MINISTERSTWO EDUKACJI NARODOWEJ
INSTYTUT INFORMATYKI UNIWERSYTETU WROCŁAWSKIEGO
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

V OLIMPIADA INFORMATYCZNA
1997/1998

WARSZAWA, 1998

MINISTERSTWO EDUKACJI NARODOWEJ
INSTYTUT INFORMATYKI UNIWERSYTETU WROCŁAWSKIEGO
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

V OLIMPIADA INFORMATYCZNA
1997/1998

WARSZAWA, 1998

Autorzy tekstów:

Guzicki
Diks
Rytter
Chrząstowski
Błaszczyk
Sobusiak
Borowski
Jakacki
Walat
Kanarek
Kubica

Autorzy programów na dyskietce:

Sobusiak
Waleń
Błaszczyk
Borowski
Jakacki
Mucha

Opracowanie i redakcja:

dr Krzysztof Diks
Grzegorz Jakacki

Skład:

Grzegorz Jakacki

ISBN 83–906301–4–1

Spis treści

Krzysztof Diks, Grzegorz Jakacki

Wstęp

W roku szkolnym 1997/98 odbyła się V Olimpiada Informatyczna. Przekazujemy czytelnikom relację z jej przebiegu, a także opracowania zadań wykorzystanych w trakcie Olimpiady. W niniejszej publikacji prezentujemy również krótkie sprawozdania z IV Olimpiady Informatycznej Centralnej Europy oraz IX Międzynarodowej Olimpiady Informatycznej. Olimpiada Centralnej Europy była zorganizowana w Nowym Sączu w lipcu 1997 roku. Olimpiada Międzynarodowa odbyła się w Kapsztadzie w grudniu tego samego roku. W obu tych imprezach brali udział laureaci IV Olimpiady Informatycznej. Oprócz zadań z tegorocznej Olimpiady zamieszczamy także zadania z tych dwóch Olimpiad Międzynarodowych.

W opracowaniu zawarliśmy oficjalne dokumenty Komitetu Głównego: „Regulamin Olimpiady Informatycznej" oraz „Zasady organizacji zawodów w 1997/1998 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 wzorcowy i testy, które posłużyły do sprawdzenia poprawności i efektywności rozwiązań zawodników. 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 V Olimpiady Informatycznej oraz testy.

* 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.

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

- *w sieci księgarni „Elektronika” w Warszawie, Łodzi i Wrocławiu,*
- *w niektórych księgarniach technicznych,*
- *w niektórych sklepach ze sprzętem komputerowym,*
- *w Komitetach Okręgowych Olimpiady Informatycznej:*
  - *w Warszawie: Ośrodek Edukacji Informatycznej i Zastosowań Komputerów, 02-026 Warszawa, ul.Raszyńska 8/10, tel. (+22) 22–40–19*
  - *we Wrocławiu: Instytut Informatyki Uniwersytetu Wrocławskiego, 51-151 Wrocław, ul.Przesmyckiego 20, tel. (+71) 3–251–271*
  - *w Toruniu: Wydział Matematyki i Informatyki UMK, 87-100 Toruń, ul.Szopena 12/18 tel. (+56) centr. 260–17, 260–18 lub 265—84 wewn. 36, dr Bolesław Wojdyło,*
- *w sprzedaży wysyłkowej za zaliczeniem pocztowym w Komitecie Głównym Olimpiady Informatycznej. Zamówienia prosimy przesyłać pod adresem 02–026 Warszawa, ul. Raszyńska 8/10.*

Niestety, nakład publikacji o pierwszej Olimpiadzie jest już wyczerpany.

Sprawozdanie z przebiegu V Olimpiady Informatycznej 1997/1998

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 młodzieżowych.

7 października 1997 r. rozesłano plakaty zawierające zasady organizacji zawodów I stopnia oraz zestaw 4-ch zadań konkursowych do 3520-tu szkół i zespołów szkół młodzieżowych ponadpodstawowych oraz do wszystkich kuratorów i koordynatorów edukacji informatycznej. Zawody I stopnia rozpoczęły się dnia 20 października 1997 roku. Ostatecznym terminem nadsyłania prac konkursowych był 17 listopada 1997 roku.

Zawody II i III stopnia były dwudniowymi sesjami stacjonarnymi, poprzedzonymi jednodniowymi sesjami próbnymi. Zawody II stopnia odbyły się w trzech okręgach oraz w Krakowie, w dniach 12–14.02.1998 r., natomiast zawody III stopnia odbyły się w ośrodku firmy Combidata Poland S.A. w Sopocie, w dniach 6–9.04.1998 r.

Uroczystość zakończenia V Olimpiady Informatycznej odbyła się w dniu 9.04.1998 r. w sali posiedzeń Urzędu Miasta w Sopocie.

SKŁAD OSOBOWY KOMITETÓW OLIMPIADY INFORMATYCZNEJ

Komitet Główny

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

z-cy przewodniczącego:
prof. dr hab. Maciej M. Sysło (Uniwersytet Wrocławski)
dr 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 Przemysława Kanarek (Uniwersytet Wrocławski)
dr Krzysztof Loryś (Uniwersytet Wrocławski)
dr Bolesław Wojdyło (Uniwersytet Mikołaja Kopernika w Toruniu)
mgr Jerzy Dałek (Ministerstwo Edukacji Narodowej)
mgr Marcin Kubica (Uniwersytet Warszawski)
mgr Krzysztof Stencel (Uniwersytet Warszawski)
mgr Krzysztof J. Święcicki (Ministerstwo Edukacji Narodowej)

sekretarz Komitetu Głównego:
Monika Kozłowska-Zając

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

Komitet Główny odbył 6 posiedzeń, a Prezydium — 5 zebrań. 30 stycznia 1998 r. przeprowadzono jednodniowe seminarium przygotowujące przeprowadzenie zawodów II stopnia.

Komitet Okręgowy w Warszawie

przewodniczący:
dr Krzysztof Diks (Uniwersytet Warszawski)

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

---

* Ośrodek Edukacji Informatycznej i Zastosowań Komputerów w Warszawie

** Na własną prośbę zwolniony przez Komitet z obowiązków dn. 12.12.1997 r.

mgr Adam Malinowski (Uniwersytet Warszawski)
mgr Wojciech Plandowski (Uniwersytet Warszawski)

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

Komitet Okręgowy we Wrocławiu

przewodniczący:
dr Krzysztof Loryś (Uniwersytet Wrocławski)
z-ca przewodniczącego:
dr Helena Krupicka (Uniwersytet Wrocławski)
sekretarz:
inż. Maria Woźniak (Uniwersytet Wrocławski)
członkowie:
dr Witold Karczewski (Uniwersytet Wrocławski)
dr 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 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
Adam Borowski
mgr Jacek Chrząszcz
mgr Marcin Engel
Grzegorz Jakacki
mgr Albert Krzymowski
mgr Marcin Kubica
Marcin Mucha
Marek Pawlicki
Marcin Sawicki
Krzysztof Sobusiak
Radosław Szklarczyk
Tomasz Waleń

ZAWODY I STOPNIA

W V Olimpiadzie Informatycznej wzięło udział 741 zawodników. Decyzją Komitetu Głównego Olimpiady do zawodów zostało dopuszczonych 6 uczniów ze szkół podstawowych:

- S.P. nr 16 w Tarnowie: Wojciech Matyjewicz,
- S.P. nr 64 w Bydgoszczy: Kamil Blank,
- S.P. nr 5 w Jarocinie: Damian Szklarczyk,
- Publiczna S.P. nr 2 im. J. Ligonia w Kaletach: Paweł Parys,
- S.P. nr 30 w Lublinie: Kamil Helman,
- S.P. nr 219 w Warszawie: Piotr Tabor.

Jury podjęło decyzję o sprawdzeniu zadań przysłanych przez 50-ciu zawodników w kopiach zapasowych:

45	rozwiązań zadania	*Pracownia malarska*
43		*Wielokąty*
50		*Suma ciągu jedynkowego*
37		*AB-słowa*
175	rozwiązań łącznie	

Wirusy usunięto z 30-tu dyskietek.

Przysłano łącznie 14 dyskietek całkowicie lub częściowo nieczytelnych, przy czym w prawie wszystkich przypadkach udało się odzyskać większość zapisów. Dyskietka jednego zawodnika pozostała nie odczytana i nie została ujęta w klasyfikacji.

Ostatecznie w V Olimpiadzie Informatycznej sklasyfikowano 740 zawodników, którzy nadesłali łącznie na zawody I stopnia:

551 rozwiązań zadania	*Pracowania malarska*
631	*Wielokąty*
703	*Suma ciągu jedynkowego*
468	*AB-słowa*
2353 rozwiązania łącznie	

Z rozwiązaniami:

czterech zadań nadeszło	368 prac
trzech zadań nadeszło	196 prac
dwóch zadań nadeszło	117 prac
jednego zadania nadeszło	59 prac

Zawodnicy reprezentowali 48 województw. Nie było zawodników tylko z województwa bialskopodlaskiego. Trzech zawodników nie podało województwa, z którego pochodzą. Województwa leszczyńskie i ostrołęckie reprezentował jeden zawodnik. Najliczniej reprezentowane były następujące województwa:

st.warszawskie	80 zawodników
katowickie	60
krakowskie	47
gdańskie	42
poznańskie	36
łódzkie	33
rzeszowskie	32
szczecińskie	28
bydgoskie	26
tarnowskie	23
wrocławskie	22
częstochowskie	18
lubelskie	18
kieleckie	15
zielonogórskie	15
piotrkowskie	14
olsztyńskie	13
białostockie	13
nowosądeckie	13
krośnieńskie	13
siedleckie	13
toruńskie	13

W zawodach I stopnia najliczniej reprezentowane były szkoły:

V L.O. im. A. Witkowskiego w Krakowie	18 zawodników
VIII L.O. im. A. Mickiewicza w Poznaniu	18

XIV L.O. im. S. Staszica w Warszawie	13
XXVII L.O. im. T. Czackiego w Warszawie	13
III L.O. im. Marynarki Wojennej RP w Gdyni	12
VI L.O. im. J. i J. Śniadeckich w Bydgoszczy	10
XXXI L.O. im. L. Zamenhoffa w Łodzi	8
L.O. im. Marii Curie-Skłodowskiej w Pile	8
I L.O. im. Adama Mickiewicza w Białymstoku	7
L.O. im. Króla W. Jagiełły w Dębicy	7
I L.O. im. St. Konarskiego w Mielcu	7
IV L.O. im. M. Kopernika w Rzeszowie	7
II L.O. im. C. K. Norwida w Tychach	7
V L.O. im. A Mickiewicza w Częstochowie	6
I L.O. im. M. Kopernika w Łodzi	6
II L.O. im. K. I. Gałczyńskiego w Olsztynie	6
I L.O. im. Ks. A. Czartoryskiego w Puławach	6
IV L.O. im. T. Kościuszki w Toruniu	6
XXVIII L.O. im. J. Kochanowskiego w Warszawie	6
III L.O. im. A. Mickiewicza we Wrocławiu	6
XIV L.O. im. Polonii Belgijskiej we Wrocławiu	6
II L.O. im. Marii Konopnickiej w Inowrocławiu	5
ZSO im. S. Dubois w Koszalinie	5
Katolickie L.O. Zakonu Pijarów im. St. Konarskiego w Krakowie	5
III L.O. Z.S.E. w Krośnie	5
I L.O. im. B. Nowodworskiego w Krakowie	5
I L.O. im. Macierzy Szkolnej w Mińsku Mazowieckim	5
II L.O. im. A. Mickiewicza w Słupsku	5
I L.O. im. M. Konopnickiej w Suwałkach	5
III L.O. im. A. Mickiewicza w Tarnowie	5

Ogólnie najliczniej reprezentowane były miasta:

Warszawa	77 uczniów	Częstochowa 12
Kraków	46	Lublin 12
Poznań	32	Mielec 11
Łódź	29	Olsztyn 11
Szczecin	23	Olkusz 10
Gdynia	22	Tychy 10
Bydgoszcz	18	Inowrocław 8
Wrocław	18	Piła 8
Gdańsk	16	Siedlce 8
Rzeszów	16	Sosnowiec 8
Tarnów	13	Toruń 8
Białystok	12	

665 zawodników podało klasę, do której uczęszczało. W tym:

do klasy I	szkoły średniej	35 zawodników
do klasy II		120
do klasy III		252
do klasy IV		230
do klasy V		22
do klasy VIII	szkoły podstawowej	5
do klasy VII		1

Zawodnicy najczęściej używali następujących języków programowania:

Pascal firmy Borland	532 prace
C/C++ firmy Borland	141 prac

Ponadto pojawiły się:

Watcom C/C++	16 prac
GNU C/C++	8 prac
Ansi C	3 prace
Visual C	2 prace
High Speed Pascal	1 praca
Sas C/C++ 6.5	1 praca
PCC 2.1 c	1 praca

Komputerowe wspomaganie umożliwiło sprawdzenie prac zawodników kompletem 48-miu testów.

Proces sprawdzania był utrudniony — choć w mniejszym stopniu niż w poprzednich olimpiadach — występowaniem takich niedokładności, jak nieprzestrzeganie przez zawodników, podanych wyraźnie w treści zadań, reguł dotyczących nazywania plików i budowania zestawów danych wynikowych; sprawdzanie takich prac musiało trwać dłużej.

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

	Pracownia malarska		*Wielokąty*		*Suma ciągu jedynkowego*		*AB-słowa*	
	liczba zawodn.	czyli	liczba zawodn.	czyli	liczba zawodn.	czyli	liczba zawodn.	czyli
100 pkt.	64	8,6%	71	9,6%	230	31,1%	0	0,0%
99–75 pkt.	9	1,2%	31	4,2%	148	20,0%	6	0,8%
74–50 pkt.	313	42,3%	296	40,0%	133	18,0%	75	10,1%
49–1 pkt.	116	15,7%	198	26,8%	162	21,9%	313	42,3%
0 pkt.	238	32,2%	144	19,4%	67	9,0%	346	46,8%

W sumie za wszystkie 4 zadania:

SUMA	liczba zawodników	czyli
400 pkt.	0	0,0%
399–300 pkt.	59	8,0%
299–200 pkt.	180	24,3%
199–1 pkt.	488	65,9%
0 pkt.	13	1,8%

Prezydium Komitetu Głównego przy udziale Jury rozpatrzyło 2 reklamacje, które w obu przypadkach nie wniosły zmian 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 134 zawodników, którzy osiągnęli w zawodach I stopnia wynik nie mniejszy niż 255 pkt. Zawody II stopnia odbyły się w dniach 12–14 lutego 1998 r. w trzech stałych okręgach i dodatkowo w Krakowie:

- w Toruniu, 20-tu zawodników z następujących województw:
  - bydgoskiego (6),
  - gdańskiego (5),
  - olsztyńskiego (2),
  - szczecińskiego (2),
  - toruńskiego (3),
  - włocławskiego (2).
- we Wrocławiu, 40-tu zawodników z następujących województw:
  - bielskiego (2),
  - częstochowskiego (6),
  - gorzowskiego (2),
  - katowickiego (5),
  - kieleckiego (4),
  - legnickiego (2),
  - łódzkiego (4),
  - poznańskiego (6),
  - wałbrzyskiego (1),

  - wrocławskiego (6),
  - zielonogórskiego (2).
- w Warszawie, 44-ch zawodników z następujących województw:
  - białostockiego (3),
  - chełmskiego (1),
  - gdańskiego (5),
  - kaliskiego (1),
  - koszalińskiego (4),
  - lubelskiego (1),
  - przemyskiego (1)
  - siedleckiego (1),
  - st.warszawskiego (18),
  - suwalskiego (2),
  - szczecińskiego (1),
  - tarnowskiego (4),
  - tarnobrzeskiego (1),
  - zamojskiego (1).
- w Krakowie, 30-tu zawodników z następujących województw:
  - krakowskiego (14),
  - krośnieńskiego (3),
  - nowosądeckiego (2),
  - rzeszowskiego (11).

Najliczniej w zawodach II stopnia reprezentowane były szkoły:

XIV L.O. im. S. Staszica w Warszawie	7 uczniów
V L.O. im. A. Witkowskiego w Krakowie	6
I L.O. im. St. Konarskiego w Mielcu	5
VIII L.O. im. A. Mickiewicza w Poznaniu	5
I L.O. im. B. Nowodworskiego w Krakowie	4
I L.O. im. St. Dubois w Koszalinie	4
IV L.O. im. M. Kopernika w Rzeszowie	4
VI L.O. im. J. i J. Śniadeckich w Bydgoszczy	3
V L.O. im. A. Mickiewicza w Częstochowie	3
I L.O. im. M. Kopernika w Gdańsku	3
III L.O. im. Marynarki Wojennej RP w Gdyni	3
IV L.O. im. T. Kościuszki w Toruniu	3

Ogólnie najliczniej reprezentowane były miasta:

Warszawa	18 zawodników
Kraków	14
Rzeszów	7
Bydgoszcz	6
Poznań	6
Gdańsk	5
Gdynia	5
Wrocław	5

12 lutego odbyła się sesja próbna, na której zawodnicy rozwiązywali nie liczące się do ogólnej klasyfikacji zadanie *Sieć dróg*. W dniach konkursowych zawodnicy rozwiązywali zadania: *Pakowanie kontenerów*, *Równania na słowach*, *Okno* i *Płetwonurek*, oceniane każde maksymalnie po 100 punktów. Decyzją Komitetu Głównego nie przyznawano punktów uznaniowych za oryginalne rozwiązania.

Do automatycznego sprawdzania 4 zadań konkursowych zastosowano łącznie 52 testy.

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

	Pakowanie kontenerów		*Równania na słowach*		*Okno*		*Płetwonurek*	
	liczba zawodn.	czyli	liczba zawodn.	czyli	liczba zawodn.	czyli	liczba zawodn.	czyli
100 pkt.	2	1,5%	3	2,2%	3	2,2%	9	6,7%
99–75 pkt.	5	3,7%	0	0,0%	6	4,5%	2	1,5%
74–50 pkt.	2	1,5%	6	4,5%	12	8,9%	9	6,7%
49–1 pkt.	23	17,2%	63	47,0%	88	65,7%	93	69,4%
0 pkt.	102	76,1%	62	46,3%	25	18,7%	21	15,7%

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

SUMA	liczba zawodników	czyli
400 pkt.	1	0,8%
399–300 pkt.	4	3,0%
299–200 pkt.	3	2,2%
199–1 pkt.	121	90,3%
0 pkt.	5	3,7%

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

ZAWODY III STOPNIA

Zawody III stopnia odbyły się w ośrodku firmy Combidata Poland S.A. w Sopocie, w dniach od 6 do 9 kwietnia 1998 r.

W zawodach III stopnia wzięło udział 40 najlepszych uczestników zawodów II stopnia, którzy uzyskali wynik nie mniejszy niż 98 pkt. Zawodnicy reprezentowali nastepujące województwa:

st.warszawskie	6 zawodników
krakowskie	5
gdańskie	3
bydgoskie	2
krośnieńskie	2
nowosądeckie	2
rzeszowskie	2
tarnowskie	2
toruńskie	2
wrocławskie	2
zielonogórskie	2
białostockie	1
częstochowskie	1
koszalińskie	1
lubelskie	1
łódzkie	1
poznańskie	1
przemyskie	1
siedleckie	1
tarnobrzeskie	1
włocławskie	1

Niżej wymienione szkoły miały w finale więcej niż jednego zawodnika:

XIV L.O. im. St. Staszica w Warszawie	3 zawodników
III L.O. im Marynarki Wojennej RP w Gdyni	2
I L.O. im. B. Nowodworskiego w Krakowie	2
I L.O. im. J. Długosza w Nowym Sączu	2
IV L.O. im. T. Kościuszki w Toruniu	2

6 kwietnia odbyła się sesja próbna, na której zawodnicy rozwiązywali nie liczące się do ogólnej klasyfikacji zadanie *Układy assemblerowe*. W dniach konkursowych zawodnicy rozwiązywali zadania: *Bankomaty*, *Gonitwa*, *Najlżejszy język*, *Gra Ulama*, *Łamane płaskie* i *Prostokąty* oceniane każde maksymalnie po 100 punktów. Decyzją Komitetu Głównego nie przyznawano punktów uznaniowych za oryginalne rozwiązania.

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

Zastosowano łącznie zestaw 79 testów.

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

- *Bankomaty*

	liczba zawodników	czyli
100 pkt.	16	41,0%
99–75 pkt.	3	7,7%
74–50 pkt.	3	7,7%
49–1 pkt.	11	28,2%
0 pkt.	6	15,4%

- *Gonitwa*

	liczba zawodników	czyli
100 pkt.	0	0,0%
99–75 pkt.	3	7,7%
74–50 pkt.	5	12,8%
49–1 pkt.	15	38,5%
0 pkt.	16	41,0%

- *Najlżejszy język*

	liczba zawodników	czyli
100 pkt.	0	0,0%
99–75 pkt.	0	0,0%
74–50 pkt.	0	0,0%
49–1 pkt.	10	25,6%
0 pkt.	29	74,4%

- *Gra Ulama*

	liczba zawodników	czyli
100 pkt.	5	12,8%
99–75 pkt.	11	28,2%
74–50 pkt.	2	5,1%
49–1 pkt.	4	10,3%
0 pkt.	17	43,6%

- *Łamane płaskie*

	liczba zawodników	czyli
100 pkt.	0	0,0%
99–75 pkt.	0	0,0%
74–50 pkt.	0	0,0%
49–1 pkt.	13	33,3%
0 pkt.	26	66,7%

- *Prostokąty*

	liczba zawodników	czyli
100 pkt.	11	28,2%
99–75 pkt.	11	28,2%
74–50 pkt.	6	15,4%
49–1 pkt.	8	20,5%
0 pkt.	3	7,7%

W sumie za wszystkie 6 zadań:

SUMA	liczba zawodników	czyli
600 pkt.	0	0,0%
599–400 pkt.	0	0,0%
399–200 pkt.	18	46,2%
199–1 pkt.	21	53,8%
0 pkt.	0	0,0%

W dniu 9 kwietnia 1998 roku w gmachu Urzędu Miasta w Sopocie ogłoszono wyniki finału V Olimpiady Informatycznej 1997/98 i rozdano nagrody ufundowane przez: PROKOM Software S.A., Ogólnopolską Fundację Edukacji Komputerowej i Wydawnictwa Naukowo-Techniczne. Poniższa tabela zawiera listę wszystkich laureatów.

(1) **Andrzej Gąsienica-Samek**, laureat I miejsca, 395 pkt.
(kamera wideo, rower górski — PROKOM, książki, roczny abonament — WNT)*

(2) **Tomasz Czajka**, laureat I miejsca, 380 pkt.
(kamera wideo, rower górski — PROKOM, książki — WNT)

(3) **Eryk Kopczyński**, laureat I miejsca, 373 pkt.
(kamera wideo, rower górski — PROKOM, książki — WNT)

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

(4) **Paweł Wolff**, laureat II miejsca, 329 pkt.
(aparat fotograficzny — OFEK, rower górski — PROKOM, książki — WNT)

(5) **Marcin Kielar**, laureat II miejsca, 320 pkt.
(aparat fotograficzny — OFEK, rower górski — PROKOM, książki — WNT)

(6) **Łukasz Anforowicz**, laureat II miejsca, 316 pkt.
(aparat fotograficzny — OFEK, rower górski — PROKOM, książki — WNT)

(7) **Gracjan Polak**, laureat III miejsca, 304 pkt.
(rower górski — PROKOM, książki — WNT)

(8) **Kamil Kloch**, laureat III miejsca, 302 pkt.
(rower górski — PROKOM, książki — WNT)

(9) **Maciej Żenczykowski**, laureat III miejsca, 302 pkt.
(rower górski — PROKOM, książki — WNT)

(10) **Piotr Przytycki**, laureat III miejsca, 300 pkt.
(rower górski — PROKOM, książki — WNT)

(11) **Konrad Kwiatkowski**, laureat III miejsca, 296 pkt.
(rower górski — PROKOM, książki — WNT)

(12) **Bazyli Blicharski**, laureat III miejsca, 291 pkt.
(rower górski — PROKOM, książki — WNT)

(13) **Bartosz Nowierski**, laureat III miejsca, 274 pkt.
(rower górski — PROKOM, książki — WNT)

Wyróżniono też następujących finalistów:

(14) **Paweł Fic**, finalista z wyróżnieniem, 247 pkt.
(książki — WNT)

(15) **Adam Leszczyński**, finalista z wyróżnieniem, 240 pkt.
(książki — WNT)

(16) **Tomasz Malesiński**, finalista z wyróżnieniem, 231 pkt.
(książki — WNT)

(17) **Grzegorz Głowaty**, finalista z wyróżnieniem, 223 pkt.
(książki — WNT)

(18) **Jerzy Ziemiański**, finalista z wyróżnieniem, 214 pkt.
(książki — WNT)

Wszyscy finaliści otrzymali książki ufundowane przez WNT oraz inne drobne upominki ufundowane przez firmę Prokom.

Ogłoszono komunikat o powołaniu reprezentacji Polski na Olimpiadę Informatyczą Centralnej Europy w Zadarze (Chorwacja) oraz Międzynarodową Olimpiadę Informatyczną w Setubal[[[akcenty?]]] (Portugalia) w składzie:

(1) **Andrzej Gąsienica-Samek**

(2) **Tomasz Czajka**

(3) **Eryk Kopczyński**

(4) **Paweł Wolff**

Zawodnikami rezerwowymi zostali:

(5) **Marcin Kielar**

(6) **Łukasz Anforowicz**

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 40 zaświadczeń o zakwalifikowaniu do zawodów III stopnia celem przedłożenia dyrekcjom szkół. Laureaci i finaliści mogą być zwolnieni z egzaminów wstępnych do wielu szkół wyższych na mocy uchwał senatów uczelni podjętych na wniosek MEN, zgodnie z art. 141 ust. 1 ustawy z dnia 12 września 1990 r. o szkolnictwie wyższym (Dz.U. nr 65, poz. 385). Sekretariat wystawił łącznie 13 zaświadczeń o uzyskaniu tytułu laureata i 27 zaświadczeń o uzyskaniu tytułu finalisty V Olimpiady Informatycznej celem przedłożenia władzom szkół wyższych.

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

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

- **Jan Gąsienica-Samek** (ICL Poland sp. z o.o. w Warszawie)
  - Andrzej Gąsienica-Samek (laureat I miejsca)
- **Iwona Waszkiewicz** (VI L.O. w Bydgoszczy)
  - Łukasz Anforowicz (laureat II miejsca)
- **Bogusław Kielar** (Urząd Gminy w Tarnowcu)
  - Marcin Kielar (laureat II miejsca)
- **Sebastian Wolff** (Politechnika Poznańska)
  - Paweł Wolff (laureat II miejsca)
- **Jacenty Kloch** (Instytut Matematyki PAN w Krakowie)
  - Kamil Kloch (laureat III miejsca)
- **Jerzy Blicharski** (Koszalin)
  - Bazyli Blicharski (laureat III miejsca)
- **Anna Kwiatkowska** (Zespół Szkół Elektrycznych w Krośnie)
  - Konrad Kwiatkowski (laureat III miejsca)
- **Zofia Leś** (Instytut Fizyki Uniwersytetu Jagiellońskiego w Krakowie)

  - Maciej Żenczykowski (laureat III miejsca)
- **Anna Kwiatkowska** (IV L.O. w Toruniu)
  - Łukasz Kamiński (finalista)
  - Marcin Poturalski (finalista)
- **Kazimierz Kulik** (I L.O. w Nowym Sączu)
  - Piotr Dobrowolski (finalista)
  - Jakub Lipiński (finalista)
- **Wojciech Tomalczyk** (III L.O. im. Marynarki Wojennej RP w Gdyni)
  - Marcin Stefaniak (finalista)
  - Dariusz Jabłonowski (finalista)
- **Jan Kucharski** (LI L.O. im. T. Kościuszki w Warszawie)
  - Adam Leszczyński (finalista z wyróżnieniem)
- **Henryk Lasko** i **Wojciech Psik** (Zespół Szkół Elektronicznych i Ogólnokształcących w Przemyślu)
  - Grzegorz Głowaty (finalista z wyróżnieniem)
- **Joanna Ewa Łuszcz** (Zespół Szkół Elektrycznych w Białymstoku)
  - Tomasz Malesiński (finalista z wyróżnieniem)
- **Krzysztof Ziemiański** (Wydział Matematyki, Informatyki i Mechaniki UW w Warszawie)
  - Jerzy Ziemiański (finalista z wyróżnieniem)
- **Marek Kryniewski** (XIII L.O. w Gdańsku)
  - Paweł Fic (finalista z wyróżnieniem)
- **Łucja Grzegórska** (I L.O. w Lublinie)
  - Marek Wieczorek (finalista)
- **Adam Grabarski** (Instytut Matematyki Politechniki Warszawskiej)
  - Przemysław Rekucki (finalista)
- **Anna Bączkowska** (III L.O. we Włocławku)
  - Jakub Górski (finalista)
- **Barbara Olechowicz** (L.O. w Dębicy)
  - Jarosław Duda (finalista)
- **Bogdan Franczyk** (II L.O. w Rzeszowie)
  - Rafał Rusin (finalista)

- **Bogumiła Hnatkowska** (Zespół Szkół Łączności we Wrocławiu)
  - Mariusz Paradowski (finalista)
- **Mariusz Blank** (Lucent Technologies w Bydgoszczy)
  - Michał Rein (finalista)
- **Rafał Wysocki** (XXVII L.O. im. T. Czackiego w Warszawie)
  - Adam Koprowski (finalista)
- **Ryszard Kruk** (Zespół Szkół Zawodowych nr 1 w Siedlcach)
  - Marcin Stefaniak (finalista)
- **Stanisław Straszkiewicz** (Zespół Szkół Ogólnokształcących i Zawodowych w Krośnie Odrzańskim)
  - Przemysław Węgrzyn (finalista)
- **Teresa Pawicka** (XIV L.O. we Wrocławiu)
  - Tomasz Pawicki (finalista)

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

Warszawa, dn. 19 czerwca 1998 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 rozwiązaniu przez uczestników Olimpiady zakwalifikowanych do tych zawodów zadań przygotowanych dla danego stopnia. Odbywa się to 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 wyróżnionych uczestników 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 ostateczny skład 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 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 pieniężną wypłacaną z budżetu Olimpiady.

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

§8 FINANSOWANIE OLIMPIADY

(1) Komitet Główny będzie się ubiegał o 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 1997/98

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 1997/98.

§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
tel. (0–22) 22–40–19

w nieprzekraczalnym terminie nadania do 17 listopada 1997 r. (decyduje data stempla pocztowego). Prosimy o zachowanie dowodu nadania przesyłki. Rozwiązania dostarczane w inny sposób nie będą przyjmowane.

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

(3) Rozwiązanie każdego zadania składa się z:

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

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

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

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

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

(5) Wszystkie nadsyłane teksty powinny być drukowane (lub czytelnie pisane) 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 X Międzynarodową Olimpiadę Informatyczną w 1998 roku na podstawie wyników zawodów III stopnia i regulaminu tej Olimpiady. Szczegółowe zasady zostaną podane po otrzymaniu formalnego zaproszenia na X 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.

Informacja o IV CEOI
Nowy Sącz, lipiec 1997

W dniach 17–24 lipca w Nowym Sączu odbyła się IV Olimpiada Informatyczna Centralnej Europy (CEOI'97). Protektorat nad imprezą sprawował Prezydent Rzeczypospolitej Polskiej Aleksander Kwaśniewski, który także ufundował nagrody dla laureatów.

Impreza była do tej pory największą regionalną olimpiadą informatyczną na świecie. Udział w olimpiadzie wzięło 14 czteroosobowych drużyn z państw środkowo-europejskich (Białoruś, Chorwacja, Estonia, Jugosławia, Litwa, Łotwa, Niemcy, Polska, Rumunia, Słowacja, Ukraina i Węgry) oraz goście spoza regionu (reprezentacje Holandii i USA).

