętów P. Wystarczy wziąć zbiór wszystkich wysokości stabilnych mniejszych lub równych w, czyli $P = WS \cap [1..w]$. Ten zbiór jest pełny (fakt 6).

Dla dowodu własności bezpieczeństwa P zauważmy, że każda liczba, która generuje się z P, generuje się także z WS, czyli $\mathrm{Domk}(P) \subseteq \mathrm{Domk}(WS)$ (bo $P = WS \cap [1..w] \subseteq WS$). Zatem, jeżeli z P generuje się liczba p nie będąca wysokością stabilną ($p \in \mathrm{Domk}(P)$ i $p \notin WS$), to generuje się ona także z WS, czyli

$$p \in \mathrm{Domk}(WS) \quad \text{oraz} \quad p \notin WS$$

Zauważmy, że d miało być najmniejszą liczbą o tej własności, więc $p \geqslant d$. Stąd otrzymujemy $p > w$ (bo $w = d-1$). Zatem jeżeli z prętów o długościach z P ustawimy słup o wysokości nie będącej wysokością stabilną, to słup ten nie zmieści się w komorze o wysokości w. □

W celu znalezienia liczby d będziemy konstruowali kolejne (coraz większe) elementy zbioru $\mathrm{Domk}(WS)$ i porównywali je z elementami zbioru WS (zapisanymi w pliku wejściowym w kolejności rosnącej). Na potrzeby opisu algorytmu potraktujmy plik wejściowy jak tablicę *we* : **array**$[1..n]$ **of integer** — ułatwi to jego zapis. Niech zmienna k przechowuje wartość wskaźnika pozycji w pliku, tj. numer pierwszego nie przeczytanego rekordu pliku (na początku $k = 1$).

Rys. 1 Schemat algorytmu znajdującego maksymalną wysokość komory

```
nast := 2 · we[1];
k := 2;
koniec := false;
while (k ⩽ n) and not koniec do            (1)
  if we[k] > nast then koniec := true
  else begin
    popraw(nast);
    k := k + 1
  end;
end;
w := nast − 1;
```

Na zmiennej *nast* będzie przechowywana najmniejsza liczba generowalna ze zbioru $we[1..(k-1)]$ nie będąca jego elementem:

$$nast = \min\bigl\{w : w \in \mathrm{Domk}\bigl(we[1..(k-1)]\bigr) \text{ oraz } w \notin we[1..(k-1)]\bigr\} \qquad (5)$$

Liczba ta jest „kandydatem" na kolejną wysokość stabilną w tym sensie, że jeżeli wysokość stabilna odczytana z pliku okaże się większa od *nast*, to znaczy, że *nast* jest pierwszą zabronioną wysokością komory (liczba d) — zob. schemat algorytmu na rys. 1.

Teraz musimy jeszcze powiedzieć, w jaki sposób zadbać o to, aby był zachowany niezmiennik (5), czyli napisać procedurę oznaczoną na rys. 1 symbolem *popraw.* W tym miejscu można sformułować naiwny algorytm przeglądający za każdym razem sumy wszystkich par ze zbioru $we[1..(k-1)]$, co daje złożoność jednego obrotu pętli (1) rzędu k^2, a stąd całego algorytmu $O(n^3)$. Spróbujmy zrobić to lepiej, konstruując przy okazji minimalny pełny i bezpieczny zbiór długości prętów P.

Sztuka polega na tym, by prócz konstruowanego zbioru długości prętów przechowywać dodatkowo zbiór „kandydatów" na wartość *nast.*

Oznaczmy przez P konstruowany zbiór długości prętów, a przez *Kand* — zbiór „kandydatów".

Na początku do zbioru P wrzucamy pierwszą (najmniejszą) wysokość stabilną. Zauważmy, że jeżeli w obrocie pętli (1) wartość $we[k] < nast$ to znaczy, że $we[k]$ nie generuje się ze zbioru $we[1..(k-1)]$. Co więcej, w takim przypadku $we[k]$ nie generuje się nawet z całego zbioru *WS* (wynika to z własności dodawania liczb dodatnich: suma jest zawsze większa od składników). Przypadek $we[k] = nast$ oznacza zaś, że $we[k]$ generuje się z $we[1..(k-1)]$, zatem $we[k]$ na pewno nie należy wkładać do P.

Tajemnicą pozostaje jednak w dalszym ciągu zbiór *Kand* i sposób uaktualniania wartości zmiennej *nast.*

Spójrzmy jeszcze raz na równanie (5) będące definicją wartości *nast.* Załóżmy, że dla zbioru $we[1..(k-1)]$ zbudowaliśmy zbiór P długości prętów. Z faktów, że $we[1..(k-1)]$ jest pełny dla wysokości $we[k-1]$ oraz że każdy element generowalny z tego zbioru jest też generowalny z P wynika, że *nast* musi mieć postać $v+p$, gdzie $v \in we[1..(k-1)]$ i $p \in P$. Liczby tej postaci będziemy właśnie przechowywać w zbiorze *Kand.* Wartość *nast* otrzymamy znajdując minimum tego zbioru.

Z każdym obrotem pętli będziemy starali się utrzymać warunki (niezmienniki):

$$Kand \cup \{nast\} = \{v + p : v \in we[1..k-1], p \in P\}$$
$$nast = \min\{v + p : v \in we[1..k-1], p \in P\}$$

Na rys. 2 przedstawiono schemat algorytmu wykorzystującego powyższe mechanizmy.

Pozostaje jeszcze kwestia efektywnej implementacji operacji na zbiorach, w tym operacji *extractMin* powodującej wyjęcie ze zbioru elementu o najmniejszej wartości.

Operacje na zbiorze P nie sprawiają w zasadzie problemu, gdyż wykonujemy tylko inicjowanie zbioru, dopisywanie nowego elementu oraz przeglądanie wszystkich jego elementów. Najprostsze wydaje się trzymanie elementów zbioru P kolejno w tablicy. Koszt operacji wstawienia przy takiej implementacji wynosi $O(1)$, przejrzenie zbioru $O(n)$.

Rys. 2 Schemat algorytmu znajdującego wysokość komory i zestaw prętów

$nast := 2 \cdot we[1];$
$k := 2;$
$Kand := \varnothing;$
$P := \{we[1]\};$
$koniec :=$ **false**;
while $(k \leqslant n)$ **and** (**not** $koniec$) **do**
  if $we[k] > nast$ **then** $koniec :=$ **true**
  else begin
    if $we[k] < nast$ **then** $P := P \cup we[k];$ (1)
    $Kand := Kand \cup \{we[k] + p : p \in P\};$ (2)
    if $we[k] = nast$ **then** $nast := extractMin(Kand);$ (3)
    $k := k + 1$
  end;
end;
$w := nast - 1;$

Zastosowanie podobnej implementacji dla zbioru *Kand* nie przyniesie jednak oczekiwanych rezultatów. Problemem jest operacja *extractMin*, która przy takiej implementacji może wymagać przejrzenia całego zbioru *Kand*, który może być rozmiaru rzędu n^2.

Rozwiązaniem powyższego problemu jest implementacja zbioru *Kand* za pomocą kopca. Kopiec jest strukturą danych umożliwiającą wykonywanie operacji wstawiania elementu i usuwania najmniejszego elementu w czasie proporcjonalnym do logarytmu rozmiaru kopca. Przy takiej implementacji otrzymujemy następujące oszacowanie na czas działania algorytmu:*

$$n \cdot \Big(\underbrace{O(1)}_{\text{wiersz (1)}} + \underbrace{O(n \log n^2)}_{\text{wiersz (2)}} + \underbrace{O(\log n^2)}_{\text{wiersz (3)}} \Big) = O(n^2 \log n)$$

W niniejszym opracowaniu nie będziemy wdawać się w szczegóły implementacji kopca. Zainteresowanych odsyłamy do pozycji [7].

Wydawało by się, że powyższego oszacowania nie da się już poprawić. Można jednak skorzystać z pewnej „technicznej” sztuczki;** po pierwsze, zakres wartości, jakie przechowujemy w kopcu jest na tyle niewielki (dokładnie [1..10000]), że kopiec można zastąpić tablicą o rozmiarze 10000 bitów, w której ustawienie bitu n na 1 będzie oznaczało, że „liczba n jest w kopcu”. Wstawienie elementu do takiej struktury polega na prostym wyliczeniu adresu i wykonaniu przypisania, zatem czas sumaryczny takiej operacji jest stały.

Co z operacją *extractMin*? Można za każdym razem przeszukiwać tablicę od początku i znajdować najmniejszy jej element. Sumaryczna liczba odczytów tablicy

* Oczywiście $O(\log n^2) = O(\log n)$.

** Sztuczka przestałaby działać, gdyby np. zrezygnowano z założenia, że wysokości są liczbami całkowitymi.

podczas pracy programu wynosiłaby około $10000^2/2$. Liczbę tę można zredukować do około 10000, gdy zauważymy, że po wyjęciu elementu *nast* ze struktury *Kand*, nie trafiają już do niej elementy mniejsze od *nast*. Zatem poszukiwania nie trzeba każdorazowo rozpoczynać od początku tablicy — wystarczy szukać od miejsca, w którym zakończono poprzednie przeszukiwanie.

W pliku ADD.PAS na załączonej dyskietce przedstawiono rozwiązanie wzorcowe z użyciem kopców. Przeróbkę implementacji struktur danych na tablice bitowe pozostawiamy jako ćwiczenie dla Czytelnika.

OPIS TESTÓW

Do sprawdzenia rozwiązań zawodników użyto 10 testów (ADD0.IN–ADD9.IN).

- ADD0.IN — test z treści zadania.
- ADD1.IN–ADD5.IN — testy poprawnościowe (liczby wysokości stabilnych: 1, 9, 45, 65, 27).
- ADD6.IN–ADD9.IN — testy wydajnościowe (liczby wysokości stabilnych: 900, 702, 950, 9980).

Przemysława Kanarek
treść zadania

Marcin Kubica
program wzorcowy, opracowanie

Genotypy

Genotypy są skończonymi łańcuchami genów. Opisujemy je za pomocą słów utworzonych z wielkich liter alfabetu angielskiego A–Z. Różne litery oznaczają różne rodzaje genów. Gen może pączkować — zmieniając się w dwójkę nowych genów. Tymi przemianami rządzi skończony zbiór reguł. Każdą regułę pączkowania można zapisać w postaci trójki wielkich liter $A_1 A_2 A_3$, co oznacza, że gen A_1 może się zmienić w dwójkę genów $A_2 A_3$.
Wielką literą S oznaczamy specjalny rodzaj genów zwanych supergenami. Hodowla genotypu rozpoczyna się od łańcucha supergenów i polega na sterowanym pączkowaniu wybranych genów zgodnie z ustalonymi regułami.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego GEN.IN skończony zbiór reguł pączkowania genów oraz ciąg słów określających genotypy, jakie należy wyhodować,*
- *dla każdego z danych genotypów bada, czy można go wyhodować z pewnego skończonego łańcucha supergenów zgodnie z danymi regułami pączkowania i jeśli tak, znajduje minimalną długość takiego łańcucha,*
- *zapisuje wyniki w pliku tekstowym GEN.OUT.*

Wejście

W pierwszym wierszu pliku tekstowego GEN.IN jest zapisana jedna liczba całkowita $1 \leqslant n \leqslant 10000$. W każdym z n kolejnych wierszy jest zapisana jedna reguła pączkowania, w postaci słowa złożonego z trzech wielkich liter A–Z.
W następnym wierszu jest zapisana jedna liczba całkowita $1 \leqslant k \leqslant 10000$. W każdym z k kolejnych wierszy jest zapisany jeden genotyp w postaci niepustego słowa złożonego z co najwyżej 100 liter A–Z.

Wyjście

W i-tym z kolejnych k wierszy pliku tekstowego GEN.OUT należy zapisać:

- *jedną liczbę całkowitą dodatnią oznaczającą minimalną długość łańcucha supergenów potrzebnego do wyhodowania i-tego danego genotypu, albo*
- *jedno słowo NIE, jeśli tego genotypu nie da się wyhodować.*

Przykład

Dla pliku GEN.IN:

```
6
```

```
SAB
SBC
SAA
ACA
BCC
CBC
3
ABBCAAABCA
CCC
BA
```

poprawnym rozwiązaniem jest plik GEN.OUT:

```
3
1
NIE
```

Twój program powinien szukać pliku GEN.IN w katalogu bieżącym i tworzyć plik GEN.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę GEN.??? gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonywalnej powinien być zapisany w pliku GEN.EXE.

UZUPEŁNIENIE TREŚCI ZADANIA

Podczas trwania zawodów treść zadania uzupełniono następującym stwierdzeniem:

> *Dopuszczalne są reguły pączkowania, w których na drugim lub trzecim miejscu występuje supergen.*

ROZWIĄZANIE

Zadanie to jest adaptacją znanego problemu informatycznego: jak sprawdzić czy dane słowo należy do języka generowanego przez gramatykę bezkontekstową (zob. książka [15]). Adaptacja polega na częściowym uproszczeniu problemu. Zamiast dwóch rodzajów symboli, terminalnych i nieterminalnych, są tylko jednoliterowe nazwy genów. Supergen S jest odpowiednikiem aksjomatu. Reguły pączkowania genów są uproszczonym odpowiednikiem produkcji — po prawej stronie znajdują się zawsze dwa symbole. Problem należenia danego słowa do języka generowanego przez gramatykę bezkontekstową można sformułować następująco: czy dany genotyp można uzyskać z pojedynczego supergenu. Różnica w liczbie supergenów od których zaczynamy zniknie, jeśli najpierw spróbujemy rozwiązać trochę bardziej ogólny problem.

Niech dany genotyp ma postać $G = g_1 \ldots g_l$. Dla $1 \leqslant i \leqslant j \leqslant l$, niech $W(i,j)$ będzie zbiorem takich pojedynczych genów, z których można uzyskać genotyp $g_i \ldots g_j$. W szczególności, jeśli $\mathrm{S} \in W(1,l)$ to cały genotyp G można uzyskać z jednego supergenu.

Zauważmy, że dla $i = 1, \ldots, l$, $W(i,i) = \{g_i\}$. Ponadto, jeśli pewien gen A może wypączkować tworząc parę genów BC oraz (dla $1 \leqslant i \leqslant k < j \leqslant l$) z genu B możemy uzyskać genotyp $g_i \ldots g_k$, a z genu C możemy uzyskać genotyp $g_{k+1} \ldots g_j$, to oczywiście z genu A możemy uzyskać genotyp $g_i \ldots g_j$. Oznaczmy przez *Sklej* następującą operację na zbiorach genów:

$$\mathit{Sklej}(V_1, V_2) = \bigcup_{v_1 \in V_1, v_2 \in V_2} \{g : g v_1 v_2 \text{ jest jedną z reguł pączkowania genów}\}$$

Wówczas dla $i < j$:

$$W(i,j) = \bigcup_{i<k\leq j} \mathit{Sklej}(W(i,k-1), W(k,j))$$

Zbiory $W(i,j)$ możemy wyliczyć za pomocą programowania dynamicznego, w dwuwymiarowej tablicy W : **array** $[1..l,\ 1..l]$ **of set of** *gen*. Najpierw inicjalizujemy przekątną tablicy jednoelementowymi zbiorami $\{g_i\}$, a następnie kolejne nadprzekątne wypełniamy korzystając z operacji *Sklej*. Ilustruje to poniższy fragment programu*.

```
for i := 1 to l do
  W[i,i] := {g_i};
for i := 1 to l − 1 do
  for j := 1 to l − i do begin
    W[j,i+j] := ∅;
    for k := 1 to i do
      W[j,i+j] := W[j,i+j] ∪ Sklej(W[j,j+k−1],W[j+k,i+j])
  end;
```

Wymaga to wykonania $\Theta(l^3)$ operacji *Sklej*. Ich implementacja jest więc kluczowa dla efektywności rozwiązania. Ponieważ w każdym pliku z danymi zestaw reguł pączkowania jest ten sam dla wszystkich genotypów, można go wstępnie przygotować na potrzeby operacji *Sklej*.