Prace Olimpiady koordynował Komitet Główny Olimpiady Informatycznej pod przewodnictwem prof. dr hab inż. Stanisława Waligórskiego.

Dwie pięciogodzinne sesje zawodów przeprowadzono w dniach 19 i 21 lipca 1997 r. Złote medale zdobyli zawodnicy:

- **Timo Burkard** (Niemcy), 186 pkt.,*
- **Daniel Adkins** (USA), 185 pkt.,
- **Krists Boitmanis** (Łotwa), 170 pkt.,
- **Valentin Gheorghita** (Rumunia), 163 pkt.,
- **Matt Craighead** (USA), 161 pkt.

Reprezentanci Polski zdobyli trzy medale srebrne i jeden brązowy:

- **Eryk Kopczyński** (Warszawa), 133 pkt., srebro,
- **Tomasz Waleń** (Ziębice, woj. wałbrzyskie), 129 pkt., srebro,
- **Piotr Sankowski** (Warszawa), 128 pkt., srebro,
- **Andrzej Gąsienica-Samek** (Warszawa), 116 pkt., brąz.

Po cztery medale zdobyły następujące reprezentacje:

USA	2 złote	1 srebrny	1 brązowy
Rumunia	1 złoty	3 srebrne	
Polski	3 srebrne	1 brązowy	

W ramach programu kulturalnego organizatorzy zapewnili uczestnikom rozliczne atrakcje, m.in. spływ Dunajcem oraz zwiedzanie kopalni soli w Wieliczce. Uczestnicy

* na 200 pkt możliwych do zdobycia

Olimpiady mieli także możliwość wysłuchania koncertu zaprezentowanego w ramach odbywającego się w tym samym czasie festiwalu Święto Dzieci Gór.

Olimpiada mogła się odbyć dzięki wsparciu następujących firm i instytucji:

- **Ministerstwo Edukacji Narodowej** pokryło większość wydatków Olimpiady,
- **firma OPTIMUS S.A.** dostarczyła wysokiej jakości sprzęt dla zawodników i organizatorów oraz ufundowała nagrody,
- **gmina Nowy Sącz** udzieliła Olimpiadzie wsparcia finansowego i organizacyjnego,
- **firma PZU S.A. Oddział Nowy Sącz** udzieliła Olimpiadzie wsparcia finansowego,
- **firma Dell Computer Poland Sp. z o.o.** udzieliła Olimpiadzie wsparcia finansowego i ufundowała upominki dla uczestników.

Informacja o IX IOI
Kapsztad, grudzień 1997

PRZEBIEG I ORGANIZACJA

Dziewiąta Międzynarodowa Olimpiada Informatyczna IOI'97 (International Olympiad in Informatics) odbyła się w Kapsztadzie w Afryce Południowej w dniach od 30 listopada do 7 grudnia 1997 roku. Polskę reprezentowali laureaci IV krajowej Olimpiady Informatycznej: Andrzej Gąsienica-Samek, Piotr Sankowski, Tomasz Waleń i Piotr Zieliński. Opiekunami polskiej ekipy byli dr Krzysztof Diks (kierownik ekipy) i dr Andrzej Walat (zastępca kierownika ekipy). Pomocą i opieką służył także prof. dr hab. inż. Stanisław Waligórski, który przebywał w Kapsztadzie jako członek Międzynarodowego Komitetu Olimpiady. Program olimpiady był następujący:

- dzień 1 — ceremonia otwarcia,
- dzień 2 — wycieczka i wybór zadań na pierwszy dzień zawodów,
- dzień 3 — pierwszy dzień zawodów (2 grudnia 1997),
- dzień 4 — wycieczka i wybór zadań na drugi dzień zawodów,
- dzień 5 — drugi dzień zawodów (4 grudnia 1997),
- dzień 6 — wycieczka i uroczysta kolacja,
- dzień 7 — ceremonia ogłoszenia wyników i wręczenia nagród.

Zadania IX Olimpiady wymagały od zawodników umiejętności pisania programów heurystycznych. Rozwiązania były oceniane za pomocą programu komputerowego. Używane oprogramowanie było podobne do oprogramowania stosowanego w latach ubiegłych. Regulamin oceniania był podobny do regulaminu olimpiady krajowej. Organizatorzy kładli jednak mniejszy nacisk na efektywność rozwiązań.

WYNIKI

Sklasyfikowano 221 zawodników z 57 krajów. Do zdobycia było w sumie 600 punktów, po 100 punktów za zadanie, po 3 zadania w każdym dniu zawodów. Polscy zawodnicy wypadli znakomicie i zdobyli:*

Andrzej Gąsienica-Samek	5	446	złoty
Piotr Zieliński	8	423	złoty
Tomasz Waleń	77	247	brązowy
Piotr Sankowski	78	245	brązowy

Pierwsze miejsce i puchar IFIP-u, ufundowany przez Komitet do spraw Kształcenia Międzynarodowej Federacji Przetwarzania Informacji, nagrodę przechodnią dla najlepszego zawodnika zdobył **Vladimir Martianov** z Rosji uzyskując 462 punkty.

Najwięcej medali zdobyły następujące kraje: Dania (2 złote, 1 srebrny), Polska (2 złote, 2 brązowe), Szwecja (2 złote), Rosja (1 złoty, 3 srebrne), Słowacja (1 złoty, 3 srebrne).

* kolejno podano: miejsce na liście wszystkich zawodników według liczby zdobytych punktów, liczbę punktów, medal.

Zawody I stopnia

opracowania zadań

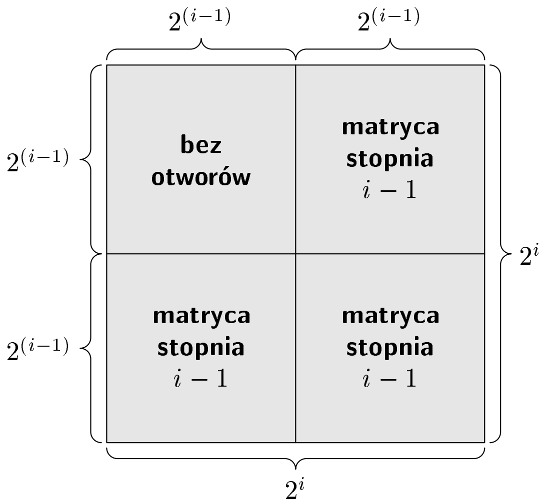
Wojciech Rytter
autor zadania, opracowanie

Marcin Engel
program wzorcowy

Pracownia malarska

Pracownia malarska przygotowuje seryjną produkcję obrazów. Obrazy będą wykonywane za pomocą kwadratowych matryc o różnych stopniach. Matryca stopnia i składa się z 2^i wierszy i 2^i kolumn. Na przecięciu pewnych wierszy i kolumn znajdują się otwory. Matryca stopnia 0 ma jeden otwór.

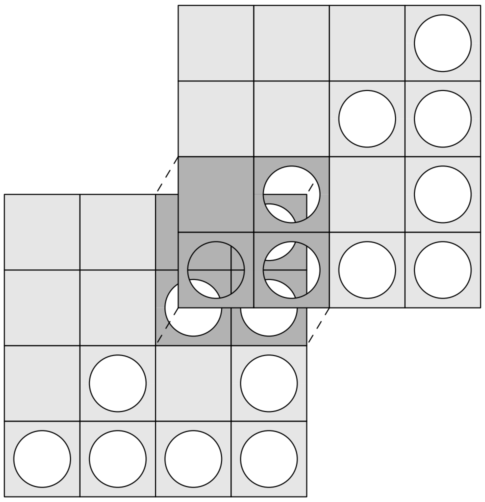
Dla $i > 0$ matryca stopnia i składa się z czterech kwadratów o rozmiarach $2^{(i-1)} \times 2^{i-1}$.



Oba prawe kwadraty oraz dolny lewy kwadrat są matrycami stopnia $i - 1$. W górnym lewym kwadracie nie ma żadnych otworów. Obraz otrzymuje się w następujący sposób. Najpierw ustala się trzy nieujemne liczby całkowite n, x, y. Następnie umieszcza się dwie matryce stopnia n jedna na drugiej i górną matrycę przesuwa się o x kolumn w prawo i y wierszy w górę. Tak otrzymany wzorzec zostaje umieszczony na białym płótnie i na wspólną część obu matryc nanosi się żółtą farbę. W efekcie na płótnie pojawią się żółte plamy tylko w tych miejscach, w których otwory obu matryc pokrywają się.

Przykład

Przyjrzyj się dwóm matorycom stopnia 2 przedstawionym na rysunku.



Górna matryca została przesunięta o 2 kolumny w prawo i o 2 wiersze w górę. W trzech miejscach otwory z obu matryc pokrywają się.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego MAL.IN stopień obu matryc oraz współrzędne przesunięcia górnej matrycy;*
- *oblicza liczbę żółtych plam na płótnie;*
- *zapisuje wynik w pliku tekstowym MAL.OUT.*

Wejście

Pierwszy wiersz pliku tekstowego MAL.IN zawiera liczbę całkowitą n, $0 \leqslant n \leqslant 100$. Jest to stopień matryc używanych w produkcji obrazów.

W drugim wierszu zapisana jest liczba całkowita x, zaś w trzecim liczba całkowita y, $0 \leqslant x, y \leqslant 2^n$. Liczba x jest liczbą kolumn, a y jest liczbą wierszy, o które należy przesunąć górną matrycę.

Wyjście

W pierwszym wierszu pliku wyjściowego MAL.OUT należy zapisać liczbę plam na płótnie.

Przykład

Dla pliku wejściowego MAL.IN:

```
2
2
2
```

poprawnym rozwiązaniem jest plik wyjściowy MAL.OUT:

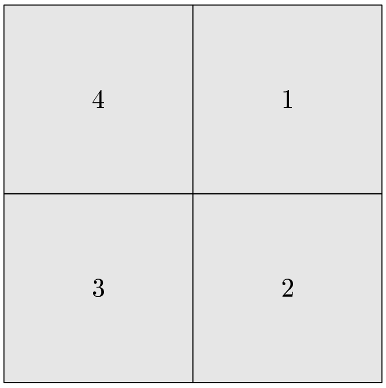
```
3
```

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

ROZWIĄZANIE

Oznaczmy przez M_i matrycę o rozmiarach $2^i \times 2^i$ skonstruowaną zgodnie z opisem zadania. Jej wiersze ponumerujmy z dołu do góry, a kolumny od lewej do prawej, rozpoczynając od zera. Ponumerujmy także ćwiartki matrycy, tak jak na rysunku 1.

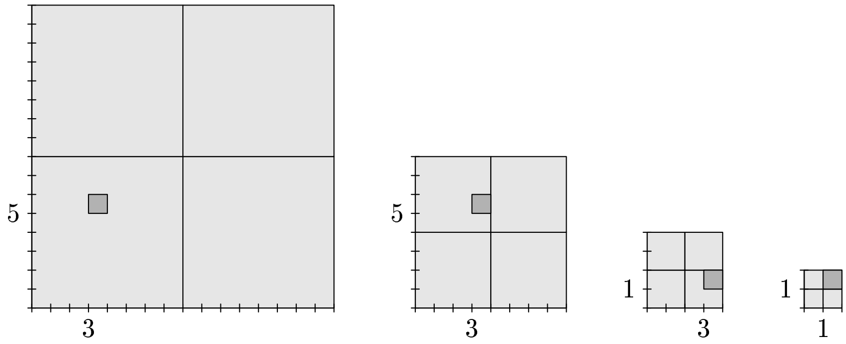
Rys. 1 Numeracja ćwiartek matrycy.



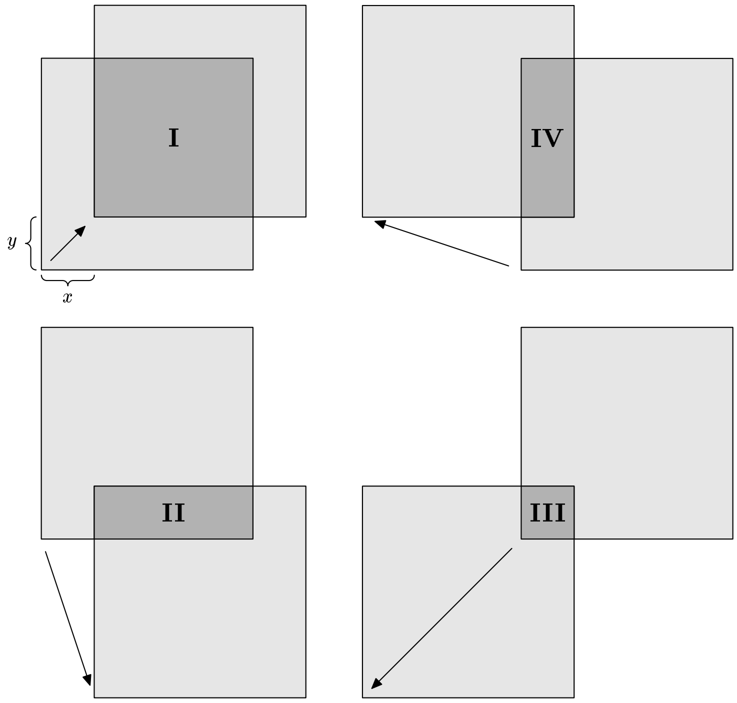
Początkowa matryca ma rozmiary $2^n \times 2^n$. Wyróżnijmy pole matrycy o współrzędnych (x, y). Należy ono do jednej z czterech ćwiartek. Zapiszmy numer tej ćwiartki i odrzućmy resztę matrycy. Otrzymaliśmy znów matrycę, na której leży wyróżnione pole (być może ma teraz inne współrzędne). Powyższy proces możemy powtarzać, dopóki nie otrzymamy matrycy o rozmiarach 1×1. Zapisywane numery ćwiartek tworzą ciąg, który nazywać będziemy **adresem ćwiartkowym** pola (x, y).

Skonstruujemy teraz algorytm, który będzie odpowiadał na pytanie nieco trudniejsze niż to postawione w treści zadania. Można w tym widzieć zjawisko podobne do występującego przy dowodzeniu przez indukcję, gdzie czasami łatwiej dowodzić silniejszej tezy indukcyjnej, gdyż tym samym możemy korzystać z silniejszego założenia indukcyjnego.

Rys. 2 Sekwencja kolejnych położeń pola $(3,5)$ w coraz mniejszych ćwiartkach. Pole jest kolejno w ćwiartce trzeciej, czwartej, drugiej i pierwszej, stąd jego adresem ćwiartkowym jest $(3,4,2,1)$.



Rys. 3 Wspólne części typu I, II, III i IV dwóch matryc o tym samym stopniu, przesuniętych względem siebie.



Spójrzmy na rysunek 3. Przedstawiono na nim cztery typy wspólnych części dwóch matryc stopnia i. Są to części wspólne matrycy stopnia i ze swoimi kopiami przesuniętymi o wektory $[x,y]$, $[x,y-2^i]$, $[2^i-x,2^i-y]$ i $[2^i-x,y]$. Będziemy je nazywać odpowiednio częściami wspólnymi typu I, II, III i IV. Niech $W_1(i)$, $W_2(i)$, $W_3(i)$ i $W_4(i)$ oznaczają odpowiednio liczby pokrywających się otworów w częściach wspólnych typu I, II, III i IV matryc stopnia i.

Załóżmy teraz, że punkt (x,y) leży w pierwszej ćwiartce matrycy stopnia i i spróbujmy wyrazić liczby $W_1(i)$, $W_2(i)$, $W_3(i)$, $W_4(i)$ przy pomocy liczb $W_1(i-1)$, $W_2(i-1)$, $W_3(i-1)$, $W_4(i-1)$. Wygodnie w tym celu posłużyć się pomocniczymi macierzami o rozmiarach 4×4. Wprowadźmy macierz

$$A_1 = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

taką, że

$$\begin{aligned} W_1(i) &= a_{11} \cdot W_1(i-1) + a_{12} \cdot W_2(i-1) + a_{13} \cdot W_3(i-1) + a_{14} \cdot W_4(i-1), \\ W_2(i) &= a_{21} \cdot W_1(i-1) + a_{22} \cdot W_2(i-1) + a_{23} \cdot W_3(i-1) + a_{24} \cdot W_4(i-1), \\ W_3(i) &= a_{31} \cdot W_1(i-1) + a_{32} \cdot W_2(i-1) + a_{33} \cdot W_3(i-1) + a_{34} \cdot W_4(i-1), \\ W_4(i) &= a_{41} \cdot W_1(i-1) + a_{42} \cdot W_2(i-1) + a_{43} \cdot W_3(i-1) + a_{44} \cdot W_4(i-1). \end{aligned}$$

Łatwo sprawdzić (umieszczając jedną matrycę nad drugą tak, by punkt przesunięcia był w pierwszej ćwiartce i badając położenia ćwiartek między sobą), że w tym przypadku

$$A_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 3 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

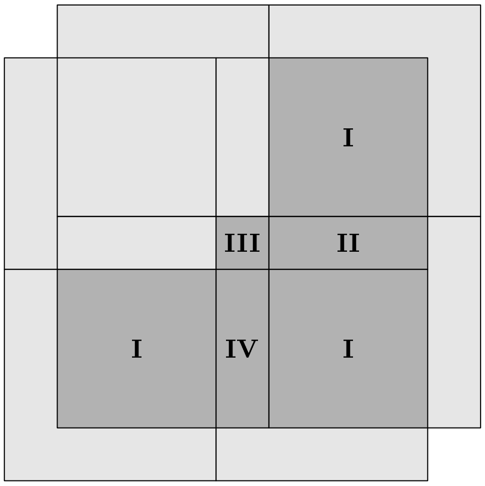
W k-tym wierszu macierzy trzeba zapisać, jak się wyraża $W_k(i)$ za pomocą $W_1(i-1)$, $W_2(i-1)$, $W_3(i-1)$ i $W_4(i-1)$. Pokażemy przykładowo, w jaki sposób obliczamy pierwszy wiersz macierzy A_3. W tym przypadku punkt (x,y) leży w trzeciej ćwiartce. Część wspólna składa się z trzech części typu I, oraz po jednej części typu II, III i IV (w sensie wspólnych części matryc stopnia $i-1$). Stąd pierwszym wierszem w macierzy A_3 jest $[3,1,1,1]$ (zob. rysunek 4).

W powyższy sposób wyznaczamy macierze A_1, A_2, A_3 i A_4 dla przypadków, gdy dolny lewy róg przesuniętej matrycy znajduje się odpowiednio w pierwszej, drugiej, trzeciej i czwartej ćwiartce. Są one podstawową strukturą danych potrzebną do rozwiązania zadania.

$$A_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 3 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad A_2 = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 3 \end{bmatrix}$$

$$A_3 = \begin{bmatrix} 3 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad A_4 = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 3 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Rys. 4 Pierwszym wierszem macierzy A_3 jest $[3,1,1,1]$, co oznacza, że jeżeli dolny lewy róg nałożonej matrycy znajduje się w pierwszej ćwiartce, to mamy trzy wspólne części typu I oraz po jednej wspólnej części typu II, III i IV dla matryc stopnia o jeden mniejszego. (Pamiętaj, że nie zaciemnione fragmenty części wspólnej nałożonych matryc nie zawierają otworów.)



Dla uproszczenia zapisu wprowadźmy jeszcze operację mnożenia wektora* przez macierz:

$$[v_1, v_2, v_3, v_4] \otimes \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = [r_1, r_2, r_3, r_4],$$

gdzie

$$\begin{aligned} r_1 &= [v_1 \cdot a_{11} + v_2 \cdot a_{21} + v_3 \cdot a_{31} + v_4 \cdot a_{41}], \\ r_2 &= [v_1 \cdot a_{12} + v_2 \cdot a_{22} + v_3 \cdot a_{32} + v_4 \cdot a_{42}], \\ r_3 &= [v_1 \cdot a_{13} + v_2 \cdot a_{23} + v_3 \cdot a_{33} + v_4 \cdot a_{43}], \\ r_4 &= [v_1 \cdot a_{14} + v_2 \cdot a_{24} + v_3 \cdot a_{34} + v_4 \cdot a_{44}]. \end{aligned}$$

Przy zastosowaniu powyższej notacji algorytm można zapisać następująco (zakładamy, że $[x, y]$ to wektor przesunięcia, a n to rozmiar matrycy):

```
adres := adresĆwiartkowy(x, y);
v := [1,0,0,0];
for t := 1 to n do begin
  i := adres[t];
  v := v ⊗ A_i
end;
```

* Pisząc „wektor” mamy tu na myśli skończony, niekoniecznie dwulementowy, ciąg liczb. Tutaj nie interesuje nas interpretacja geometryczna wektora. Słowo „wektor” w takim znaczeniu jest powszechnie używane w algebrze liniowej.

7: $wypisz(v[1])$

Poprawność algorytmu można uzasadnić zauważając, że zawsze po wykonaniu linii 5 prawdziwy jest warunek (tzw. niezmiennik):

$$wynik = v[1] \cdot W_1(n-t) + v[2] \cdot W_2(n-t) + v[3] \cdot W_3(n-t) + v[4] \cdot W_4(n-t),$$

gdzie *wynik* oznacza wartość wypisywaną przez algorytm w linii 7. (Przyjmujemy $W_1(0) = 1$ i $W_2(0) = W_3(0) = W_4(0) = 0$.)

Mnożenie wektora przez macierz możemy zastąpić gotowym ciągiem operacji, na przykład mnożenie wektora v przez macierz A_4 sprowadza się do:

$w[1] := v[1] + v[2];$
$w[2] := 3 \cdot v[2];$
$w[3] := v[2] + v[3];$
$w[4] := v[1] + v[3];$
$v := w$

Możemy wcześniej wyznaczyć ręcznie 4 ciągi operacji i zastąpić mnożenie przez macierz tymi operacjami. Będzie to na pewno bardziej efektywne obliczeniowo (w sensie czasu), chociaż notacja z macierzami wydaje się bardziej przekonywająca.

Na przykład dla matrycy o rozmiarze 16×16 i przesunięcia [3,5] poprawną odpowiedzią będzie 14, co obliczamy następująco:

inicjacja:	$adres := [3,4,2,1]$
iteracja 1:	$[1,0,0,0] * A_3 := [3,1,1,1]$
iteracja 2:	$[3,1,1,1] * A_4 := [4,3,2,4]$
iteracja 3:	$[4,3,2,4] * A_2 := [8,6,6,12]$
iteracja 4:	$[8,6,6,2] * A_1 := [14,12,18,18]$
zakończenie:	$wypisz(14)$

TESTY

Oprócz testów poprawnościowych dla danych wejściowych o niedużych rozmiarach, przeprowadzono testy wydajnościowe dla danych o maksymalnym rozmiarze. Pozwoliło to na odsianie rozwiązań nadmiernie zużywających pamięć. Do sprawdzenia rozwiązań zawodników użyto 11-tu testów MAL0.IN–MAL10.IN.

- MAL0.IN — test z treści zadania.
- MAL1.IN–MAL3.IN, MAL10.IN — testy poprawnościowe dla $n \leqslant 5$ i x, y w przedziale [0..31].
- MAL4.IN, MAL5.IN — testy dla przypadków brzegowych: przesunięcie zerowe, maksymalne przesunięcie po obu współrzędnych.
- MAL6.IN–MAL9.IN — testy wydajnościowe.

Krzysztof Diks
autor zadania, opracowanie

Marcin Mucha
program wzorcowy

Wielokąty

Powiemy, że dwa trójkąty przecinają się*, jeśli ich wnętrza mają co najmniej jeden punkt wspólny. Wielokąt jest* wypukły*, jeśli każdy odcinek łączący dowolne dwa punkty tego wielokąta jest w nim zawarty.*

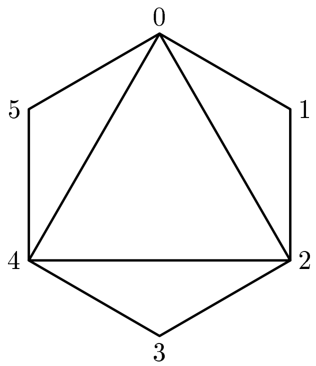
Trójkątem elementarnym *w wielokącie wypukłym nazywamy każdy trójkąt, którego wierzchołkami są wierzchołki tego wielokąta.*

Triangulacją *wielokąta wypukłego nazywamy każdy zbiór elementarnych trójkątów w tym wielokącie, w którym żadne dwa trójkąty nie przecinają się, a wszystkie razem pokrywają cały wielokąt.*

Dane są wielokąt wypukły i jego triangulacja. Jaka jest największa liczba trójkątów w tej triangulacji, które może przeciąć jeden elementarny trójkąt w tym wielokącie?

Przykład

Rozważmy następującą triangulację:



Trójkąt elementarny (1, 3, 5) przecina wszystkie trójkąty w tej triangulacji.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego WIE.IN liczbę wierzchołków wielokąta i jego triangulację;*
- *oblicza największą liczbę trójkątów tej triangulacji, które przecina pojedynczy, elementarny trójkąt w danym wielokącie;*
- *zapisuje wynik w pliku tekstowym WIE.OUT.*

Wejście

Pierwszy wiersz pliku WIE.IN zawiera liczbę n, $3 \leqslant n \leqslant 1000$. Jest to liczba wierzchołków wielokąta.

Wierzchołki wielokąta są ponumerowane kolejno $0, 1, \ldots, n-1$ zgodnie z ruchem wskazówek zegara.

Kolejne $n-2$ wiersze zawierają opisy trójkątów w triangulacji. W wierszu $i+1$, $1 \leqslant i \leqslant n-2$, zapisane są trzy liczby całkowite a, b, c oddzielone pojedynczymi odstępami. Są to numery wierzchołków i-tego trójkąta w triangulacji.

Wyjście

W pierwszym i jedynym wierszu pliku WIE.OUT należy zapisać jedną liczbę całkowitą — największą liczbę trójkątów w triangulacji, które przecinają jeden elementarny trójkąt w wielokącie wejściowym.

Przykład

Dla pliku wejściowego WIE.IN:

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

poprawnym rozwiązaniem jest plik tekstowy WIE.OUT:

```
4
```

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

ROZWIĄZANIE

Zadanie Wielokąt bardzo łatwo rozwiązać za pomocą algorytmu działającego w czasie rzędu n^4. W tym celu wystarczy dla każdego trójkąta elementarnego policzyć liczbę trójkątów w triangulacji, z którymi taki trójkąt przecina się, a następnie wybrać największą z obliczonych liczb. Ponieważ trójkątów w triangulacji n-kąta wypukłego jest zawsze $n-2$, a wszystkich trójkątów elementarnych jest $n(n-1)(n-2)/6$, to taki algorytm rzeczywiście działa w czasie rzędu n^4.

Istnieje znacznie lepszy algorytm dla naszego zadania. Algorytm, który opisujemy poniżej, działa w czasie liniowym ze względu na rozmiar danych wejściowych n.

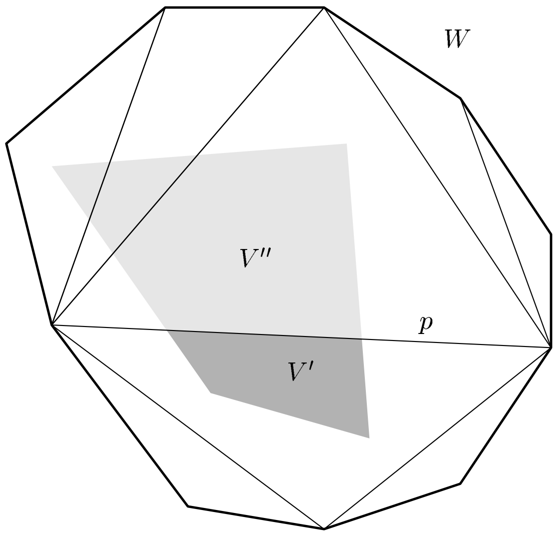
Niech W będzie n-kątem wypukłym, a T jego dowolną triangulacją. Nietrudno zauważyć, że taka triangulacja jest jednoznacznie wyznaczona przez $n-3$ przekątnych wielokąta W, przecinających się co najwyżej w wierzchołkach wielokąta. Od tej chwili pisząc *przekątna* będziemy mieli zawsze na myśli przekątną ze zbioru przekątnych wyznaczających triangulację T.

Obserwacja 1: Niech K będzie dowolnym trójkątem elementarnym w wielokącie W i niech k będzie liczbą przekątnych, które mają niepuste przecięcie z wnętrzem K. Wówczas liczba tójkątów w triangulacji T, które przecinają K, wynosi $k + 1$.

Obserwacja 1 wynika natychmiast z następującego ogólniejszego twierdzenia.

Twierdzenie 2: Niech V będzie dowolnym wielokątem wypukłym zawartym w W i niech v będzie liczbą przekątnych, które mają niepuste przecięcie z wnętrzem V. Wówczas, liczba trójkątów z T, które przecinają* V wynosi $v + 1$ (zobacz rysunek 1).

Rys. 1 Ilustracja dowodu twierdzenia 2



Dowód: Indukcja po liczbie wierzchołków wielokąta W. Dla $n = 3$ twierdzenie jest oczywiście prawdziwe. Niech $n > 3$ i załóżmy, że twierdzenie jest prawdziwe dla każdego wielokąta W o mniejszej niż n liczbie wierzchołków. Jeśli wielokąta V nie przecina żadna przekątna, to jest on całkowicie zawarty w pewnym trójkącie z triangulacji T. Przyjmijmy zatem, że V przecina co najmniej jedna przekątna. Każda przekątna przecinająca wielokąt V dzieli go na dwa podwielokąty wypukłe (zobacz rysunek 1). Nietrudno zauważyć, że wśród przekątnych można znaleźć co najmniej jedną taką, która dzieli wielokąt na takie dwa podwielokąty, że jednego z nich nie przecina żadna przekątna. Niech p będzie taką przekątną, a V' i V'' będą podwielokątami V powstającymi w wyniku podziału V za pomocą przekątnej p. Załóżmy, że V' nie przecina żadna przekątna. Przekątna p dzieli też wielokąt W na dwa wielokąty wypukłe: W' — zawierający V', i W'' — zawierający V''. Niech T' będzie triangulacją T „obciętą" do wielokąta W', natomiast T'' triangulacją T „obciętą" do W''. Jeśli v'' jest liczbą przekątnych przecinających V'', to, z założenia indukcyjnego, liczba trójkątów z T'' (a

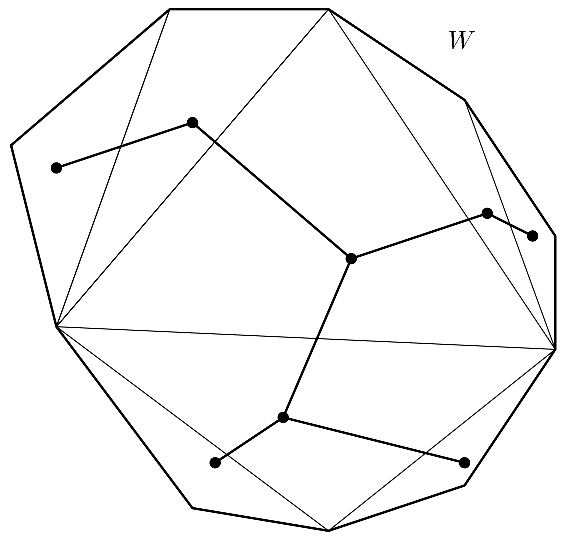
* Przyjmujemy, że dwa wielokąty przecinają się, gdy ich wnętrza mają co najmniej jeden punkt wspólny.

stąd z T) przecinających V'' wynosi $v''+1$. Podwielokąt V' jest zawarty w dokładnie jednym trójkącie z T' (z T). Otrzymujemy zatem, że liczba trójkątów przecinających V wynosi $v''+2$. Z drugiej strony widzimy, że wielokąt V przecina $v = v''+1$ przekątnych, co dowodzi tezy twierdzenia.

Zauważmy jeszcze, że liczba przekątnych o tej samej własności co p wynosi co najwyżej 3. □

W jaki sposób wyznaczyć szybko trójkąt elementarny, który przecina największa liczba przekątnych z danej triangulacji? Żeby odpowiedzieć na to pytanie sformułujemy nasze zadanie w terminach teorii grafów.

Rys. 2 Drzewo triangulacji

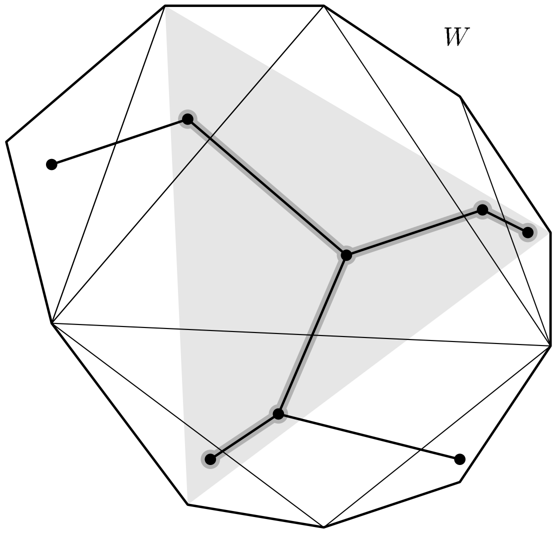


Niech DT będzie grafem, którego wierzchołkami są trójkąty danej triangulacji T. Krawędź w grafie DT łączy dwa wierzchołki-trójkąty wtedy i tylko wtedy, gdy mają one wspólny bok (zobacz rysunek 2). Jest oczywiste, że z każdego wierzchołka wychodzą co najwyżej trzy krawędzie. Nietrudno zauważyć, że graf DT jest spójnym grafem bez cykli. Grafy o takich własnościach nazywa się **drzewami**. Graf DT będziemy nazywali **drzewem triangulacji**. Zauważmy, że krawędzie w drzewie DT odpowiadają przekątnym w triangulacji. Niech K będzie dowolnym trójkątem elementarnym i niech T_K będzie zbiorem wszystkich tych trójkątów z triangulacji T, które przecinają K. Oznaczmy przez $DT(T_K)$ taki maksymalny podgraf drzewa DT, którego wierzchołkami są tylko trójkąty z T_K. Można pokazać, że:

Twierdzenie 3: Graf $DT(T_K)$ jest drzewem, w którym co najwyżej trzy wierzchołki mają stopień* 1. Pozostałe wierzchołki mają stopień 2, poza być może jednym, który ma stopień 3 (zobacz rysunek 3). □

* Stopniem wierzchołka w grafie nazywamy liczbę wychodzących z niego krawędzi.

Rys. 3 Drzewo DT_K



Prawdziwe jest też twierdzenie "odwrotne".

Twierdzenie 4: Niech DT' będzie poddrzewem drzewa DT, w którym co najwyżej trzy wierzchołki maja stopień 1, a pozostałe są stopnia 2, poza być może jednym, który ma stopień 3. Wówczas istnieje trójkąt elementarny K taki, że $DT' = DT(T_K)$. $\square$

Powyższe rozważania prowadzą nas do następującego algorytmu:

(1) Zbuduj drzewo triangulacji DT.

(2) Oblicz rozmiar (liczbę wierzchołków) największego poddrzewa drzewa DT, w którym co najwyżej 3 wierzchołki mają stopień 1, a pozostałe mają stopień 2, być może poza jednym, który ma stopień 3.

Zauważmy, że jeżeli w drzewie DT są co najmniej 3 wierzchołki stopnia 1, to największy rozmiar spośród poddrzew o podanych wyżej własnościach będzie miało poddrzewo, w którym są trzy wierzchołki stopnia 1 (a stąd wynika, że także dokładnie jeden wierzchołek stopnia 3), będące jednocześnie wierzchołkami o stopniu 1 w drzewie triangulacji DT.

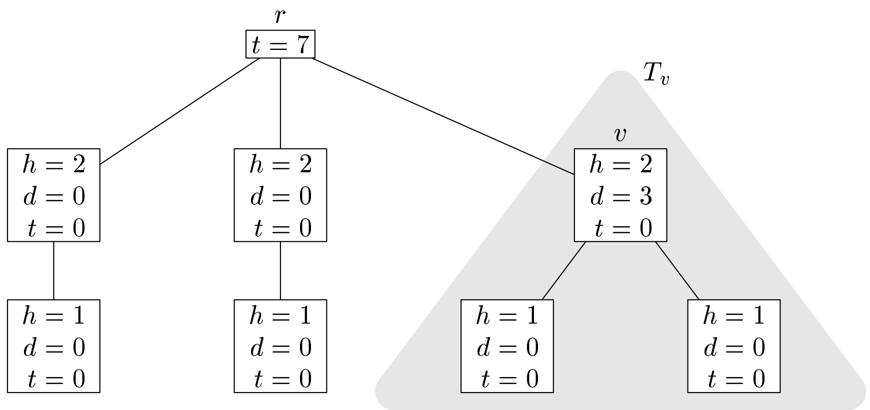
Budowa drzewa triangulacji jest czysto techniczna i nie powinna nastręczać żadnych trudności. Format danych wejściowych umożliwia zbudowanie drzewa triangulacji w czasie liniowym. Szczegóły znajdzie Czytelnik w programie WIE.PAS na załączonej dyskietce. Tutaj omówimy ciekawszy problem efektywnej implementacji kroku (2). Rozważmy 3 przypadki:

- Drzewo DT składa się tylko z jednego wierzchołka. Ten przypadek zachodzi tylko dla $n = 3$ i odpowiedzią jest wówczas 1.

- Drzewo DT jest ścieżką, tzn. dokładnie dwa wierzchołki mają stopień 1, a pozostałe mają stopień 2. W tym przypadku poddrzewem o największym rozmiarze jest całe drzewo i poszukiwaną liczbą jest $n-2$.
- W drzewie DT co najmniej 3 wierzchołki mają stopień 1. W takim drzewie musi istnieć wierzchołek o stopniu większym niż 1. Niech r będzie dowolnym takim wierzchołkiem. Na drzewo DT możemy teraz spojrzeć jak na drzewo z korzeniem r, w którym wszystkie krawędzie są skierowane w kierunku liści — wierzchołków o stopniu 1 w drzewie DT. Dla każdego wierzchołka v, niech DT_v będzie poddrzewem drzewa DT o korzeniu w v i zawierającym wszystkie wierzchołki, do których ścieżki z r prowadzą przez v (zobacz rysunek 4). W celu wyznaczenia rozmiaru największego poddrzewa w drzewie triangulacji, które zawiera dokładnie 3 wierzchołki o stopniu 1 i jeden wierzchołek o stopniu 3, a pozostałe wierzchołki mają stopień 2, obliczamy dla każdego wierzchołka v trzy wielkości:
  - $h(v)$ — wysokość drzewa DT_v liczoną w liczbie wierzchołków;
  - $d(v)$ — rozmiar największego poddrzewa w DT_v o korzeniu w v, zawierającego dokładnie dwa liście;
  - $t(v)$ — rozmiar największego poddrzewa w DT_v, niekoniecznie o korzeniu w v, zawierającego dokładnie trzy liście.

  Poszukiwaną liczbą jest $t(r)$.

Rys. 4 Ukorzenione drzewo triangulacji



Liczby $h(v)$, $d(v)$, $t(v)$ obliczamy rekurencyjnie korzystając z następujących zależności:

- jeśli v jest liściem, to:

$$h(v) = 1, \qquad d(v) = 0, \qquad t(v) = 0.$$

- jeśli v ma jedynego syna x, to:

$$h(v) = h(x) + 1, \qquad d(v) = d(x) + 1, \qquad t(v) = t(x).$$

- jeśli v ma dokładnie dwóch synów x i y, to:

$$\begin{aligned} h(v) &= \max(h(x),\ h(y)) + 1, \\ d(v) &= h(x) + h(y) + 1, \\ t(v) &= \max(t(x),\ t(y),\ h(x) + d(y) + 1,\ d(x) + h(y) + 1). \end{aligned}$$

- jeśli v ma dokładnie trzech synów x, y i z (przypadek ten zachodzi tylko wtedy, gdy v jest korzeniem drzewa ($v = r$), wystarczy więc policzyć tylko $t(v)$), to:

$$t(v) = \max \begin{cases} t(x) \\ t(y) \\ t(z) \\ d(x) + \max(h(y),\ h(z)) + 1 \\ d(y) + \max(h(x),\ h(z)) + 1 \\ d(z) + \max(h(x),\ h(y)) + 1 \\ h(x) + h(y) + h(z) + 1 \end{cases}$$

UWAGA: dla uproszczenia zapisu przyjmujemy, że w przypadku gdy $d(\cdot)$ jest składnikiem sumy i ma wartość 0, to cała suma też ma wartość 0.

Ponieważ obliczenia w każdym wierzchołku zajmują stały czas, łączny czas obliczeń jest rzędu rozmiaru drzewa, czyli $O(n)$.

TESTY

Rozwiązania zawodników były sprawdzane za pomocą 11 testów WIE0.IN–WIE10.IN:

- WIE0.IN — test z treści zadania.
- WIE1.IN–WIE3.IN — przypadki szczególne: trójkąt, czworokąt, pięciokąt. Wynikiem jest zawsze $n - 2$.
- WIE4.IN, WIE5.IN — testy poprawnościowe: drzewo triangulacji jest ścieżką.
- WIE6.IN, WIE7.IN — testy poprawnościowe: ukorzenione drzewa triangulacji przypominają pełne drzewa binarne o wysokościach, odpowiednio, 2 i 7.
- WIE8.IN–WIE10.IN — testy wydajnościowe: losowe triangulacje wielokątów o dużych rozmiarach.

Grzegorz Jakacki
autor zadania, opracowanie

Krzysztof Sobusiak
program wzorcowy

Suma ciągu jedynkowego

Ciąg liczbowy o wartościach będących liczbami całkowitymi nazywamy jedynkowym*, jeżeli dowolne jego sąsiednie wyrazy różnią się od siebie dokładnie o jeden oraz jego pierwszy wyraz jest równy 0. Bardziej precyzyjnie: niech* $[a_1, a_2, \ldots, a_n]$ *będzie ciągiem o wartościach całkowitych; powiemy, że ten ciąg jest* jedynkowy*, jeżeli:*

- *dla dowolnej liczby całkowitej* k *spełniającej nierówność* $1 \leqslant k < n$ *zachodzi warunek* $|a_k - a_{k+1}| = 1$ *oraz*
- $a_1 = 0$.

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego SUM.IN dwie liczby całkowite: długość ciągu i sumę elementów ciągu;*
- *wyznaczy ciąg jedynkowy o zadanej długości i sumie elementów lub stwierdzi, że taki ciąg nie istnieje;*
- *zapsize rezultat w pliku tekstowym SUM.OUT.*

Wejście

W pliku tekstowym SUM.IN są zapisane:

- *w pierwszym wierszu — liczba* n *elementów ciągu, spełniająca nierówność* $1 \leqslant n \leqslant 10000$;
- *w drugim wierszu — liczba* S *będąca żądaną sumą elementów ciągu, spełniająca nierówność* $|S| \leqslant 50\,000\,000$.

Wyjście

W pierwszych n *wierszach pliku tekstowego SUM.OUT należy zapisać* n *liczb całkowitych (po jednej w wierszu) będących kolejnymi wyrazami ciągu jedynkowego (k-ty wyraz w k-tym wierszu) o zadanej sumie* S *lub słowo NIE, jeżeli taki ciąg nie istnieje.*

Przykład

Dla pliku wejściowego SUM.IN o zawartości:

```
8
4
```

poprawnym rozwiązaniem jest plik wyjściowy SUM.OUT o zawartości:

```
0
1
2
1
0
-1
0
1
```

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

ROZWIĄZANIE

Niech A będzie pewnym skończonym podzbiorem zbioru liczb naturalnych. Oznaczmy przez ΣA sumę* elementów zbioru A, a przez $\Sigma^* A$ zbiór sum elementów wszystkich podzbiorów A, tj. $\Sigma^* A = \{\Sigma X \ : \ X \subseteq A\}$. Jeżeli $A = \{2, 5, 7, 8\}$, to $\Sigma A = 22$ oraz $\Sigma^* A = \{0, 2, 5, 7, 8, 9, 10, 12, 13, 14, 15, 17, 20, 22\}$. Niech $\mathcal{S}_n$ będzie sumą wszystkich liczb naturalnych od 1 do n, tj. $\mathcal{S}_n = 1 + 2 + \ldots + n = \Sigma[1..n]$. Ponieważ $\mathcal{S}_n$ jest sumą elementów ciągu arytmetycznego, nietrudno zauważyć, że

Fakt 1: $\mathcal{S}_n = n(n+1)/2$,

Fakt 2: $n \leqslant \mathcal{S}_n$.

Zanim przejdziemy do rozwiązania zadania, rozważmy następujący problem pomocniczy: dane są liczby n i S, znaleźć zbiór $Z \subseteq [1..n]$ taki, że $\Sigma Z = S$, o ile taki zbiór istnieje. Po pierwsze zauważmy, że:

Twierdzenie 3: $\Sigma^*[1..n] = [0..\mathcal{S}_n]$.

Dowód: Dowodzimy przez indukcję względem n. Dla $n = 1$ po lewej stronie równości mamy

$$\Sigma^*[1..1] = \Sigma^*\{1\} = \{0, 1\} = [0..1],$$

natomiast po prawej

$$[0..\mathcal{S}_1] = [0..1].$$

Przypuśćmy teraz, że twierdzenie jest prawdziwe dla $n = k$. Pokażemy, że jest ono prawdziwe także dla $n = k + 1$. Po pierwsze wiadomo, że największym elementem zbioru $\Sigma^*[1..(k+1)]$ jest $\mathcal{S}_{k+1}$. Pozostaje tylko udowodnić, że wszystkie liczby naturalne mniejsze od $\mathcal{S}_{k+1}$ też należą do $\Sigma^*[1..(k+1)]$. Weźmy zatem dowolną liczbę m ze zbioru $[1..\mathcal{S}_{k+1}]$. Mamy następujące przypadki:

* Przyjmujemy, że $\Sigma\varnothing = 0$, gdzie $\varnothing$ oznacza zbiór pusty.

- $m \leqslant k$; w tym przypadku na pewno $m \leqslant \mathcal{S}_k$ (fakt 2), czyli $m \in [0..\mathcal{S}_k]$. Stąd na mocy założenia indukcyjnego wnioskujemy, że $m \in \Sigma^*[1..k]$, więc także $m \in \Sigma^*[1..(k+1)]$.
- $m > k$, czyli $m \geqslant k+1$; w takim przypadku m można przedstawić jako sumę $m = m' + (k+1)$. Ponieważ $m \geqslant k+1$, to $m' \geqslant 0$. Ponieważ $m \leqslant \mathcal{S}_{k+1}$, to $m' \leqslant \mathcal{S}_k$. Ostatecznie $m' \in [0..\mathcal{S}_k]$. Z założenia indukcyjnego wiemy, że m' daje się przedstawić jako suma elementów pewnego podzbioru zbioru $[1..k]$. Wnioskujemy zatem, że m daje się przedstawić jako suma elementów pewnego podzbioru zbioru $[1..(k+1)]$ (do podzbioru zbioru [1..k] o sumie elementów równej m' trzeba dołożyć element $k+1$). Stąd $m \in \Sigma^*[1..(k+1)]$.

Powyższe stwierdzenia kończą dowód. □

Na podstawie dowodu twierdzenia można skonstruować algorytm, który dla danej liczby m ze zbioru $[0..\mathcal{S}_n]$ wypisuje podzbiór zbioru $[1..n]$ o sumie elementów równej m. Schemat algorytmu wygląda następująco:

```
procedure rozkład(m, n : integer);
begin
  if m ⩽ 1 then begin
    if m = 1 then wypisz(1)
  end
  else begin
    if m ⩾ n then begin
      wypisz(n);
      rozkład(m − n, n − 1)
    end
    else
      rozkład(m, n − 1)
  end
end
```

Uważny czytelnik dostrzeże, że powyższy algorytm wynika niemalże wprost z dowodu indukcyjnego: rozpatrywanie przypadków w dowodzie odpowiada instrukcjom **if then else**, natomiast powoływanie się na założenie indukcyjne odpowiada rekurencyjnemu wywołaniu procedury (dla danych o mniejszym rozmiarze). Czytelnik z pewnością zauważy także, że rekurencja w procedurze *rozkład* może być zastąpiona przez iterację. Iteracyjna wersja algorytmu wygląda następująco:

```
procedure rozkład(m, n : integer);
begin
  while m > 0 do begin
    if m ⩾ n then begin
      wypisz(n);
      m := m − n
    end;
    n := n − 1
  end
end
```

Zbiór $[1..n]$ może zawierać wiele podzbiorów o sumie elementów równej m (np. dla $[1..4]$ oraz $m = 5$ mamy $\Sigma\{1,4\} = \Sigma\{2,3\} = 5$). Nasz algorytm wyznacza jeden z wielu możliwych rozkładów liczby m.

Powróćmy teraz do problemu pomocniczego (znaleźć podzbiór zbioru $[1..n]$ o sumie elementów równej S). Korzystając z twierdzenia oraz skonstruowanego algorytmu można sformułować następujące rozwiązanie: jeżeli S jest spoza przedziału $[0..\mathcal{S}_n]$, to nie istnieje $Z \subseteq [1..n]$ o sumie elementów równej m. Jeżeli natomiast $m \in [0..\mathcal{S}_n]$, to Z można wyznaczyć za pomocą procedury *rozkład*.

Powyższy rezultat pozwala wreszcie zmierzyć się z zadaniem olimpijskim. Mamy znaleźć ciąg $a_1, \ldots, a_n$ taki, że $a_1 + \ldots + a_n = S$, $a_1 = 0$ oraz różnica pomiędzy kolejnymi wyrazami wynosi 1 lub -1. Oznaczmy przez b_k różnicę pomiędzy wyrazem k-tym i $(k+1)$-szym, tj. $b_k = a_{k+1} - a_k$. Wówczas

$$\begin{aligned} a_1 + a_2 + a_3 + \ldots + a_n &= 0 + b_1 + (b_1 + b_2) + \ldots + (b_1 + b_2 + \ldots + b_{n-1}) = \\ &= (n-1) \cdot b_1 + (n-2) \cdot b_2 + \ldots 1 \cdot b_{n-1}. \end{aligned}$$

Zadanie sprowadza się zatem do wyznaczenia wartości współczynników b_k (równych 1 lub -1), żeby zachodziła równość

$$S = (n-1) \cdot b_1 + (n-2) \cdot b_2 + \ldots + 1 \cdot b_{n-1}.$$

Spróbujemy teraz przekształcić tę równość w ten sposób, żeby po prawej stronie, zamiast sumy liczb z przedziału [1..(n-1)] ze współczynnikami ze zbioru $\{-1, 1\}$, otrzymać sumę ze współczynnikami ze zbioru $\{0, 1\}$, dla której wartości współczynników umiemy wyznaczyć przy pomocy omówionego wcześniej rozwiązania problemu pomocniczego. W tym celu do każdego b_k trzeba dodać jeden (otrzymujemy wtedy współczynniki ze zbioru $\{0, 2\}$), a następnie podzielić przez 2. Dodanie jedynki do b_k oznacza zwiększenie prawej strony równania o wartość wyrazu mnożonego przez b_k. Ponieważ współczynniki b_k stoją przy kolejnych liczbach z przedziału $[1..(n-1)]$, to „dodanie jedynki” do wszystkich b_k powoduje zwiększenie wartości prawej strony równania o $\mathcal{S}_{n-1}$. Stąd otrzymujemy:

$$\frac{S + \mathcal{S}_{n-1}}{2} = (n-1) \cdot \frac{b_1 + 1}{2} + (n-2) \cdot \frac{b_2 + 1}{2} + \ldots + 1 \cdot \frac{b_{n-1} + 1}{2} \qquad (1)$$

Algorytm będący rozwiązaniem zadania powinien stwierdzić, posługując się rozwiązaniem zadania pomocniczego, czy istnieje podzbiór zbioru $[1..(n-1)]$ o sumie elementów $(S + \mathcal{S}_{n-1})/2$ i ew. wyznaczyć go. Jeśli taki podzbiór nie istnieje, to nie istnieje także ciąg spełniający warunki zadania. W przeciwnym przypadku zawartość podzbioru określa, które współczynniki w sumie (1) mają wartość 0, a które 1, czyli które współczynniki b_k są równe -1, a które 1.

Należy pamiętać, że w zadaniu pomocniczym zakładaliśmy, iż rozkładana na składniki suma ma wartość całkowitą. W rozwiązaniu zadania olimpijskiego warunek ten należy sprawdzić, gdyż może się zdarzyć, że $S + \mathcal{S}_{n-1}$ jest liczbą nieparzystą. Oczywiście w takim przypadku ciąg spełniający warunki zadania nie istnieje.

Pozostaje kwestia implementacji opisanego algorytmu, co jednak, ze względu na jego prostotę, nie powinno nastręczać trudności. Algorytm wzorcowy najpierw sprawdza parzystość liczby $S + \mathcal{S}_{n-1}$ oraz przynależenie liczby $(S + \mathcal{S}_{n-1})/2$ do przedziału

$[0..\mathcal{S}_{n-1}]$ (wystarczy sprawdzić przynależenie S do przedziału $[-\mathcal{S}_{n-1}..\mathcal{S}_{n-1}]$), a następnie rozkłada ją na składniki ze zbioru $[1..(n-1)]$ za pomocą iteracyjnej wersji procedury *rozkład*, konstruując na bieżąco ciąg (a_n). Jeżeli kolejny element zbioru $[1..(n-1)]$ należy do podzbioru o sumie elementów $(S+\mathcal{S}_{n-1})/2$, to stojący przy nim we wzorze (1) współczynnik b_k jest równy 1, czyli kolejny wyraz ciągu (a_n) musi być o jeden większy od poprzedniego. W przeciwnym przypadku, na mocy analogicznego rozumowania, kolejny wyraz ciągu (a_n) musi być o jeden mniejszy od poprzedniego. Ponadto wiadomo, że $a_1 = 0$. Na załączonej dyskietce Czytelnik znajdzie program SUM.PAS będący implementacją opisanego algorytmu.

Złożoność obliczeniowa algorytmu jest liniowa względem n. Algorytm korzysta z pamięci o stałym rozmiarze, niezależnym od danych wejściowych.

OPIS TESTÓW

Do sprawdzenia rozwiązań zawodników użyto 13-tu testów.

- SUM0.IN — test z treści zadania.
- SUM1.IN, SUM2.IN — niewielkie testy poprawnościowe.
- SUM3A.IN — brak rozwiązania, $S+\mathcal{S}_{n-1}$ nieparzyste.
- SUM3B.IN — prosty test posiadający rozwiązanie (do zgrupowania).
- SUM4A.IN — brak rozwiązania, zbyt małe S.
- SUM4B.IN — prosty test posiadający rozwiązanie (do zgrupowania).
- SUM5.IN — przypadek graniczny ($n = 1$, $S = 0$).
- SUM6.IN–SUM8.IN — testy wydajnościowe (duże n i S).
- SUM9.IN — maksymalne S.
- SUM10.IN — minimalne S.

Dla dwóch z nich poprawną odpowiedzią był plik zawierający słowo „NIE”. By uniknąć punktowania rozwiązań, które zawsze wypisują „NIE” do pliku wyjściowego, zastosowano tzw. grupowanie testów, tj. punkty za pary testów SUM3A.IN i SUM3B.IN oraz SUM4A.IN i SUM4B.IN przyznawano tylko wtedy, gdy oba testy zostały zaliczone.

Krzysztof Diks
autor zadania, opracowanie

Adam Borowski
program wzorcowy

AB-słowa

Każdy niepusty ciąg, którego elementami są małe litery a i b, a także ciąg pusty nazywamy ab-słowem. Jeżeli $X = [x_1, \ldots, x_n]$ jest ab-słowem, a i, j takimi dowolnymi liczbami całkowitymi, że $1 \leqslant i \leqslant j \leqslant n$, to przez $X[i..j]$ będziemy oznaczali podsłowo X składające się z kolejnych liter $x_i, \ldots, x_j$. Powiemy, że ab-słowo $X = [x_1, \ldots, x_n]$ jest ładnie zbudowane, *jeżeli zawiera tyle samo liter a, co b i dla każdego $i = 1, \ldots, n$ podsłowo $X[1..i]$ zawiera co najmniej tyle samo liter a, co liter b.*

Podamy teraz indukcyjną definicję podobieństwa *ładnie zbudowanych ab-słów:*

- *każde dwa puste ab-słowa (tzn. nie zawierające żadnych liter) są podobne,*
- *dwa niepuste, ładnie zbudowane ab-słowa $X = [x_1, \ldots, x_n]$ i $Y = [y_1, \ldots, y_m]$ są podobne, jeżeli są tej samej długości ($n = m$) i jest spełniony jeden z następujących warunków:*
  - *$x_1 = y_1$, $x_n = y_n$ oraz $X[2..n-1]$ i $Y[2..n-1]$ są ładnie zbudowanymi ab-słowami*
  - *istnieje i, $1 \leqslant i < n$, takie, że słowa $X[1..i]$, $X[i+1..n]$ są ładnie zbudowanymi ab-słowami i*
    - ▹ *$Y[1..i]$, $Y[i+1..n]$ są ładnie zbudowanymi ab-słowami i $X[1..i]$ jest podobne do $Y[1..i]$ oraz $X[i+1..n]$ jest podobne do $Y[i+1..n]$ lub*
    - ▹ *$Y[1..n-i]$, $Y[n-i+1..n]$ są ładnie zbudowanymi ab-słowami i $X[1..i]$ jest podobne do $Y[n-i+1..n]$ oraz $X[i+1..n]$ jest podobne do $Y[1..n-i]$.*

Stopniem zróżnicowania *niepustego zbioru S ładnie zbudowanych ab-słów nazywamy największą liczbę ab-słów, które można wybrać z S tak, żeby żadne dwa wybrane słowa nie były do siebie podobne.*

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego ABS.IN elementy zbioru S;*
- *policzy stopień zróżnicowania zbioru S;*
- *zapisze wynik w pliku tekstowym ABS.OUT.*

Wejście

W pliku tekstowym ABS.IN są zapisane:

- *w pierwszym wierszu liczba n elementów zbioru S, $1 \leqslant n \leqslant 1000$;*
- *w kolejnych n wierszach elementy zbioru S, tj. ładnie zbudowane ab-słowa, po jednym w każdym wierszu; pierwsza litera każdego ab-słowa jest pierwszym symbolem w wierszu*

i między kolejnymi literami w słowie nie ma żadnych innych znaków; każde słowo ma długość co najmniej 1, a co najwyżej 200.

Wyjście

W pierwszym i jedynym wierszu pliku tekstowego ABS.OUT należy zapisać jedną liczbę całkowitą — stopień zróżnicowania S.

Przykład

Dla pliku wejściowego ABS.IN o zawartości:

```
3
aabaabbbab
abababaabb
abaaabbabb
```

poprawnym rozwiązaniem jest plik ABS.OUT o zawartości:

```
2
```

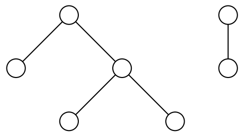
Twój program powienien szukać pliku ABS.IN w katalogu bieżącym i tworzyć plik ABS.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę ABS.???, gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania.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

ROZWIĄZANIE

Łatwo zauważyć, że ab-słowo jest „ładnie zbudowane” wtedy i tylko wtedy, gdy po zamianie każdej litery a na nawias otwierający „(” i każdej litery b na nawias zamykający „)” otrzymamy poprawnie zbudowane **wyrażenie nawiasowe**. Każdemu poprawnie zbudowanemu wyrażeniu nawiasowemu odpowiada jednoznacznie ciąg drzew (z korzeniami) otrzymany w sposób zilustrowany na rysunku 1.

Rys. 1 Przykład odpowiedniości ab-słów, wyrażeń nawiasowych i ciągów drzew

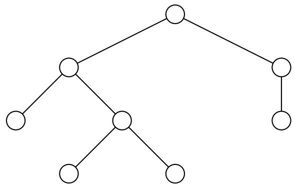
$aabaababbbaabb =$ `(()(()()))(())` $=$



Podczepiając kolejne drzewa w ciągu do jednego, dodatkowego wierzchołka, otrzymamy drzewo jednoznacznie reprezentujące dane ab-słowo lub równoważne mu wyrażenie nawiasowe. Nowy wierzcholek ustanawiamy korzeniem drzewa. Drzewo powstałe z dodania korzenia do ciągu drzew z rysunku 1 przedstawiono na rysunku 2.

Powiemy, że dwa drzewa (posiadające wyróżnione korzenie) T_1 i T_2 są **izomorficzne** wtedy i tylko wtedy, gdy spełniony jest jeden z następujących warunków:

Rys. 2 Ciąg (dwóch) drzew z rysunku 1 połączonych w jedno drzewo nowym wierzchołkiem (aktualnym korzeniem)

$aabaababbbaabb =$ 

- Oba drzewa składają się tylko z pojedynczych wierzchołków.
- Jeśli $v_1, \ldots, v_k$ są synami korzenia drzewa T_1, a $u_1, \ldots, u_m$ są synami korzenia drzewa T_2, to $k = m$ (liczby synów korzeni w obu drzewach są takie same) i istnieje wzajemnie jednoznaczne przyporządkowanie synów korzenia T_1 synom korzenia T_2 takie, że poddrzewa zaczepione w odpowiadających sobie wierzchołkach są izomorficzne.

Na przykład drzewa odpowiadające wyrażeniom nawiasowym

(() (()) ((())) ((())()))
((()) ((())()) ((())) ())
((()) (()(())) () ((())))

są parami izomorficzne.

Nietrudno zauważyć, że relacja izomorfizmu drzew jest relacją:

- zwrotną (każde drzewo jest izomorficzne z samym sobą),
- symetryczną (jeśli drzewo T jest izomorficzne z drzewem T', to drzewo T' jest izomorficzne z drzewem T),
- przechodnią (jeśli drzewo T jest izomorficzne z drzewem T' i drzewo T' jest izomorficzne z drzewem T'', to drzewo T jest izomorficzne z drzewem T'').

O relacji, która jest zwrotna, symetryczna i przechodnia mówimy, że jest **relacją równoważności**.

Ponieważ ab-słowom jednoznacznie odpowiadają drzewa, to relację izomorfizmu drzew można w naturalny sposob przenieść na izomorfizm ab-słów: dwa ab-słowa są izomorficzne wtedy i tylko wtedy, gdy odpowiadające im drzewa są izomorficzne.

Analogicznie relacja podobieństwa na słowach zdefiniowana w treści zadania przenosi się w naturalny sposób na podobieństwa odpowiadających im drzew — podobieństwu podsłów odpowiada podobieństwo odpowiadających im ciagów poddrzew.

Niestety, relacja podobieństwa zdefiniowana w treści zadania nie jest relacją równoważności. Relacja podobieństwa ab-słów jest zwrotna, symetryczna, ale nie jest przechodnia, co, jak się okaże, bardzo utrudnia skonstruowanie efektywnego algorytmu dla naszego zadania.

Jeśli weźmiemy słowa (dla przejrzystości używamy nawiasów zamiast liter a i b)

$A =$ (() (()) ((())) (((()))))
$B =$ ((()) (((()))) ((())) ())
$C =$ ((()) (((()))) () ((())))

to A jest podobne do B, B jest podobne do C, ale A nie jest podobne do C!