Ze względu na małą liczbę rodzajów genów, zbiory genów można reprezentować za pomocą liczb typu `longint`, przeznaczając na każdy gen jeden bit. Reguły pączkowania genów możemy zapamiętać w dwuwymiarowej tablicy zbiorów genów P : **array** [*gen*, *gen*] **of longint** — $P[g_1, g_2]$ to zbiór genów, z których można wypączkować parę genów $g_1 g_2$. Funkcja *Sklej* może mieć następującą implementację:

```
function Sklej (V1, V2 : longint) : longint;
var
  g1, g2 : gen;
  zb : longint;
```

* Program, którego fragmenty tu przytaczamy, można znaleźć na dyskietce, w pliku GEN.PAS

```
begin
  zb := 0;
  for g1 ∈ gen do
    if g1 ∈ V1 then
      for g2 ∈ gen do
        if g2 ∈ V2 then
          zb := zb or P[g1, g2];
  Sklej := zb
end;
```

Mając wyliczoną tablicę W możemy wyznaczyć liczbę potrzebnych supergenów — ponownie stosując programowanie dynamiczne. Wypełniamy jednowymiarową tablicę S:**array** [1..l] **of integer** tak, aby $S[i]$ było równe minimalnej liczbie supergenów potrzebnych do uzyskania genotypu $g_1 \ldots g_i$. Jeśli supergen należy do zbioru $W[1,i]$ to oczywiście $S[i] = 1$. W przeciwnym przypadku szukamy takiego podziału genotypu $g_1 \ldots g_i$ na dwie części $g_1 \ldots g_{k-1}$, $g_k \ldots g_i$, że pierwszą z nich można uzyskać z jak najmniejszej liczby supergenów, a drugą z jednego supergenu. Ilustruje to poniższy fragment programu.

```
for i := 1 to l do begin
  if supergen ∈ W[1, i] then
    S[i] := 1
  else begin
    S[i] := ∞;
    for k := 2 to i do
      if supergen ∈ W[k, i] then
        S[i] := min(S[i], S[k − 1] + 1);
  end
end;
```

Wynik znajduje się w $S[l]$, przy czym jeśli $S[l] = \infty$, to danego genotypu nie da się uzyskać.

Koszt czasowy tej części programu wynosi $\Theta(l^2)$, jest więc zaniedbywalny w porównaniu z kosztem wyliczenia tablicy W.

TESTY

Do sprawdzenia rozwiązań zawodników użyto 15 testów GEN0.IN – GEN14.IN. Poniżej zamieszczamy krótkie komentarze do testów.

- GEN1.IN — z jednego supergenu można uzyskać takie genotypy złożone z genów A i B, w których jest tyle samo genów A i B. Test ten bada ogólną poprawność rozwiązania;
- GEN2.IN — z jednego supergenu można uzyskać genotypy przypominające wyrażenia nawiasowe. Pierwszy i ostatni gen tworzą parę nawiasów: C. . . D lub A. . . Z.

W środku znajduje się jeden gen X lub para genotypów przypominających wyrażenia nawiasowe. Test ten bada ogólną poprawność rozwiązania;

- GEN3.IN — jeśli geny A–J zastąpimy cyframi 0–9, to z jednego supergenu można uzyskać genotypy będące zapisami liczb podzielnych przez 3 zakończonymi pojedynczym supergenem. Test ten bada ogólną poprawność rozwiązania;
- GEN4.IN — z jednego supergenu można uzyskać genotypy postaci

  $$\mathrm{A}\ldots\mathrm{AB}\ldots\mathrm{BC}\ldots\mathrm{C}$$

  w których występuje przynajmniej jeden gen A, liczba genów B jest dowolna, a liczba genów C jest równa liczbie genów A plus podwojona liczba genów B. Test ten bada ogólną poprawność rozwiązania;
- GEN5.IN — z jednego supergenu można uzyskać takie genotypy złożone z genów A, B i C, w których liczba genów A jest równa liczbie genów B plus podwojona liczba genów C. Test ten eliminuje rozwiązania badające rekurencyjnie jakie genotypy można uzyskać z jednego supergenu;
- GEN6.IN — z jednego supergenu można uzyskać ciąg 4-ech lub 7-miu genów A. Ile supergenów potrzeba, aby uzyskać ciąg 67-miu genów A? Test ten eliminuje rozwiązania badające rekurencyjnie, które fragmenty genotypu powinny powstać z pojedynczych supergenów;
- GEN7.IN, GEN8.IN — w testach tych geny reprezentują pewne zbiory. Reguły pączkowania pokrywają wszystkie takie sytuacje, w których suma zbiorów reprezentowanych przez geny powstałe w pączkowaniu zawiera się w zbiorze reprezentowanym przez gen poddawany pączkowaniu. Testy te zawierają dużo reguł pączkowania i badają efektywność implementacji operacji *Sklej*;
- GEN9.IN, GEN10.IN — w testach tych geny reprezentują pewne ciągi. Reguły pączkowania pokrywają wszystkie takie sytuacje, w których sklejenie ciągów reprezentowanych przez geny powstałe w pączkowaniu daje podciąg ciągu reprezentowanego przez gen poddawany pączkowaniu. Testy te zawierają dużo reguł pączkowania i badają efektywność implementacji operacji *Sklej*;
- GEN11.IN–GEN14.IN — w testach tych z jednego supergenu można uzyskać takie genotypy złożone z genów A i B, w których jest tyle samo genów A i B. Testy te zawierają coraz dłuższe genotypy i badają wydajność rozwiązań.

Wojciech Guzicki
treść zadania, opracowanie

Marcin Engel
program wzorcowy

Lotniska

W państwie X istnieją lotniska w n miastach. Znane są maksymalne **przepustowości** *tych lotnisk — lotnisko w mieście M_i może mieć co najwyżej d_i połączeń lotniczych z innymi miastami. Należy zaplanować sieć połączeń lotniczych między tymi miastami w taki sposób, by miasto M_i miało dokładnie d_i połączeń z innymi miastami, przy czym zakładamy, że każde połączenie jest dwukierunkowe i każde miasto może mieć z innym tylko jedno połączenie.*

Zadanie

Ułóż program, który:

- *wczytuje z pliku tekstowego LOT.IN liczbę n miast oraz liczby d_i,*
- *układa sieć połączeń lotniczych, taką że dla każdego i od 1 do n miasto M_i ma dokładnie d_i połączeń z innymi miastami,*
- *zapisuje w pliku LOT.OUT listę wszystkich tworzących sieć połączeń.*

Dane są tak dobrane, by rozwiązanie zadania istniało. Jeśli zadanie ma wiele rozwiązań, Twój program powinien znajdować tylko jedno.

Wejście

W pierwszym wierszu pliku tekstowego LOT.IN jest zapisana liczba całkowita n spełniająca nierówność $3 \leqslant n \leqslant 500$. Jest to liczba miast.
W kolejnych n wierszach są zapisane liczby całkowite dodatnie d_i, po jednej w każdym wierszu.

Wyjście

W pliku tekstowym LOT.OUT należy zapisać wszystkie połączenia lotnicze sieci utworzonej przez Twój program. Każde połączenie należy zapisać w osobnym wierszu w postaci dwóch liczb całkowitych dodatnich oddzielonych pojedynczym odstępem, tj. numerów dwóch połączonych miast. Numery miast w wierszu mogą występować w dowolnej kolejności; również kolejność zapisywania połączeń w pliku jest dowolna.

Przykład

Dla pliku LOT.IN

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

```
2
```

przykładem poprawnego rozwiązania jest plik LOT.OUT

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

Twój program powinien szukać pliku LOT.IN w katalogu bieżącym i tworzyć plik LOT.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę LOT.??? gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonywalnej powinien być zapisany w pliku LOT.EXE.

UZUPEŁNIENIE TREŚCI

W trakcie trwania zawodów treść zadania uzupełniono następującym stwierdzeniem:

> *Może się zdarzyć, że podróż z jednego miasta do drugiego, nawet z przesiadkami, nie jest możliwa.*

ROZWIĄZANIE

Liczbę d_i połączeń lotniska o numerze i z innymi lotniskami nazwiemy *przepustowością* tego lotniska. Szukaną sieć połączeń znajdujemy w następujący sposób:

(1) Znajdujemy lotnisko mające największą przepustowość. Przypuśćmy, że jego numerem jest i.

(2) Łączymy lotnisko i z d_i lotniskami o największych przepustowościach spośród pozostałych lotnisk. Przepustowości tych lotnisk zmniejszamy o 1.

(3) Lotnisko o numerze i usuwamy z listy lotnisk. Podobnie usuwamy wszystkie lotniska mające (po ostatnim zmniejszeniu) przepustowość równą 0.

(4) Powtarzamy kroki (1)–(3) dotąd, aż usuniemy wszystkie lotniska.

Poprawność tej procedury wynika łatwo przez indukcję względem n. Łatwo sprawdzić, że podstawa indukcji zachodzi dla $n = 3$. Załóżmy, że opisany wyżej sposób postępowania prowadzi do rozwiązania zadania, gdy liczba lotnisk jest mniejsza od n i przypuśćmy, że mamy do czynienia z n lotniskami. Bez zmniejszenia ogólności można założyć, że lotniska zostały ponumerowane w ten sposób, by przepustowości kolejnych lotnisk tworzyły ciąg nierosnący: $d_1 \geqslant d_2 \geqslant d_3 \geqslant \ldots \geqslant d_n$. Opisany wyżej algorytm polega na wybraniu lotniska o numerze 1 i połączeniu go z lotniskami o numerach $2, 3, \ldots, d_1 + 1$. Wykażemy, że jeśli istnieje jakakolwiek sieć połączeń, w której lotnisko i ma d_i połączeń z innymi lotniskami, to istnieje również sieć taka, w której lotnisko i ma d_i połączeń z innymi lotniskami i ponadto lotnisko 1 jest połączone z lotniskami $2, 3, \ldots, d_1 + 1$. Stąd wynika, że po wykonaniu kroków (1)–(3) nadal możemy otrzymać rozwiązanie zadania. Liczba lotnisk uległa jednak zmniejszeniu co najmniej o 1 (bo usunęliśmy lotnisko 1 z listy), więc powtarzanie kroków (1)–(3) doprowadzi do rozwiązania. To kończy dowód indukcyjny.

Pozostaje dowieść, że istnieje rozwiązanie, w którym lotnisko 1 jest połączone z lotniskami $2, 3, \ldots, d_1 + 1$. Weźmy pod uwagę jakąkolwiek sieć połączeń spełniającą warunki zadania. Jeśli lotnisko 1 jest połączone z lotniskami $2, 3, \ldots, d_1 + 1$, to dowód jest zakończony. Przypuśćmy więc, że lotnisko 1 nie jest połączone z którymś z tych lotnisk. Niech i będzie najmniejszym numerem lotniska, które nie jest połączone z lotniskiem 1. Oczywiście $i \leqslant d_1 + 1$. Ale lotnisko 1 jest połączone z d_1 lotniskami, musi więc być połączone z jakimś lotniskiem o numerze większym od $d_1 + 1$. Niech $j > d_1 + 1$ i niech j będzie numerem lotniska, z którym połączone jest lotnisko nr 1. Ponieważ przepustowości lotnisk są uporządkowane nierosnąco, więc $d_i \geqslant d_j$. To znaczy, że lotnisko i ma co najmniej tyle połączeń, co lotnisko j. Lotnisko j jest połączone z lotniskiem 1, z którym nie jest połączone lotnisko i. Stąd wynika, że istnieje lotnisko, na przykład o numerze k, które jest połączone z lotniskiem i i nie jest połączone z lotniskiem j. Mamy więc cztery lotniska: 1, i, j, k takie, że i jest połączone z k oraz 1 z j, ale ani 1 nie jest połączone z i, ani j z k. Teraz zmieniamy połączenia: lotnisko i zamiast z k łączymy z 1, lotnisko j zamiast z 1 łączymy z k. Zauważamy, że stopnie lotnisk nie uległy zmianie, natomiast lotnisko 1 jest już połączone z lotniskiem i. W ten sposób powiększyliśmy o jedno liczbę połączeń lotniska 1 z lotniskami o numerach $2, 3, \ldots, d_1 + 1$. Powtarzając to postępowanie wielokrotnie (ścisłe rozumowanie znów wymaga indukcji), doprowadzimy do sytuacji, w której lotnisko 1 będzie połączone z d_1 następnymi lotniskami. To ostatecznie kończy dowód poprawności algorytmu.

Może się zdarzyć, że za pomocą opisanej wyżej procedury stworzymy sieć w której z niektórych lotnisk nie będzie można dolecieć (nawet z przesiadkami) do niektórych innych. Można pokazać, że jeśli suma stopni lotnisk jest równa co najmniej $2n - 2$, to istnieje sieć „spójna", tzn. taka, w której każde lotnisko ma połączenie (niekoniecznie bezpośrednie) ze wszystkimi innymi lotniskami. Główny pomysł dowodu polega na tym, że jeśli $d_1 + \ldots + d_n \geqslant 2n - 2$, to sieć połączeń albo jest spójna, albo ma cykl. Jeśli połączone ze sobą lotniska x i y nie mają żadnych połączeń z lotniskami pewnego cyklu, to wybieramy z tego cyklu dwa sąsiednie lotniska u i v, a następnie zmieniamy połączenia: łączymy x z u i y z v. W ten sposób zmniejszamy o 1 liczbę tzw. składowych, tzn. spójnych części całej sieci. Powtarzając to postępowanie odpowiednią liczbę razy otrzymamy w końcu sieć spójną.

Jedna z możliwych realizacji zadania polega na dokładnym powtórzeniu powyższego rozumowania. Umieszczamy w tablicy pary: numer lotniska i jego stopień, a następnie sortujemy nierosnąco względem stopni. Następnie bierzemy kolejne lotniska (o numerach Nr od 1 do n) i jeśli stopień lotniska jest większy od zera, to zmniejszamy stopnie następnych d_{Nr} lotnisk o 1. Teraz musimy ponownie posortować tablicę. To sortowanie nie jest trudne, zamiany wymagają tylko dwa położone obok siebie bloki wierzchołków. Znalezione połączenia od razu zapisujemy w pliku wyjściowym.

Należy zauważyć, że jeśli w tablicy lotnisk występują kolejno lotniska mające ten sam stopień, to w czasie zmniejszania stopni możliwe są dwa przypadki:

- wszystkie lotniska tego bloku lotnisk będą miały zmniejszony stopień; wtedy do tego miejsca tablica jest nadal posortowana,
- zmniejszanie stopni zakończy się wewnątrz tego bloku i otrzymamy blok postaci:

  $$k-1, k-1, k-1, \ldots, k-1, k, k, k, \ldots, k.$$

  Jeśli w tym bloku występuje L lotnisk mających stopień $k-1$ i P lotnisk mających stopień k, to trzeba zamienić ze sobą dwa bloki: jeśli $L < P$, to zamieniamy L lotnisk z lewej strony z L lotniskami z prawej strony, a jeśli $L \geqslant P$, to P lotnisk z prawej strony z P lotniskami z lewej strony. Nie mamy istotnych ograniczeń pamięci, więc możemy przepisać lotniska z jednej strony do dodatkowej tablicy, wpisać do zwolnionych miejsc lotniska z drugiej strony i w końcu wpisać na właściwe miejsce lotniska z tablicy pomocniczej.