Problem obliczania stopnia zróżnicowania danego zbioru ab-słów można wyrazić w terminach teorii grafów.

Powiemy, że wierzchołki grafu są **niezależne** wtedy i tylko wtedy, gdy nie są połączone krawędzią. Podzbiór wierzchołków grafu nazwiemy **zbiorem wierzchołków niezależnych** jeżeli wierzchołki w tym zbiorze są parami niezależne (tzn. jeżeli nie ma pomiędzy nimi żadnej krawędzi). Problem obliczania w grafie zbioru wierzchołków niezależnych o największym rozmiarze jest problemem bardzo trudnym obliczeniowo. Nie jest dla tego problemu znany żaden algorytm, który działałby w czasie wielomianowym, a istnienie takiego algorytmu jest bardzo mało prawdopodobne. Czytelnik, który znajdzie wielomianowy algorytm wyznaczający naliczniejszy zbiór wierzchołków niezależnych rozwiąże jeden z fundamentalnych problemów w informatyce, znany pod nazwą $P \stackrel{?}{=} NP$, zdobędzie sławę i pieniądze. Więcej na ten temat można znaleźć w [12].

Problem znajdowania stopnia zróżnicowania danego zbioru ab-słow można sprowadzić do obliczania rozmiaru najliczniejszego zbioru wierzchołków niezależnych w pewnym grafie. Tym grafem jest graf podobieństwa ab-słów, w którym wierzchołkami są dane ab-słowa, a dwa wierzchołki są połączone krawędzią wtedy i tylko, gdy odpowiadające im ab-słowa są podobne. Zauważmy, że gdyby relacją na słowach byla relacja izomorfizmu, to dla policzenia stopnia zróżnicowania zbioru ab-słów w tej relacji wystarczyłoby znaleźć liczbę spójnych składowych w grafie izomorfizmu, co daje się zrealizować w czasie wielomianowym.

Pan Adam Borowski pokazał, że istnieje wielomianowy algorytm, który dla dowolnego n-wierzchołkowego grafu G konstruuje n ab-słów, dla których graf podobieństwa jest właśnie grafem G. Stopień zróżnicowania takiego zbioru ab-słów jest zatem równy rozmiarowi najliczniejszego zbioru wierzchołków niezależnych w G. Wynika stąd, że gdybyśmy potrafili rozwiązać nasze zadanie w czasie wielomianowym, to rozwiązalibyśmy problem $P = NP$.*

Algorytm wzorcowy dla zadania ab-słowa działa w czasie wykładniczym. Na taką złożoność algorytmu ma wpływ sposób obliczania maksymalnego zbioru niezależnego w grafie podobieństwa. Zbiór ten jest obliczany przy pomocy metody zwanej **przeszukiwaniem z powrotami**, która w tym wypadku polega na generowaniu w systematyczny sposób wszystkich niezależnych podzbiorów i wybraniu z nich najliczniejszego. Taka metoda ma szanse powodzenia tylko wtedy, gdy graf, w którym szukamy najliczniejszego zbioru niezależnych wierzchołków, jest małego rozmiaru. Algorytm wzorcowy gwarantuje, że dla danych o rozmiarze wyspecyfikowanym w treści zadania, grafy, w których obliczane są zbiory niezależnych wierzchołków, są małych rozmiarów.

W jaki sposób to osiągamy? Nietrudno zauważyć, że dwa ab-słowa, które nie są izomorficzne, nie są do siebie podobne. Wynika stąd, że możemy zbiór wszystkich ab-słów podzielić na klasy, w których słowa są parami izomorficzne i dla każdej z nich

---

* Pojawia się tu subtelna różnica polegająca na tym, że z rozwiązania wynika algorytm, który znajduje w czasie wielomianowym rozmiar najliczniejszego zbioru wierzchołków niezależnych, a nie sam zbiór. Dysponując jednakże takim algorytmem możemy w prosty sposób wyznaczyć w czasie wielomianowym również elementy tego zbioru. Konstrukcję takiego rozszerzenia algorytmu pozostawiamy jako ćwiczenie dla Czytelnika.

policzyć oddzielnie stopień zróżnicowania. Poszukiwanym stopniem zróżnicowania całego zbioru jest suma stopni zróżnicowania poszczególnych klas. Algorytm podziału zbioru wszystkich ab-słów na klasy słów parami izomorficznych jest następujący:

- każdemu ab-słowu przyporządkowujemy sygnaturę, która jednoznacznie identyfikuje ab-słowa z tej samej klasy;
- sortujemy ab-słowa względem ich sygnatur;
- wyznaczamy klasy słów parami izomorficznych, jako ciągi kolejnych słów o takich samych sygnaturach.

Sygnaturę o potrzebnych nam własnościach można wybrać na wiele sposobów. Tutaj za sygnaturę przyjęliśmy drzewo ab-słowa przekształcone do **postaci kanonicznej** za pomocą permutowania synów każdego wierzchołka w tym drzewie.

Zdefiniujmy na drzewach następujący porządek liniowy (zob. [11]), oznaczany przez $\prec$:

(1) Jeżeli korzeń drzewa T ma mniej synów od korzenia drzewa T', to $T \prec T'$.

(2) Jeżeli dla uporządkowanych (w porządku $\prec$) ciągów synów korzeni T i T', odpowiednio $v_1, \ldots, v_k$ i $u_1, \ldots, u_k$, zachodzi $v_k = u_k, \ldots, v_{j+1} = u_{j+1}, v_j \prec u_j$ to $T \prec T'$.

Powiemy, że drzewo jest w **postaci kanonicznej**, jeśli synowie każdego wierzchołka w tym drzewie są uporządkowani nierosnąco zgodnie z porządkiem $\prec$ dla poddrzew, których są korzeniami.

Zdefiniujemy dodatkowo relację silnego podobieństwa, która pozwala na zmniejszenie rozmiaru grafu podobieństwa, w którym mamy znajdować najliczniejszy zbiór niezależnych wierzchołków.

Dwa ab-słowa A i B są **silnie podobne** wtedy i tylko wtedy, gdy dla każdego ab-słowa C, słowo A jest podobne do słowa C wtedy i tylko wtedy, gdy słowo B jest podobne do C.

Tak jak w przypadku podobieństwa, silne podobieństwo przenosi się w naturalny sposób na drzewa odpowiadające ab-słowom.

Łatwo sprawdzić, że wszystkie słowa, które są do siebie silnie podobne, są nierozróżnialne w sensie relacji podobieństwa — każdą grupę silnie podobnych słów możemy zastąpić jej jednym reprezentantem („skleić”), a stopień zróżnicowania zbioru słów nie zmieni się. Problem w tym, że nie dysponujemy szybkim algorytmem sprawdzającym silne podobieństwo słów. Spróbujemy jednak posłużyć się algorytmem, który czasami nie stwierdza silnego podobieństwa tam, gdzie ono zachodzi, ale za to jest szybki. Niedoskonałość algorytmu spowoduje, że po wykonaniu operacji sklejania pozostaną słowa, które są silnie podobne, ale nie zostały sklejone. No cóż, trudno. Okazuje się jednak, że zmniejszenie rozmiaru grafu, które dokonuje się w wyniku sklejania wierzchołków, jest wystarczające dla naszych celów.

Oto szybki algorytm „częściowego sprawdzania” silnego podobieństwa dwóch słów A i B:

- Jeśli liczba synów korzenia drzewa słowa A jest różna od liczby synów korzenia drzewa słowa B, to słowa nie są silnie podobne.

Załóżmy zatem, że liczby synów obu korzeni są takie same.

- Jeśli liczba synów jest mniejsza niż 4, synów korzeni drzew A i B sortujemy wg sygnatur zaczepionych w nich poddrzew. Jeśli odpowiadające sobie w tym porządku poddrzewa z obu drzewach A i B są silnie podobne, to słowa A i B też są silnie podobne.
- Jesli liczba synów jest większa od 3, sprawdzamy silne podobieństwo kolejnych (bez sortowania!) par poddrzew drzew słów A i B, których korzenie są synami korzeni tych drzew. Jeśli wszyskie pary poddrzew są silnie podobne, to drzewa A i B są też silnie podobne.

Oczywiście, jeśli słowa A i B są silnie podobne, to są podobne.

A oto wzorcowy algorytm obliczania stopnia zróżnicowania wejściowego zbioru ab-słów.

- Dzielimy dany zbiór ab-słów na klasy słów parami izomorficznych.
- Wewnątrz każdej klasy:
  - Znajdujemy zbiory słów silnie podobnych. Z każdego zbioru słów silnie podobnych usuwamy wszystkie słowa oprócz jednego.
  - Dla pozostałych słów budujemy graf podobieństwa.
  - W grafie podobieństwa znajdujemy największy zbiór wierzcholkow niezależnych.

Suma rozmiarów największych zbiorów wierzchołków niezależnych w każdej z klas słów parami izomorficznych jest poszukiwanym stopniem zróżnicowania wejściowego zbioru ab-słów.

TESTY

Do sprawdzania rozwiązań zawodników użyto 13-tu testów ABS0.IN–ABS12.IN. Test ABS0.IN był testem z treści zadania. W testach ABS3.IN, ABS5.IN, ABS6.IN, ABS7.IN, ABS11.IN relacja podobieństwa na zbiorze słów wejściowych pokrywała się z relacją izomorfizmu, przy czym testy ABS5.IN–ABS7.IN były testami o dużych rozmiarach (1000 słów o długościach dochodzących do 200 znaków). Pozostałe testy zaliczały tylko programy, których autorzy zauważyli, że relacja podobieństwa nie jest relacją rownoważności. W szczególności rozwiązanie dla zestawu danych ABS9.IN sprowadzało się do znalezienia najliczniejszego zbioru wierzchołków niezależnych w pewnym grafie ośmiowierzchołkowym.

Zawody II stopnia

opracowania zadań

Piotr Chrząstowski-Wachtel — autor zadania, opracowanie
Marcin Mucha — program wzorcowy

Sieć dróg

Do mapy drogowej dołączona została dyskietka z tabelą długości najkrótszych dróg (odległości) między każdymi dwoma miastami na mapie. Wszystkie drogi są dwukierunkowe. Położenie miast na mapie ma następującą ciekawą własność. Jeżeli długość najkrótszej drogi z miasta A do B jest równa sumie długości najkrótszych dróg z A do C i z C do B, to miasto C leży na (pewnej) najkrótszej drodze z A do B. Powiemy, że dwa miasta A i B sąsiadują ze sobą, jeżeli nie istnieje miasto C takie, że długość najkrótszej drogi z miasta A do B jest równa długości najkrótszych dróg z A do C i z C do B.
Na podstawie danej tabeli odległości znajdź wszystkie pary miast sąsiadujących ze sobą.

Przykład

Jeżeli tabela odległości ma postać

	A	B	C
A	0	1	2
B	1	0	3
C	2	3	0

wówczas sąsiednimi miastami są A i B oraz A i C.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego SIE.IN tabelę odległości;*
- *znajduje wszystkie pary sąsiednich miast;*
- *zapisuje wynik w pliku tekstowym SIE.OUT.*

Wejście

W pierwszym wierszu pliku wejściowego SIE.IN znajduje się liczba całkowita n, $1 \leqslant n \leqslant 200$. Jest to liczba miast na mapie. Miasta ponumerowane są jednoznacznie od 1 do n.
Tabela odległości jest zapisana w kolejnych n wierszach pliku SIE.IN. W wierszu $i+1$, $1 \leqslant i \leqslant n$, pliku SIE.IN jest zapisanych n nieujemnych liczb całkowitych nie większych od 200, oddzielonych pojedynczymi odstępami. Liczba j-ta jest odległością między miastami i oraz j.

Wyjście

Twój program powinien wypisać w pliku wyjściowym SIE.OUT wszystkie pary (numerów) sąsiednich miast, po jednej parze w każdym wierszu.
Każda para powinna pojawić się tylko raz. Liczby w parze powinny być uporządkowane rosnąco i oddzielone pojedynczym odstępem.

Kolejność wypisywania par powinna być taka, że dla pary (a, b) poprzedzającej parę (c, d), $a < c$ lub ($a = c$ i $b < d$).

Przykład

Dla pliku tekstowego SIE.IN:

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

poprawnym rozwiązaniem jest plik wyjściowy SIE.OUT:

```
1 2
1 3
```

UZUPEŁNIENIE TREŚCI ZADANIA

Podczas trwania zawodów treść zadania uzupełniono następującym stwierdzeniem:

Odległość między dwoma różnymi miastami jest zawsze większa od zera.

ROZWIĄZANIE

Zadanie powstało, gdy planowałem przejazd na wakacje samochodem nad morze. Wpatrywałem się w mapę drogową Polski i szukałem najkrótszej drogi z Warszawy do Świnoujścia. Zacząłem od wydrukowanej na odwrocie mapy tabelki odległości drogowych między miastami. Odczytałem tam, że do Świnoujścia mam 567 km, ale nijak nie mogłem wykombinować, jak tę wartość osiągnąć. Usiłowałem odszukać w tabelce takie miejscowości, przez które przejeżdżając będę miał w sumie te 567 km. Dobrze wyglądała międzynarodowa droga przez Poznań. Ale okazało się, że nie była ona najkrótsza. Przekonałem się o tym sprawdzając sumę odległości między Warszawą a Poznaniem i między Poznaniem a Świnoujściem. Z Warszawy do Poznania jest 308 km, a z Poznania do Świnoujścia 297 km — w sumie o 38 km za dużo. Podobnie było z Płockiem, Włocławkiem, Bydgoszczą, Piłą, Koszalinem — większymi miejscowościami, które optycznie leżą mniej więcej po drodze. Żadna z nich jakoś nie chciała mieć tej własności, żeby łączna liczba kilometrów z Warszawy do niej i od niej do Świnoujścia wyniosła 567.

W końcu znalazłem takie miasto: Toruń. Okazało się, że z Warszawy do Torunia jest 209 km, a z Torunia do Świnoujścia 358 km. Wiedziałem już, że powinienem jechać przez Toruń. Żeby opracować resztę trasy trzeba było rozwiązać to samo zadanie dla przejazdu z Warszawy do Torunia i z Torunia do Świnoujścia. Niestety, w zestawie większych miast, dla których była skonstruowana tabelka, nie było ani jednego między Warszawą a Toruniem i ani jednego między Toruniem a Świnoujściem, przez które prowadziłaby najkrótsza droga.

Oderwałem się więc od tabelki i wziąłem się za studiowanie mapy. Tabelka bowiem była zbyt zgrubna, żeby do końca podpowiedzieć marszrutę. Zawierała za mało miast. Pomyślałem sobie, że szkoda, bo gdyby zawierała wszystkie miasta w Polsce (a jest ich kilkanaście tysięcy), to wyszłoby ciekawe zadanie. W odróżnieniu od mapy, której reprezentacja graficzna jest niewygodna do automatyzacji, tabelka zawiera konkretne odległości, a dodatkowo posiada wszystkie dane, aby odtworzyć strukturę sąsiedztwa. Oczywiście sporządzenie takiej tabelki, w której uwzględnione będą wszystkie widniejące na mapie miasta w Polsce jest mało realne: przy n miastach rozmiar tabelki byłby wtedy równy $n(n+1)/2$ — mniej więcej połowa kwadratu ich liczby. Dałoby to wielkość rzędu setek milionów danych — za dużo, żeby w praktyce taka reprezentacja mogła być użyteczna. Dlatego na mapach podaje się zaledwie fragment takiej tabeli, wybierając jedynie niektóre ważniejsze węzły drogowe.

Znacznie prościej jest mapę drogową pamiętać podając odległości między sąsiednimi węzłami sieci samochodowej. Ze względu na to, że z takiego węzła możemy wyjechać przeważnie w 2–5 kierunkach, liczba zapisów dotyczących odległości łączących sąsiadów jest równa mniej więcej $5n$, czyli dla Polski kilkadziesiąt tysięcy. Oczywiście odtworzenie najkrótszych tras na podstawie takiej informacji byłoby zadaniem niejako dualnym do zadania konkursowego. Można się spodziewać, że część algorytmów rozwiązujących któryś z tych problemów będzie miała swoją dualną wersję rozwiązującą problem przeciwny. Dualne do naszego zadanie odtworzenia najkrótszych połączeń na podstawie opisu odległości między sąsiadami znane jest pod nazwą problemu szukania najkrótszych ścieżek w grafie. Tutaj mamy do czynienia ze specyficznym rodzajem grafu: takim, że wagi krawędzi łączących węzły są interpretowane, jako odległości. Na dobrą sprawę znajdowanie najkrótszej marszruty na podstawie tabelki odległości można rozbić na dwa etapy: najpierw odtworzyć strukturę sąsiedztwa, a następnie rozwiązać problem szukania najkrótszej ścieżki w grafie. Treścią zadania konkursowego był pierwszy etap.

Zadanie odtworzenia relacji sąsiedztwa jest proste. Wystarczy dla każdej pary węzłów sprawdzić warunek definiujący sąsiedztwo między dwoma węzłami. Dwa węzły są sąsiadami wtedy i tylko wtedy, gdy nie istnieje pomiędzy nimi węzeł odległy od nich w sumie o tyle, ile wynosi bezpośrednia odległość między nimi. Zorganizujmy więc trzy zagnieżdżone pętle: dwie zewnętrzne przebiegające zbiór par węzłów i najbardziej zagnieżdżoną przebiegającą wszystkie węzły i badającą, czy nie występują one na ścieżce między tamtą parą:

```
{tablica A zawiera odległości między miastami,
przy czym na przekątnej A[i,i] wstawione są  jedynki}
for i:=1 to n − 1 do
  for j:=i + 1 to n do begin
    {badamy parę węzłów (i,j); ze względu na symetrię
    odległości możemy ograniczyć się do uporządkowanych par z i < j}
    sąsiedzi := true; {domniemamy, że i oraz j są sąsiadami}
    k := 1;
    while sąsiedzi and (k ⩽ n) do
      if A[i,k] + A[k,j] = A[i,j] then sąsiedzi := false;
      else k:=k + 1;
    if sąsiedzi then wypisz(i, j)
  end;
```

Jak widać, zadanie było wyjątkowo proste. Większość zawodników nie miała problemów z jego rozwiązaniem.

TESTY

Do sprawdzenia rozwiązań zawodników użyto 12-tu testów SIE0.IN–SIE11.IN:

- SIE0.IN — test z treści zadania.
- SIE1.IN — najmniejszy możliwy zestaw danych.
- SIE2.IN, SIE7.IN — węzły ułożone na prostej; sąsiadami są węzły o kolejnych numerach.
- SIE3.IN, SIE8.IN — węzły ułożone gwiaździście wokół jednego wyróżnionego węzła; wyróżniony węzeł jest sąsiadem każdego innego. Żadne inne węzły nie sąsiadują ze sobą.
- SIE4.IN, SIE9.IN — węzły ułożone na pierścieniu; sąsiadami są węzły o kolejnych numerach oraz węzeł pierwszy z ostatnim.
- SIE5.IN, SIE10.IN — każde dwa węzły sąsiadują ze sobą.
- SIE6.IN, SIE11.IN — sąsiedztwa węzłów generowane losowo.

Testy były podzielone na trzy grupy, przy czym w ramach grup drugiej i trzeciej powtarzały się schematy generowania danych. Pierwszą grupę stanowiły testy podstawowe: test z treści zadania SIE0.IN i test SIE1.IN o rozmiarze najmniejszym z możliwych (dwa węzły połączone krawędzią).

Druga grupa (SIE2.IN–SIE6.IN) to testy o małych rozmiarach, dziesięciowęzłowe. Trzecia grupa (SIE7.IN–SIE11.IN) zawierała dokładnie takie same typy połączeń, jak grupa druga, tyle że dla 200 węzłów.

Wojciech Rytter
autor zadania, opracowanie

Krzysztof Sobusiak
program wzorcowy

Pakowanie kontenerów

Pewna fabryka pakuje swoje wyroby do pudełek, które mają kształt walca. Podstawy wszystkich pudełek są takie same. Wysokość każdego pudełka jest zawsze nieujemną, całkowitą potęgą dwójki, tzn. liczbą postaci 2^i, dla pewnego $i = 0, 1, 2, \ldots$. Liczbę i (wykładnik potęgi), która charakteryzuje wielkość pudełka, nazywamy rozmiarem tego pudełka. We wszystkich pudełkach znajduje się taki sam towar, ale w różnych pudełkach wartość towaru może być różna — towar wyprodukowany wcześniej jest tańszy. Dyrekcja fabryki wydała zarządzenie, żeby starać się wydawać najpierw towar najstarszy, czyli o najmniejszej wartości. Towar jest wywożony z magazynu w kontenerach. Każdy kontener, podobnie jak pudełka, ma kształt walca. Średnica kontenera jest niewiele większa od średnicy pudełek tak, że pudełka można swobodnie umieszczać w kontenerach. Wysokość każdego kontenera jest też nieujemną, całkowitą potęgą dwójki. Podobnie jak w przypadku pudełek, rozmiarem kontenera nazywamy wykładnik tej potęgi. Żeby bezpiecznie przewozić towar w kontenerze, należy go szczelnie wypełnić pudełkami, tzn. suma wysokości umieszczonych w kontenerze pudełek musi być równa wysokości tego kontenera.
Do magazynu dostarczono zestaw kontenerów. Sprawdź, czy można pudełkami z magazynu wypełnić szczelnie wszystkie kontenery. Jeśli tak, to podaj minimalną wartość towaru jaki można wywieźć z magazynu w tych, szczelnie wypełnionych, kontenerach.

Przykład

Rozważmy magazyn, w którym znajdują się 4 pudełka o następujących rozmiarach i wartościach przechowywanego w nich towaru:

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

Dwa kontenery o rozmiarach 1 i 2 można szczelnie wypełnić dwoma pudełkami o łącznej wartości 3, 4 lub 5, lub trzema pudełkami o łącznej wartości 9.
Kontener o rozmiarze 5 nie da się szczelnie wypełnić pudełkami z magazynu.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego PAK.IN charakterystyki pudełek w magazynie (rozmiar, wartość) oraz charakterystyki kontenerów (ile jest kontenerów danego rozmiaru);*
- *sprawdza, czy wszystkie kontenery w zestawie można szczelnie wypełnić pudełkami z magazynu, a jeśli tak, to oblicza minimalną wartość towaru jaką można w tych kontenerach wywieźć z magazynu;*
- *zapisuje wynik w pliku tekstowym PAK.OUT.*

Wejście

W pierwszym wierszu pliku wejściowego PAK.IN znajduje się liczba całkowita n, $1 \leqslant n \leqslant 10000$. Jest to liczba pudełek w magazynie.
W każdym z n kolejnych wierszy zapisane są dwie nieujemne liczby całkowite oddzielone pojedynczym odstępem. Są to charakterystyki jednego pudełka, odpowiednio, rozmiar pudełka i wartość zawartego w nim towaru. Rozmiar pudełka jest nie większy od 1000, natomiast wartość jest nie większa od 10000.
W kolejnym wierszu podana jest dodatnia liczba całkowita q określająca, ile różnych rozmiarów mają kontenery dostarczone do magazynu.
W każdym z kolejnych q wierszy jest zapisana para dodatnich liczb całkowitych oddzielonych pojedynczym odstępem. Pierwsza z tych liczb jest rozmiarem kontenera, natomiast druga jest liczbą dostarczonych kontenerów o tym rozmiarze.
Wiadomo, że maksymalna liczba kontenerów nie przekracza 5000, a rozmiar kontenera jest nie większy od 1000.

Wyjście

Twój program powinien zapisać w pierwszym i jedynym wierszu pliku wyjściowego PAK.OUT:

- *jedno słowo NIE, jeżeli nie da się szczelnie wypełnić pudełkami z magazynu kontenerów z podanego zestawu, lub*
- *jedną liczbę całkowitą, której wartość jest równa minimalnej, łącznej wartości towaru z pudełek, którymi można szczelnie wypełnić wszystkie kontenery z podanego zestawu.*

Przykład

Dla pliku wejściowego PAK.IN:

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

poprawnym rozwiązaniem jest plik wyjściowy PAK.OUT:

```
3
```

ROZWIĄZANIE

Rozwiązanie opiera się na pewnej własności sumowania całkowitych potęg dwójki. Niech liczba 2^k będzie sumą przynajmniej dwóch potęg dwójki:

$$2^k = 2^{i_1} + 2^{i_2} + \ldots + 2^{i_s}, \qquad i_1 \leqslant i_2 \leqslant \ldots \leqslant i_s.$$

Wówczas wykładnik dowolnego składnika sumy jest mniejszy od k, a ponadto

$$i_1 = i_2.$$

Innymi słowy, składnik sumy o najmniejszym wykładniku musi się powtarzać.

Oznaczmy przez r minimalny rozmiar pudełka, a przez R minimalny rozmiar kontenera. Z opisanej własności sumowania potęg dwójki wynikają następujące własności optymalnego rozmieszczenia pudełek w kontenerach:

(1) $r = R$ — jeśli istnieje jakiekolwiek rozwiązanie, to istnieje rozwiązanie optymalne, w którym w najmniejszym kontenerze jest pudełko rozmiaru r o najmniejszej wartości,

(2) $r < R$ — jeśli istnieje jakiekolwiek rozwiązanie, to:

- jeśli jest tylko jedno pudełko rozmiaru r, to możemy je pominąć,
- jeśli są co najmniej dwa pudełka rozmiaru r, to możemy dwa z nich, najmniej wartościowe, zastąpić jednym o rozmiarze $r+1$ i wartości będącej sumą wartości tych dwóch. Rozwiązanie optymalne dla tak zmodyfikowanego zestawu pudełek jest takie samo, jak dla zestawu wyjściowego.

(3) $r > R$ — zadanie nie ma rozwiązania, ponieważ najmniejszego kontenera nie da się zapełnić; pudełka są za duże w stosunku do niego.

Najciekawszy jest punkt (2) — mówi nam, że w pewnych sytuacjach dwa pudełka można **skleić** i traktować je zawsze jako jedno większe pudełko.

Z powyższych spostrzeżeń otrzymujemy **algorytm zachłanny** polegający na stosowaniu za każdym razem któregoś z punktów (1)–(3), w zależności od aktualnych wartości r i R. Albo zapełniamy całkowicie jakiś kontener (i liczba kontenerów maleje), albo sklejamy dwa pudełka (i liczba pudełek maleje), albo się zatrzymujemy, ponieważ zadanie nie ma rozwiązania.

Powyższa analiza pozwala już napisać program. W celu zmniejszenia złożoności zastosujemy pewną „sztuczkę": scalanie posortowanych list. Oznaczmy taką operację przez *scal*.

Załóżmy, że mamy listę *rozmiaryKontenerów*, na której zapamiętane są rozmiary kontenerów w kolejności niemalejącej (przy tym jeśli jest np. 6 kontenerów o rozmiarze 7, to na liście jest 6 siódemek pod rząd). Niech $pudełka_p, \ldots, pudełka_q$ są listami wartości pudełek o poszczególnych rozmiarach $p, \ldots, q$, gdzie p jest minimalnym rozmiarem pudełka, a q jest maksymalnym rozmiarem pudełka (na liście $pudełka_k$ przechowujemy pudełka o rozmiarze k posortowane względem wartości).

Załóżmy, że funkcja *pobierzPierwszy*(**var** *l* : *lista*) : *element* pobiera pierwszy (a więc najmniejszy) element z posortowanej listy (i zwraca jako wynik wywołania funkcji) i przy okazji usuwa ten element z listy. Umówmy się przy tym, że jeśli zastosujemy tę operację do pustej listy to algorytm się zatrzyma stwierdzając, że nie ma rozwiązania. Niech ponadto procedura *wstaw*(**var** *l* : *lista*; *e* : *element*) wstawia element na koniec listy.

Korzystamy z pomocniczej listy *podwójne*, na którą wrzucamy pudełka rozmiaru $r+1$ powstające przez sklejenie dwóch mniejszych pudełek z listy $pudełka_r$. Po wykorzystaniu listy $pudełka_r$ scalamy pomocniczą listę z listą $pudełka_{r+1}$.*

* W opisie algorytmu, symbolem „_" (*podkreślenie*) oznaczono zmienną, której wartość nie jest w programie odczytywana (wykorzystaną w celu zapisania w języku Pascal wywołania funkcji z jednoczesnym zignorowaniem jej wyniku).

```
···
{przydziel pudełka do list pudełka_p, ..., pudełka_q wg ich rozmiarów}
{posortuj każdą z list pudełka_p, ..., pudełka_q wg wartości}
···
r := p;
podwójne := ∅;
wartość := 0;
brakRozwiązania := false;
while rozmiaryKontenerów ≠ ∅ and not brakRozwiązania do begin
  if pudełka_r = ∅ then begin
    scal(podwójne, pudełka_{r+1});
    podwójne := ∅
  end;
  R := min(rozmiaryKontenerów);
  r := min {i : pudełka_i ≠ ∅};
  if r = R then begin
    wartość := wartość + pobierzPierwszy(pudełka_r);
    _ := pobierzPierwszy(rozmiaryKontenerów);
  end
  else
    if r < R then begin
      if długość listy pudełka_r ⩾ 2 then
        wstaw(podwójne,
                    pobierzPierwszy(pudełka_r) + pobierzPierwszy(pudełka_r))
      else
        _ := pobierzPierwszy(pudełka_r)
    end
    else
      brakRozwiązania := true
end
if brakRozwiązania then wypisz('NIE')
else wypisz(wartość)
```

Ciekawą własnością algorytmu jest to, że da się on zaimplementować w czasie liniowym. Gdybyśmy zawsze wstawiali **sklejoną** parę pudełek do listy pudełek większego rozmiaru, to za każdym razem potrzeba byłoby około $\log n$ kroków, w sumie $n \cdot \log n$. My robimy takie wstawienia „hurtowo", zamiast wstawiania pojedynczych elementów scalamy (za pomocą operacji *scal*) listy złożone z potencjalnie bardzo wielu elementów. Scalanie list uporządkowanych można wykonać w czasie liniowym. Przyjmijmy, że każdy element scalanych list jest obciążany „jednostką kosztu". Zauważmy, że dla każdego r, każde pudełko o rozmiarze r jest albo umieszczane w kontenerze, albo usuwane, albo sklejane z innym pudełkiem o tym samym rozmiarze i umieszczane na liście *podwójne*. Wynika stąd, że sumaryczny koszt scalań jest proporcjonalny do liczby pudełek pojawiających się w trakcie działania algorytmu (zliczamy zarówno pudełka dane na początku, jak i powstałe w wyniku scalań). Nietrudno policzyć, że takich pudełek jest co najwyżej dwa razy tyle, ile pudełek początkowych.

Pudełka pamiętamy w listach posortowanych rosnąco wg wartości, osobno dla każdego z rozmiarów. Na początku pudełka są nieuporządkowane, więc musimy je posortować. Można to zrobić w czasie liniowym korzystając z faktu, że wartości pudełek są liczbami całkowitymi z małego przedziału. Pozwala na zastosowanie sortowania kubełkowego* po rozmiarach oraz po wartościach elementów (czyli leksykograficznie, tzn. jeżeli rozmiary dwóch elementów są równe, to porównujemy ich wartości).

Z powyższych rozważań wynika, że złożoność algorytmu jest liniowa.

TESTY

Do sprawdzenia rozwiązań zawodników użyto 11-tu testów. Testy PAK3.IN–PAK7.IN zostały wygenerowane automatycznie w taki sposób, by w algorytmach działających na listach rozmiarów, operacja scalania działała zawsze na listach o podobnych długościach oraz rozkładach wartości. Dla danej liczby pudełek n, liczby kontenerów m oraz rozmiaru najmniejszego pudełka r w takim teście występuje $d = \log_2(n+1)$ różnych rozmiarów. Jego postać wygląda w przybliżeniu następująco (C oznacza maksymalną dopuszczalną wartość pudełka na wejściu):

rozmiar r	$\frac{n}{2}$ pudełek o wartościach z $\langle \frac{C}{2^d}, \frac{C}{2^{d-1}} \rangle$
rozmiar $r+1$	$\frac{n}{4}$ pudełek o wartościach z $\langle \frac{C}{2^{d-2}} - \frac{C}{2^{d-1}}, \frac{C}{2^{d-2}} \rangle$, $\frac{n}{4}$ kontenerów
rozmiar $r+2$	$\frac{n}{8}$ pudełek o wartościach z $\langle \frac{C}{2^{d-3}} - \frac{C}{2^{d-1}}, \frac{C}{2^{d-3}} \rangle$, $\frac{n}{8}$ kontenerów
...	
rozmiar $r+d-1$	$\frac{n}{2^{d-1}}$ pudełek o wartościach z $\langle C - \frac{C}{2^{d-1}}, C \rangle$, $\frac{n}{2^{d-1}}$ kontenerów

Dla takiego testu w algorytmach operujących na listach rozmiarów, z rozmiaru $r+i$ do rozmiaru $r+i+1$ przechodzi około $\frac{n}{2^{i+2}}$ pudełek o wartościach z przedziału (mniej więcej) $\langle \frac{C}{2^{d-i-1}} - \frac{C}{2^{d-1}}, \frac{C}{2^{d-i-1}} \rangle$.

- PAK0.IN — test z treści zadania.
- PAK1.IN — prosty test poprawnościowy.
- PAK2A.IN — prosty test z odpowiedzią „NIE”.
- PAK2B.IN — prosty test poprawnościowy, zgrupowany z testem PAK2A.IN
- PAK3.IN–PAK6.IN — testy opisane powyżej.
- PAK7.IN–PAK10.IN — testy wydajnościowe o dużych rozmiarach.

---

* zob. [12] str. 215–217

Wojciech Rytter autor zadania

Krzysztof Diks opracowanie

Adam Borowski program wzorcowy

Równanie na słowach

Słowem dwójkowym nazywamy każdy niepusty ciąg złożony z 0 lub 1. Równanie na słowach ma postać $x_1x_2\ldots x_l = y_1y_2\ldots y_r$*, gdzie* x_i *i* y_j *sa cyframi dwójkowymi (0 lub 1) lub zmiennymi, to jest małymi literami alfabetu angielskiego. Dla każdej zmiennej jest ustalona długość słów dwójkowych, które można podstawiać w jej miejsce. Długość tę nazywamy długością zmiennej. Rozwiązanie równania na słowach polega na przypisaniu każdej zmiennej słowa dwójkowego o odpowiadającej tej zmiennej długości, w taki sposób, by po zastąpieniu zmiennych w równaniu przez przypisane im słowa, obie strony równania (słowa dwójkowe) były takie same.*

Dla danego równania na słowach policz ile jest różnych rozwiązań tego równania.

Przykład

Niech 4, 2, 4, 4, 2 będą długościami zmiennych a, b, c, d, e *w poniższym równaniu:*

$$1bad1 = acbe$$

To równanie ma 16 różnych rozwiązań.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego ROW.IN liczbę równań i ich opisy;*
- *znajduję liczbę rozwiązań każdego równania;*
- *zapisuje wyniki w pliku tekstowym ROW.OUT.*

Wejście

W pierwszym wierszu pliku wejściowego ROW.IN znajduje się liczba całkowita x*,* $1 \leqslant x \leqslant 5$*. Jest to liczba równań.*
Następne wiersze zawierają opisy x *równań. Na każdy opis składa się 6 wierszy. Między kolejnymi opisami nie ma pustych wierszy. Każde równanie jest opisane w następujący sposób: W pierwszym wierszu opisu znajduje się liczba całkowita* k*,* $0 \leqslant k \leqslant 26$*. Jest to liczba różnych zmiennych w równaniu. Przyjmujemy, że zawsze zmiennymi jest* k *pierwszych małych liter alfabetu angielskiego. W drugim wierszu jest zapisany ciąg* k *liczb całkowitych dodatnich oddzielonych pojedynczymi odstępami. Te liczby to długości* k *kolejnych zmiennych* $a, b, \ldots$ *występujących w równaniu. Trzeci wiersz opisu zawiera tylko jedną dodatnią liczbę całkowitą* l*. Jest to długość lewej strony równania, tj. długość słowa utworzonego z cyfr 0 lub 1 i jednoliterowych zmiennych. W następnym wierszu jest zapisana lewa strona równania jako ciąg* l *cyfr lub zmiennych bez odstępów między nimi. Następne dwa wiersze zawierają opis prawej strony równania.*
Pierwszy z tych wierszy zawiera dodatnią liczbę r*. Jest to długość prawej strony równania. Drugi z wierszy zawiera prawą stronę równania zapisaną w taki sam sposób jak jego lewa*

strona. Liczba cyfr plus suma długości wszystkich zmiennych (licząc wszystkie wystąpienia każdej zmiennej) po każdej stronie równania nie przekracza 10000.

Wyjście

Dla każdego i, $1 \leqslant i \leqslant x$, Twój program powinien zapisać w i-tym wierszu pliku wyjściowego ROW.OUT liczbę różnych rozwiązań i-tego równania.

Przykład

Dla pliku tekstowego ROW.IN:

```
1
5
4 2 4 4 2
5
1bad1
4
acbe
```

poprawnym rozwiązaniem jest plik wyjściowy ROW.OUT:

```
16
```

ROZWIĄZANIE

Rozwinięciem zmiennej x nazywamy ciąg $(x,1),(x,2),\ldots,(x,d_x)$, gdzie d_x jest długością tej zmiennej. **Rozwinięciem lewej (prawej) strony równania** nazywamy ciąg powstający w wyniku zastąpienia wystąpień wszystkich zmiennych po lewej (prawej) stronie równania przez ich rozwinięcia. Jeżeli długości rozwinięć obu stron są różne, równanie nie ma rozwiązania. Przyjmijmy zatem, że te długości są takie same. Problem obliczenia liczby rozwiązań danego równania sprowadza się do policzenia liczby spójnych składowych w **grafie zależności** tego równania. Wierzchołkami grafu zależności są pozycje w rozwinięciu zmiennych z równania i dwa symbole — cyfry „0" i „1". Dokładniej, jeśli x jest zmienną w równaniu, to w grafie zależności tego równania występują wierzchołki $(x,1),(x,2),\ldots,(x,d_x)$. Krawędzie w grafie zależności tworzymy w następujący sposób: dla każdej pozycji p w rozwinięciu lewej strony równania łączymy krawędzią element występujący w tym rozwinięciu na pozycji p z elementem występującym na tej samej pozycji w rozwinięciu prawej strony równania.

Na rysunku 1 przedstawiono graf zależności dla równania

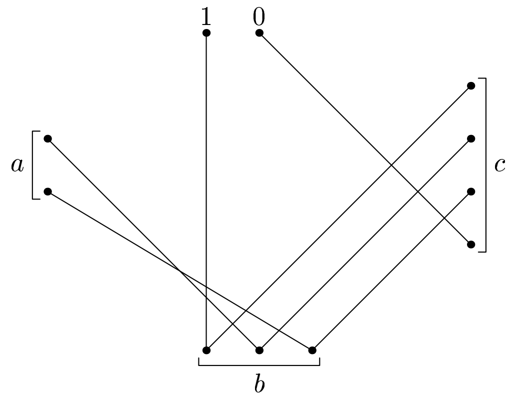
$$1ab0 = bc,$$

w którym długości zmiennych wynoszą:

$$a = 2, \qquad b = 3, \qquad c = 4.$$

Nietrudno zauważyć, że równanie nie ma rozwiązania wtedy i tylko wtedy, gdy w grafie zależności wierzchołki „0" i „1" należą do tej samej spójnej składowej. Jeżeli

Rys. 1 Graf zależności dla równania $1ab0 = bc$ z przykładu.



„0” i „1” należą do różnych spójnych składowych, to liczba rozwiązań równania jest równa dwa do potęgi będącą liczbą składowych w grafie zależności, które nie zawierają, ani „0”, ani „1”. Liczba rozwiązań równania z przykładu wynosi 4.

Z naszych rozważań wynika, że algorytm obliczania liczby rozwiązań danego równania na słowach jest bardzo prosty:

(1) Jeśli długości rozwinięć lewej i prawej strony równania są różne, to STOP — równanie nie ma rozwiązania.

(2) Zbuduj graf zależności równania.

(3) Jeśli do jednej składowej w grafie zależności należą „0” i „1”, to STOP — równanie nie ma rozwiązania.

(4) Liczba rozwiązań równania wynosi

$$2^{\text{liczba składowych w grafie zależności}}.$$

Spójne składowe grafu można obliczyć na jeden ze sposobów przedstawionych w opisie zadania Prostokąty. W rozwiązaniu wzorcowym do wyszukiwania spójnych składowych użyto przeszukiwania wszerz.

TESTY

Do sprawdzania rozwiązań zawodników użyto 12 testów ROW0.IN–ROW11.IN.

- ROW0.IN — test z treści zadania.
- ROW1.IN — dwa równania, jedno z nich nie zawiera żadnych zmiennych.
- ROW2.IN — dwa równania, jedno z nich nie ma rozwiązania.
- ROW3.IN — graf zależności składa się z trzech drzew i jednego cyklu.

- ROW4.IN — graf zależności składa się z dwóch składowych: cyklu 4-wierzchołkowego i dwóch cykli połączonych wspólnym wierzchołkiem.
- ROW5.IN — graf zależności posiada 10000 spójnych składowych; liczba rozwiązań znacznie (!!!) przekracza zasięg standardowych typów całkowitoliczbowych.
- ROW6.IN — w grafie zależności krawędzie można uzyskać na wiele różnych sposobów.
- ROW7.IN — dwa równania, dla jednego z nich graf zależności składa się z jednej spójnej składowej, w której wierzchołki "0" i "1" leżą daleko od siebie.
- ROW8.IN — graf zależności zawiera 125 składowych — 124 z nich to pojedyncze krawędzie, a 125-ta ma 3000 wierzchołków i około 6000 krawędzi.
- ROW9.IN, ROW10.IN — testy wydajnościowe o dużych rozmiarach.
- ROW11.IN — dwa równania, w pierwszym z nich długości rozwinięć lewej i prawej strony są różne.

Wojciech Guzicki
autor zadania, opracowanie

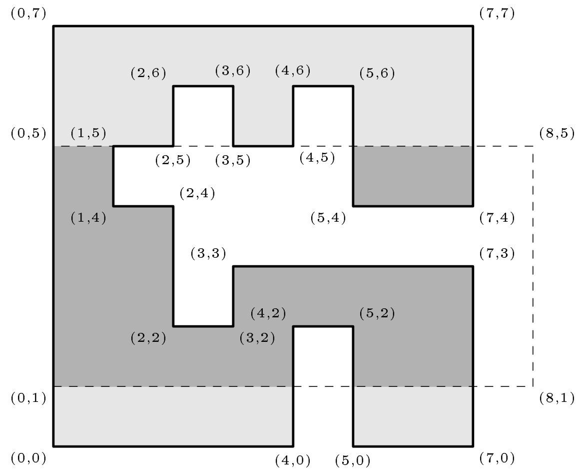
Grzegorz Jakacki
program wzorcowy

Okno

W prostokątnym układzie współrzędnych dany jest wielokąt o bokach równoległych do osi układu. Każde dwa kolejne boki w tym wielokącie są do siebie prostopadłe, a każdy wierzchołek ma współrzędne całkowitoliczbowe. Dane jest również okno, czyli prostokąt o bokach równoległych do osi układu. Wnętrze wielokąta (ale nie jego brzeg) zostało pokolorowane na czerwono. Jaka jest liczba rozłącznych czerwonych fragmentów wielokąta widzianych przez okno?

Przykład

Przyjrzyjmy się rysunkowi poniżej:



Liczba parami rozłącznych fragmentów wielokąta widzianych przez okno wynosi 2.

Zadanie

Napisz program, który: wczytuje opisy okna i wielokąta z pliku tekstowego OKN.IN; oblicza liczbę parami rozłącznych czerwonych fragmentów wielokąta widzianych przez okno; zapisuje wynik w pliku tekstowym OKN.OUT.

Wejście

W pierwszym wierszu pliku wejściowego OKN.IN znajdują się cztery liczby całkowite x_1, y_1, x_2, y_2 z przedziału [0..10000] oddzielone pojedynczymi odstępami.
Liczby x_1, y_1 są współrzędnymi górnego-lewego rogu okna. Liczby x_2, y_2 są współrzędnymi dolnego-prawego rogu okna.

Następny wiersz pliku wejściowego zawiera liczbę całkowitą n, $4 \leqslant n \leqslant 5000$. Jest to liczba wierzchołków wielokąta.
W kolejnych n wierszach znajdują się współrzędne kolejnych wierzchołków wielokąta danych w kierunku przeciwnym do ruchu wskazówek zegara, tzn. wnętrze wielokąta leży po lewej stronie jego brzegu, gdy przesuwamy się wzdłuż boków wielokąta zgodnie z zadanym porządkiem.
Każdy wiersz zawiera dwie liczby całkowite x, y oddzielone pojedynczym odstępem, $0 \leqslant x \leqslant 10000$, $0 \leqslant y \leqslant 10000$. Liczby w wierszu $i+2$, $1 \leqslant i \leqslant n$, w pliku OKN.IN są współrzędnymi i-tego wierzchołka wielokąta.

Wyjście

Twój program powinien zapisać w pierwszym i jedynym wierszu pliku wyjściowego OKN.OUT jedną liczbę całkowitą, a mianowicie liczbę rozłącznych czerwonych fragmentów wielokąta widzianych przez okno.

Przykład

Dla pliku wejściowego OKN.IN:

```
0 5 8 1
24
0 0
4 0
4 2
5 2
5 0
7 0
7 3
3 3
3 2
2 2
2 4
1 4
1 5
2 5
2 6
3 6
3 5
4 5
4 6
5 6
5 4
7 4
7 7
0 7
```

poprawnym rozwiązaniem jest plik wyjściowy OKN.OUT:

```
2
```

UZUPEŁNIENIE TREŚCI ZADANIA

Podczas trwania zawodów treść zadania uzupełniono następującym stwierdzeniem:

Brzeg wielokąta jest łamaną zwykłą zamkniętą, co oznacza, że każdy wierzchołek należy do dokładnie dwóch (kolejnych) boków, a każdy inny punkt łamanej należy do dokładnie jednego boku.

ROZWIĄZANIE

Zadanie *Okno* jest wzorowane na problemach często występujących w grafice komputerowej. Przypuśćmy, że w przestrzeni mamy daną pewną liczbę brył, których rzut musimy narysować. Istnieją stosunkowo proste algorytmy pozwalające obliczyć współrzędne rzutów końców wszystkich odcinków, które mamy narysować. Następnie rysujemy odcinki o końcach w tych wyznaczonych punktach. Możemy jednak zażądać, by nasze pole widzenia było w pewien sztuczny sposób ograniczone, na przykład, gdy rysujemy tylko to, co widać z okna. Musimy więc narysować tylko fragment tego, co dotychczas, mianowicie ten fragment, który mieści się w pewnym ograniczonym fragmencie płaszczyzny. Najczęściej tym ograniczonym fragmentem płaszczyzny jest prostokąt. Nasze zadanie ma podobny charakter: należy rozpoznać widoczne fragmenty większej całości. Oczywiście to zadanie nabrałoby więcej realizmu, gdyby odpowiedzią miał być ciąg list wierzchołków widocznych w oknie wielokątów. Utrudniłoby to jednak i tak dość zawiłą implementację rozwiązania, a samo rozwiązanie nie byłoby jednoznaczne. Dlatego ograniczono się tylko do żądania wskazania liczby widocznych wielokątów.

Algorytm rozwiązania jest bardzo prosty. Spróbujmy sobie wyobrazić „ręczne” rozwiązanie, nie zastanawiając się na razie nad sposobem implementacji. Przyjmijmy, że powierzchnia rysunku jest tak duża, że nie możemy ogarnąć jej wzrokiem. Na przykład, wielokąt i okno są narysowane na dużym boisku, na które nie możemy spojrzeć z góry. Będziemy zliczać wielokąty zawarte w oknie spacerując po obwodach wielokątów widocznych w oknie. Zaczniemy od znalezienia jakiegokolwiek wierzchołka naszego dużego wielokąta z treści zadania, leżącego na zewnątrz okna. Jeśli takiego wierzchołka nie ma, to cały wielokąt znajduje się wewnątrz okna i jest on widoczny w całości: odpowiedzią jest 1, bo widoczny jest jeden wielokąt. Rozpoznanie tego przypadku jest łatwe: każdy napotkany wierzchołek leży wewnątrz okna, a to można stwierdzić dość łatwo. Przypuśćmy więc, że znaleźliśmy wierzchołek leżący na zewnątrz okna. Rozpoczynamy wędrówkę w kierunku zgodnym z podanym w zadaniu (tzn. w kierunku przeciwnym do ruchu wskazówek zegara, czyli w taki sposób, by wnętrze wielokąta leżało po naszej lewej stronie). Wędrujemy dotąd, aż dojdziemy do krawędzi okna. Oczywiście może się zdarzyć, że nasza droga nigdy nie przetnie krawędzi okna. Jest to możliwe w jednym z trzech przypadków:

(1) cały wielokąt leży wewnątrz okna i odpowiedzią jest 1;

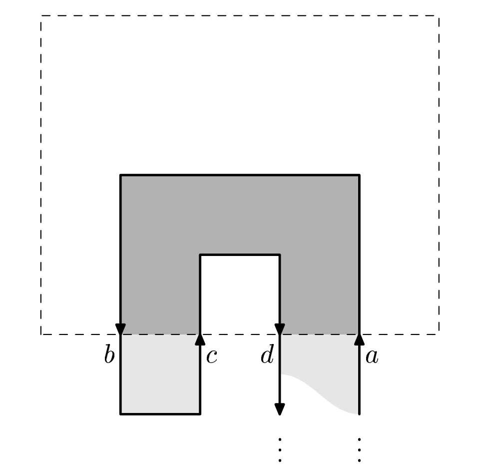
(2) całe okno leży wewnątrz wielokąta i odpowiedzią jest 1;

(3) wnętrze okna jest rozłączne z wnętrzem wielokąta i odpowiedzią jest 0.

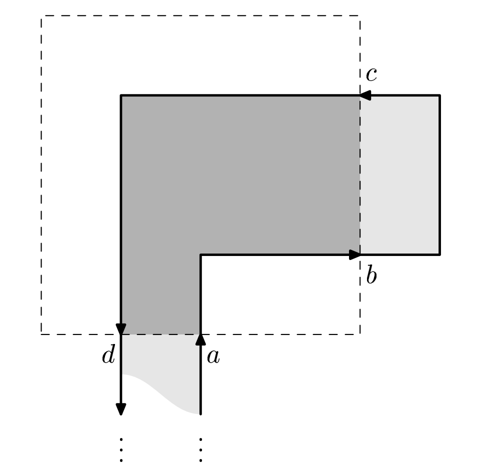
Kwestią, jak rozróżnić te przypadki, zajmiemy się później.

Tak więc kontynuujemy naszą wędrówkę dotąd, aż dojdziemy do granicy okna **i przekroczymy ją**. Zauważmy, że nie musimy przejmować się sytuacjami, w których dochodzimy do granicy okna, następnie skręcamy i podążamy wzdłuż tej granicy i jeszcze raz skręcamy w tę stronę, z której przyszliśmy (tzn. wracamy na zewnątrz okna, jeśli doszliśmy do granicy okna z zewnątrz lub wracamy do wewnątrz, jeśli przyszliśmy z wnętrza okna). Wynika to stąd, że mamy zliczać te wielokąty, których wnętrza są zawarte w oknie. Przekraczamy więc granicę okna i wchodzimy do jego wnętrza. Teraz podążamy wzdłuż obwodu wielokąta do momentu, w którym przekroczymy granicę okna próbując wyjść na zewnątrz. Taki moment oczywiście musi nastąpić: zaczęliśmy[[[zaczeliśmy?]]] wędrówkę na zewnątrz, jesteśmy wewnątrz, więc musimy kiedyś znaleźć się znów na zewnątrz. Teraz nie opuszczamy okna (a więc porzucamy naszą dotychczasową strategię polegającą na obchodzeniu obwodu całego wielokąta, natomiast chodzimy po obwodach wielokątów widocznych w oknie), ale skręcamy w lewo (by nadal mieć wielokąt po naszej lewej stronie) i podążamy wzdłuż granicy okna do **najbliższego** punktu, w którym znów droga wzdłuż obwodu wielokąta wchodzi do wielokąta z zewnątrz. W tym punkcie powtarzamy poprzednie postępowanie: idziemy wewnątrz okna do najbliższego punktu wyjścia z wielokąta, skręcamy w lewo i znów idziemy do **najbliższego** punktu wejścia itd. Oczywiście kiedyś to postępowanie się zakończy: dojdziemy do pierwszego punktu wejścia do okna. Wtedy nasza wędrówka wokół jednego widocznego fragmentu się kończy i (pamiętaną) liczbę widocznych fragmentów aktualizujemy (tzn. zwiększamy o 1). Przykładowe sytuacje są pokazane na rysunkach 1 i 2. W każdym przypadku wchodzimy do okna w punkcie a, podążamy wewnątrz okna do punktu b, skręcamy w lewo i podążamy wdłuż granicy okna do punktu c, znów wchodzimy do okna i wewnątrz niego dochodzimy do punktu d, ponownie skręcamy w lewo i powracamy do punktu a. Widzimy, że nie ma znaczenia, czy wewnątrz okna skręcimy tak, że wyjdziemy „na lewo”, czy „na prawo” od punktu a.

Rys. 1



Rys. 2



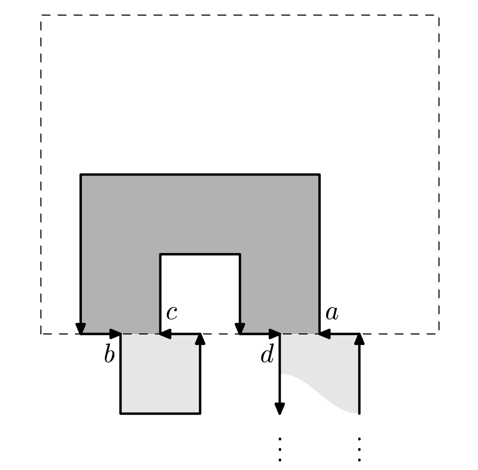
Teraz szukamy następnego punktu wejścia do okna i od niego zaczynamy tę samą procedurę. Robimy to dotąd, aż wyczerpiemy wszystkie punkty wejścia do okna i otrzymaną liczbę widocznych wielokątów podajemy jako wynik. Następnego wejścia do okna możemy szukać na różne sposoby. Jednym z nich jest obejście najpierw całego obwodu i zapamiętanie wszystkich punktów wejścia. W trakcie opisanej wyżej procedury usuwamy z listy punktów wejścia wszystkie te punkty, które wykorzystaliśmy. Po zamknięciu cyklu bierzemy pierwszy na liście punkt wejścia i od niego zaczynamy podobną wędrówkę.

Teraz już łatwo możemy się domyślić, w jaki sposób zaimplementować ten algorytm. Po pierwsze, odczytując z pliku wejściowego współrzędne wierzchołków wielokąta znajdujemy wszystkie punkty, w których przekraczamy granicę okna. Te punkty dzielimy na wejściowe (tzn. te, w których wchodzimy z zewnątrz do wnętrza wielokąta) i wyjściowe (gdy przekraczamy granicę okna w drugą stronę). Następnie tworzymy listę takich punktów wejścia i wyjścia. Dla punktu wejścia do okna następnym po nim na liście ma być odpowiedni punkt wyjścia (tzn. ten punkt wyjścia, który osiągniemy poruszając się wzdłuż obwodu wielokąta wewnątrz okna: dla punktów a i c na rysunkach 1 i 2 będą to odpowiednio punkty b i d). Dla punktu wyjścia następnym na liście ma być **najbliższy**, licząc w kierunku przeciwnym do ruchu wskazówek zegara, punkt wejścia do okna (na rysunkach 1 i 2 dla punktów b i d będą to odpowiednio punkty c i a). Widać więc, że wszystkie punkty wejścia musimy posortować (w programie wzorcowym każdemu punktowi na obwodzie okna przypisujemy *wagę* w taki sposób, by wagi punktów zwiększały się przy obchodzeniu obwodu okna w kierunku przeciwnym do ruchu wskazówek zegara, a następnie sortujemy punkty względem tego atrybutu). Pewnej staranności wymaga wybór punktu przekroczenia granicy okna, gdy przez pewien czas poruszamy się wzdłuż tej granicy; porównaj rysunki 1 i 2 z odpowiednimi rysunkami 3 i 4. Należy zauważyć, że w punktach b i d już nie skręcamy w lewo — po prostu przechodzimy do następnego punktu wejścia, nie przejmując się nawet tym, że pewne odcinki przejdziemy dwukrotnie (np. z punktu d do a na rysunku 4). Ważne są tylko same punkty wejścia i wyjścia oraz ich uporządkowanie na obwodzie okna.

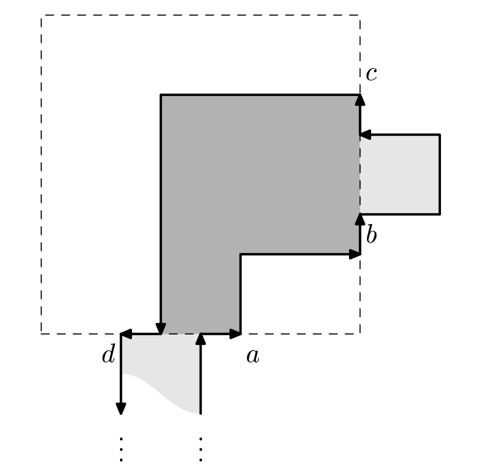
Ostatnim problemem implementacyjnym jest kwestia rozróżnienia sytuacji (1), (2) i (3) w przypadku, gdy obwód wielokąta nie przecina obwodu okna. Zauważmy, że jeżeli obwód wielokąta nie przecina obwodu okna, to stwierdzenie, czy wielokąt znajduje się w całości w oknie, sprowadza się do sprawdzenia, czy dowolny jego punkt (np. „pierwszy" punkt obwodu wielokąta) leży w oknie. To pozwala nam stwierdzić, czy mamy do czynienia z przypadkiem (1). Jeżeli jednak okaże się, że punkt obwodu wielokąta nie leży wewnątrz okna, to musimy jeszcze rozstrzygnąć pomiędzy przypadkiem (2) i (3). W tym celu wystarczy wybrać dowolny punkt okna (na przykład lewy dolny róg) i sprawdzić, czy znajduje się on wewnątrz wielokąta. Jak to zrobić? Bierzemy prostą pionową przechodzącą przez ten róg okna i sprawdzamy, czy poniżej rogu przecięła ona obwód wielokąta. Jeśli nie, to róg okna leży na zewnątrz wielokąta. Jeśli tak, to możemy postąpić dwojako: albo policzyć, ile razy ta prosta przecięła obwód wielokąta (róg okna leży wewnątrz wielokąta wtedy i tylko wtedy, gdy obwód wielokąta przetniemy nieparzystą liczbę razy), albo nieco prościej, wybrać najwyższy punkt przecięcia (tzn. leżący najbliżej rogu) i sprawdzić, w którym kierunku obwód wielokąta przebija tę prostą. Szczegóły tego rozwiązania pozostawiamy Czytelnikowi.

Warto jeszcze wspomnieć o wyborze metody sortowania. Może się zdarzyć, że ob-

Rys. 3



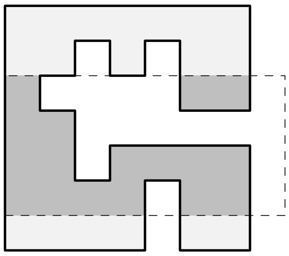
Rys. 4



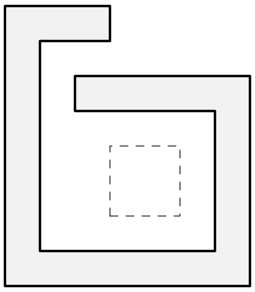
wód wielokąta przecina brzeg okna w bardzo wielu punktach (nawet powyżej tysiąca). Wtedy od wyboru metody sortowania zależy w głównej mierze szybkość działania programu. W programie wzorcowym wybrano metodę sortowania szybkiego (ang. *quicksort*).

TESTY

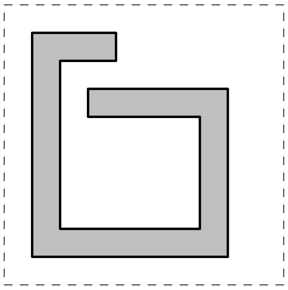
Rozwiązania zawodników oceniano przy pomocy zestawu 16-tu testów: PRO0.IN––PRO15.IN. Test PRO0.IN był testem z treści zadania. W teście PRO1.IN wielokąt był w całości poza oknem, w teście PRO2.IN w całości wewnątrz okna, w teście PRO3.IN pokrywał się z oknem. Testy PRO4.IN i PRO5.IN miały na celu sprawdzenie, czy program radzi sobie z sytuacjami, w których obwód wielokąta częściowo pokrywa się z brzegiem okna. Test PRO6.IN to długi meander przecięty oknem (21 widocznych kawałków). Test PRO7.IN miał sprawdzić poprawność programu na małym przykładzie, zaś testy PRO8.IN i PRO9.IN na przykładzie dwóch „spiral”, mniejszej i większej. Testy PRO10.IN i PRO11.IN wykorzystywały „spirale” podwójnie skręcone, stosunkowo niewielkich rozmiarów (współrzędne wierzchołków nie przekraczały 240). Wielokąt w teście PRO12.IN miał ten sam kształt, co w teście PRO10.IN, ale był dziesięciokrotnie powiększony. Testy PRO13.IN, PRO14.IN i PRO15.IN miały maksymalne rozmiary danych i były to dwie spirale i jeden meander. Na kolejnych stronach przedstawiono graficzne schematy niektórych testów.