Takie rozwiązanie znajduje się w pliku `LOT1.PAS`. Sortowanie tablicy zostało dokonane za pomocą procedury sortowania szybkiego (quicksort). Rozwiązania mniej efektywne otrzymamy używając innych, wolniejszych procedur sortowania oraz zastępując opisaną wyżej zamianę bloków ponownym sortowaniem całej tablicy. Implementacja tej wersji algorytmu wraz z procedurami sortowania przez wstawianie, przez wybieranie, bąbelkowego, przez łączenie oraz sortowania szybkiego znajduje się w pliku `LOT2.PAS`.

Implementacja algorytmu z nawrotami znajduje się w pliku `LOT3.PAS`. Jest to algorytm nieefektywny zarówno czasowo, jak i pamięciowo.

Rozwiązanie wzorcowe bazuje na strukturze danych, w której numery lotnisk są zapisywane w kolejkach. Przy wczytywaniu danych tworzymy dla każdej przepustowości d kolejkę, do której wrzucamy numery lotnisk o przepustowości d. Przy okazji zapamiętujemy maksymalną przepustowość.

Powiemy, że kolejka przechowująca lotniska o przepustowości d jest rzędu d.

Załóżmy, że w pewnej chwili działania algorytmu maksymalny rząd niepustej kolejki wynosi d. Pojedynczy krok algorytmu polega na wyjęciu z kolejki rzędu d jednego lotniska. Następnie wyjmujemy z tej kolejki, a następnie w miarę ubywania elementów z kolejek o rzędach coraz mniejszych d lotnisk. Każde takie lotnisko zostaje umieszczone w kolejce o jeden rząd mniejszej niż ta, z której zostało wyjęte. Oczywiście, lotnisk wyjmowanych z kolejki rzędu 1 nie umieszczamy już w strukturze. Przy realizacji tego algorytmu należy uważać, aby nie pobierać ponownie wierzchołków, które zostały przerzucone do kolejki o mniejszym rzędzie.

Złożoność tego algorytmu jest liniowa względem liczby połączeń w konstruowanej sieci, gdyż po każdym wyjęciu lotniska z kolejki mamy „w ręku” kolejne połączenie, a operacje na kolejce mają koszt stały.

Rozwiązanie to zostało zaimplementowane w pliku `LOT.PAS`. Czasy działania programów `LOT.PAS` i `LOT1.PAS` dla przeprowadzonych testów nie odbiegały bardzo od siebie.

TESTY

Do sprawdzenia zadania zaprojektowano 9 testów. Pierwszym (LOT0.IN) był test z treści zadania. Następne 4 testy były również dość małych rozmiarów

(LOT1.IN–LOT4.IN) i miały na celu sprawdzenie poprawności algorytmów. Dwa testy (LOT5.IN i LOT6.IN) były nieco większe, miały po ok. 150 lotnisk i różniły się liczbą połączeń. Ostatnie dwa testy miały odpowiednio 300 i 500 lotnisk i ich zadaniem było sprawdzenie efektywności programów.

Tomasz Śmigielski — treść zadania
Albert Krzymowski — program wzorcowy, opracowanie

Paliwo

Firma transportowa X otrzymała nowe zlecenie; będzie dostarczała paczki z miasta A do miasta B. Paczki będą przewożone samochodami firmy. Na trasie AB jest wiele stacji benzynowych oferujących paliwo po różnych cenach. Pierwsza z nich znajduje się na początku trasy. Wszystkie samochody firmy X zużywają jednakowo standardową jednostkę paliwa na przejechanie jednej mili, ale mają różne pojemności zbiorników. Koszt paliwa potrzebnego do przejechania trasy AB zależy od pojemności zbiornika na paliwo w samochodzie i wybranego planu tankowania paliwa na stacjach leżących wzdłuż trasy. Zakładamy, że na każdej stacji jest zawsze wystarczająco dużo paliwa, by kierowca mógł napełnić zbiornik do pełna oraz że leżą one dostatecznie gęsto, aby każdy samochód firmy X mógł przejechać całą trasę.

Zadanie

Ułóż program, który:

- *wczytuje z pliku tekstowego PAL.IN następujące dane:*
  - *pojemność zbiornika na paliwo w samochodzie,*
  - *liczbę stacji benzynowych na trasie AB,*
  - *cenę paliwa na każdej stacji oraz odległości między kolejnymi stacjami,*
- *znajduje minimalny koszt tankowania paliwa na trasie AB dla samochodu o danej pojemności zbiornika,*
- *zapisuje wynik w pliku tekstowym PAL.OUT.*

Wejście

W pierwszym wierszu pliku tekstowego PAL.IN jest zapisana pojemność p zbiornika samochodu. Jest to liczba całkowita spełniająca nierówność $1 \leqslant p \leqslant 1000000$.
W drugim wierszu jest zapisana liczba n stacji benzynowych na trasie AB. Jest to liczba całkowita spełniająca nierówność $1 \leqslant n \leqslant 1000000$.
W każdym z kolejnych n wierszy jest zapisana para liczb całkowitych dodatnich, oddzielonych pojedynczym odstępem c_i, d_i. *Liczba* c_i *— to cena standardowej jednostki paliwa na stacji i (licząc od A do B), zaś* d_i *to odległość tej stacji w milach od następnej stacji w kierunku B* (d_n *— to odległość ostatniej stacji od końca trasy AB). Liczby te spełniają nierówności:* $1 \leqslant c_i \leqslant 1000$, $1 \leqslant d_i \leqslant 1000000$.
Długość trasy AB (suma d_i*) jest nie większa niż 1000000.*

Wyjście

W pliku tekstowym PAL.IN należy zapisać jedną liczbę całkowitą — minimalny koszt tankowania paliwa na trasie AB dla samochodu o danej pojemności zbiornika.

PRZYKŁAD

Dla pliku PAL.IN

```
40
3
2 10
1 15
2 5
```

poprawnym rozwiązaniem jest plik PAL.OUT:

```
40
```

Twój program powinien szukać pliku PAL.IN w katalogu bieżącym i tworzyć plik PAL.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę PAL.??? gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonywalnej powinien być zapisany w pliku PAL.EXE.

ROZWIĄZANIE

Powyższy problem można przedstawić jako podział rozważanej drogi na rozłączne odcinki o długości jednej mili i przyporządkowanie każdemu z nich ceny benzyny zużytej na jego przejechanie. Przyjmijmy następujące oznaczenia:

- $stacja(i)$ — stacja, na której zostało zatankowane paliwo zużyte na i-tej mili,
- $cena(s)$ — cena benzyny na stacji s,
- $koszt(i)$ — koszt przejechania i-tego odcinka drogi, $koszt(i) = cena(stacja(i))$,
- P — pojemność baku samochodu,
- K — koszt podróży, który wyraża się sumą:

$$K = \sum_{i=1}^{D} cena(stacja(i))$$

gdzie D jest odległością między miastami A i B.

Nasze zadanie polega na zminimalizowaniu K.

Obserwacja 1: Dla każdego planu tankowania *stacja*, o całkowitym koszcie równym $K = \sum_{i=1}^{D} cena(stacja(i))$, istnieje „plan tankowania” $stacja'$ o koszcie $K' = \sum_{i=1}^{D} cena(stacja'(i))$ taki, że $K = K'$ oraz

$$\forall_{0 \leqslant i < j \leqslant D} \ stacja(i) \leqslant stacja'(j) \qquad (*)$$

Warunek $(*)$ mówi, że paliwo w baku jest zużywane w kolejności tankowania. Wynika z niego natychmiast, że żadna jednostka paliwa nie jest przewożona w zbiorniku samochodu przez więcej niż P mil. Innymi słowy, jest ona zużyta nie dalej niż w odległości P od miejsca zakupu.

Powyższe obserwacje pozwalają na szukanie rozwiązania optymalnego tylko wśród rozwiązań spełniających $(*)$.

Prezentowany poniżej program, wybiera, dla każdego odcinka drogi długości jeden, najtańszą stację znajdującą się nie dalej niż P (pojemność baku samochodu) w stronę miasta A. Wyszukiwanie rozpoczynamy od miasta A pamiętając dane o stacjach z ostatnich P mil. Można łatwo zauważyć, że nie musimy przechowywać danych o wszystkich stacjach z danego obszaru — wystarczy dla każdej ceny paliwa pamiętać najbliższą stację sprzedającą benzynę w tej samej ofercie. Różnych cen paliwa jest niewiele w porównaniu z potencjalną liczbą miejsc, w których można tankować.

Zastosowanie tablicy przechowującej tylko informacje o cenach paliwa pozwolą uzyskać stały koszt pamięciowy.

Tablica

$$Ceny\ :\ \textbf{array}[TCena]\ \textbf{of}\ \textbf{0}..CMAX\_BAK + CMAX\_DYSTANS;$$

informuje jak daleko można dojechać korzystając z benzyny o danej cenie.

Główna pętla programu, obliczająca potrzebną ilość pieniędzy, przedstawia się następująco:

```
gotowka := 0;
mila := 0;
Readln(PlikWe, doNastepnejStacji, min);
Ceny[min] := PojBaku;
Dec(PozostaloStacji);
repeat
  min := znajdzNajtansza;
  if mila + doNastepnejStacji < Ceny[min] then
    skok := doNastepnejStacji
  else
    skok := Ceny[min] − mila;
  gotowka := gotowka + min · skok;
until not jedz(skok);
```

Funkcja *znajdzNajtanszą* wyszukuje najtańszą stację położoną nie dalej niż B od rozważanego odcinka w stronę miasta A.

```
function znajdzNajtansza : TCena;
var
  i : TCena;
begin
  i := 1;
  while Ceny[i] ⩽ mila do Inc(i);
  znajdzNajtansza := i;
end;
```

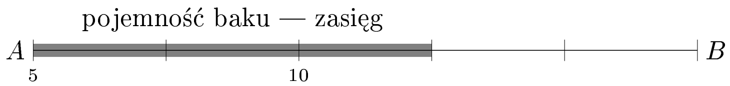
Funkcja *jedz* przesuwa samochód o zadaną liczbę mil.

```
function jedz(dystans : TDystans) : Boolean;
var
  cena : TCena;
begin
  mila := mila + dystans;
  doNastepnejStacji := doNastepnejStacji − dystans;
  if (doNastepnejStacji = 0) and (PozostaloStacji > 0) then begin
    Readln(PlikWe, doNastepnejStacji, cena);
    Ceny[cena] := mila + PojBaku;
    Dec(PozostaloStacji);
  end;
  jedz := (PozostaloStacji > 0) or (doNastepnejStacji ≠ 0)
end;
```

Można łatwo zauważyć, że poruszanie się co jedną jednostkę prowadziłoby do nieefektywnego programu. Często zdarza się, że koszt przejechania kolejnych odcinków trasy jest taki sam. Dlatego też, w przedstawionej implementacji, nie rozważamy każdego jednostkowego odcinka oddzielnie ale „przeskakujemy” do najbliższego miejsca, w którym należy uaktualnić dane (zmienna *skok*). Ważne są dwa typy sytuacji: dojeżdżamy do następnej stacji lub skończyło się paliwo o danej cenie i należy wybrać kolejne najtańsze.

Konieczność sprawdzania czy nie skończyła się benzyna o zadanej cenie ilustruje następujący przykład. Plik PAL.IN:

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



Przedstawiony algorytm działa w czasie $O(n)$ i zużywa pamięć o rozmiarze stałym, niezależnym od n.

Inne rozwiązanie o takich samych parametrach złożoności opiera się na założeniu, że na dowolnej stacji możemy odsprzedać zakupioną i niezużytą benzynę po cenie jej zakupu.

Analizujemy trasę przejazdu zgodnie z kierunkiem jazdy samochodu. Obserwacja 1 pozwala przyjąć, że paliwo zużywa się w takiej kolejności, w jakiej było zatankowane. Na każdej stacji:

- jeżeli posiadamy droższą benzynę niż ta na stacji, na której się zatrzymaliśmy, „odsprzedajemy ją” to bez straty;
- tankujemy do pełna.

Dla uproszczenia przyjmujemy, że w mieście B znajduje się stacja oferująca paliwo po cenie zerowej.

Powyższy algorytm jest równoważny czasowo i pamięciowo rozwiązaniu przedstawionemu jako pierwsze. Zauważmy, że sposób tankowanie zapewnia, iż paliwo w baku jest zawsze „ułożone” w kolejności od najtańszego do najdroższego. Po każdym tankowaniu mamy zatem najtańsze pokrycie dla znajdującego się przed nami odcinka drogi do następnej stacji.

Obserwacja 1 i wniosek pozwoliły stworzyć rozwiązanie optymalne, działające w stałej pamięci. Istnieją także inne, mniej efektywne rozwiązania. Na przykład rozwiązanie, które składa się z dwóch etapów. Przeglądamy drogę w kierunku od miasta B do A wyznaczając dla każdej stacji odległość do następnego tańszego miejsca tankowania, a następnie symulujemy jazdę samochodem. Na każdej stacji kupujemy tylko tyle paliwa, ile jest potrzebne aby dojechać do najbliższej tańszej. Jeżeli jest ona poza zasięgiem tankujemy do pełna.

Ponieważ musimy trzymać w pamięci całą potrzebną informację, koszt pamięciowy jest w tym przypadku liniowy, natomiast czas działania algorytmu jest asymptotycznie taki sam, jak poprzednio.

Wyszukiwanie następnej tańszej stacji można oczywiście prowadzić na bieżąco. W tym przypadku uzyskamy algorytm działający w czasie $O(n)$.

TESTY

- PAL0.IN–PAL5.IN — testy sprawdzające poprawność rozwiązania;
- PAL6.IN–PAL8.IN — testy sprawdzające szybkość działania algorytmu. Ta grupa testów eliminowała rozwiązania o złożoności czasowej $O(n^2)$;
- PAL9–PAL10.IN — testy sprawdzające zużycie pamięci. Wielkości testów zostały tak dobrane, aby wychwycić rozwiązania pamiętające dane o wszystkich stacjach.

Wojciech Rytter
treść zadania, opracowanie

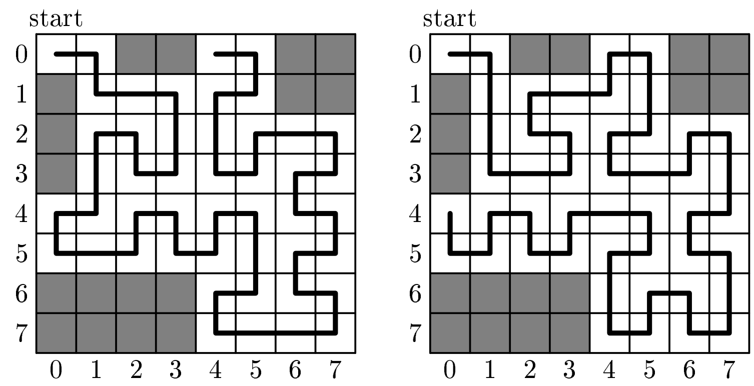
Krzysztof Sobusiak
program wzorcowy

Rekurencyjna mrówka

Mrówka ma obejść wszystkie pola szachownicy $2^n \times 2^n$ poza polami zabronionymi — każde pole dokładnie jeden raz. Obchodzenie musi się zacząć w polu startowym leżącym w lewym górnym rogu i zakończyć na polu leżącym na brzegu szachownicy (mrówka na końcu opuszcza szachownicę). Zakładamy, że pole startowe nie jest zabronione. W jednym kroku mrówka może przejść do jednego z co najwyżej czterech sąsiednich pól szachownicy (w górę, dół, lewo lub prawo).