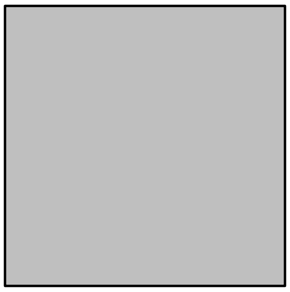
OKN0.IN



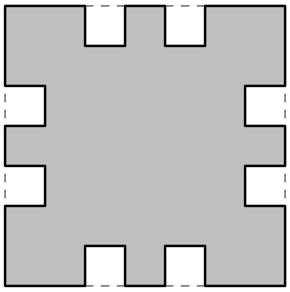
OKN1.IN



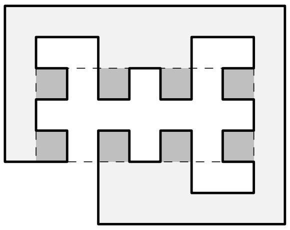
OKN2.IN



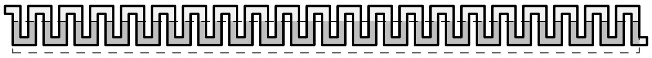
OKN3.IN



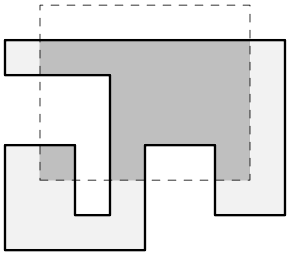
OKN4.IN



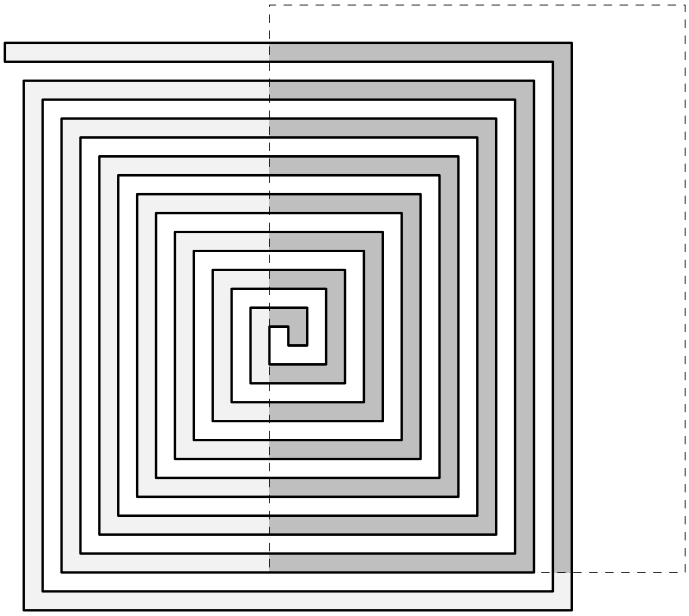
OKN5.IN



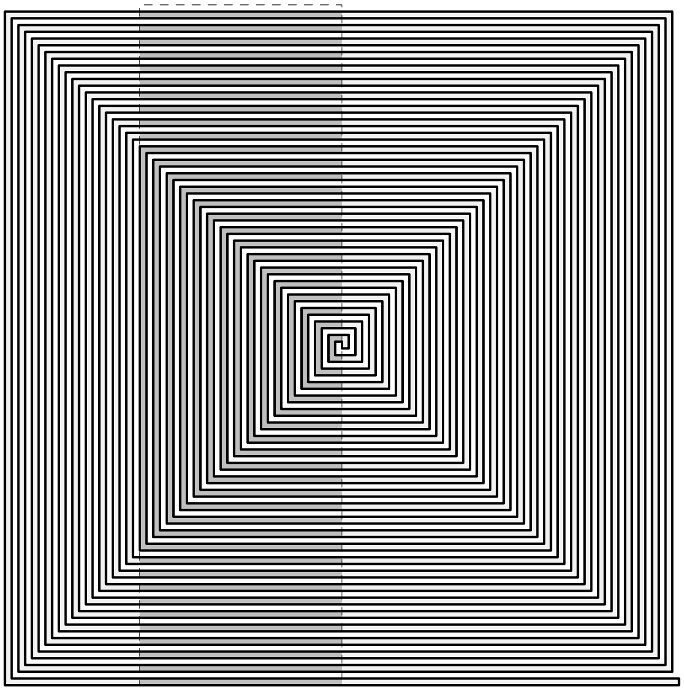
OKN6.IN



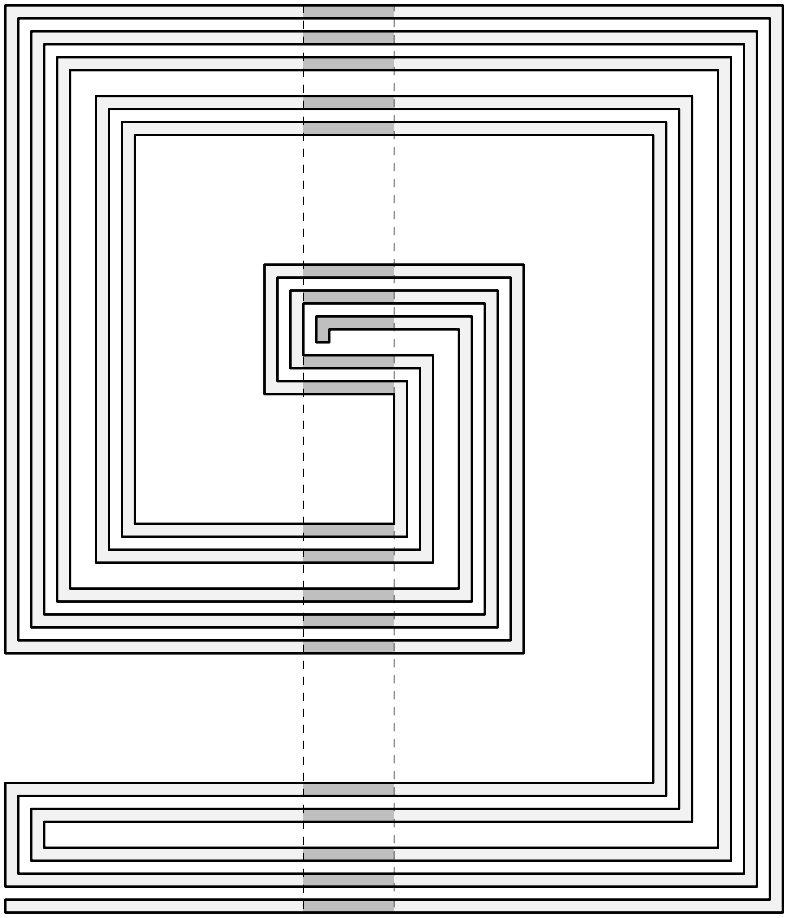
OKN7.IN



OKN8.IN



OKN9.IN



OKN10.IN

Autor nieznany **Tomasz Błaszczyk**
program wzorcowy, opracowanie

Płetwonurek

Płetwonurek do nurkowania używa butli, w której są dwa zbiorniki: z tlenem i z azotem. W zależności od czasu przebywania pod wodą i głębokości nurek potrzebuje różnych ilości tlenu i azotu. Płetwonurek ma do dyspozycji pewną liczbę butli. Każda butla charakteryzuje się wagą oraz objętością zawartego w niej tlenu i azotu. Do wykonania zadania nurek potrzebuje określonych ilości tlenu i azotu. Jaka jest najmniejsza sumaryczna waga butli, które nurek musi zabrać ze sobą, żeby mógł wykonać zadanie?

Przykład

Nurek ma do dyspozycji 5 butli o następujących charakterystykach (odpowiednio: objętość tlenu w litrach, objętość azotu w litrach, waga butli w dekagramach):

```
3 36 120
10 25 129
5 50 250
1 45 130
4 20 119
```

Jeżeli do wykonania zadania nurek potrzebuje 5 litrów tlenu i 60 litrów azotu, to musi zabrać ze sobą dwie butle o łącznej wadze 249, np. pierwszą i drugą lub czwartą i piątą.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego PLE.IN zapotrzebowanie nurka na tlen i azot, liczbę dostępnych butli oraz ich charakterystyki;*
- *oblicza najmniejszą sumaryczną wagę butli, które nurek musi zabrać ze sobą, żeby wykonać zadanie;*
- *zapisuje wynik w pliku tekstowym PLE.OUT.*

Uwaga: dany zestaw butli zawsze gwarantuje wykonanie zadania.

Wejście

W pierwszym wierszu pliku wejściowego PLE.IN znajdują się dwie liczby całkowite t i a oddzielone pojedynczym odstępem, $1 \leqslant t \leqslant 21$ i $1 \leqslant a \leqslant 79$. Są to, odpowiednio, ilości tlenu i azotu potrzebne do wykonania zadania.
Drugi wiersz pliku wejściowego zawiera tylko jedną liczbę n, $1 \leqslant n \leqslant 1000$. Jest to liczba dostępnych butli.
Kolejne n wierszy zawiera charakterystyki butli.
Wiersz $i + 2$ pliku PLE.IN zawiera trzy liczby całkowite t_i, a_i, w_i, pooddzielane pojedynczymi odstępami ($1 \leqslant t_i \leqslant t$, $1 \leqslant a_i \leqslant a$, $1 \leqslant w_i \leqslant 800$). Są to kolejno: objętości tlenu i azotu w i-tej butli oraz waga tej butli.

Wyjście

Twój program powinien zapisać jedną liczbę całkowitą w pierwszym i jedynym wierszu pliku wyjściowego PLE.OUT. Tą liczbą powinna być najmniejsza sumaryczna waga butli, które nurek musi zabrać ze sobą, żeby mógł wykonać zadanie.

Przykład

Dla pliku tekstowego PLE.IN:

```
5 60
5
3 36 120
10 25 129
5 50 250
1 45 130
4 20 119
```

poprawnym rozwiązaniem jest plik wyjściowy PLE.OUT:

```
249
```

ZMIANA W TREŚCI ZADANIA

Podczas trwania zawodów zostały zmienione ograniczenia: zamiast $1 \le t_i \le t$ i $1 \le a_i \le a$ wprowadzono $1 \le t_i \le 21$ i $1 \le a_i \le 79$.

ROZWIĄZANIE

Przy omawianiu rozwiązania zostaną użyte następujące oznaczenia:

- n — liczba butli,
- t, a — zapotrzebowanie na, odpowiednio, tlen i azot,
- $t[k]$, $a[k]$, $w[k]$ — liczba jednostek tlenu i azotu w k-tej butli oraz waga tej butli.

Zadanie można rozwiązać metodą programowania dynamicznego. Rozwiązanie zadania jest analogiczne do rozwiązania problemu plecakowego dla plecaka o ograniczonej pojemności (zob. [12], str. 381–383), rozszerzonego na przypadek dwuwymiarowy.

Algorytm działa w n fazach. W k-tej fazie rozwiązywane jest zadanie dla k pierwszych butli, na podstawie rozwiązania znalezionego w fazie poprzedniej, tj. dla $k-1$ pierwszych butli. Rozwiązania częściowe są przechowywane w tablicy T. Oto schemat algorytmu:

```
var
   T : array [0..t, 0..a] of word;
   ti, ai, td, ad, k : word;
begin
   T := [+∞, ..., +∞];
   T[0,0] := 0;
```

```
 7:   for k := 1 to n do
 8:     for ti := t downto 0 do
 9:       for ai := a downto 0 do begin
10:         td := min(t, ti + t[k]);
11:         ad := min(a, ai + a[k]);
12:         T[td, ad] :=min(T[td, ad], T[ti, ai] + w[k])
13:       end;
14:   write(T[t, a])
15: end
```

Przez cały czas działania algorytmu spełniona jest następująca zależność (tzw. niezmiennik):

> *Po zakończeniu k-tej fazy każdy element $T[t', a']$ zawiera minimalną wagę zestawu butli wybranego spośród k pierwszych butli, w którym liczba jednostek tlenu wynosi:*
>
> $$\begin{cases} \textit{dokładnie } t' & \textit{gdy } t' < t \\ \textit{przynajmniej } t' & \textit{gdy } t' = t, \end{cases}$$
>
> *natomiast liczba jednostek azotu wynosi:*
>
> $$\begin{cases} \textit{dokładnie } a' & \textit{gdy } a' < a \\ \textit{przynajmniej } a' & \textit{gdy } a' = a. \end{cases}$$

Umawiamy się, że $+\infty$ oznacza, iż nie istnieje zestaw butli spełniający te warunki. Przyjmujemy także, że „po wykonaniu zerowej fazy” oznacza „przed wejściem do pętli”.

Przed wykonaniem do wiersza nr 10 podtablica $T[0..(t_i + t[k] - 1),0..a]$ oraz fragment wiersza $T[t_i + t[k], 0..a_i + a[k]]$ zawierają rozwiązanie dla fazy $(k - 1)$-wszej, natomiast pozostała część tablicy T — dla fazy k-tej.

W wierszach 10 i 11 obliczane są liczby jednostek tlenu i azotu, jakie można uzyskać dodając aktualnie rozpatrywaną butlę do zestawu umożliwiającego uzyskanie ti litrów tlenu i ai litrów azotu.

W wierszu 12 waga najlepszego znalezionego zestawu pozwalającego uzyskać td litrów tlenu i ad litrów azotu jest modyfikowana, jeżeli właśnie znaleziono zestaw o mniejszej wadze, niż najlżejszy znaleziony do tej pory. Stąd, po wykonaniu przypisania w wierszu 12, część tablicy zawierająca rozwiązanie dla fazy k-tej powiększa się o element $T[td, ad]$.

W algorytmie ważna jest kolejność przeglądania tablicy (wiersze 4 i 5). Wykazanie tego faktu pozostawiamy jako ćwiczenie dla Czytelnika.

Złożoność czasowa powyższego algorytmu wynosi $\Theta(tan)$, natomiast pamięciowa — $\Theta(ta)$.

Program wzorcowy jest bezpośrednią implementacją przedstawionego algorytmu. Zauważmy, że z podanych w zadaniu ograniczeń wynika, że waga zestawu butli może osiągnąć $79 \cdot 800 = 63200$ dekagramów, dlatego w Turbo Pascalu zamiast typu **integer** należy użyć typu **longint**.

TESTY

Do sprawdzenia rozwiązań zawodników użyto 10-ciu testów.

- PLE0.IN — test z treści zadania.
- PLE1.IN — minimalne wymagania, jedna butla.
- PLE2.IN — test poprawnościowy, wartości zmiennych przekraczają zakres typu **integer**.
- PLE3.IN — mały test poprawnościowy.
- PLE4.IN — test faworyzujący rozwiązania heurystyczne, wystarcza jedna butla o wadze 101.
- PLE5.IN — test poprawnościowy; dwie grupy butli, opłaca się wybierać tylko z drugiej.
- PLE6.IN–PLE9.IN — różne testy wydajnościowe.

Zawody III stopnia

opracowania zadań

Marcin Kubica — autor zadania, opracowanie
Krzysztof Sobusiak — program wzorcowy

Układy asemblerowe

Firma Bajtel postanowiła usprawnić produkowane przez nią komputery, przez zastąpienie zaszytych w komputerach programów asemblerowych specjalnymi układami elektronicznymi, zwanymi układami asemblerowymi. Programy asemblerowe składają się wyłącznie z przypisań. Każde przypisanie jest określone przez cztery elementy:

- *dwa rejestry, z których pobierane są dane,*
- *operację elementarną, wykonywaną na tych danych,*
- *rejestr, w którym umieszczany jest wynik.*

Zakładamy, że jest co najwyżej 26 rejestrów oznaczonych małymi literami alfabetu angielskiego od `a` *do* `z` *oraz są co najwyżej 4 różne operacje elementarne oznaczone wielkimi literami angielskiego alfabetu od* `A` *do* `D`*.*

Układ asemblerowy ma:

- *wejścia oznaczone nazwami rejestrów, na każde wejście podawana jest wartość początkowa odpowiedniego rejestru, oraz*
- *wyjścia oznaczone również nazwami rejestrów, których wartości końcowe są kierowane do tych rejestrów.*

Wewnątrz układu znajdują się bramki. Każda bramka ma dwa wejścia i jedno wyjście. Wykonuje ona jedną operację elementarną na danych, które pojawiają się na jej wejściach i udostępnia wynik na swoim wyjściu. Wejścia bramek oraz wyjścia całego układu są połączone z wyjściami innych bramek lub wejściami układu. Wyjścia bramek oraz wejścia układu mogą być połączone z wieloma wejściami innych bramek lub wyjściami układu. Połączenia między bramkami nie tworzą cykli.

Układ asemblerowy jest równoważny programowi asemblerowemu, gdy dla dowolnego stanu początkowego rejestrów daje taki sam — jak program — stan końcowy rejestrów.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego UKL.IN opis programu asemblerowego;*
- *oblicza najmniejszą liczbę bramek, które musi zawierać układ asemblerowy równoważny z danym programem;*
- *zapisuje wynik w pliku tekstowym UKL.OUT.*

Wejście

W pierwszym wierszu pliku wejściowego UKL.IN znajduje się jedna liczba całkowita n ($1 \leqslant n \leqslant 1000$). Jest to liczba instrukcji programu.

W n kolejnych wierszach opisane są kolejne instrukcje programu. Każdy opis ma postać czteroliterowego słowa, zaczynającego się od symbolu operacji elementarnej: `A`, `B`, `C` *albo* `D`. *Pozostałe trzy litery są małe. Druga i trzecia — to nazwy rejestrów, z których należy pobrać dane; czwarta — to nazwa rejestru, w którym należy umieścić wynik.*

Wyjście

Twój program powinien zapisać w pierwszym i jedynym wierszu pliku wyjściowego UKL.OUT jedną liczbę całkowitą — minimalną liczbę bramek układu asemblerowego, równoważnego danemu programowi.

Przykład

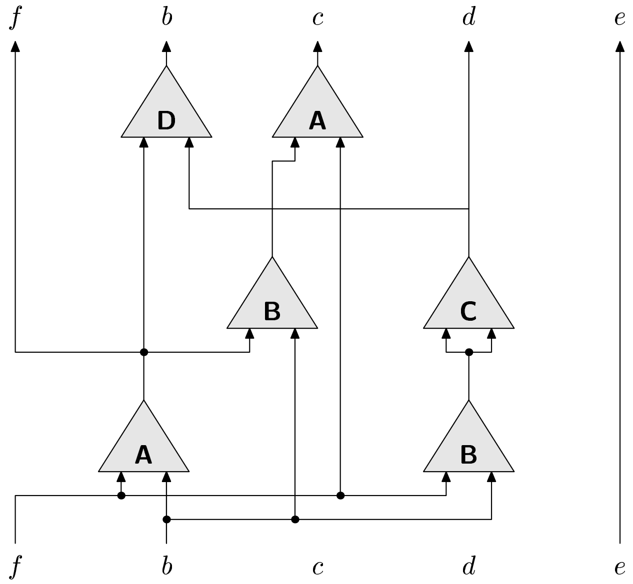
Dla pliku tekstowego UKL.IN:

```
8
Afbc
Bfbd
Cddd
Bcbc
Afcc
Afbf
Cfbb
Dfdb
```

poprawnym rozwiązaniem jest plik wyjściowy UKL.OUT:

```
6
```

Układ równoważny danemu programowi przedstawiono na rysunku.



ROZWIĄZANIE

Zadanie to jest adaptacją problemu występującego przy optymalizacji kodu generowanego przez kompilatory, zwanego numerowaniem wartości (zob. [6]). Problem ten polega na takim przeorganizowaniu ciągu instrukcji, aby pozbyć się tych, które niepotrzebnie się powtarzają oraz tych, których wyniki są nieistotne.

Rozwiązanie polega na zbudowaniu grafu, którego wierzchołki reprezentują wartości pojawiające się w trakcie obliczeń. Są to wierzchołki reprezentujące początkowe wartości rejestrów oraz „bramki" — wierzchołki reprezentujące wyniki operacji elementarnych. Dla każdej bramki pamiętamy kod operacji elementarnej wykonywanej przez bramkę oraz dwie krawędzie prowadzące od wierzchołków reprezentujących argumenty operacji do bramki. Krawędzie prowadzące do bramki są pamiętane tak, aby było wiadomo, która reprezentuje pierwszy, a która drugi argument operacji.

Graf budujemy na bieżąco, podczas wczytywania danych. W danej chwili pamiętamy też, które wierzchołki reprezentują wartości rejestrów. Początkowo graf składa się wyłącznie z wierzchołków reprezentujących inicjalne wartości rejestrów. Dla każdej kolejnej instrukcji or_1r_2r sprawdzamy, czy w budowanym grafie istnieje bramka o kodzie operacji o, do której prowadzą krawędzie, odpowiednio z wierzchołków reprezentujących wartości rejestrów r_1 i r_2. Jeśli taka bramka nie istnieje, to dodajemy ją do grafu. Następnie przyjmujemy, że bramka ta reprezentuje wartość rejestru r. Zauważmy, że po utworzeniu grafu instrukcje dające te same wyniki są reprezentowane przez jedną bramkę. Jednocześnie wiadomo, które wierzchołki grafu reprezentują końcowe wartości rejestrów.

Tak utworzony graf może jednak zawierać bramki, które nie mają wpływu na końcowe wartości rejestrów. Aby wyznaczyć, które bramki są potrzebne, przechodzimy po krawędziach grafu wstecz, począwszy od wierzchołków reprezentujących końcowe wartości rejestrów. Jednocześnie zliczamy odwiedzane bramki. Bramki, które nie zostaną odwiedzone, nie mają wpływu na końcowe wartości rejestrów. Tak więc liczba odwiedzonych bramek jest szukanym wynikiem.

Dla efektywności algorytmu kluczowy jest sposób reprezentacji danych oraz sprawdzania, czy w grafie jest bramka o zadanym kodzie operacji, do której prowadzą krawędzie z zadanych wierzchołków. Najprostsza implementacja polega na każdorazowym przeglądaniu wszystkich bramek w grafie. Wówczas pierwsza faza algorytmu — budowa grafu — ma koszt $\Theta(n^2)$ (gdzie n, to liczba instrukcji). Lepszy koszt można uzyskać pamiętając opisy bramek (złożone z kodu operacji oraz wierzchołków reprezentujących argumenty) w słowniku o logarytmicznym czasie dostępu (np. w drzewie AVL, por. [9]). Wówczas koszt pierwszej fazy algorytmu wynosi $\Theta(n \log n)$. Implementacja takiej struktury danych jest jednak dosyć pracochłonna. Kolejną możliwością jest pamiętanie opisów bramek w tablicy mieszającej. W tym ostatnim przypadku, z uwagi na fakt, że liczba bramek jest stosunkowo niewielka, łatwo uzyskać koszt oczekiwany $\Theta(n)$.

Koszt drugiej fazy algorytmu, przy zastosowaniu np. przeszukiwania grafu w głąb, jest liniowy ze względu na liczbę wierzchołków w grafie. Tak więc koszt pierwszej fazy dominuje nad drugą.

Wśród rozwiązań przedstawionych przez zawodników kolejność opisanych faz była czasem odwrócona. Korzyścią płynącą z takiego rozwiązania jest obniżenie kosztu fazy utożsamiającej operacje, które mogą być reprezentowane przez te same bramki (jest to istotne, gdy ma ona koszt $\Theta(n^2)$). Jednak praktycznie oznacza to dwukrotną konstrukcję grafu. W rezultacie, gdy liczba instrukcji nie przekracza 1000, taka zamiana kolejności faz oznacza nieznaczne spowolnienie programu.

Okazuje się, że dla danych z takiego zakresu jak w treści zadania, wszystkie opisane rozwiązania działają praktycznie tak samo szybko. Biorąc pod uwagę to spostrzeżenie, a także fakt, że zadanie to było zadaniem rozgrzewkowym, przy ocenie rozwiązań przyjęto za punkt odniesienia rozwiązanie o koszcie $\Theta(n^2)$. Wszystkie sprawdzane rozwiązania, które uzyskały 100 pkt. działały w czasie 0,3–0,4 s, bez względu na rozmiar testów. Widać więc, że czas uruchomienia programu oraz wczytywania danych jest istotnie większy niż czas trwania obliczeń.

Przedstawimy teraz bardziej szczegółowo rozwiązanie o koszcie $\Theta(n^2)$. Konstruowany graf możemy reprezentować pamiętając opisy bramek (w tablicy *Bramki*). Każdy taki opis składa się z kodu operacji elementarnej `op` oraz opisów `a`, `b` krawędzi prowadzących do bramki. Krawędzie reprezentujemy za pomocą liczb całkowitych — liczby dodatnie oznaczają pozycje w tablicy bramek, od których prowadzą krawędzie, a liczby ujemne oznaczają początkowe wartości poszczególnych rejestrów. Podobnie reprezentujemy wartości rejestrów w trakcie konstrukcji grafu (w tablicy *Rejestry*). Biorąc pod uwagę przeszukiwanie grafu w drugiej fazie algorytmu, w opisie bramki pamiętamy również (pole *odw*), czy dana bramka została odwiedzona. Liczbę bramek pamiętamy na zmiennej *nb*.

```
const MAXN = 1000;
type
  TBramka = record
  op : char;
  a, b : integer;
  odw : boolean
end;
var
  Rejestry : array ['a'..'z'] of integer;
  Bramki : array [1..MAXN] of TBramka;
  nb : word;
```

Wyszukanie bramki o zadanym kodzie operacji elementarnej oraz węzłach, z których prowadzą do niej krawędzie, jest realizowane przez poniższą funkcję. Wynikiem funkcji jest pozycja szukanej bramki w tablicy *Bramki*, a jeśli takiej bramki nie ma, to wynikiem jest 0.

```
function ZnajdźBramkę(lop : char; la, lb : integer) : word;
var
  i : word;
begin
  for i := 1 to nb do with Bramki[i] do
    if (op = lop) and (a = la) and (b = lb) then begin
      ZnajdźBramkę := i;
      exit
    end;
```

```
   ZnajdźBramkę := 0
end;
```

Uwzględnienie kolejnej instrukcji w trakcie budowy grafu może być realizowane przez następującą prostą procedurę:

```
procedure PrzetwórzInstrukcję(lop, ra, rb, lr : char);
var i : word;
begin
   i := ZnajdźBramkę(lop, Rejestry[ra], Rejestry[rb]);
   if i = 0 then begin
      inc(nb);
      with Bramki[nb] do begin
         op := lop;
         a := Rejestry[ra];
         b := Rejestry[rb];
         odw := false
      end;
      i := nb
   end;
   Rejestry[lr] := i
end;
```

Poniżej przedstawiamy jeszcze funkcję realizującą drugą część algorytmu. Chcąc obliczyć wynik należy zsumować wyniki tej funkcji dla wszystkich wierzchołków grafu reprezentujących końcowe wartości rejestrów.

```
function Przejdź(i : integer) : word;
begin
   if (i ⩽ 0) or (Bramki[i].odw) then Przejdz := 0
   else with Bramki[i] do begin
      odw := true;
      Przejdź := 1 + Przejdz(a) + Przejdz(b)
   end
end
```

Pełną treść programu można znaleźć na załączonej dyskietce.

TESTY

W poniższej tabelce zestawiono liczbowe charakterystyki wszystkich testów. Instrukcje „zdublowane”, to te, które są usuwane w pierwszej fazie algorytmu — w trakcie budowy grafu. Instrukcje „zbędne” to te, które nie są odwiedzane w drugiej fazie algorytmu.

Test	Liczba instrukcji wszystkich	Liczba instrukcji zdublowanych	Liczba instrukcji zbędnych	Liczba bramek
UKL0.IN	8	1	1	6
UKL1.IN	10	1	3	6
UKL2.IN	20	3	8	9
UKL3.IN	48	24	0	24
UKL4.IN	100	0	99	1
UKL5.IN	6	1	2	3
UKL6.IN	100	0	0	100
UKL7.IN	6	2	3	1
UKL8.IN	200	55	14	131
UKL9.IN	1000	743	175	82
UKL10.IN	1000	498	0	502
UKL11.IN	1000	0	0	1000
UKL12.IN	6	1	1	4
UKL13.IN	1000	999	0	1
UKL14.IN	6	1	1	4

Aby wychwycić te programy, które zawsze dawały odpowiedź jeden, albo zawsze wypisywały na wyjście liczbę wszystkich instrukcji, zgrupowano niektóre testy z prostymi testami zawierającymi 6 instrukcji. Rozwiązanie uzyskiwało punkty za grupę testów tylko wtedy, gdy było poprawne dla wszystkich testów z grupy. I tak, zgrupowano testy: 4 i 5, 6 i 7, 11 i 12, oraz 13 i 14. Poniżej przedstawiamy krótką charakterystykę ciekawszych testów.

- UKL3.IN — 4 warstwy, po 6 bramek w każdej warstwie, każdej bramce odpowiadają dokładnie dwie instrukcje.
- UKL4.IN — w każdej ze 100-u instrukcji wynik jest umieszczany w tym samym rejestrze (`d`), który nie jest argumentem żadnej instrukcji (zatem wszystkie instrukcje, poza ostatnią, są zbędne).
- UKL6.IN — instrukcja `Ahhh` powtarza się 100 razy (szukany układ bramek przypomina drabinkę o 100-u szczeblach).

Piotr Chrząstowski-Wachtel
autor zadania, opracowanie

Adam Borowski
program wzorcowy

Bankomaty

Każdy członek Bajtlandzkiej Kasy Pożyczkowej ma prawo pożyczyć dowolną sumę mniejszą niż 10^{30} bajtlandzkich dukatów, ale musi ją w całości zwrócić do Kasy nie później niż po upływie 7 dni. W sali obsługi klientów Kasy ustawiono 100 bankomatów ponumerowanych od 0 do 99. Każdy bankomat wykonuje tylko jedną operację: wypłaca albo przyjmuje ustaloną kwotę. Bankomat o numerze i wypłaca 2^i dukatów, jeśli i jest parzyste, zaś przyjmuje 2^i dukatów, jeśli i jest nieparzyste. Gdy klient zamierza wypożyczyć ustaloną kwotę, trzeba zbadać, czy będzie mógł ją pobrać, korzystając co najwyżej raz z każdego z bankomatów i jeśli tak, wyznaczyć numery bankomatów, z których należy skorzystać. Trzeba również zbadać, czy będzie mógł ją zwrócić w podobny sposób i jeśli tak, wyznaczyć numery bankomatów, z których należy skorzystać w celu wykonania tej operacji.

Przykład

Klient, który zamierza pożyczyć 7 dukatów, pobiera najpierw 16 dukatów w bankomacie nr 4 i 1 dukata w bankomacie nr 0, a następnie oddaje 8 dukatów w bankomacie nr 3 i 2 dukaty w bankomacie nr 1. Żeby zwrócić pożyczoną kwotę 7 dukatów, pobiera najpierw 1 dukata w bankomacie nr 0, a następnie oddaje 8 dukatów w bankomacie nr 3.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego BAN.IN liczbę klientów n i dla każdego klienta wysokość kwoty, jaką zamierza pożyczyć w Kasie;*
- *dla każdego klienta sprawdza, czy będzie mógł pobrać ustaloną kwotę korzystając co najwyżej raz z każdego bankomatu i jeśli tak, wyznacza numery bankomatów, z których należy skorzystać, oraz czy będzie mógł ją zwrócić w podobny sposób i jeśli tak, wyznacza numery bankomatów, z których należy skorzystać w tym celu;*
- *zapisuje wyniki w pliku tekstowym BAN.OUT.*

Wejście

W pierwszym wierszu pliku wejściowego BAN.IN znajduje się jedna liczba całkowita dodatnia $n \leqslant 1000$. Jest to liczba klientów.

W każdym z kolejnych n wierszy jest jedna liczba całkowita dodatnia, mniejsza niż 10^{30}, zapisana za pomocą co najwyżej 30 cyfr dziesiętnych.

Liczba w i-tym z tych wierszy to wysokość kwoty, którą zamierza pożyczyć klient nr i.

Wyjście

W każdym z 2n kolejnych wierszy pliku wyjściowego BAN.OUT należy zapisać malejący ciąg liczb całkowitych dodatnich z zakresu [0..99] oddzielonych pojedynczymi odstępami albo jedno słowo NIE:

- *w pierwszym wierszu i-tej pary wierszy — numery bankomatów, w porządku malejącym, z których powinien skorzystać klient numer i, by pobrać pożyczkę, albo słowo NIE, gdy nie może jej pobrać zgodnie z ustalonymi regułami;*
- *w drugim wierszu i-tej pary — numery bankomatów, w porządku malejącym, z których powinien skorzystać klient numer i, oddając pożyczkę, albo słowo NIE.*

Przykład

Dla pliku tekstowego BAN.IN:

```
2
7
633825300114114700748351602698
```

poprawnym rozwiązaniem jest plik wyjściowy BAN.OUT:

```
4 3 1 0
3 0
NIE
99 3 1
```

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

ROZWIĄZANIE

Zadanie o bankomatach prezentowało ciekawy system zapisywania liczb całkowitych: system o podstawie -2. System ten jest binarnym systemem pozycyjnym, czyli takim, że pozycja cyfry decyduje o związanej z nią wartości, a jedynymi dozwolonymi cyframi są 0 oraz 1, z tym że na $i-$tej pozycji od prawej w zapisie liczby występuje cyfra odpowiadająca wartości $(-2)^i$. Zatem bitom, licząc od prawej, odpowiadają wartości $1, -2, 4, -8, 16, -32, \ldots$. Pojawienie się w zapisie pozycyjnym liczby bitu 1 oznacza dodanie odpowiadającej mu wartości $(-2)^i$, a 0 — pominięcie jej. Kolejne bankomaty odpowiadały kolejnym pozycjom bitów, a wybór bankomatu albo jego pominięcie — jedynce lub zeru na tej pozycji.