Mrówka obchodzi szachownicę w sposób w pewnym sensie rekurencyjny: aby obejść kwadrat $2^k \times 2^k$ dzieli go na cztery części (rekurencyjne ćwiartki) o rozmiarach $2^{k-1} \times 2^{k-1}$, a następnie obchodzi każdą z nich, to znaczy, jeżeli mrówka wejdzie do jednej z ćwiartek, to może z niej wyjść dopiero wtedy, gdy obejdzie wszystkie nie zabronione pola w tej ćwiartce.

Rysunek 1 przedstawia dwie trasy rekurencyjnej wędrówki mrówki na szachownicy o rozmiarach $2^3 \times 2^3$, od pola startowego (0,0) do pola leżącego odpowiednio na górnym oraz lewym brzegu szachownicy.



Rysunek 1

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego REK.IN liczbę całkowitą dodatnią n określającą jednoznacznie rozmiar szachownicy $2^n \times 2^n$ oraz liczbę m pól zabronionych i ich listę,*
- *na każdym z czterech brzegów szachownicy znajduje pole, na którym może się zakończyć rekurencyjna wędrówka mrówki, spełniająca podane wyżej warunki lub stwierdza, że takiego pola nie ma,*
- *zapisuje wyniki w pliku tekstowym REK.OUT.*

UWAGA: *Każde z czterech pól narożnych należy do dwóch brzegów szachownicy.*

Wejście

W pierwszym wierszu pliku tekstowego REK.IN jest zapisana liczba całkowita dodatnia $n \leqslant 30$. W drugim wierszu jest zapisana liczba pól zabronionych $m \leqslant 50$.
W każdym z kolejnych m wierszy jest zapisana para liczb całkowitych nieujemnych i, j oddzielonych pojedynczym odstępem, są to dwie współrzędne: numer wiersza i oraz numer kolumny j odpowiedniego zabronionego pola. Wiersze szachownicy numerujemy od góry w dół, a kolumny od lewej do prawej, od 0 do $2^n - 1$.

Wyjście

W każdym z czterech kolejnych wierszy pliku tekstowego REK.OUT należy zapisać dwie liczby całkowite nieujemne oddzielone pojedynczym odstępem — współrzędne (numer wiersza i numer kolumny) odpowiedniego końcowego pola rekurencyjnej trasy mrówki leżącego na: górnym, prawym, dolnym oraz lewym brzegu szachownicy (w takiej kolejności) lub jedno słowo NIE, jeśli takiego pola nie ma.

Przykład

Dla pliku REK.IN będącego opisem szachownicy przedstawionej na rysunku:

```
3
15
2 0
1 0
3 0
6 0
7 0
6 1
7 1
6 2
7 2
6 3
7 3
0 2
0 3
0 6
0 7
1 6
1 7
```

poprawnym rozwiązaniem jest plik REK.OUT

```
0 4
NIE
NIE
4 0
```

Twój program powinien szukać pliku REK.IN w katalogu bieżącym i tworzyć plik REK.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę REK.??? gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonywalnej powinien być zapisany w pliku REK.EXE.

ROZWIĄZANIE

Miarę złożoności danych wejściowych określają dwie liczby: n, m, gdzie m jest liczbą zabronionych pól na szachownicy o rozmiarach $2^n \times 2^n$. Największą trudnością (oraz najbardziej istotnym punktem) w zadaniu jest potencjalnie olbrzymi rozmiar szachownicy względem n. Zadanie jest o tyle łatwe, że liczba tras zaczynających się w danym polu na zadanym boku szachownicy i kończących się na zadanym (innym lub tym samym) boku nie przekracza 1. W pewnym sensie mrówka chodzi deterministycznie, jeśli ustalimy na którym boku ma się zakończyć trasa. Trzeba wtedy zacząć chodzić zgodnie z regułami, ale czasami trzeba w jednym kroku pokonywać duże puste podszachownice (w przeciwnym wypadku trasa może być astronomicznie długa). Przez **pustą podszachownicę** rozumiemy część szachownicy bez zabronionych pól. Można łatwo wyliczyć gdzie wychodzimy z pustej ćwiartki nie obchodząc jej bezpośrednio. Możemy skorzystać z następującego faktu kombinatorycznego.

Fakt 1: Przy ustalonym polu początkowym leżącym na brzegu pustej szachownicy oraz ustalonym brzegu b, na którym mrówka ma zakończyć swoją podróż, istnieje dokładnie jedna poprawna droga rekurencyjnej mrówki, kończąca się na brzegu b (zob. rys. 1).

Dowód: Indukcja ze względu na n. Pozostawiamy go jako ćwiczenie dla czytelnika.□

Podstawowym elementem rozwiązania jest ustalenie brzegu szachownicy na którym ma być pole końcowe. Jeśli ten brzeg ustalimy, to trasa mrówki jest wyznaczona jednoznacznie, lub może jej nie być. Mamy cztery możliwości ustalenia brzegu i w związku z tym cztery wywołania podobnego algorytmu. W dalszym tekście zakładamy, że końcowy brzeg jest ustalony.

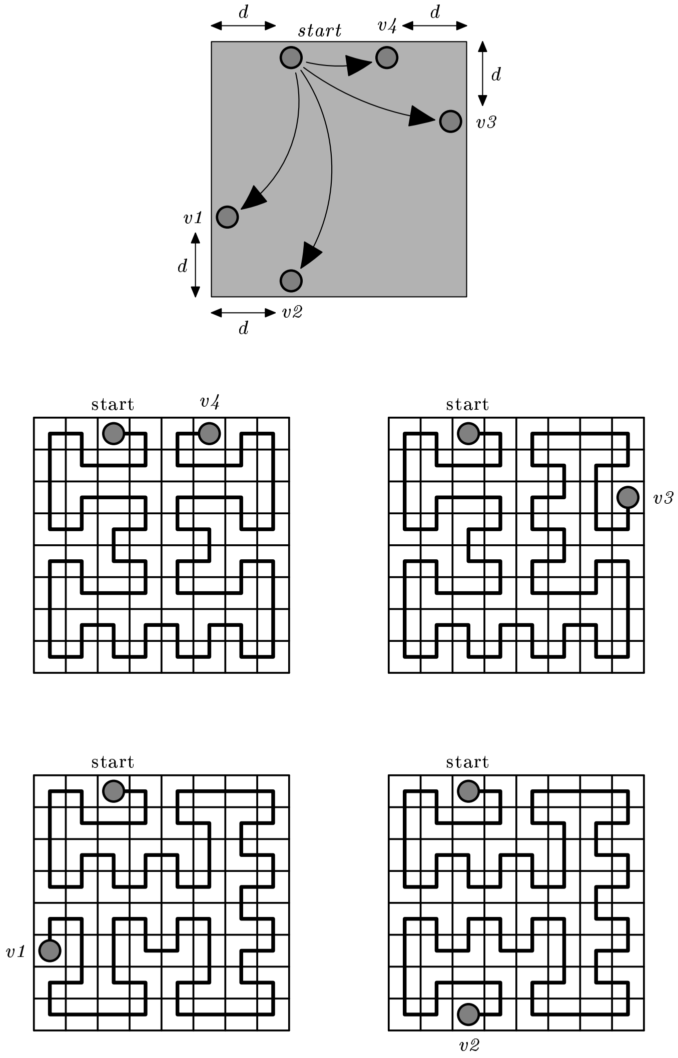
Sformułowanie zadania narzuca rozwiązanie rekurencyjne. Aby znaleźć drogę mrówki na szachownicy należy rekurencyjnie obejść jej cztery ćwiartki, za wyjątkiem tych ćwiartek, których wszystkie pola są zabronione. Kolejność przechodzenia ćwiartek jest jednoznacznie wyznaczona przez brzeg końcowy.

Przy przechodzeniu przez puste ćwiartki wykonujemy **przeskok**: w jednym kroku przechodzimy do pola końcowego na zadanym brzegu końcowym danej ćwiartki zgodnie ze schematem z rys. 1. Przykładowa strategia przeskoków jest pokazana przykładowo na rys. 2.

IMPLEMENTACJA

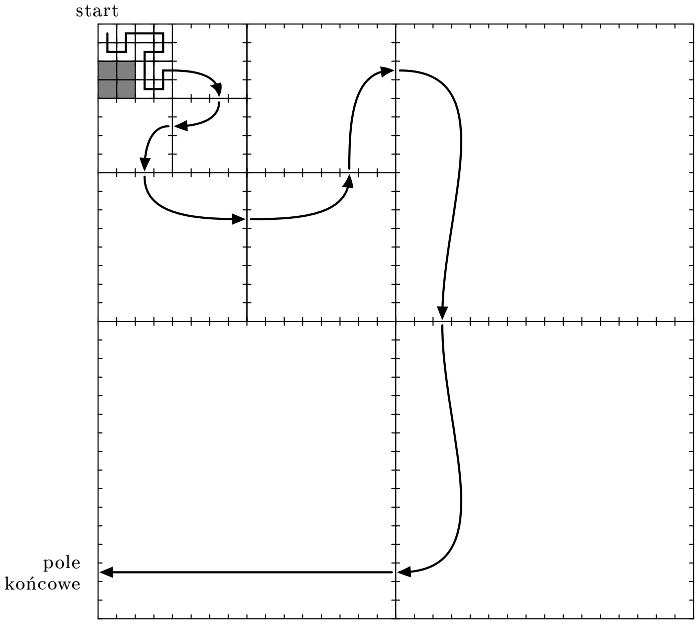
Podstawową częścią programu jest rekurencyjna procedura znajdowania pola końcowego mrówki dla następujących danych: pola początkowego, kierunków wejścia i wyjścia, poziomu rekurencji oraz listy pól zabronionych. Procedura jest uruchamiana czterokrotnie dla każdego z czterech brzegów szachownicy. Procedura ta sprawdza najpierw, czy przekazana jej lista pól zabronionych nie jest pusta. Jeśli lista jest pusta, to oblicza współrzędne pola końcowego zgodnie rys. 1. W przeciwnym razie rozdziela przekazaną jej listę pól zabronionych zawartych w części szachownicy, dla której ma znaleźć trasę mrówki, pomiędzy jej cztery ćwiartki.

Rys. 1 Schemat ogólny i konkretny przykład z $d = 2$ dla szachownicy bez zabronionych pól. Jeśli rozpoczniemy w polu *start* to możemy zakończyć jedynie w jednym z pól v_1, v_2, v_3, v_4, jeśli *start* jest polem narożnym to $v_1 = v_2$ oraz $v_3 = v_4$.



Operację tę można wykonać szybko, przez sprawdzenie dla każdego pola jednego bitu (o numerze zależnym od poziomu rekurencji) z jego współrzędnych. Procedura sprawdza też, czy któraś z ćwiartek nie jest w całości pokryta przez zabronione pola. Następnie ustala kierunek przechodzenia ćwiartek i obchodzi je rekurencyjnie, przekazując każdej swej rekurencyjnej kopii odpowiednią listę pól zabronionych.

Dla pojedynczego pola zabronionego złożoność tej procedury będzie liniowa (n wywoływań rekurencyjnych). Zatem przy m polach, pesymistycznie dostajemy czas $O(mn)$. Operacje rozdzielania list pól zabronionych amortyzują się do czasu $O(mn)$, Rys. 2 Obliczanie końcowego pola na lewym brzegu kwadratu o rozmiarach 32×32 mającego 4 pola zabronione. Pole kończace trasę mrówki w każdym z dużych podkwadratów liczymy (w czasie stałym) zgodnie z rys. 1.



gdyż każdy bit ze współrzędnych tych pól jest rozważany tylko raz.

UWAGA: Zakładamy, że operacje na liczbach kodujących współrzędne wykonujemy w czasie stałym.

TESTY

Do sprawdzenia rozwiązań zawodników użyto 11 testów REK0.IN–REK10.IN:

- REK0.IN — test z treści zadania;
- REK1.IN — szachownica zdegenerowana, $n = 0$;
- REK2.IN — niewielki test ($n = 3$), z czterema różnymi poprawnymi trasami mrówki;
- REK3.IN — nieduży test ($n = 4$), z jedną trasą mrówki;
- REK4.IN — mały test ($n = 4$), zawierający wszelkie dopuszczalne konfiguracje pól zabronionych w kwadracie 2×2;
- REK5.IN — test bez poprawnych tras mrówki ($n = 5$);

- REK6.IN — test z jednym polem zabronionym, bez poprawnych tras mrówki ($n = 30$);
- REK7.IN — losowy test gęsty ($n = 5$, $m = 50$);
- REK8.IN — test losowy ($n = 15$, $m = 25$);
- REK9.IN — test losowy ($n = 30$, $m = 50$);
- REK10.IN — szachownica bez pól zabronionych ($n = 30$).

Zawody III stopnia

opracowania zadań

Piotr Chrząstowski-Wachtel
treść zadania, opracowanie

Marcin Kubica
program wzorcowy

Alibaba

Aby otworzyć Sezam, trzeba mieć komplet co najmniej z żetonów złotych, s — srebrnych i m — miedzianych.
Alibaba ma początkowo pewną liczbę żetonów każdego rodzaju i może je wymieniać u strażnika Sezamu według ściśle określonych reguł.
Każda reguła jest postaci:

$$z_1, s_1, m_1 \rightarrow z_2, s_2, m_2 \qquad (z_i, s_i, m_i \in \{0, 1, 2, 3, 4\})$$

gdzie: z_1, s_1, m_1 oznaczają odpowiednio liczby żetonów złotych, srebrnych i miedzianych, jakie Alibaba musi dać strażnikowi, zaś z_2, s_2, m_2 — liczby żetonów złotych, srebrnych i miedzianych, które otrzyma w wyniku transakcji wymiany.
Żetony otrzymane w wyniku transakcji, można wymieniać w kolejnych transakcjach.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego ALI.IN zestawy danych, z których każdy zawiera:*
  - *liczby żetonów złotych, srebrnych i miedzianych znajdujących się początkowo w posiadaniu Alibaby,*
  - *opis kompletu żetonów otwierającego Sezam, oraz reguły transakcji;*
- *dla każdego zestawu danych stwierdza, czy istnieje skończony ciąg transakcji, który pozwoli Alibabie zgromadzić komplet żetonów otwierający Sezam i jeśli tak, znajduje i zapisuje w pliku tekstowym ALI.OUT minimalną długość takiego ciągu, a w przeciwnym przypadku zapisuje w pliku ALI.OUT odpowiedź NIE.*

Wejście

W pierwszym wierszu pliku tekstowego ALI.IN jest zapisana jedna liczba całkowita dodatnia $d \leqslant 10$. Jest to liczba zestawów danych.
Dalej są zapisane kolejno zestawy danych. Każdy zestaw danych składa się z wielu wierszy.
W pierwszym wierszu są zapisane trzy liczby całkowite nieujemne $z_p, s_p, m_p \in \{0, 1, 2, 3, 4\}$. Są to liczby żetonów złotych, srebrnych i miedzianych, jakie na początku ma Alibaba.
W drugim wierszu są kolejne trzy liczby całkowite $z, s, m \in \{0, 1, 2, 3, 4\}$. Są to liczby żetonów złotych, srebrnych i miedzianych, jakie trzeba mieć, aby otworzyć Sezam.
W trzecim wierszu jest zapisana liczba reguł r, gdzie $1 \leqslant r \leqslant 10$.
W każdym z kolejnych r wierszy jest zapisany ciąg sześciu liczb $z_1, s_1, m_1, z_2, s_2, m_2$, gdzie $z_i, s_i, m_i \in \{0, 1, 2, 3, 4\}$. Każda taka szóstka liczb jest zapisem jednej reguły transakcji: $z_1, s_1, m_1 \rightarrow z_2, s_2, m_2$.
Liczby w każdym wierszu są pooddzielane pojedynczymi odstępami.

Wyjście

W i-tym wierszu pliku tekstowego ALI.OUT należy zapisać wynik dla i-tego zestawu danych:

- *jedną liczbę całkowitą nieujemną, tj. minimalną liczbę transakcji wymiany, jakich Alibaba musi dokonać, aby zgromadzić komplet żetonów otwierający Sezam,*
- *albo jedno słowo NIE.*

Przykład

Dla pliku tekstowego ALI.IN:

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

poprawnym rozwiązaniem jest plik tekstowy ALI.OUT:

```
NIE
9
```

Twój program powinien szukać pliku ALI.IN w katalogu bieżącym i tworzyć plik ALI.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę ALI.??? gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonalnej powinien być zapisany w pliku ALI.EXE.

ROZWIĄZANIE

Zadanie o Alibabie było jednym z łatwiejszych zadań trzeciego etapu. Kilkunastu uczestników uzyskało za jego rozwiązanie maksymalną liczbę punktów, co nota bene nie zawsze oznaczało bezbłędność rozwiązania. Zadanie przypominało te zadania z poprzednich olimipiad, które rozwiązywało się za pomocą przeszukiwania grafu wszerz. Wytrawni uczestnicy, gdy widzą w treści sformułowanie, które wymaga znalezienia najkrótszej drogi prowadzącej do jakiegoś celu, od razu zastanawiają się nad grafem, który można by zbudować, aby zastosować znany algorytm.

Większość uczestników wpadła na to, że tym razem węzłami grafu będą trójki liczb określające liczbę poszczególnych rodzajów żetonów, które akurat są w posiadaniu Alibaby.

Tym razem główna trudność polegała na tym, żeby zatrzymać przeszukiwanie grafu w odpowiednim momencie. Graf bowiem potencjalnie był nieskończony i, w przypadku odpowiedzi negatywnej, bez trudu można było wpaść w nieskończoną pętlę. Mogło się bowiem zdarzyć, że Alibaba przy zadanych regułach nie miał szans dostać się do Sezamu, choć mógł wykonywać nieskończoną sekwencję ruchów tworząc coraz to nowe konfiguracje.

Przykładem takiej sytuacji były następujące dane: Alibaba ma początkowo jeden żeton złoty. Jeden żeton złoty można wymienić na jeden srebrny lub jeden brązowy. Jeden srebrny można wymienić na dwa srebrne, a jeden brązowy na dwa brązowe. Aby otworzyć Sezam trzeba mieć jeden żeton srebrny i jeden brązowy. Widać, że Alibaba może uzyskać dowolną liczbę żetonów srebrnych bądź brązowych, ale nie może mieć jednocześnie i srebrnego i brązowego. Graf powstający przez stosowanie powyższych reguł jest nieskończony i naiwne przeszukiwanie wszerz prowadziłoby do zapętlenia.

Co można zatem zrobić, żeby uniknąć nieskończonej pętli w algorytmie? Istnieją co najmniej dwie metody. Pierwsza, to uciąć przeszukiwanie, gdy tylko zorientujemy się, że dalsze szukanie jest już beznadziejne. Tę metodę stosowała większość zawodników, z różnym zresztą skutkiem. Podstawowa trudność polega na rozpoznaniu sytuacji, w której już dalej szukać nie warto, bo wiadomo, że rozwiązania dalej na badanej ścieżce być nie może.

Najbardziej naturalnym pomysłem jest w tej metodzie ograniczenie wartości współrzędnych. Chodzi o wyznaczenie takiej wartości, żeby nie było konieczne dalsze rozważanie ścieżki, jeśli taką wartość osiągniemy na którejś współrzędnej. Taką wartością było przy przyjętych ograniczeniach danych 100. Jeżeli któraś ze współrzędnych osiągnęła 100, to dalsze szukanie było bezcelowe. Oto dane, dla których możliwe było osiągnięcie 100:

- początkowa konfiguracja Alibaby: $(0, 0, 4)$,
- reguły wymiany: $(0, 0, 0) \to (0, 1, 4)$, $(0, 4, 0) \to (1, 0, 4)$
- konfiguracja otwierająca Sezam: $(4, 4, 0)$.

Łatwo zauważyć, że aby osiągnąć wartość 4 na pierwszej współrzędnej, trzeba czterokrotnie skorzystać z drugiej reguły. Zastosowanie jej wymaga każdorazowo zgromadzenia 4 żetonów srebrnych, a wygenerowanie każdego z nich wymaga czterokrotnego zastosowania reguły pierwszej. Podsumowując, aby uzyskać jeden złoty żeton, trzeba czterokrotnie skorzystać z reguły pierwszej i raz z drugiej. Każde zastosowanie którejkolwiek z reguł zwiększa liczbę żetonów brązowych o 4. Wygenerowanie jednego żetonu złotego wiąże się zatem, czy chcemy tego, czy nie z otrzymaniem 20 żetonów brązowych. Po uzyskaniu w ten sposób 4 żetonów złotych mamy więc 84 żetony brązowe i 0 srebrnych. Pozostaje jeszcze czterokrotnie zastosować regułę pierwszą, aby uzyskać dodatkowo 4 żetony srebrne (i przy okazji 16 brązowych). Kończymy więc nasz algorytm mając 4 żetony złote, 4 srebrne i 100 brązowych.

Mając to ograniczenie 100, modyfikujemy algorytm przechodzenia grafu wszerz w następujący sposób. Zaczynamy od konfiguracji początkowej, którą, o ile nie otwiera od razu Sezamu, wkładamy do kolejki etykietując ją liczbą 0 (etykieta będzie oznaczała minimalną liczbę zamian doprowadzającą do związanej z nią konfiguracji). Następnie, aż do momentu, w którym znajdziemy element otwierający Sezam, lub kolejka nie stanie się pusta, wykonujemy następujące operacje:

(1) Pobierz z kolejki kolejny element v wraz z towarzyszącą mu etykietą,

(2) Dla każdej z możliwych wymian, które można dokonać z pobranego elementu:

- sprawdź, czy nie została osiągnięta konfiguracja otwierająca Sezam. Jeśli tak, to zakończ algorytm wypisując wartość etykiety badanej konfiguracji z początku kolejki, powiększoną o 1;
- jeśli Sezamu nie da się otworzyć za pomocą otrzymanej konfiguracji, to sprawdź, czy któraś ze współrzędnych nie przekracza 100. Jeśli tak, to zignoruj tę konfigurację.
- w przeciwnym razie wstaw do kolejki uzyskaną konfigurację, etykietując ją liczbą o jeden większą, niż etykieta, która towarzyszyła konfiguracji pobranej z kolejki.

Pokazanie w ścisły sposób, że faktycznie 100 jest właściwym ograniczeniem, nie jest proste. Wielu uczestników wpadło na inne, niepoprawne ograniczenia (do typowych należały $16, 40, 64$) i zapewne też byli przekonani, że ich ograniczenia są właściwe.

Tutaj od razu pewna dygresja. Niestety wśród testów nie znalazł się taki, jak opisany powyżej, który by wymagał ograniczenia wartości współrzędnej do 100. W związku z tym uczestnicy, którzy przyjęli niepoprawnie mniejsze ograniczenia, mogli uzyskać (i czasem uzyskiwali) maksymalne liczby punktów trochę niezasłużenie. Słusznie ci, którzy przyjmowali właściwe ograniczenia mieli później pretensje, że testy nie wychwyciły wyższości ich rozwiązania nad niezupełnie poprawnymi rozwiązaniami kolegów. Jest to pewna, niestety trudna do uniknięcia wada sprawdzania za pomocą systemu testów. Jestem przekonany, że i w przeszłości takie przypadki, choć niekoniecznie uświadomione, mogły się wydarzyć. Jakkolwiek organizatorzy Olimpiady starają się tak dobrać testy, aby wychwycić potencjalnie złe rozwiązania, jednak nie zawsze udaje się doścignąć inwencję zawodników w generowaniu niepoprawnych kodów. Tym razem niedopatrzenie było widoczne gołym okiem. Niewykluczone, że w przyszłych edycjach Olimipiady system punktowania i przygotowywania testów zostanie nieco zmodyfikowany, na przykład przez dodanie punktu umożliwiającego wprowadzenie dodatkowych testów po obejrzeniu rozwiązań zawodników.

Powstaje przy okazji tego rozwiązania następujące pytanie. Na ile rozwiązanie nieco przypadkowo prawidłowe jest godne najwyższej oceny? To jest trudny problem. Wyobraźmy sobie taką sytuację. Dany problem można rozwiązać za pomocą dwóch algorytmów. Jeden z nich, wolniejszy, posiada dowód poprawności. Drugi, oparty na jakimś fajnym pomyśle, działa szybciej, ale nie za bardzo wiadomo dlaczego, i ogólnie nigdy nie ma pewności, czy jest do końca poprawny. Profesjonaliści zdecydowanie wolą pierwsze rozwiązanie. Najczęściej praca z niepewnym algorytmem jest dość stresująca i, na dłuższą skalę, nieopłacalna.

Czy potrafimy zatem przedstawić algorytm rozwiązujący to zadanie w sposób pewny? Chodzi tu jeszcze o jedną rzecz: łatwość uogólnienia. Czy bowiem bylibyśmy w stanie podać równie łatwo odpowiednie ograniczenie, gdyby liczba rodzajów żetonów była większa, a początkowe konfiguracje ograniczone przez liczby większe niż 4? Czy to ograniczenie byłoby w ogóle użyteczne (co nam po nim, jeśli powiedzmy byłoby rzędu 10^{12} zamiast 10^2)?

Okazuje się, że rozwiązać nasz problem można za pomocą algorytmu jakby dualnego do podanego powyżej. Zastosujemy również przechodzenie grafu wszerz, ale tym razem będziemy rozwiązania szukali od końca. Wyjdziemy zatem od Sezamu, jako konfiguracji początkowej i będziemy zastanawiali się, jakie konfiguracje mogą doprowadzić do Sezamu, albo do konfiguracji od niego większej.

W tym celu, mając w ręku konfigurację v pobraną z kolejki, będziemy badali, ile co najmniej trzeba mieć żetonów każdego rodzaju, aby stosując daną regułę zamiany otrzymać konfigurację równą v, lub od niej większą. Zilustrujmy to dwoma przykładami. Jeżeli $v = (4,4,4)$, a badana reguła zamiany jest postaci $(2,2,2) \rightarrow (2,1,3)$, to najmniejsza liczba żetonów, którą trzeba mieć aby otrzymać v lub więcej niż v, wynosi odpowiednio $(4,5,3)$. Jeżeli natomiast inna reguła zamiany jest postaci, $(6,2,4) \rightarrow (5,3,3)$, to taka najmniejsza konfiguracja jest równa $(6,3,5)$. Ogólny wzór wyliczający i-tą współrzędną takiej konfiguracji, to $\max(wez[i], daj[i] - wez[i] + v[i])$, gdzie $daj[i]$ oraz $wez[i]$, to odpowiednio liczby żetonów, które trzeba dać i które się dostanie stosując i-tą regułę. Wyliczenie tych wartości dokonuje się w procedurze *oktant* (w programie ALI.PAS) w czasie generowania nowych elementów kolejki.

Jeżeli tak otrzymana konfiguracja będzie mniejsza, niż początkowa konfiguracja Alibaby, to kończymy algorytm. Jeżeli nie, to zanim wstawimy ją do kolejki, sprawdzamy, czy nie jest ona gorsza od którejś z już przebadanych konfiguracji (w tym celu w programie tworzymy listę *osiag*, w której pamiętamy wszystkie osiągnięte konfiguracje). Gorsza, to znaczy mniejsza lub równa na wszystkich współrzędnych od konfiguracji już przebadanej, która ze względu na właściwości algorytmu przechodzenia grafu wszerz jest na domiar bliższa Sezamowi. Nie dopuszczamy zatem takiej sytuacji, że w kolejce znalazłyby się dwie porównywalne ze sobą konfiguracje (tzn. takie, że wszystkie współrzędne jednej z nich byłyby nie mniejsze, niż wszystkie współrzędne drugiej).

Powstaje pytanie, w czym podany algorytm różni się od poprzednio omawianego. Na pierwszy rzut oka może się wydawać, że struktury, z którymi mamy do czynienia, czyli kolejka oraz lista napotkanych konfiguracji, mogą potencjalnie urosnąć do nieograniczonych rozmiarów. Udowodnimy teraz, że tak się stać nie może: liczba nieporównywalnych ze sobą punktów o współrzędnych naturalnych nie może być nieskończona. Fakt ten znany w różnych wersjach pod nazwą lematu Dicksona jest bardzo użytecznym wynikiem. Podamy tu jego intuicyjny dowód. Uściślijmy na początku parę pojęć.

Rozważamy przestrzeń wektorów $\mathbb{N}^n$, czyli $n-$wymiarowych wektorów o współrzędnych całkowitych, nieujemnych. Dla $v \in \mathbb{N}^n$ przez $v[i]$ będziemy oznaczali i-tą współrzędną wektora v dla $i = 1, \cdots, n$. Dwa wektory $v, w \in \mathbb{N}^n$ nazwiemy **nieporównywalnymi**, jeśli istnieją takie $1 \leqslant i, j \leqslant n$, że $v[i] < w[i]$ oraz $v[j] > w[j]$. Czyli ze względu na jedną ze współrzędnych „lepszy” jest jeden wektor, a ze względu na inną — drugi. W przeciwnym razie powiemy, że wektory są **porównywalne**, i wtedy możemy je uszeregować według wielkości, bowiem wszystkie wartości współrzędnych jednego z nich nie mogą przekroczyć odpowiadających im wartości drugiego. W takim przypadku, podobnie jak dla liczb naturalnych, możemy mówić o relacjach mniejszości, niewiększości, niemniejszości i większości między dwoma wektorami.

Twierdzenie 1: (Lemat Dicksona) W każdym nieskończonym ciągu wektorów $\mathbb{N}^n$ istnieje nieskończony podciąg niemalejący.

Dowód: Indukcja ze względu na wymiar n.

Podstawa indukcji: $n = 1$. Jednowymiarowe wektory $\mathbb{N}^1$ można traktować, jak liczby naturalne. Załóżmy przeciwnie, że nie istnieje w pewnym nieskończonym ciągu $a_1, a_2, \ldots$ liczb naturalnych podciąg niemalejący. Musi wtedy istnieć element a_i taki, że dla wszystkich $j > i : a_j < a_i$. Gdyby takiego elementu nie było, to mielibyśmy sposób konstrukcji nieskończonego niemalejącego podciągu: jako pierwszy wzięlibyśmy element a_1, jako drugi — element nie mniejszy od niego o większym indeksie, jako trzeci — element nie mniejszy od drugiego o większym indeksie, niż drugi, itd. Skoro zatem żaden z elementów ciągu nie przekracza a_i, to zbiór wszystkich wartości występujących w ciągu jest skończony, a ponieważ sam ciąg jest nieskończony, to przynajmniej jedna z wartości musi powtórzyć się nieskończenie wiele razy. Podciąg tych wartości jest nieskończonym podciągiem niemalejącym, wbrew założeniu. Sprzeczność. Skorzystaliśmy tu z nieskończonego wariantu zasady szufladkowej Dirichleta.