Okazuje się, że w tym systemie można zapisać każdą liczbę całkowitą i to dokładnie na jeden sposób. (Rozważamy tylko najkrótszy możliwy zapis.) Kolejnych kilka liczb całkowitych w okolicy zera ma następujące reprezentacje:

-5	1111
-4	1100
-3	1101
-2	10
-1	11
0	0
1	1
2	110
3	111
4	100
5	101

System ten był badany na początku lat 60-tych jako alternatywny do układu dwójkowego w procesorach komputerowych. Miał on kilka zalet, np. ujednolicenie zapisu i działań na liczbach dodatnich i ujemnych, brak wyróżnionych bitów, a także kilka wad, np. niesymetryczność zakresu liczb dodatnich i ujemnych w n-bitowej reprezentacji. Wadą tego systemu był też sposób obliczania liczby przeciwnej do danej, co okazało się niebanalną operacją. Widać, że postać każdej liczby w zapisie o podstawie -2 oraz postać liczby przeciwnej do niej różnią się od siebie zasadniczo.

Zadanie o bankomatach polegało właśnie na znalezieniu reprezentacji bitowej zadanej liczby i liczby przeciwnej do niej w układzie o podstawie -2.

Algorytm przedstawienia dowolnej nieujemnej liczby x w układzie pozycyjnym o podstawie $b \geqslant 2$ jest następujący. W tablicy cyfr $C[0..\mathit{liczbacyfr}-1]$ **of** $0..b-1$ umieszczamy od końca kolejne cyfry przedstawienia:

```
for i:=0 to liczbacyfr − 1 do
  begin
  C[i] := x mod b;
    x := x div b
end
```

Dzielimy zatem liczbę x przez b tyle razy, ile cyfr przedstawienia sobie zażyczymy. W tablicy cyfr C od prawej do lewej odnotowujemy reszty z tego dzielenia. Jeżeli wartość x po kilku iteracjach spadnie do zera, to pozostała część tablicy, aż do lewego końca, zostanie uzupełniona zerami. Możemy też wyjść wcześniej z pętli, zmieniając jej nagłówek na **while** $x \neq 0$ **do**. W tej wersji trzeba albo wypełnić na początku tablicę C zerami, albo pamiętać, gdzie się zaczyna zapis liczby.

Powyższy algorytm dla $b = 2$ nazywany jest niekiedy algorytmem *zamiany liczby z układu dziesiętnego na dwójkowy*, co jest mylną nazwą, bo z układem dziesiętnym ma on niewiele wspólnego. Może tyle tylko, że do reprezentowania liczby jesteśmy przyzwyczajeni używać układu dziesiętnego. W algorytmie bowiem odwołujemy się do tego, co z liczby pozostanie w kolejnych przebiegach pętli algorytmu, a nie do rozwinięcia dziesiętnego tej liczby i jej pozostałości „wewnątrz” algorytmu.

Ponieważ mamy znaleźć przedstawienie liczb w układzie o podstawie -2, więc najprościej byłoby użyć tego algorytmu dla $b = -2$. Problem polega na tym, że nie jest jasne, jakie znaczenie nadać wyrażeniom x **mod** b oraz x **div** b, gdy b lub x są ujemne. Aby wyjaśnić tę kwestię można spróbować użyć dwóch metod: napisać prosty program w Pascalu i najzwyczajniej w świecie przyjrzeć się kilku wartościom, co oczywiście nie da stuprocentowej pewności co do poprawności, ale przynajmniej pozwoli się upewnić co do koncepcji. Drugi sposób to oczywiście spróbować udowodnić poprawność algorytmu dla ujemnych b.

Napisanie w Pascalu lub w C programu, który realizuje pierwszy pomysł dla $b = -2$ przynosi rozczarowanie: w tablicy C, np. dla $x = 11$, pojawiają się 3 wartości: $-1, 0, 1$ — zupełnie nie to, czego się spodziewaliśmy.

Spróbujmy zatem drugiej metody — dowodu poprawności algorytmu. Oczywiście możemy to potraktować w formie żartu, bo skoro pierwsza metoda zawiodła, to po co dalej rozmyślać? Nie jest to jednak do końca taki czczy żart, bo jeśli przyjrzymy się wynikom, to zobaczymy, że te rozwinięcia, które nie zawierają minus jedynek są dobre, a te, które zawierają — też są sensowne, tyle że sugerują wzięcie odpowiednich wartości z minusem. Na przykład liczba 11 ma rozwinięcie (-1)0(-1)1, czyli jakby $(-1)*(-8)+0*(4)+(-1)*(-2)+1*(1) = 11$. Coś się tu jednak zgadza. W innych przypadkach jest podobnie. Może zatem wystarczy się jakoś uwolnić od tych minus jedynek i w ogóle będzie dobrze? Wróćmy do pomysłu dowodu algorytmu zamiany podstawy układu pozycyjnego i zobaczmy, gdzie się nam dowód załamie.

Jaka jest interpretacja operacji **div** i **mod**? Oczywiście chodzi o iloraz i resztę dzielenia całkowitoliczbowego, czyli o takie dwie liczby d i r, że

$$x = db + r, \qquad \text{gdzie} \quad 0 \leqslant r < |b|. \tag{1}$$

Liczby d i r są wyznaczone jednoznacznie przez dowolne x oraz b, takie że $|b| > 1$.

Twierdzenie 1: Przy przyjętej interpretacji operacji **div** i **mod** algortym z poprzedniej strony jest poprawny dla każdej podstawy b takiej, że $|b| \geqslant 2$.

Dowód: Indukcja ze względu na $|x|$. Zacznijmy od przypadku $b \geqslant 2$ oraz $x \geqslant 0$. Najpierw odnotujmy, że liczba 0 daje właściwe przedstawienie przy każdej podstawie $b > 1$: wszystkie operacje **div** i **mod** dają w wyniku zero, więc tablica C zostanie wypełniona samymi zerami. Teraz załóżmy, że nasz algorytm działa poprawnie dla wszystkich wartości nieujemnych mniejszych niż pewne $x > 0$ (czyli wypełnia dla takich wartości tablicę C właściwymi cyframi). Udowodnimy, że w tym przypadku będzie poprawnie działał również dla liczby x.

Prześledźmy, co się stanie, gdy wykonamy pierwszy obrót pętli naszego algorytmu. Do tablicy C na pozycji 0 zostanie wpisany wynik x **mod** b, a liczba x stanie się równa x **div** b. Ponieważ $b > 1$, więc liczba x **div** b jest mniejsza niż x. Zatem na mocy założenia indukcyjnego liczba x **div** b zostanie w dalszej części algorytmu prawidłowo rozwinięta w komórkach $C[1], C[2], \ldots$, czyli do tablicy zostaną wpisane właściwe cyfry rozwinięcia $x\mathbf{div}b$ przy podstawie b, tyle że przesunięte o 1 w lewo. Ze względu na to, że przesunięcie o 1 w lewo oznacza w każdym potęgowym układzie pozycyjnym przemnożenie przez b, otrzymana liczba będzie równa

$$b \cdot (x \ \mathbf{div} \ b) + (x \ \mathbf{mod} \ b) = x,$$

co było do pokazania (na razie dla $x \geqslant 0$, $b \geqslant 2$).

Teraz najważniejsze odkrycie pozwalające nam przenieść ten dowód na ujemne wartości b oraz x. Zauważmy, że jedynym miejscem dowodu, w którym zakładaliśmy, że $b \geqslant 2$ i $x \geqslant 0$ było wykorzystanie kroku indukcyjnego: chodziło o to, żeby liczba

x **div** b była mniejsza od x. Reszta rachunków nie wymagała ani dodatności b, ani nieujemności x.

Jak zatem zabrać się za ujemne wartości b i x? Wystarczy wymusić istotny postęp w „zmniejszaniu” się liczby x. W tym celu zauważmy, że dla $b \leqslant -2$, dla każdego x, poza jednym przypadkiem, wartość $|x \textbf{ div } b|$ jest mniejsza, niż $|x|$. Tym jedynym przypadkiem jest $x = -1$. Wtedy jednak x **div** $b = 1$ i w zasadzie można na tym poprzestać, bo w kolejnym kroku 1 **div** $b = 0$. Czyli poza -1, zarówno dla dodatnich, jak i dla ujemnych liczb x, mamy $|x \textbf{ div } b| < |x|$. Ze względu na to, że również dla -1 w następnym kroku osiągniemy zero uzyskując właściwą reprezentację: $-1 = (0 \cdots 01(b-1))_b$, rozumowanie możemy tu zakończyć. Formalnie powiększamy bazę indukcji o przypadki $x = -1$ i $x = 1$. Następnie zakładamy, że teza zachodzi dla wszystkich liczb co do modułu mniejszych od $|x|$, po czym przeprowadzamy wnioskowanie identyczne z poprzednim, które przekona nas, że algorytm działa poprawnie dla każdej liczby całkowitej x i dla każdej liczby b o module większym od 1. □

Zaraz, zaraz! To dlaczego nasz program w Pascalu dał złe wyniki? Skąd się wzięły te minus jedynki? Ja tego szczerze powiedziawszy nie rozumiem. To znaczy nie rozumiem, dlaczego tak wybitny informatyk, jakim jest Niklaus Wirth, projektując Pascala, w tak dziwaczny sposób zdefiniował operacje **div** i **mod**. Okazuje się, że dla liczb ujemnych pascalowe odpowiedniki wyników operacji **mod** i **div** nie mają nic wspólnego ze wzorem (1). Na przykład, zgodnie ze wzorem (1), (-7) **div** $(-2) = -4$, (-7) **mod** $(-2) = 1$, podczas gdy w Pascalu odpowiednie wyniki, to 3 i -1.

Widać więc, że przy nieograniczonej liczbie bankomatów każdą sumę dukatów da się pożyczyć i oddać, a algorytm wyboru bankomatów jest bardzo prosty. Jego istotę (z pominięciem problemów związanych z realizacją omawianych operacji) oddaje następujący kod poprawny dla każdej podstawy b:

```
for i:=0 to LiczbaBankomatow − 1 do
  begin
    Bankomat[i] := x mod (abs(b)); {u nas b = −2 }
    x := (x − Bankomat[i]) div b {odejmujemy, żeby uniknąć niejasności
                                                      z operacją div}
  end
```

Rzecz jasna, gdyby po zakończeniu pętli wartość x była różna od zera, to oznaczałoby to, że nie starcza nam bankomatów na wyrażenie aż tak dużej co do modułu liczby i byłaby to jedyna przyczyna, dla której odpowiedzią powinno być NIE. W rzeczywistości ograniczenia w zadaniu były tak dobrane, żeby na końcach przedziału występowały wartości niewyrażalne na 100 pozycjach w układzie o podstawie (-2).

Tym razem obowiązek zrealizowania operacji **div** i **mod** spoczywał na zawodnikach, bowiem zakres liczb przekraczał rozmiar wszelkich standardowych typów, w których te operacje są zdefiniowane na poziomie języka (zresztą, o czym była mowa, niezbyt szczęśliwie z punktu widzenia naszych potrzeb). Nie było to trudne, bo pierwsza operacja wymaga sprawdzenia jednego bitu, a druga — generując na przemian wyniki dodatnie i ujemne — sprowadzała się do ucinania ostatniej cyfry z ewentualnym odjęciem jedynki od wyniku dla argumentu ujemnego.

TESTY

Testy były podzielone na 11 grup.

- BAN0.IN — test z treści zadania.
- BAN1.IN — jedna liczba równa 1.
- BAN2.IN — 3 niewielkie liczby jednobajtowe.
- BAN3.IN — 100 liczb mieszczących się w typie **longint** (4 bajty).
- BAN4.IN — wszystkie liczby od 1 do 1000.
- BAN5.IN — różne liczby, w tym kilka trudnych, na granicy reprezentowalności.
- BAN6.IN — 442 liczby postaci $2^i, 2^i + 1, 2^i - 1, F(i)$, gdzie $F(i)$, to i-ta liczba Fibonacciego.
- BAN7.IN — 978 losowych liczb o liczbie cyfr z przedziału 1..30.
- BAN8.IN — 1000 losowych liczb o maksymalnej długości 30 cyfr.
- BAN9.IN — 1000 największych liczb trzydziestocyfrowych cyfr (harmonia dziewiątek z różnymi końcówkami).
- BAN10.IN — 1000 losowych liczb 28–30-cyfrowych

Adam Borowski
autor zadania, opracowanie

Krzysztof Sobusiak
program wzorcowy

Gonitwa

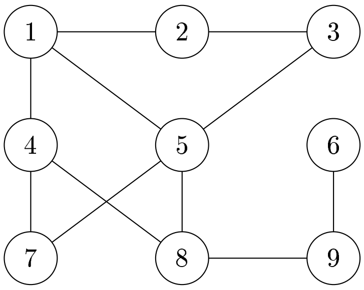
Gonitwa jest grą planszową dla dwóch osób A i B. Plansza do gry składa się z pól ponumerowanych kolejnymi liczbami naturalnymi, poczynając od 1. Dla każdej pary różnych pól wiadomo, czy są sąsiednie, czy nie. Każdy z graczy dysponuje jednym pionem, który początkowo znajduje się na wskazanym z góry polu planszy, każdy pion na innym polu. Ruch gracza polega na przesunięciu własnego piona na jedno z sąsiednich pól lub pozostawieniu piona na miejscu. Plansza ma następujące właściwości:

- *nie zawiera trójkątów, tzn. nie istnieją trzy różne pola, takie że każde dwa z nich są sąsiednie,*
- *każde pole może być osiągnięte zarówno przez gracza A, jak i przez gracza B.*

Gra składa się z wielu rund następujących po sobie. W jednej rundzie każdy z graczy wykonuje jeden ruch, przy czym gracz A zawsze wykonuje swój ruch przed ruchem gracza B.

Powiemy, że gracz B dogonił gracza A, jeżeli oba piony znajdą się na tym samym polu. Rozstrzygnij, czy dla danych początkowych położeń obu pionów, gracz B może dogonić gracza A, niezależnie od tego jak dobrze gra gracz A. Jeśli tak, to po ilu rundach gracz B dogoni gracza A, jeżeli gracz A stara się uciekać jak najdłużej, a gracz B stara się dogonić gracza A jak najszybciej i obaj grają optymalnie?

Przykład



Rozważmy planszę przedstawioną na rysunku. Sąsiednie pola planszy są połączone odcinkami. Jeżeli na początku gry pion gracza A znajduje się na polu 9, natomiast pion gracza B jest na polu 4, to gracz B dogoni A w trzeciej rundzie, o ile obaj gracze grają optymalnie. Gdyby pion gracza A znajdował się początkowo na polu 8, to ruszając z pola 4 gracz B nie ma szans na dogonienie A, jeżeli tylko ten gra optymalnie.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego GON.IN opis planszy i numery pól, na których początkowo umieszczone są piony graczy;*

- *sprawdza, czy gracz B może dogonić gracza A i jeżeli tak, to oblicza liczbę rund, po których B dogoni A przy założeniu, że obaj gracze grają optymalnie;*
- *zapisuje wynik w pliku tekstowym GON.OUT.*

Wejście

W pierwszym wierszu pliku wejściowego GON.IN znajdują się cztery liczby całkowite n, m, a oraz b, oddzielone pojedynczymi odstępami, gdzie: $2 \leqslant n \leqslant 3000$, $n-1 \leqslant m \leqslant 15000$, $1 \leqslant a, b \leqslant n$ oraz $a \leqslant b$. Są to odpowiednio: liczba pól na planszy, liczba wszystkich różnych par (nieuporządkowanych) tych pól, które ze sobą sąsiadują, numer pola, na którym jest umieszczony pion gracza A oraz numer pola, na którym jest umieszczony pion gracza B.

W każdym z następnych m wierszy znajdują się dwie różne dodatnie liczby całkowite oddzielone pojedynczym odstępem. Liczby w każdym z tych wierszy są numerami dwóch pól, które ze sobą sąsiadują.

Wyjście

Twój program powinien zapisać w pierwszym i jedynym wierszu pliku GON.OUT jedno słowo NIE, jeśli gracz B nigdy nie dogoni gracza A, a w przeciwnym przypadku jedną liczbę całkowitą — liczbę rund, po których, gracz B dogoni gracza A.

Przykład

Dla pliku wejściowego GON.IN:

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

poprawnym rozwiązaniem jest plik wyjściowy GON.OUT:

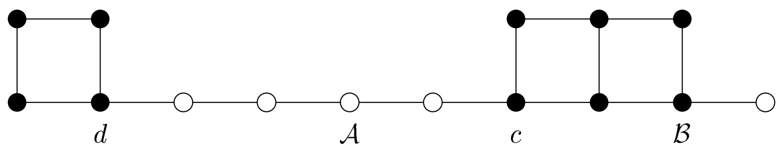
```
3
```

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

ROZWIĄZANIE

Dość oczywiste jest spostrzeżenie, że gracz $\mathcal{A}$ może uciekać nieskończenie długo tylko wtedy, jeśli uda mu się dotrzeć do grupy pól tworzących zamkniętą pętlę. Takich grup może być wiele, także jedno pole może należeć do wielu pętli. Pole należące do jakiejkolwiek pętli będziemy nazywać **bezpiecznym**.

Rys. 1 Przykładowa plansza z zaznaczonymi (przez zaczernienie) polami bezpiecznymi.



Kiedy gracz $\mathcal{A}$ dotrze do jakiegokolwiek z pól bezpiecznych może od tej chwili poruszać się po dowolnej z pętli przechodzących przez to pole, zmieniając ewentualnie kierunek ruchu na przeciwny. Jedyną strategią dla gracza $\mathcal{B}$ jest zatem niedopuszczenie, by $\mathcal{A}$ dotarł do któregoś z pól bezpiecznych. W tym miejscu łatwo zrobić błąd: jeśli gracz $\mathcal{A}$ dotrze do pola bezpiecznego, do którego **w tej samej rundzie** może dotrzeć też gracz $\mathcal{B}$, gra zakończy się wygraną gracza $\mathcal{B}$! Przykładem takiego pola jest pole c na rysunku 1.

Nieskończona trasa ucieczki istnieje wtedy i tylko wtedy, gdy na planszy istnieje pole bezpieczne x takie, że gracz $\mathcal{A}$ ma do pola x bliżej niż gracz $\mathcal{B}$. Możemy to sprawdzić znajdując odległości wszystkich pól zarówno od pola, z którego rozpoczyna grę gracz $\mathcal{A}$, jak i pola startowego gracza $\mathcal{B}$. Robimy to korzystając z algorytmu przeszukiwania grafu planszy wszerz (zob. [9]):

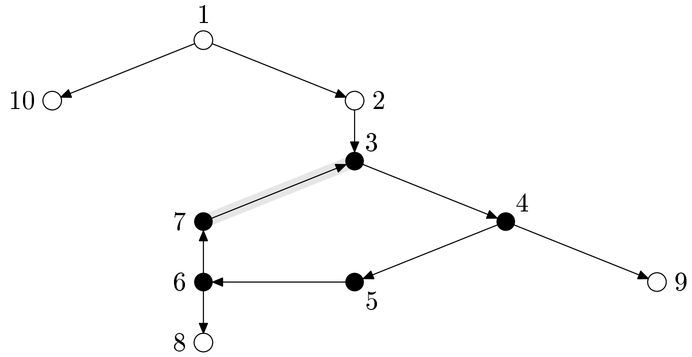
- wiemy, że jedynym polem odległym od $\mathcal{A}$ (odp. $\mathcal{B}$) o 0 jest jego pole startowe,
- dla każdego $k = 1, 2, 3, \ldots$ sprawdzamy, czy któreś z pól odległych od $\mathcal{A}$ (odp. $\mathcal{B}$) o $k-1$ ma jeszcze nie odwiedzonego sąsiada. Jeśli tak, zapamiętujemy że odległość tego sąsiada od $\mathcal{A}$ (odp. $\mathcal{B}$) wynosi k oraz dodajemy go do kolejki pól do sprawdzenia w kolejnym ruchu,
- obliczenia kończymy, gdy kolejka pól do sprawdzenia będzie pusta.

Teraz wystarczy sprawdzić, czy któreś z pól, którego odległość od $\mathcal{A}$ jest mniejsza niż odległość od $\mathcal{B}$, jest bezpieczne. I to jest właśnie najtrudniejsza część zadania: określenie, które pola są bezpieczne, a które nie.

Najbardziej efektywnym sposobem jest wykorzystanie przeszukiwania grafu w głąb (zob. [9] lub [12]) począwszy od dowolnego z wierzchołków grafu. Odwiedzanym polom nadajemy kolejne numery, a procedura przeszukująca wywołana rekurencyjnie zwraca najmniejszy numer pola, do którego można dojść idąc w danym kierunku i kończąc na polu, które było już odwiedzone lub z którego nie ma wyjścia.

Jak widać z przykładu (zob. rysunek 2), jeśli dane pole nie jest bezpieczne, zwrócony przez procedurę rekurencyjną numer musi być większy od numeru wierzchołka, z którego procedura ta była wywołana. Z drugiej strony, jeśli zwrócony numer jest

Rys. 2 Przykładowy przebieg przeszukiwania w głąb, mającego na celu wyznacznie pól bezpiecznych. Numery przy wierzchołkach oznaczają kolejność odwiedzin, strzałki symbolizują kierunek przechodzenia krawędzi grafu.



mniejszy lub równy numerowi aktualnego wierzchołka, oznacza to, że jesteśmy na polu bezpiecznym.

Jeśli nie istnieje nieskończona trasa ucieczki, wynikiem zadania jest odległość od początkowej pozycji gracza $\mathcal{B}$ do najdalszego pola, którego odległość od początkowej pozycji gracza $\mathcal{A}$ jest nie większa niż odległość od $\mathcal{B}$. Najlepszą strategią dla gracza $\mathcal{A}$ jest umieszczenie piona na tym polu i czekanie, aż $\mathcal{B}$ tam dotrze.

Program wzorcowy najpierw wyznacza pola bezpieczne. Wierzchołki nie są numerowane kolejnymi liczbami, lecz liczbami oznaczającymi, na którym poziomie rekursji w przeszukiwaniu w głąb dany wierzchołek został znaleziony. Pomimo tej różnicy program daje poprawne wyniki. Uzasadnienie tego faktu pozostawiamy Czytelnikowi. Następnie wyznaczana jest tablica odległości wszystkich pól od początkowego pola gracza $\mathcal{A}$, oraz taka sama tablica z odległościami od początkowego pola gracza $\mathcal{B}$. Teraz mozemy już sprawdzić, czy istnieje pole bezpieczne leżące bliżej $\mathcal{A}$ niż $\mathcal{B}$. Jeśli tak, wypisujemy odpowiedź 'NIE', w przeciwnym przypadku określamy, na którym polu gracz $\mathcal{A}$ może się najdłużej chować. Złożoność powyższego algorytmu jest liniowa ze względu na rozmiar grafu.

TESTY

Do sprawdzenia rozwiązań zawodników użyto 21 testów. Test GON0.IN był testem z treści zadania. W celu eleminacji przyznawania punktów rozwiązaniom wypisującym zawsze 'NIE' zgrupowano każdy test o numerze $2i-1$ z testem o numerze $2i$, dla $i = 1, \ldots, 10$. Testy GON1.IN–GON8.IN sprawdzały głównie poprawność programów, natomiast testy GON9.IN–GON20.IN były testami wydajnościowymi.

Wojciech Rytter
autor zadania, opracowanie

Tomasz Waleń
program wzorcowy

Najlżejszy język

Dany jest alfabet A_k złożony z k początkowych liter alfabetu angielskiego. Każda litera w tym alfabecie ma określoną wagę, która jest dodatnią liczbą całkowitą. Wagą słowa zbudowanego z liter alfabetu A_k nazywamy sumę wag wszystkich jego liter. Językiem nad alfabetem A_k nazywamy dowolny skończony zbiór różnych słów utworzonych z liter tego alfabetu. Wagą języka nazywamy sumę wag wszystkich jego słów.

Mówimy że język jest **bezprefiksowy**, *gdy żadne jego słowo nie jest prefiksem (początkiem) jego innego słowa.*

Chcemy ustalić, jaka może być najmniejsza waga n-elementowego, bezprefiksowego języka nad alfabetem A_k.

Przykład

Załóżmy, że $k = 2$ waga litery a — $W(a) = 2$ oraz waga litery b — $W(b) = 5$.

Wtedy: waga słowa ab — $W(ab) = 2 + 5 = 7$. $W(aba) = 2 + 5 + 2 = 9$.

Waga języka $J = \{ab,\ aba,\ b\}$ — $W(J) = 21$.

Język J nie jest bezprefiksowy, ponieważ jego słowo ab jest prefiksem słowa aba.

Najlżejszym trzyelementowym bezprefiksowym językiem nad alfabetem A_2, o określonych powyżej wagach liter, jest $\{b,\ aa,\ ab\}$; jego waga wynosi 16.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego NAJ.IN dwie liczby naturalne n oraz k, a następnie wagi wszystkich k liter alfabetu A_k;*
- *oblicza najmniejszą wagę bezprefiksowego n-elementowego języka nad alfabetem A_k;*
- *zapisuje wynik w pliku tekstowym NAJ.OUT.*

Wejście

W pierwszym wierszu pliku wejściowego NAJ.IN znajdują się dwie dodatnie liczby całkowite n i k oddzielone pojedynczym odstępem, $2 \leqslant n \leqslant 10000$, $2 \leqslant k \leqslant 26$. Są to odpowiednio: liczba słów języka i liczba liter alfabetu.

Drugi wiersz zawiera k dodatnich liczb całkowitych nie większych niż 10000, oddzielonych pojedynczymi odstępami — i-ta liczba jest wagą i-tej litery.

Wyjście

W pierwszym i jedynym wierszu pliku wyjściowego NAJ.OUT Twój program powinien wypisać jedną liczbę całkowita — wagę najlżejszego, bezprefiksowego, n-elementowego języka nad alfabetem A_k.

Przykład

Dla pliku wejściowego NAJ.IN:

```
3 2
2 5
```

poprawnym rozwiązaniem jest plik wyjściowy NAJ.OUT:

```
16
```

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

ROZWIĄZANIE

Na początek sformułujemy zadanie i algorytm w terminach drzew. Każdy bezprefiksowy język L na alfabetem A odpowiada drzewu, którego krawędzie są etykietowane literami z A. Każdemu prefiksowi słowa języka L odpowiada dokładnie jeden wierzchołek drzewa. Wagą krawędzi drzewa jest waga litery, będącej etykietą tej krawędzi.

Tak jak to jest przeważnie w informatyce, nasze drzewa rosną do góry nogami (korzeniami). **Wierzchołkiem zewnętrznym** (lub **liściem**) drzewa nazywamy wierzchołek, który nie ma wierzchołków potomnych (dzieci). **Wierzchołkiem wewnętrznym** nazywamy każdy wierzchołek drzewa, który nie jest wierzchołkiem zewnętrznym.

Ścieżką w drzewie nazywamy dowolny ciąg kolejnych krawędzi tego drzewa w kierunku od korzenia do pewnego liścia. Ścieżka nie musi kończyć się w liściu. Przez wagę ścieżki będziemy rozumieć sumę wag krawędzi leżących na tej ścieżce. Wagą całego drzewa nazwiemy sumę wag wszystkich ścieżek prowadzących do liści. Czasami zamiast mówić „waga ścieżki prowadzącej do wierzchołka v” będziemy mówili po prostu „waga wierzchołka v”.

Zadanie można teraz sformułować następująco: policz minimalną wagę drzewa, które ma n liści. Narysujmy wierzchołki drzewa tak, żeby każdy wierzchołek leżał na głębokości odpowiadającej jego wadze (korzeń drzewa na poziomie 0), tak jak na rysunkach 2 i 3.

Rozpatrzmy następujący przykład. Niech $n = 10$, $k = 4$ oraz niech wagi liter wynoszą:

$$W(a) = 2, \qquad W(b) = 3, \qquad W(c) = 8, \qquad W(d) = 9.$$

Wtedy ostatnie drzewo z rysunku 3 jest drzewem o minimalnym koszcie (być może nie jest to jedyne takie drzewo). Język

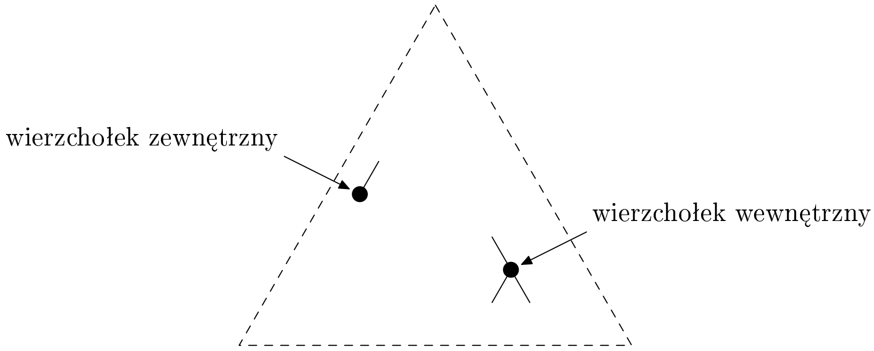
$$L = \{aaa,\ aab,\ aba,\ abb,\ ac, baa,\ bab,\ bb,\ c,\ d\}$$

jest 10-elementowym bezprefiksowym językiem o minimalnej wadze. Język ten składa się ze słów, którego słowa powstają ze ścieżek ostatniego drzewa na rysunku 3 (słowo otrzymujemy czytając po kolei etykiety krawędzi na ścieżce). Podstawową (oczywistą) własnością drzewa o minimalnej wadze jest to, że:

- liść nie może być na wyższym poziomie (mieć mniejszą wagę) niż jakikolwiek węzeł wewnętrzny.

W związku z powyższym **nie może** zajść sytuacja taka jak na rysunku 1. Jeśli mamy węzeł wewnętrzny o wadze w, to nie ma liścia (wierzchołka zewnętrznego) o wadze mniejszej niż w. Zatem w drzewie nie ma "dziur".

Rys. 1 Sytuacja, która nie może zajść w drzewie o minimalnej wadze.



Powyższa własność implikuje następujący schemat algorytmu. Budujemy drzewo poziomami z góry na dół (korzeń jest na górze). Za każdym razem staramy się **rozwinąć** liść na najwyższym poziomie. Operację rozwijania kontynuujemy do momentu, gdy mamy n liści i próba rozwinięcia liścia nie zmniejsza wagi drzewa.