Krok indukcyjny: Załóżmy prawdziwość twierdzenia dla wszystkich wymiarów aż do $n-1$ włącznie, $n > 1$. Wybierzmy z danego ciągu $a_1, a_2, \ldots$ nieskończony podciąg $a_{i_1}, a_{i_2}, \ldots$ niemalejący ze względu na początkowe $n-1$ wymiarów. Na mocy założenia indukcyjnego taki podciąg istnieje. Ciąg liczb naturalnych $a_{i_1}[n], a_{i_2}[n], \ldots$ zawiera nieskończony podciąg niemalejący $a_{i_{j_1}}[n], a_{i_{j_2}}[n], \ldots$. Rozważmy nieskończony podciąg wektorów $a_{i_{j_1}}, a_{i_{j_2}}, \ldots$. Ze względu na pierwszych $n-1$ współrzędnych jest on niemalejący, ale również ze względu na ostatnią współrzędną jest niemalejący. Stąd jest niemalejący ze względu na wszystkie współrzędne, co kończy dowód. $\square$

Z przedstawionego powyżej lematu wynika

Wniosek 2: Nie istnieje nieskończony zbiór nieporównywalnych wektorów z $\mathbb{N}^n$.

Gdyby taki zbiór miał istnieć, to ustawilibyśmy elementy tego zbioru w nieskończony ciąg, w którym znaleźlibyśmy nieskończony podciąg (a więc znaleźlibyśmy i podzbiór) niemalejący.

Z powyższego faktu wynika poprawność konstrukcji algorytmu. Nie ma niebezpieczeństwa, że algorytm nasz mógłby się zapętlić. Zbiór wektorów, które mają szansę znaleźć się w liście *osiag* jest skończony. Algorytm nasz musi się więc zakończyć w skończonym czasie. Z każdym krokiem algorytmu lista musiałaby się bowiem albo powiększyć, albo któryś z jej elementów musiałby ulec zmniejszeniu. Gdyby algorytm działał w nieskończoność, wówczas pierwsza z tych rzeczy byłaby wykluczona na mocy udowodnionego właśnie faktu, a druga na mocy tego, że skończonych wektorów nie jesteśmy w stanie nieskończenie wiele razy zmniejszać.

Zauważmy, że dowód poprawności (a tak naprawdę własności stopu) algorytmu jest niekonstruktywny. Nie wiemy, jakie jest ograniczenie czasu jego działania, nie wiemy, jak będą wyglądały budowane zbiory, wiemy tylko, że nie mogą urosnąć one do nieskończonych rozmiarów.

Oczywiście czasami, z pewnego punktu widzenia, ze względu na dużą złożoność algorytmu, możemy mówić o jego praktycznej nieskończoności. Gdyby na przykład okazało się, że dane początkowe wymagają, żeby przechowywany zbiór miał powiedzmy 2^{100} elementów, to praktycznie byłby to zbiór nieskończony. I to, czy sam algorytm jest nieskończony, czy też jest skończony, ale niewyobrażalnie duży, nie miałoby znaczenia. Po prostu i w jednym i w drugim przypadku byłby nie do użycia. W naszym zadaniu, przyjęte dane powodują, że niewątpliwie zbiór ten nie będzie zbyt duży. Oszacowanie 100^3 (na każdej współrzędnej co najwyżej 100) jest bardzo grubym oszacowaniem i w rzeczywistości lista *osiag* nie przekracza nigdy kilkuset elementów.

TESTY

Do sprawdzenia rozwiązań zawodników użyto testów zebranych w sześciu plikach ALI0.IN–ALI5.IN

- ALI0.IN — 2 testy z treści zadania,
- ALI1.IN — 4 proste testy testy niskowymiarowe: z jedną lub dwiema współrzędnymi aktywnymi;
- ALI2.IN — 2 niewielkie testy: pierwszy na „nie”, drugi generujący sporą liczbę wymian;
- ALI3.IN — 2 testy, w których długość ścieżki była dość duża (ponad 100).
- ALI4.IN — 2 testy badające wydajność algorytmu. Oba generowały duże grafy osiągalności (przy opisanym algorytmie działającym wstecz), przy czym pierwszy z nich nie miał rozwiązania, a drugi — miał;
- ALI5.IN — 2 spore testy z nieco większą liczbą reguł wymiany.

Wojciech Rytter **Marcin Mucha**
treść zadania, opracowanie program wzorcowy

Kajaki

Organizując spływ, wypożyczamy na przystani kajaki. Wszystkie kajaki są jednakowe. W jednym kajaku mogą popłynąć co najwyżej dwie osoby, a suma ich wag nie może przekroczyć ustalonego maksymalnego obciążenia. Aby zapłacić jak najmniej, szukamy sposobu rozmieszczenia wszystkich uczestników spływu w minimalnej liczbie kajaków.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego KAJ.IN maksymalne obciążenie kajaka, liczbę uczestników spływu i wagę każdego z nich,*
- *oblicza minimalną liczbę kajaków, jakie należy wynająć, aby rozmieścić w nich wszystkich uczestników spływu w zgodzie z przepisami,*
- *zapisuje wynik w pliku tekstowym KAJ.OUT.*

Wejście

W pierwszym wierszu pliku tekstowego KAJ.IN jest zapisana jedna liczba całkowita w, spełniająca nierówności $80 \leqslant w \leqslant 200$. Jest to maksymalne obciążenie kajaka.
W drugim wierszu jest zapisana jedna liczba całkowita n, spełniająca nierówności $1 \leqslant n \leqslant 30000$. Jest to liczba uczestników spływu.
W każdym z kolejnych n wierszy jest zapisana jedna liczba całkowita z przedziału $[5..w]$. Jest to waga jednego uczestnika spływu.

Wyjście

W pierwszym wierszu pliku tekstowego KAJ.OUT należy zapisać jedną liczbę całkowitą — minimalną liczbę kajaków, jakie trzeba wynająć.

Przykład

Dla pliku tekstowego KAJ.IN:

```
100
9
90
20
20
30
50
60
70
```

```
80
90
```

poprawnym rozwiązaniem jest plik tekstowy KAJ.OUT

```
6
```

Twój program powinien szukać pliku KAJ.IN w katalogu bieżącym i tworzyć plik KAJ.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę KAJ.??? gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonalnej powinien być zapisany w pliku KAJ.EXE.

ROZWIĄZANIE

Zadanie jest bardzo proste. Do jego rozwiązania możemy zastosować metodę **przeciwnych wag**: najlżejszą osobę wsadzamy do pierwszego kajaka z jak najcięższą osobą, która się z nią zmieści (tzn. nie zostanie przekroczone dopuszczalne obciążenie). Następnie kontynuujemy postępowanie dla grupy pozostałych osób. Na końcu zostaną pojedyncze „grubasy”, których wsadzamy do pojedynczych kajaków.

Powyższe rozwiązanie można zaimplementować na różne sposoby. Najprościej uporządkować osoby niemalejąco względem wag, a następnie poszukiwać pasażerów kajaków wybierając lekkich kandydatów z lewej strony, a ich partnerów od strony prawej.

Można podać jeszcze prostszy algorytm. W jednym kroku wybieramy **jakąkolwiek** wolną osobę, po czym wsadzamy ją do kajaka razem z jak najcięższą jeszcze wolną osobą, która się z nią mieści. Poprzednie rozwiązanie jest szczególnym przypadkiem tej ogólnej strategii.

Dlaczego taka strategia jest poprawna? Załóżmy, że na początku bierzemy osobę i. Jeśli i nie mieści się do kajaka z nikim innym, to w każdym optymalnym układzie i płynie samotnie, tak więc nasz optymalny układ dla pozostałych osób zagwarantuje optymalność rozwiązania globalnego. Jeśli i wsadzimy do kajaka z osobą j, a w pewnym optymalnym układzie i płynie z j', to w tym układzie możemy zamienić j z j', ponieważ waga j' jest nie większa od wagi j.

IMPLEMENTACJA

Efektywność implementacji przedstawionego algorytmu zależy od doboru metody sortowania, ponieważ dobieranie osób do kajaków wykonuje się w czasie liniowym. W rozwiązaniu wzorcowym zastosowano sortowanie kubełkowe. Poza tym nie kompletowano pojedynczych kajaków, ale całe grupy. Jeśli przykładowo mamy dziesięciu uczestników spływu o wadze 60 i pięciu o wadze 90, a maksymalne obciążenie kajaka wynosi 150, to możemy od razu obsadzić pięć kajaków.

TESTY

Do sprawdzenia rozwiązań zawodników użyto 10 testów (KAJ0.IN–KAJ9.IN). Test KAJ0.IN jest przykładem z treści zadania. Testy KAJ1.IN–KAJ4.IN są prostymi testami sprawdzającymi poprawność implementacji algorytmu. Testy KAJ5.IN–KAJ9.IN są dużymi (po 30 tys. uczestników spływu) testami wydajności badającymi głównie efektywność zastosowanej metody sortowania.

Wojciech Rytter
treść zadania, opracowanie

Krzysztof Sobusiak
program wzorcowy

Liczba zbiorów n-k-specjalnych

Mówimy, że zbiór liczb naturalnych X jest n-k-specjalny, gdy:

- *każdy element $x \in X$ jest liczbą naturalną spełniającą nierówność $1 \leqslant x \leqslant n$,*
- *suma wszystkich liczb należących do X jest większa od k,*
- *X nie zawiera żadnej pary kolejnych liczb naturalnych.*

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego LIC.IN dwie liczby naturalne n oraz k,*
- *znajduje liczbę wszystkich zbiorów n-k-specjalnych,*
- *zapisuje wynik w pliku tekstowym LIC.OUT.*

Wejście

W pierwszym wierszu pliku tekstowego LIC.IN są zapisane dwie liczby naturalne n oraz k oddzielone pojedynczym odstępem, gdzie $1 \leqslant n \leqslant 100$, $0 \leqslant k \leqslant 400$.

Wyjście

W pierwszym wierszu pliku tekstowego LIC.OUT należy zapisać jedną liczbę całkowitą nieujemną — liczbę wszystkich zbiorów n-k-specjalnych dla danych n oraz k.

Przykład

Dla pliku tekstowego LIC.IN:

```
5 6
```

poprawnym rozwiązaniem jest plik tekstowy LIC.OUT

```
3
```

Twój program powinien szukać pliku LIC.IN w katalogu bieżącym i tworzyć plik LIC.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę LIC.??? gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonalnej powinien być zapisany w pliku LIC.EXE.

ROZWIĄZANIE

Rozpoczniemy od wprowadzenia niezbędnych definicji. W tym opracowaniu pisząc „zbiór” mamy na myśli zawsze podzbiór liczb całkowitych. **Wagą** zbioru nazywamy sumę jego elementów. Wagą zbioru pustego jest 0. Powiemy, że zbiór jest **specjalny**, jeżeli nie zawiera dwóch kolejnych liczb całkowitych. Zbiór jest j-**lekki**, gdzie j jest nieujemną liczbą całkowitą, jeżeli jego waga nie przekracza j.

Przez $S(i)$ będziemy oznaczali liczbę wszystkich podzbiorów specjalnych zbioru $\{1, 2, \ldots, i\}$ dla $i = 0, 1, \ldots$, natomiast $ILE(i, j)$ będzie oznaczało liczbę wszystkich j-lekkich podzbiorów zbioru $\{1, 2, \ldots, i\}$. Dodatkowo przyjmijmy $ILE(i, j) = 0$ dla $j < 0$ oraz $ILE(i, j) = 1$ dla $i < 0$ i $j \geqslant 0$.

Zadanie rozwiązuje się metodą tablicowania (lub inaczej programowania dynamicznego), wprowadzając pomocniczą tablicę dla zapamiętywania wyników pośrednich. Istota zadania polega na odpowiednim wydefiniowaniu dodatkowych pomocniczych danych oraz zależności rekurencyjnych. Zadanie jest w pewnym sensie podobne do liczenia liczb Fibonnaciego. Łatwo zauważyć następujący fakt:

Fakt 1: Liczba $S(i)$ podzbiorów specjalnych zbioru $\{1, 2, \ldots, i\}$ jest $(i+1)$-szą liczbą Fibonacciego, tzn. $S(0) = 1$, $S(1) = 2$, $S(i) = S(i-1) + S(i-2)$ dla $i > 1$.*

Nasz algorytm będzie wykonywać kolejne iteracje dla $i = 1, 2, \ldots n$. W i-tej iteracji policzymy $ILE(i, j)$ dla $1 \leqslant j \leqslant k$. Po zakończeniu ostatniej iteracji obliczymy wynik odejmując od odpowiedniej liczby Fibonacciego $S(n)$ liczbę $ILE(n, k)$, (zob. rys. 1). W i-tej iteracji obliczamy i-ty wiersz tablicy ILE korzystając z rekurencji:

$$ILE(i, j) = ILE(i-1, j) + ILE(i-2, j-i)$$

Rekurencja ta oznacza, że każdy j-lekki podzbiór zbioru $\{1, 2, \ldots, i\}$, albo zawiera i (wtedy po usunięciu i jest $(j-i)$-lekkim podzbiorem zbioru $\{1, .., i-2\}$), albo nie zawiera i (wtedy jest j-lekkim podzbiorem zbioru $\{1, .., i-1\}$).

UWAGA: W danej iteracji potrzebne nam są tylko trzy wiersze tablicy ILE: aktualnie obliczany wiersz i poprzednie dwa wiersze. Spostrzeżenie to istotnie zmniejsza zapotrzebowanie programu na pamięć.

Ze względu na możliwość wystąpienia dużych liczb naturalnych należy zaimplementować własną arytmetykę z operacjami + i −. Złożoność czasowa algorytmu jest rzędu $n \cdot k$.

Można wyróżnić trzy klasy rozwiązań, które prowadziły do programów niepoprawnych lub nieefektywnych:

- rozwiązania starające się trzymać w pamięci całą tablicę ILE, co przy dużych danych wejściowych wykraczało poza dostępną pamięć operacyjną;
- rozwiązania rekurencyjne o złożoności wykładniczej;

* Równanie rekurencyjne wynika z dodania liczby podzbiorów, do których nie należy element i, tj. $S(i-1)$, do liczby podzbiorów, do których ten element należy, tj. $S(i-2)$.

Rys. 1 Tablice *ILE* i *S* oraz przykładowy schemat liczenia $ILE(3,5)$. Jeśli $n = 5$, $k = 5$ (dane z przykładu w treści zadania) to wynikiem jest $S(5) - ILE(5,5) = 5$.

	1	2	3	4	5
1	2	2	2	2	2
2	2	2	3	3	3
3	2	3	4	4	4
4	2	3	5	6	6
5	2	3	5	7	8

ILE

1	2
2	3
3	5
4	8
5	13

S

- rozwiązania bez zaimplementowanej arytmetyki dużych liczb.

TESTY

Do sprawdzenia rozwiązań zawodników użyto 11 testów LIC.0IN–LIC10.IN. Test LIC0.IN był testem z treści zadania. Celem testów LIC1.IN–LIC6.IN było sprawdzenie poprawności rozwiązań zawodników dla danych o stosunkowo niewielkich rozmiarach. Testy LIC7.IN–LIC10.IN to testy wydajności, sprawdzające zarówno szybkość działania programów, jak i zużycie pamięci.

Wojciech Guzicki
treść zadania, opracowanie

Tomasz Błaszczyk
program wzorcowy

Rezerwacja sal wykładowych

Rozporządzamy jedną salą wykładową. Wykładowcy, którzy chcą korzystać z sali, składają zamówienia określając czas rozpoczęcia i zakończenia wykładu. Układamy plan wykorzystania sali akceptując pewne wykłady i odrzucając inne, tak aby czas wykorzystania sali był jak najdłuższy. Zakładamy, że w momencie zakończenia jednego wykładu może się rozpocząć następny wykład.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego REZ.IN zamówienia wykładowców,*
- *oblicza maksymalny czas wykorzystania sali (przy odpowiednio ułożonym planie wykładów),*
- *zapisuje wynik w pliku tekstowym REZ.OUT.*

Wejście

W pierwszym wierszu pliku tekstowego REZ.IN jest zapisana jedna liczba całkowita dodatnia $n \leqslant 10000$. Jest to liczba zamówień.
W każdym z kolejnych n wierszy są zapisane dwie liczby całkowite p oraz k, oddzielone pojedynczym odstępem spełniające nierówności $0 \leqslant p \leqslant k \leqslant 30000$. Jest to zamówienie na salę wykładową w przedziale czasu od p do k.

Wyjście

W pierwszym wierszu pliku tekstowego REZ.OUT należy zapisać maksymalny czas wykorzystania sali.

Przykład

Dla pliku tekstowego REZ.IN:

```
12
1 2
3 5
0 4
6 8
7 13
4 6
9 10
9 12
11 14
```

```
15 19
14 16
18 20
```

poprawnym rozwiązaniem jest plik tekstowy REZ.OUT

```
16
```

Twój program powinien szukać pliku REZ.IN w katalogu bieżącym i tworzyć plik REZ.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę REZ.??? gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonalnej powinien być zapisany w pliku REZ.EXE.

UZUPEŁNIENIE TREŚCI

Podczas trwania zawodów treść zadania uzupełniono następującym stwierdzeniem:

> *Zamówienie w postaci pary liczb całkowitych p, k oznacza, że wykładowca potrzebuje sali w czasie reprezentowanym na osi czasu przez odcinek otwarty*

ROZWIĄZANIE

Zadanie to jest modyfikacją jednego z zadań z książki [10]. Oryginalne zadanie polegało na takim przydziale sal wykładowcom, by zadowolić jak największą liczbę zainteresowanych. W takim sformułowaniu zadanie można było rozwiązać za pomocą algorytmu zachłannego.

Obecne zadanie można rozwiązać za pomocą programowania dynamicznego. Dane wczytujemy do tablicy T typu

$$\mathbf{array}[1..MAXN]\ \mathbf{of}\ \mathit{TRekord}$$

gdzie typ *TRekord* jest zdefiniowany w następujący sposób:

$$\mathit{TRekord} = \mathbf{record}\ \mathit{Pocz}, \mathit{Kon} : \mathbf{Word\ end};$$

Będziemy także używać tablicy *Czas* typu

$$\mathbf{array}[\mathbf{0}..MAXN]\ \mathbf{of\ Word};$$

Oczywiście $T[i].Pocz$ oznacza czas rozpoczęcia i-tego wykładu, a $T[i].Kon$ czas zakończenia tego wykładu. Następnie tablicę tę sortujemy niemalejąco względem czasów zakończenia wykładów i zaczynamy kolejno wyznaczać liczby $Czas[i]$. Liczba $Czas[i]$

oznacza najlepszy możliwy czas wykorzystania sali przez wykładowców od pierwszego do i-tego (numery wykładowców po posortowaniu tablicy T). Oczywiście

$$Czas[1] = T[1].Kon - T[1].Pocz$$

Dla uproszczenia przyjmujemy także $Czas[0] = 0$. Dla $i > 1$ porównujemy dwie liczby. Pierwszą jest $t1 = Czas[i-1]$, oznaczająca optymalny czas wykorzystania sali w przypadku, gdybyśmy zrezygnowali z przydzielenia sali i-temu wykładowcy. Drugą, w programie oznaczoną przez $t2$, otrzymamy dodając czas trwania i-tego wykładu do najlepszego możliwego czasu wykorzystania sali przez wykłady kończące się nie później niż w chwili $T[i].Pocz$. Ten czas znajdujemy w następujący sposób. Zauważamy najpierw, że liczby w tablicy *Czas* są uporządkowane niemalejąco. Niemalejąco są również uporządkowane rekordy w tablicy T względem pola *Kon*. Możemy więc zastosować wyszukiwanie binarne. Funkcja *znajdź_indeks* znajduje najwcześniej występujący wykład kończący się później niż w chwili $T[i].pocz$. Jeśli jest to pierwszy wykład, to żaden nie mógł rozpocząć się przed i-tym wykładem i dotychczasowy najlepszy czas wynosi 0 (zauważmy, że jest to $Czas[0]$). Jeśli nie jest to pierwszy wykład, to w poprzednim miejscu w tablicy *Czas* znajduje się szukany najlepszy dotychczasowy czas. W obu przypadkach jest to $Czas[j-1]$, gdzie j jest liczbą znalezioną za pomocą funkcji *znajdź_indeks*, a więc najmniejszą liczbą taką, że $T[j].Kon > T[i].Pocz$. Teraz jako $Czas[i]$ przyjmujemy większą z liczb $t1$ i $t2$. Liczba $Czas[n]$ jest szukanym najlepszym czasem wykorzystania sali.

Nietrudno dowieść przez indukcję względem i, że na i-tym miejscu w tablicy *Czas* rzeczywiście znajduje się optymalny czas wykorzystania sali przez wykładowców od pierwszego do i-tego.

Implementacja tego algorytmu jest już prosta. Najpierw sortujemy wczytane dane. W programie wzorcowym zastosowano sortowanie z użyciem kopca (zob. [7]). Działa ono w miejscu (tzn. sortuje elementy tablicy nie korzystając z dodatkowej pamięci — używa tylko kilku dodatkowych zmiennych pomocniczych) i przy tym dość szybko. Czas działania $O(n \log n)$ nie zależy też w istotny sposób od danych, w przeciwieństwie do sortowania szybkiego (*quicksort*). W przypadku, gdy mamy do czynienia z tablicą posortowaną i w sortowaniu szybkim wybieramy jako element dzielący pierwszy element, to sortowanie szybkie będzie wymagało czasu kwadratowego. Oczywiście implementacje sortowania szybkiego wybierające element dzielący staranniej (np. losowo) też będą działały szybko.

Następnie wypełniamy tablicę *Czas* w sposób opisany powyżej. Wreszcie zapisujemy w pliku wyjściowym liczbę $Czas[n]$.

Złożoność obliczeniowa podanego algorytmu wynosi $O(n \log n)$. Sortowanie wstępne wykonujemy w tym właśnie czasie, a następnie n razy dokonujemy wyszukiwania binarnego w czasie $O(\log n)$.

Można rozwiązać to zadanie z pominięciem sortowania. Wtedy złożoność wyniesie $O(n^2)$. Można zastosować sortowanie szybkie. Przy niestarannym wyborze elementu dzielącego można też uzyskać czas rzędu n^2. Można też użyć algorytmu z sortowaniem, ale bez wyszukiwania binarnego (z wyszukiwaniem liniowym). Ten algorytm też ma złożoność $O(n^2)$.

Istnieje również niebezpieczeństwo zastosowania algorytmu zachłannego, służącego do rozwiązania zadania oryginalnego. Taki algorytm często da wynik błędny.

TESTY

Zastosowano 13 testów (REZ0.IN–REZ12.IN). Pierwszy (REZ0.IN) był testem z treści zadania. Drugi był testem złożonym z tylko jednego odcinka. Trzeci był posortowanym ciągiem 20 zamówień, z których każde zaczynało się w momencie zakończenia poprzedniego. Czwarty składał się z nieposortowanego ciągu dwóch serii zamówień o długości 600 i 601, przy czym jedna seria była przesunięta względem drugiej o 30 jednostek. Dwa następne testy miały wyeliminować rozwiązania korzystające z algorytmu zachłannego. Siedem następnych testów to testy składające się z dużej liczby odcinków różnie uporządkowanych. Dwa z tych testów (REZ10.IN–REZ11.IN) zawierały dane losowe.

Wojciech Guzicki
treść zadania, opracowanie

Marcin Engel
program wzorcowy

Trójkąty jednobarwne

W przestrzeni rozmieszczono n punktów w taki sposób, że żadne trzy z nich nie są współliniowe. Następnie każdą parę tych punktów połączono odcinkiem i każdy odcinek pokolorowano na czarno albo na czerwono. Trójkątem jednobarwnym nazwiemy każdy trójkąt mający wszystkie trzy boki tego samego koloru. Mamy daną listę wszystkich czerwonych odcinków. Chcemy znaleźć liczbę wszystkich trójkątów jednobarwnych.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego TRO.IN: liczbę punktów, liczbę odcinków czerwonych oraz ich listę,*
- *znajduje liczbę trójkątów jednobarwnych,*
- *zapisuje wynik w pliku tekstowym TRO.OUT.*

Wejście

W pierwszym wierszu pliku tekstowego TRO.IN jest zapisana jedna liczba całkowita n spełniająca nierówności $3 \leqslant n \leqslant 1000$. Jest to liczba punktów.
W drugim wierszu pliku tekstowego TRO.IN jest zapisana jedna liczba całkowita m spełniająca nierówności $0 \leqslant m \leqslant 250000$. Jest to liczba odcinków czerwonych.
W każdym z kolejnych m wierszy są zapisane dwie liczby całkowite p oraz k, oddzielone pojedynczym odstępem i spełniające nierówność: $1 \leqslant p \leqslant k \leqslant n$. Są to numery wierzchołków będących końcami kolejnego odcinka czerwonego.

Wyjście

W pierwszym wierszu pliku tekstowego TRO.OUT należy zapisać jedną liczbę całkowitą — liczbę trójkątów jednobarwnych.

Przykład

Dla pliku tekstowego TRO.IN:

```
6
9
1 2
2 3
2 5
1 4
1 6
3 4
```

```
4 5
5 6
3 6
```

poprawnym rozwiązaniem jest plik tekstowy TRO.OUT

```
2
```

Twój program powinien szukać pliku TRO.IN w katalogu bieżącym i tworzyć plik TRO.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę TRO.??? gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonalnej powinien być zapisany w pliku TRO.EXE.

ROZWIĄZANIE

Rozwiązanie narzucające się, polegające na przeszukaniu wszystkich trójek punktów, jest zbyt czasochłonne (czas rzędu n^3). Pomysł na szybszy algorytm polega na zliczaniu trójkątów różnobarwnych. Jeśli z wierzchołka o numerze k wychodzi $d(k)$ odcinków czerwonych i $n-1-d(k)$ odcinków czarnych, to liczba trójkątów różnobarwnych, o różnokolorowych bokach i wspólnym końcu w punkcie k, jest równa $d(k)\cdot(n-1-d(k))$. Liczba trójkątów różnobarwnych jest wtedy równa

$$S = \frac{1}{2} \cdot \sum_{k=1}^{n} d(k) \cdot (n-1-d(k))$$

Oczywiście liczby $d(k)$ wyznaczamy w czasie wczytywania danych o odcinkach czerwonych. Czas działania algorytmu będzie więc liniowy względem liczby danych (a więc co najwyżej $O(n^2)$). Liczbę trójkątów jednobarwnych wyznaczamy odejmując otrzymaną sumę od liczby wszystkich trójkątów:

$$J = \binom{n}{3} - S.$$

Wszystkie liczby mieszczą się w zakresie typu *longint*. Program wzorcowy znajduje się w pliku `TRO.PAS`, program działający w czasie rzędu n^3 znajduje się w pliku `TRO1.PAS`.

TESTY

Pierwszy test (TRO0.IN) był testem z treści zadania. Drugi, dla $n = 4$, nie miał odcinków czerwonych. Pięć następnych testów to testy losowe dla liczb n równych odpowiednio 10, 30, 100, 300, 600.

Grzegorz Jakacki
program wzorcowy, treść zadania, opracowanie

Wiarygodność świadków

Sąd dysponuje zeznaniami świadków oznaczonych kolejno liczbami naturalnymi $1, 2, \ldots, n$. Każde zeznanie jest stwierdzeniem postaci: „świadek i zgadza się ze świadkiem j" albo „świadek i nie zgadza się ze świadkiem j".
Jeśli świadek i zgadza się ze świadkiem j, oznacza to, że świadek i zgadza się też ze wszystkimi świadkami, z którymi zgadza się się świadek j oraz nie zgadza się ze wszystkimi świadkami, z którymi nie zgadza się świadek j.
Mówimy, że świadek nie jest wiarygodny*, jeśli z zeznań świadków, będących w posiadaniu sądu wynika, że jednocześnie zgadza się on i nie zgadza z pewnym dowolnym świadkiem.*

Zadanie

Ułóż program, który:

- *wczytuje z pliku tekstowego WIA.IN liczbę świadków i ich zeznania,*
- *znajduje wszystkich świadków, którzy nie są wiarygodni,*
- *zapisuje wynik w pliku tekstowym WIA.OUT*

Wejście

W pierwszym wierszu pliku tekstowego WIA.IN jest zapisana liczba całkowita n, spełniająca nierówność $1 \leqslant n \leqslant 3000$. Jest to liczba świadków.
W drugim wierszu pliku tekstowego WIA.IN jest zapisana jedna liczba całkowita m, spełniająca nierówność $0 \leqslant m \leqslant 8000$. Jest to liczba zeznań.
W każdym z kolejnych m wierszy jest zapisana para liczb całkowitych i j, oddzielonych pojedynczym odstępem, gdzie $1 \leqslant i \leqslant n$, $1 \leqslant |j| \leqslant n$. Jeśli liczba j jest dodatnia — jest to zapis zeznania: „świadek i zgadza się ze świadkiem j". Jeśli jest to liczba ujemna – jest to zapis zeznania: „świadek i nie zgadza się ze świadkiem $-j$".

Wyjście

W pliku tekstowym WIA.OUT należy zapisać:

- *jedno słowo NIKT, jeśli na podstawie zeznań nie można znaleźć żadnego świadka, który nie jest wiarygodny,*
- *albo — w porządku rosnącym — numery świadków, którzy nie są wiarygodni.*

Przykład

Dla pliku tekstowego WIA.IN:

```
6
12
```

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

Poprawnym rozwiązaniem jest plik tekstowy WIA.OUT:

```
1
3
4
6
```

UZUPEŁNIENIE TREŚCI

Podczas zawodów treść uzupełniono następującym stwierdzeniem:

Przyjmuje się przez domniemanie, że każdy świadek stwierdza: „Zgadzam się ze sobą”.

ROZWIĄZANIE

Zestaw zeznań świadków będziemy traktować jako tzw. graf skierowany (digraf) z etykietowanymi krawędziami. Wierzchołki grafu będziemy utożsamiać ze świadkami, natomiast jego krawędzie z zeznaniami. Krawędź od wierzchołka a do wierzchołka b ma oznaczać, iż świadek reprezentowany przez wierzchołek a wypowiada się o świadku reprezentowanym przez wierzchołek b. Typ wypowiedzi („zgadzam się” lub „nie zgadzam się”) będzie reprezentowany przez etykiety krawędzi. Etykietami będą słowa „tak” i „nie” jako skróty zeznań „tak, zgadzam się” i „nie, nie zgadzam się”.

Krawędź od wierzchołka a do wierzchołka b etykietowaną „tak” lub „nie” będziemy oznaczać odpowiednio $a \xrightarrow{T} b$ i $a \xrightarrow{N} b$. Ścieżkę (być może pustą) od wierzchołka a do wierzchołka b złożoną tylko z krawędzi typu „tak” lub „nie” będziemy oznaczać odpowiednio $a \xrightarrow{T}{}^{*} b$ i $a \xrightarrow{N}{}^{*} b$ (zakładamy, że dla każdego wierzchołka a istnieje ścieżka $a \xrightarrow{T}{}^{*} a$, bo świadek zgadza się z samym sobą). Gdy istnieje ścieżka $a \xrightarrow{T}{}^{*} b$, to będziemy mówić, że a **popiera** b.

Niech a będzie pewnym wierzchołkiem grafu. Zauważmy, że:

Obserwacja 1: Świadek a zgadza się ze świadkiem b wtedy i tylko wtedy, gdy istnieje ścieżka $a \xrightarrow{T}{}^{*} b$.

O świadku możemy stwierdzić, że nie jest wiarygodny, jeżeli jednocześnie zgadza się i nie zgadza z pewnym dowolnym świadkiem.

Jeżeli a zgadza się b, to nie zgadza się również ze świadkami, z którymi nie zgadza się b. Stąd otrzymujemy:

Obserwacja 2: Świadek a nie zgadza się ze świadkiem c jeżeli istnieje świadek b taki, że istnieje ścieżka $a \xrightarrow{T}^* b \xrightarrow{N} c$

Zauważmy, że jeżeli świadek a osobiście nie zgadza się ze świadkiem c, to obserwacja też pozostaje prawdziwa (bo świadkiem b jest w takim przypadku sam a).

Świadek nie jest wiarygodny, jeżeli jednocześnie zgadza się z jakimś świadkiem i z nim się nie zgadza. Zatem:

Obserwacja 3: Świadek a nie jest wiarygodny, jeżeli istnieją świadkowie b i c tacy, że istnieją ścieżki $a \xrightarrow{T}^* b \xrightarrow{N} c$ oraz $a \xrightarrow{T}^* c$.

W powyższej obserwacji świadek c jest tym świadkiem, z którym a jednocześnie zgadza się i nie zgadza.

Na tym etapie rozważań jesteśmy gotowi sformułować algorytm będący rozwiązaniem zadania. Dla każdego świadka a będziemy wyznaczać zbiór świadków, z którymi się on zgadza

$$\mathit{ZgadzaSię}(a) = \{x : a \xrightarrow{T}^* x\}$$

Świadka a należy uznać za niewiarygodnego, jeżeli pomiędzy elementami $\mathit{ZgadzaSię}(a)$ istnieje krawędź $\xrightarrow{N}$. Taka sytuacja oznacza, że wśród świadków, z którymi zgadza się a istnieją świadkowie b i c, tacy, że $a \xrightarrow{T}^* b \xrightarrow{N} c$, co rzeczywiście każe uznać a za świadka niewiarygodnego. Z drugiej strony, jeżeli pomiędzy elementami zbioru $\mathit{ZgadzaSię}(a)$ nie ma takiej krawędzi, to znaczy, że świadka a należy uznać za wiarygodnego.

Zbiór $\mathit{ZgadzaSię}(a)$ będziemy konstruować przy pomocy metody znanej jako „przechodzenie grafu w głąb” (ang. *depth-first search, DFS*), pozwalającej m.in. wyznaczyć zbiór wierzchołków grafu, do których istnieją ścieżki z zadanego wierzchołka (tzw. zbiór wierzchołków osiągalnych z danego wierzchołka).

Schemat algorytmu DFS przedstawiono na rys. 1. Algorytm uruchomiony dla wierzchołka a zaznacza go jako odwiedzony, a następnie uruchamia się rekurencyjnie dla wszystkich wierzchołków, do których prowadzą krawędzie z wierzchołka a i które nie zostały odwiedzone wcześniej.

Po zakończeniu pracy algorytmu wierzchołki odwiedzone tworzą zbiór wierzchołków osiągalnych z wierzchołka startowego.

Czas pracy algorytmu DFS uruchomionego dla jednego wierzchołka grafu jest rzędu $O(m)$, gdzie m jest liczbą krawędzi. Nie trudno to zauważyć wziąwszy pod uwagę, że każdą krawędź rozważamy tylko jeden raz.

Głębokość rekursji nie przekroczy na pewno długości najdłuższej ścieżki, na której nie powtarzają się wierzchołki, więc jest nie większa niż n.

Przed wywołaniem procedury DFS należy wszystkim wierzchołkom grafu ustawić atrybut *odwiedzony* na **false**. Założono, że *sąsiedzi*(w) oznacza zbiór wszystkich wierzchołków, do których prowadzą krawędzie z w. Taki zbiór można zaimplementować np. jako listę lub tablicę (jeżeli wiadomo z góry, ilu maksymalnie sąsiadów może mieć wierzchołek). Pętla w wierszu (1) musi, przy konkretnej implementacji, Rys. 1 Schemat algorytmu DFS

---

procedure *DFS*(w : *wierzcholek*)
var
  v : *wierzchołek*;
begin
  w.*odwiedzony* := **true**
  for $v \in$ *sąsiedzi*(w) **do** (1)
    if not v.*odwiedzony* **then** *DFS*(v)
end

---

realizować w jakiś sposób przeglądanie tego zbioru (kolejność przeglądania nie ma znaczenia).

W algorytmie będącym rozwiązaniem zadania posłużymy się zmodyfikowaną wersją algorytmu DFS. Po pierwsze, do zbioru *sąsiedzi*(a) zaliczymy jedynie te wierzchołki, do których można dojść z a po krawędziach $\xrightarrow{T}$, dzięki czemu po zakończeniu pracy algorytmu uruchomionego dla wierzchołka a wartość atrybutu *odwiedzony* ustawiona na **true** wskaże elementy zbioru *ZgadzaSię*(a). W rozwiązaniu wzorcowym zamiast nazwy atrybutu *odwiedzony* użyto nazwy *popierany* (gdyż ustawienie tego atrybutu oznacza, że świadek reprezentowany przez dany wierzchołek jest *popierany* w swych zeznaniach przez świadka, którego wiarygodność aktualnie się sprawdza).

Wyznaczenie zbioru *ZgadzaSię*(a) to jednak jeszcze nie wszystko. Kolejną fazą sprawdzania wiarygodności świadka jest sprawdzenie, czy pomiędzy elementami tego zbioru nie istnieje krawędź $\xrightarrow{N}$. By usprawnić tę operację, przy okazji konstruowania zbioru *ZgadzaSię*(a) algorytm zapisuje numery jego elementów do pomocniczej tablicy (dokładniej: w procedurze DFS oprócz zaznaczenia wierzchołka jako odwiedzonego, zapisuje się jego numer na pierwsze wolne miejsce w tablicy *ZgadzaSięZ*). W kolejnej fazie tablica jest przeglądana, a dla każdego zapisanego w niej wierzchołka sprawdza się, czy wierzchołki do których prowadzą zeń krawędzie $\xrightarrow{N}$ nie mają ustawionego atrybutu *popierany*. Ten sposób umożliwia szybkie odszukanie ew. krawędzi $\xrightarrow{N}$ pomiędzy elementami zbioru *ZgadzaSię*. Na tym kończy się sprawdzanie wiarygodności jednego świadka. Przed przystąpieniem do podobnej procedury dla następnego, należy ponownie zainicjalizować na **false** atrybut *popierany* oraz opróżnić tablicę *ZgadzaSięZ*. Zauważmy, że atrybut *popierany* wystarczy zainicjować tylko w tych wierzchołkach, których numery znajdują się w tablicy *ZgadzaSięZ*.

Złożoność sprawdzania wiarygodności dla jednego świadka jest rzędu $O(m)$, gdzie m jest liczbą zeznań. Dla każdego z n świadków trzeba ją jednak powtórzyć, co daje złożoność rzędu nm. Gdy każdy świadek wypowiada się o każdym, wówczas $m = n(n-1)$, co daje złożoność rzędu n^3.

Na rys. 2 przedstawiono schemat algorytmu. Atrybuty wierzchołka oraz zbiory wierzchołków sąsiadujących z danym wierzchołkiem przechowuje się w tablicy *świadkowie*. Zbiór sąsiadów zaimplementowano jako jednokierunkową listę numerów wierzchołków. Dla każdego wierzchołka oddzielnie przechowuje się listę sąsiadów, do których prowadzą krawędzie $\xrightarrow{N}$ (lista *nie*) oraz $\xrightarrow{T}$ (lista *tak*). Informację o tym, czy świadek został uznany za niewiarygodnego przechowuje się dla każdego wierzchołka

Rys. 2 Schemat rozwiązania

```
type
  wsk = ↑sąsiad;
  sąsiad = record
    nr : integer;
    nast : wsk
  end;
  świadek = record
    tak : wsk;
    nie : wsk;
    popierany : boolean;
    niewiarygodny : boolean
  end;
var
  świadkowie : array [1..MaxŚwiadków] of świadek;
  zgadzaSięZ : array [1..MaxŚwiadkow] of integer;
  zIluSięZgadza : integer;

procedure Odwiedź(s : integer);
var
  odwiedzany : wsk;
begin
  świadkowie[s].popierany := true;
  zIluSięZgadza := zIluSięZgadza + 1;
  zgadzaSięZ[zIluSięZgadza] := s;
  odwiedzany := świadkowie[s].tak;
  while (odwiedzany ≠ nil) do begin                    pętla (1)
    if not świadkowie[odwiedzany↑.nr].popierany then
      Odwiedź(odwiedzany↑.nr);
    odwiedzany := odwiedzany↑.nast
  end
end;
```

za pomocą dodatkowego atrybutu *niewiarygodny*.

Powyższy program można nieco poprawić zauważywszy, że jeżeli $a \xrightarrow{T} b$ oraz świadek b został już wcześniej uznany za niewiarygodnego, to a też jest niewiarygodny. Ulepszenie nie zmienia zachowania programu w przypadku pesymistycznym, jednakże w wielu sytuacjach może zapobiec wielokrotnemu przechodzeniu grafu. Wprowadzenie poprawki wymaga zmian w pętlach (1) i (2), opisanych na rys. 3.

Pełen tekst rozwiązania wzorcowego zamieszczono w pliku WIA.PAS na załączonej dyskietce.

Rys. 2 Schemat rozwiązania — c.d.

```
procedure TestujNaSprzeczność(s : integer);
var
  oskarżany : wsk;
  i : integer;
begin
  i := 1;
  while i < zIluSięZgadza and not (świadkowie[s].niewiarygodny) do
    oskarżany := świadkowie[zgadzaSięZ[i]].nie;
    while (oskarżany ≠ nil) and (notświadkowie[s].niewiarygodny) do begin
      świadkowie[s].niewiarygodny := świadkowie[oskarżany↑.nr].popierany;
      oskarżany := oskarżany↑.nast
    end;
    i := i + 1
  end
end;

begin
  for i := 1 to iluŚwiadkow do świadkowie[i].popierany := false;
  zIluSięZgadza := 0;
  for i := 1 to iluŚwiadkow do begin          pętla (2)
    Odwiedź(i);
    TestujNaSprzeczność(i);
    AnulujPoparcie
  end;
  for i := 1 to iluŚwiadkow do
  if not świadkowie[i].niewiarygodny then writeln(i)
end;
```

UWAGI

W poszukiwaniu metod zmniejszenia rzeczywistego czasu pracy algorytmu można rozważyć tzw. sklejenie silnie spójnych składowych grafu. Zauważmy, że jeżeli w grafie krawędzie $\xrightarrow{T}$ tworzą cykl, to obalenie wiarygodności jednego ze świadków w cyklu powoduje obalenie wiarygodności pozostałych (*jeden za wszystkich, wszyscy za jednego*). Podzbiór wierzchołków grafu skierowanego o tej własności, że z każdego jego elementu prowadzi ścieżka do dowolnego innego i odwrotnie, nazywamy **silnie spójną składową**. Właśnie silnie spójne składowe względem krawędzi $\xrightarrow{T}$ tworzą takie „grupy wspólnej odpowiedzialności”.

Optymalizacja polega na poprzedzeniu właściwej fazy algorytmu operacją sklejenia silnie spójnych składowych, tzn. zastąpienia każdej z nich jednym nowym wierz-

Rys. 3 Usprawnienia rozwiązania

modyfikacja pętli (1):

while (*odwiedzany* $\neq$ **nil**) **and not** *świadkowie*[*s*].*niewiarygodny* **do begin**
  if not *świadkowie*[*odwiedzany*↑.*nr*].*popierany* **then** *Odwiedź*(*odwiedzany*↑.*nr*);
  świadkowie[*s*].*niewiarygodny* := *świadkowie*[*odwiedzany*↑.*nr*].*niewiarygodny*;
  odwiedzany := *odwiedzany*↑.*nast*
end

modyfikacja pętli (2):

for *i* := 1 **to** *iluŚwiadków* **do begin**
  Odwiedź(*i*);
  if not *świadkowie*[*i*].*niewiarygodny* **then** *TestujNaSprzeczność*(*i*);
  AnulujPoparcie
end

chołkiem. Wszystkie krawędzie, których jednym z końców był jakiś element silnie spójnej składowej, zostają w wyniku operacji podłączone do nowego wierzchołka. Precyzyjny opis algorytmu znajdowania spójnych składowych zainteresowany czytelnik znajdzie w książce [3] (w opisie zadania „Agenci") lub [7].

Powyższa optymalizacja, niestety, nie zmniejsza kosztu działania algorytmu w przypadku pesymistycznym. Jest przy tym o wiele trudniejsza do zaimplementowania, niż optymalizacja polegająca na oznaczaniu niewiarygodnych świadków.

TESTY

Do sprawdzenia rozwiązań zawodników użyto 11 testów (WIA1.IN–WIA11.IN):

- WIA1.IN — tylko jeden świadek, brak zeznań;
- WIA2.IN — test z treści zadania;
- WIA3.IN — trzech świadków, każdy wypowiada się o każdym, brak niewiarygodnych;
- WIA4.IN–WIA6.IN — różne testy poprawnościowe.
- WIA7.IN, WIA8.IN — 400-tu świadków, połowa niewiarygodna.
- WIA9.IN, WIA10,IN — 3000 świadków, połowa niewiarygodna.
- WIA11.IN — 3000 świadków, 8000 zeznań, brak niewiarygodnych.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[20] R. J. Wilson *Wprowadzenie do teorii grafów*, PWN, Warszawa 1985

[21] N. Wirth *Algorytmy + struktury danych = programy*, WNT, Warszawa 1989

Niniejsza publikacja stanowi wyczerpujące źródło informacji o zawodach IV Olimpiady Informatycznej, przeprowadzonych w roku szkolnym 1996/1997. Książka zawiera zarówno informacje dotyczące organizacji zawodów, regulaminu oraz klasyfikacji, jak i treści oraz wyczerpujące omówienia zadań Olimpiady.

IV 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.

ISBN 83–906301–3–3