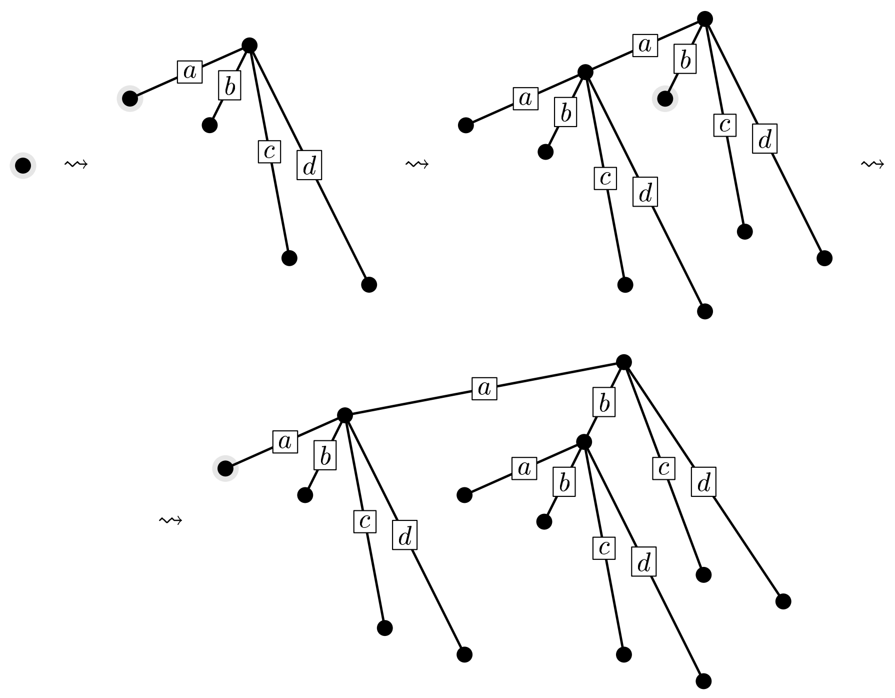
Oznaczmy początkowe k liter przez $a_1, a_2, \ldots, a_k$. Niech operacja *NowySyn(v, i)* oznacza utworzenie nowego węzła v', podczepionego do węzła v krawędzią o etykiecie a_i. Podstawową operacją jest następująca procedura:

```
procedure RozwińLiść(v : wierzchołek);
begin
  NowySyn(v, 1);
  for i = 2 to k do
  if liczbaLiści < n then NowySyn(v, i)[[[czy coś źle?]]]
  else
    if istnieje liść w tż. W(w) > W(v) + W(a_i)
    then begin
      Usuń(w);
      NowySyn(v, i)
    end
end
```

Teraz możemy sformułować algorytm:

```
waga := +∞;
nowaWaga := 0;
drzewo := ε; { sam korzeń }
while (waga > nowaWaga) or (liczbaLiści < n) do begin
  waga := nowaWaga;
```

Rys. 2 Pierwsze 3 iteracje algorytmu, rozwijamy „do pełna” 3 najlżejsze liście (zaznaczone na szaro) rozpoczynając od drzewa jednowierzchołkowego.



v := liść o minimalnej wadze;
RozwińLiść(v);
nowaWaga := aktualna waga drzewa
end

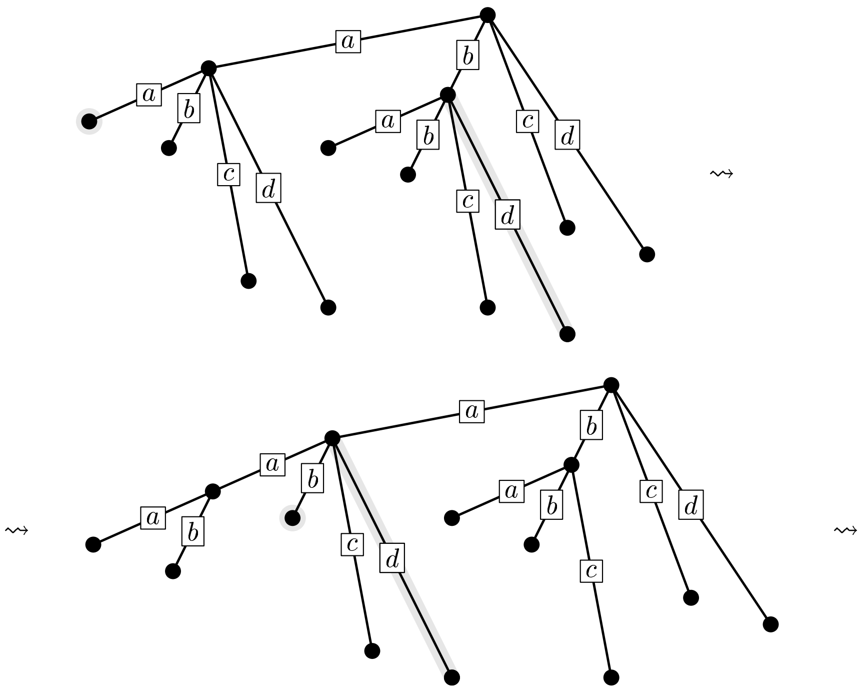
Powróćmy do rozważanego przykładu. Na rysunku 2 przedstawiony jest proces konstrukcji drzewa do momentu gdy powstanie $n = 10$ liści. Na rysunku 3 są przedstawione następne 3 iteracje, za każdym razem rozwijamy liść zaznaczony na szaro i usuwamy pewną zaznaczoną na szaro krawędź. Rozwijanie ostatniego zaznaczonego liścia nie zmniejsza kosztu i algorytm się zatrzymuje.

W algorytmie można pominąć konstrukcję drzewa i operować jedynie na wagach liści. Dane, na których działamy, to co najwyżej n liczb interpretowanych jako aktualne wagi liści.

Załóżmy, że mamy strukturę danych, która pozwala na wykonywanie następujących operacji:

- *ExtractMin*(**var** S : *struktura*) : **integer** — podaje najmniejszy element znajdujący się w strukturze i usuwa go,
- *Add*(**var** S : *struktura*; x : **integer**) — dodaje x do struktury, jeśli liczba elementów w strukturze jest większa niż n, to zostawia w niej tylko n najmniejszych elementów (a pozostałe wyrzuca),

Rys. 3 Następne 3 iteracje, za każdym razem rozwijamy liść o najmniejszej wadze (zaznaczony na szaro) i usuwamy inne liście (krawędzie do usuwanych liści są zaznaczone na szaro). Pierwsze drzewo na tym rysunku jest ostatnim drzewem z rysunku 2.



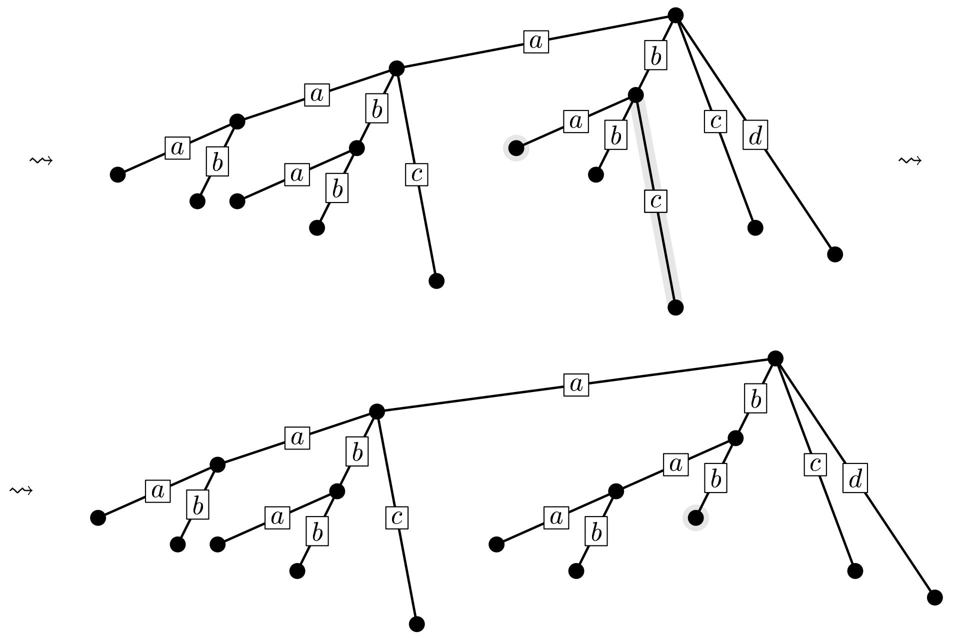
- *Sum*(**var** S : *struktura*) : **integer** — zwraca sumę wszystkich elementów w strukturze,
- *Count*(**var** S : *struktura*) : **integer** — zwraca liczbę elementów w strukturze,
- *Init*(**var** S : *struktura*) — inicjuje strukturę tak, że zawiera jedynie element 0.

Dysponując opisaną strukturą, algorytm możemy zapisać następująco:

```
minSuma := +∞;
Init(S);
for i := 1 to k do Add(S, W(i));
while (Count(S) < n) or ((n = Count(S)) and (Sum(S) < minSuma))
                                                        do begin
  if (n = Count(S)) and (Sum(S) < minSuma) then
    minSuma := Sum(S);
  x := ExtractMin(S);
  for i := 1 to k do Add(S, x + W(i))
end
```

Strukturę można zaprogramować jako dwa **kopce** (opis kopca znajdzie czytelnik w [9]). Elementy jednego kopca są uporządkowane rosnąco, drugiego — malejąco.

Rys. 3 dokończenie



Każdy element struktury S jest wkładany do obu kopców, przy czym każda kopia posiada wskaźnik na swojego „bliźniaka” (który to wskaźnik trzeba aktualizować przy wszystkich operacjach na kopcu). Dla takiej implementacji struktury, złożoność operacji *ExtractMin* i *Add* wynosi $O(\log n)$. Ponieważ maksymalna liczba obrotów pętli wynosi n, cały algorytm ma złożoność $O(kn \log n)$.

TESTY

Do oceny rozwiązań zawodników wykorzystano 10 testów. Oto ich krótka charakterystyka:

- NAJ0.IN — test z treści zadania.
- NAJ1.IN, NAJ2.IN — proste testy poprawnościowe.
- NAJ3.IN, NAJ4.IN — test, dla którego wartości zmiennych przekraczają wartość typu **word**.
- NAJ5.IN–NAJ10.IN — testy wydajnościowe.

Andrzej Pelc
autor zadania, opracowanie

Marcin Mucha
program wzorcowy

Gra Ulama

Wybitny polski matematyk Stanisław Ulam zaproponował następującą grę dwuosobową.

Gracz A wybiera ze zbioru liczb całkowitych $X = \{0, 1, ..., 2047\}$ *liczbę* x*, nie zdradzając jej graczowi B.*

Gracz B musi dowiedzieć się, jaką liczbę wybrał gracz A. W tym celu może zadawać pytania postaci: „Czy x *jest elementem zbioru* Y*?", gdzie* Y *jest dowolnym podzbiorem zbioru* X*.*

Gracz A udziela odpowiedzi „TAK" lub „NIE", przy czym wolno mu co najwyżej raz skłamać.

Twoim zadaniem jest znalezienie i zaprogramowanie strategii zadawania pytań dla gracza B, która pozwoli mu wskazać poprawnie liczbę wybraną przez gracza A, przy możliwie najmniejszej liczbie zadanych pytań.

Przykład

Rozważmy grę Ulama dla zbioru $X = \{0, 1, 2, 3\}$*. Przyjrzyjmy się następującemu ciągowi pytań i odpowiedzi:*

B: Czy x *należy do* $\{0, 1\}$*?*
A: TAK
B: Czy x *należy do* $\{0\}$*?*
A: TAK

W tym miejscu B nie może jeszcze być pewnym, że liczbą wybraną przez A jest 0, ponieważ A mógł skłamać odpowiadając na drugie pytanie.

B: Czy x *należy do* $\{0\}$*?*
A: NIE

B nie wie, która z dwóch ostatnich odpowiedzi jest prawdziwa.

B: Czy x *należy do* $\{0\}$*?*
A: NIE

Szukaną liczbą na pewno nie jest 0. Odpowiedź na pierwsze pytanie musiała być prawdziwa, zatem szukaną liczbą jest 1.

B: x=1.

Zadanie

Twoim zadaniem jest zaprogramowanie strategii zadawania pytań umożliwiającej graczowi B znalezienie liczby x *przy pomocy jak najmniejszej liczby pytań. Rolę gracza A będzie pełnił program dostarczony przez organizatorów.*

Musisz zaprogramować moduł zawierający następujące trzy procedury (funkcje w przypadku języka C):

- `nowa_gra` — *ta procedura będzie wywoływana na początku każdej gry i tylko wtedy, możesz za jej pomocą dokonywać inicjalizacji swoich struktur danych;*
- `daj_pytanie` — *ta procedura służy do zadawania pytania i w tym celu powinna odpowiednio wypełniać tablicę pytanie opisaną w następnym paragrafie;*
- `analizuj_odpowiedz` — *ta procedura powinna odczytać z opisanej dalej zmiennej globalnej odpowiedź na ostatnio zadane pytanie i dokonać analizy tej odpowiedzi, jeśli w wyniku analizy ostatniej odpowiedzi liczba x zostanie zidentyfikowana, procedura powinna zasygnalizować ten fakt nadając właściwe wartości odpowiednim zmiennym globalnym.*

Program nadzorujący grę na początku wywoła napisaną przez Ciebie procedurę `nowa_gra` *po czym cyklicznie będzie wykonywał następujęce czynności:*

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

do momentu, gdy Twój program stwierdzi, że odgadł liczbę x (procedura `analizuj_odpowiedz` *umieści właściwą wartość w odpowiedniej zmiennej).*

UWAGA: Nie zakładaj, że program symulujący gracza A faktycznie ustala liczbę do odgadnięcia przed rozpoczęciem gry. Może dobierać ją w trakcie gry w taki sposób, aby zmusić Twój program do zadania największej liczby pytań lecz by po zakończeniu gry, kiedy znana jest „pomyślana" liczba oraz lista zadanych pytań i odpowiedzi, nie można było udowodnić, że gracz A skłamał więcej niż raz. Tak więc, powinieneś dążyć do tego, aby liczba pytań, jakie Twój program będzie musiał zadać w najgorszym przypadku, była jak najmniejsza.

Komunikacja

Komunikacja pomiędzy napisanym przez Ciebie modułem, a programem nadzorującym grę odbywa się przez zmienne globalne.

Zadanie pytania „Czy x należy do Y?" w procedurze `daj_pytanie` *polega na wypełnieniu tablicy* pytanie *typu* **array***[0..2047]* **of boolean** *(w języku C tablicy* **char** pytanie *[2048]) w taki sposób, że liczba i jest elementem Y wtedy i tylko wtedy, gdy* pytanie*[*i*]* = TAK*, gdzie* TAK *to predefiniowana stała o wartości* **true** *w Pascalu oraz 1 w C (podobnie predefiniowano stałą* NIE *o wartości* **false** *w Pascalu oraz 0 w C).*

Odpowiedź na zadane pytanie jest umieszczana w zmiennej odpowiedz *typu* **boolean** *(w języku C typu* **char***):* odpowiedz = TAK *wtedy i tylko wtedy, gdy odpowiedź jest twierdząca.*

O odnalezieniu szukanej liczby x Twój program powinien poinformować w procedurze `analizuj_odpowiedz`*, zapisując w zmiennej* wiem *typu* **boolean** *(w języku C typu* **char***)* TAK*, a następnie w zmiennej* x *znalezioną liczbę. Podanie odpowiedzi kończy grę, jeśli jest to odpowiedź błędna, gracz B przegrywa.*

Katalog i pliki

Programujący w Pascalu powinni przygotowywać swoje rozwiązanie w katalogu `C:\GRAPAS`*, natomiast piszący w* `C` *i* `C++` *w katalogu* `C:\GRAC`*.*

W katalogu `C:\GRAPAS` *(odpowiednio* `C:\GRAC`*) znajdziesz następujące pliki:*

- `gramod.pas` *(odpowiednio* `gramod.h` *i* `gramod.c`*) — moduł zawierający podstawowe definicje oraz deklaracje zmiennych komunikacyjnych;*
- `gra.pas` *— szkielet modułu grającego; powinieneś uzupełnić ten plik definicjami funkcji* `nowa_gra`*,* `daj_pytanie` *i* `analizuj_odpowiedz` *(dla piszących w języku C nagłówki powyższych 3 funkcji znajdują się w pliku* `gra.h`*, musisz napisać plik* `gra.c` *z definicjami tych funkcji);*
- `rywal.pas` *(odpowiednio* `rywal.c`*) — przykładowy program nadzorujący grę i korzystający z Twojego modułu, możesz z jego pomocą sprawdzić, czy Twój moduł spełnia specyfkację zadania.*

Wyjście

Wynikiem Twojej pracy powinien być zapisany na dyskietce tylko jeden plik `gra.pas` *lub* `gra.c`*, ale nie oba jednocześnie.*

Uwagi dla piszących w C i C++

Piszący w języku C (C++) spotkają się z koniecznością kompilacji programu wieloplikowego. Aby tego dokonać należy utworzyć projekt poleceniem „New Project" z menu „Project", po czym dodać (klawisz „Insert") do nowo utworzonego projektu wszystkie pliki `*.c`*. Od tego momentu polecenia „Make" i „Build" będą automatycznie kompilować i łączyć wszystkie pliki wymienione w projekcie.*

ROZWIĄZANIE

Każdy etap gry po odpowiedzi gracza $\mathcal{A}$ kodujemy jako parę liczb naturalnych (x, y) zwaną **stanem gry**. x jest mocą „zbioru prawd": zbioru tych liczb, które spełniają wszystkie dotychczasowe odpowiedzi. y jest mocą „zbioru kłamstw": zbioru tych liczb, które spełniają wszystkie dotychczasowe odpowiedzi, poza jedną. Oczywiście tylko element sumy tych zbiorów może być prawidłowym rozwiązaniem, inne liczby wskazywałyby bowiem, że gracz $\mathcal{A}$ skłamał co najmniej 2 razy.

Intuicyjnie jest jasne, że graczowi $\mathcal{B}$ trudniej sobie poradzić z elementami zbioru prawdy, bo na temat tych elementów gracz $\mathcal{A}$ może jeszcze skłamać, podczas gdy, jeśli wybrana liczba jest w zbiorze kłamstw, musi on do końca mówić prawdę. Jeśli do końca gry pozostało jeszcze j pytań, nadajemy elementom zbioru prawd wagę $j + 1$ odpowiadającą j możliwościom kłamstwa w każdej z pozostałych do końca odpowiedzi plus możliwości powstrzymania się od kłamstwa. Nadajmy też elementom zbioru kłamstwa wagę 1 odpowiadającą jedynej w ich przypadku możliwości: stałego mówienia prawdy. Łączna **waga** stanu gry (x, y) wynosi więc $w_j(x, y) = x(j + 1) + y$.

Każde pytanie w stanie gry (x, y) można potraktować jako pytanie typu: *czy nieznana liczba jest w danym podzbiorze P wielkości p zbioru prawdy lub w danym podzbiorze K wielkości k zbioru kłamstw?* Takie pytanie można zakodować jako „(p, k)?" i odpowiedzi TAK/NIE prowadzą do stanów odpowiednio (x_1, y_1) i (x_2, y_2), gdzie $x_1 = p$, $x_2 = x - p$, $y_1 = k + x - p$, $y_2 = y - k + p$. Łatwo zauważyć, że prawdziwa jest w tym wypadku równość:

$$w_j(x, y) = w_{j-1}(x_1, y_1) + w_{j-1}(x_2, y_2).$$

Równość ta sugeruje optymalną strategię dla gracza $\mathcal{B}$: należy tak zadawać pytania, by dzielić wagę stanu na połowy. Przy tej strategii gracz $\mathcal{B}$ zawsze znajduje nieznaną liczbę w co najwyżej 15-tu pytaniach i można pokazać, że jest to minimalna liczba pytań w najgorszym przypadku, bowiem $w_{15}(2^{11}, 0) = 2^{15}$.

A oto strategia. Stan początkowy to $(2^{11}, 0)$. Zadajemy pytanie $(2^{10}, 0)$?. Obie odpowiedzi na to pytanie prowadzą do stanów $(2^{10}, 2^{10})$. Następnie zadajemy pytanie $(2^9, 2^9)$?, uzyskując przy obu odpowiedziach stany $(2^8, 2^9)$, itd. Przez pierwszych 11 pytań w stanie (x, y) zadajemy pytanie $(x/2, y/2)$. Po tych 11 pytaniach, niezależnie od odpowiedzi uzyskujemy stan $(1, 11)$. Pozostały nam 4 pytania, zatem $w_4(1, 11) = 16$. Zadajemy pytanie $(1, 4)$?. Odpowiedzi TAK/NIE prowadzą odpowiednio do stanów $(1, 4)$ i $(0, 8)$. Drugi z nich jest łatwy: zwykłe wyszukiwanie binarne przy pomocy trzech pytań. W pierwszym zaś zadajemy pytanie $(1, 1)$? prowadzące do stanów $(1, 1)$ i $(0, 4)$. Zostały dwa pytania. Znów stan $(0, 4)$ jest łatwy, a w pierwszym zadajemy pytanie $(1, 0)$? uzyskując możliwe stany $(1, 0)$ i $(0, 2)$. W pierwszym przypadku poszukiwanie jest zakończone, a w drugim jedno dodatkowe pytanie wystarcza.

Przy implementowaniu tej strategii należy pamiętać parę zbiorów (X, Y), gdzie X jest aktualnym zbiorem prawd, a Y aktualnym zbiorem kłamstw. Pytanie (p, k)? należy interpretować jako pytanie *Czy nieznana liczba jest w zbiorze T?*, gdzie T jest sumą zbiorów pierwszych p liczb ze zbioru X i pierwszych k liczb ze zbioru Y.

TESTY

Testy wykonano przy pomocy modułu arbitra, którego kod był konsolidowany z kodem programów dostarczonych przez zawodników. Program arbitra był sparametryzowany tak, by mógł grać wg różnych strategii, poczynając od najprostszej, polegającej na wybraniu liczby i odpowiadaniu na pytania w odniesieniu do tej liczby, a kończąc na strategii gracza $\mathcal{A}$ wymuszającej zadanie 15 pytań. Szczegóły znajdzie Czytelnik w załączonym na dyskietce kodzie źródłowym.

Krzysztof Sobusiak
autor zadania, opracowanie

Tomasz Waleń
program wzorcowy

Łamane płaskie

Rozważmy kartezjański układ współrzędnych narysowany na kartce papieru. Łamaną płaską w tym układzie nazywamy łamaną, którą można wykreślić prowadząc ołówek bez odrywania od papieru w kierunku od lewej do prawej strony kartki i taką, że każda prosta zawierająca odcinek łamanej jest nachylona do osi OX *pod kątem należącym do przedziału domkniętego* $[-45°, 45°]$.

Na kartce narysowano n *różnych punktów o obu współrzędnych całkowitych. Jaka jest najmniejsza liczba łamanych płaskich, które należałoby wykreślić, żeby każdy z* n *punktów należał do co najmniej jednej z nich?*

Przykład

Dla 6 punktów o współrzędnych $(1,6)$, $(10,8)$, $(1,5)$, $(2,20)$, $(4,4)$, $(6,2)$ *najmniejszą liczbą łamanych płaskich pokrywających te punkty jest 3.*

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego LAM.IN liczbę punktów i ich współrzędne;*
- *oblicza najmniejszą liczbę łamanych płaskich, które należałoby wykreślić, żeby pokryły one wszystkie punkty;*
- *zapisuje wynik w pliku tekstowym LAM.OUT.*

Wejście

W pierwszym wierszu pliku wejściowego LAM.IN znajduje się dodatnia liczba całkowita n *nie większa od 30000. Jest to liczba punktów.*

W kolejnych n *wierszach znajdują się współrzędne punktów.*

Każdy wiersz zawiera dwie liczby całkowite x, y *oddzielone pojedynczym odstępem,* $0 \leqslant x \leqslant 30000$, $0 \leqslant y \leqslant 30000$. *Liczby w wierszu* $i+1$, $1 \leqslant i \leqslant n$, *pliku LAM.IN są współrzędnymi* i*-tego punktu.*

Wyjście

Twój program powinien zapisać jedną liczbę całkowitą w pierwszym i jedynym wierszu pliku wyjściowego LAM.OUT. Tą liczbą powinna być najmniejsza liczba łamanych płaskich, które należy wykreślić, żeby pokryć wszystkie dane punkty.

Przykład

Dla pliku wejściowego LAM.IN:

```
6
1 6
10 8
1 5
2 20
4 4
6 2
```

poprawnym rozwiązaniem jest plik wyjściowy LAM.OUT:

```
3
```

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

ROZWIĄZANIE

Kluczową dla rozwiązania zadania „operacją" jest obrót kartki papieru o 45 stopni w lewo, tzn. obrót każdego punktu o 45 stopni w lewo względem początku układu współrzędnych. Wtedy łamaną płaską będzie łamana wykreślona bez odrywania ołówka od papieru w kierunku od lewej do prawej i taka, że każda prosta zawierająca odcinek łamanej jest nachylona do osi OX pod kątem należącym do przedziału domkniętego $[0°, 90°]$.

Aby rozwiązać takie zmodyfikowane zadanie zastosujemy technikę zwaną **zamiataniem**, często spotykaną w algorytmach geometrycznych.

Sortujemy najpierw wszystkie punkty niemalejąco względem współrzędnych x, przy czym jeśli dwa punkty mają równe współrzędne x wtedy przyjmujemy, że wcześniejszy w porządku jest punkt o mniejszej współrzędnej y (posortować możemy algorytmem Heapsort lub innym algorytmem sortującym o złożoności $O(n \log n)$, zob. [12]).

Wysokością łamanej płaskiej będziemy nazywać wspólrzędną y ostatniego jej punktu. **i-tym prefiksem** zbioru łamanych płaskich nazwiemy początkowe fragmenty łamanych z tego zbioru, rozpięte na i pierwszych punktach (w określonym wyżej porządku). **i-tym przekrojem** rozwiązania nazwiemy malejący ciąg $a_1, \ldots, a_{p_i}$ wysokości łamanych w i-tym prefiksie zbioru łamanych będących rozwiązaniem zadania.

Nasz algorytm będzie przeglądał uporządkowane punkty tworząc kolejne przekroje rozwiązania optymalnego.

Przekrój zerowy jest oczywiście pusty.

Dla kolejnego punktu P_i o współrzędnych (x_i, y_i) znajdujemy w $(i-1)$-tym przekroju maksymalny element a_k spełniający nierówność $a_k \leqslant y_i$. Punkt P_i podczepimy do łamanej o tej właśnie wysokości — stanie się on jej ostatnim punktem, zatem i-ty przekrój powstanie z przekroju $(i-1)$-ego poprzez zamianę a_k na y_i.

Jeśli wszystkie łamane mają większe wysokości niż y_i, wówczas tworzymy nową łamaną złożoną tylko z tego jednego punktu, czyli przekrój i-ty otrzymujemy z przekroju $(i-1)$-ego poprzez dodanie elementu y_i na koniec ciągu.

Rozwiązaniem zadania jest długość n-tego przekroju.

Zauważmy, że kolejne przekroje możemy obliczać w jednej tablicy. Wyszukiwań łamanych do rozszerzenia możemy wtedy dokonywać binarnie w czasie $O(\log n)$. Takich wyszukiwań wykonamy dokładnie n, zatem łącznie ze wstępną fazą sortowania złożoność naszego algorytmu wynosi $O(n \log n)$.

Opisany powyżej algorytm należy do klasy algorytmów **zachłannych**, tzn. takich, które znajdują optymalne rozwiązanie dla danego problemu wykonując lokalne działania, które są w danej chwili najkorzystniejsze. Takie „intuicyjnie najlepsze" zachowanie algorytmu nie musi jednak zawsze prowadzić do rozwiązania optymalnego. Na szczęście opisany wyżej algorytm zawsze znajduje optymalne rozwiązanie, co udowodniamy poniżej.

Powiemy, że przekrój $a_1, \ldots, a_p$ jest **niższy** od przekroju $b_1, \ldots, b_r$, jeśli $p \leqslant r$ oraz istnieje podciąg $b'_1, \ldots, b'_p$ ciągu $b_1, \ldots, b_r$ taki, że dla każdego k zachodzi $a_k \leqslant b'_k$, tzn. dla każdej łamanej z przekroju $a_1, \ldots, a_p$ mamy łamaną o nie mniejszej wysokości w przekroju $b_1, \ldots, b_r$.

Pokażemy indukcyjnie, że dla dowolnego rozwiązania B (niekoniecznie optymalnego), i-ty przekrój rozwiązania A wyznaczanego przez nasz algorytm jest niższy od i-tego przekroju rozwiązania B.

Dla $i = 0$ powyższe stwierdzenie jest w oczywisty sposób prawdziwe.

Załóżmy więc, że $(i-1)$-szy przekrój $a_1, \ldots, a_p$ rozwiązania A jest niższy od $(i-1)$-ego przekroju $b_1, \ldots, b_r$ rozwiązania B. Teraz musimy rozpatrzeć 2 przypadki:

- punkt $P_i(x_i, y_i)$ jest podłączony w rozwiązaniu B do łamanej o wysokości b_k.
- punkt $P_i(x_i, y_i)$ w rozwiązaniu B jest początkiem nowej łamanej.

W obu z nich można w miarę łatwo pokazać, że i-ty przekrój rozwiązania A jest niższy od i-tego przekroju rozwiązania B. Pozostawiamy dowód tego faktu Czytelnikowi jako pouczające ćwiczenie.

Jak obliczyć współrzędne punktu (x', y') powstałego przez obrót punktu (x, y) o 45 stopni w lewo względem początku układu współrzędnych?

W dowolnej książce poświęconej geometrii analitycznej znajdujemy (lub lepiej: sami wyprowadzamy) wzory dla dowolnego kąta ϕ:

$$x' = x\cos\phi - y\sin\phi,$$
$$y' = x\sin\phi + y\sin\phi.$$

Ponieważ $\phi = 45^\circ$, to:

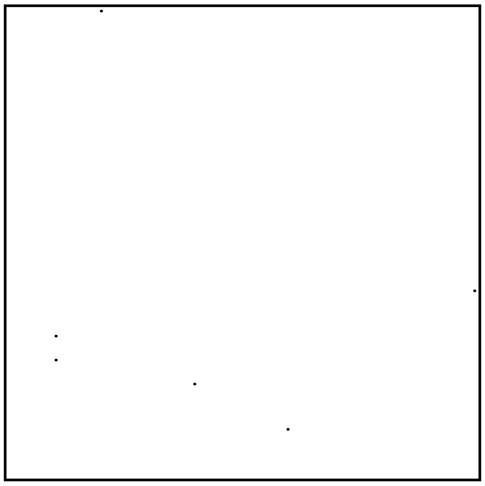
$$x' = \frac{\sqrt{2}}{2}(x-y),$$
$$y' = \frac{\sqrt{2}}{2}(x+y).$$

Zwykle we wszelkich algorytmach staramy się unikać (jeśli to możliwe) liczb rzeczywistych ze względu na ich niedokładną reprezentację. Z tego powodu skalujemy wszystkie punkty, dzieląc ich obie współrzędne przez $\sqrt{2}/2$. Ostatecznie mamy:

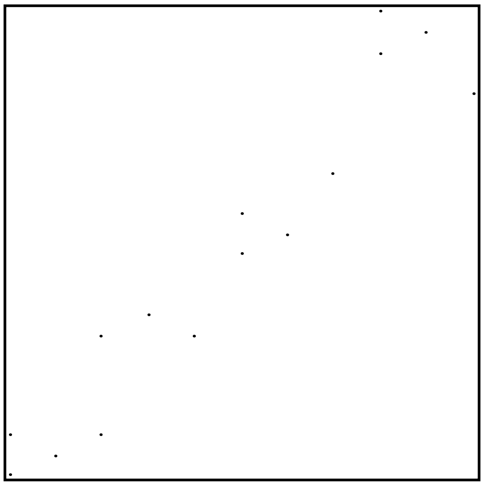
$$x' = x - y,$$
$$y' = x + y.$$

TESTY

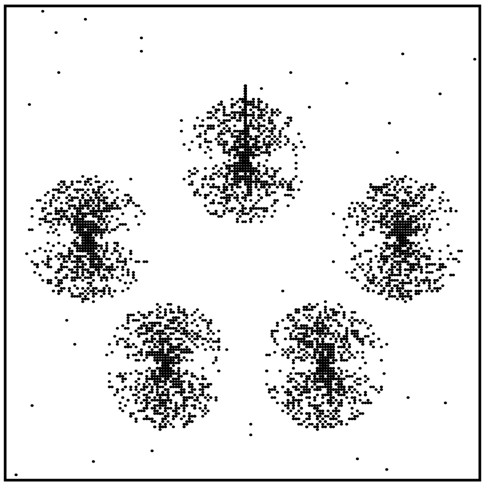
Do sprawdzenia rozwiązań zawodników użyto 11-tu testów LAM0.IN–LAM10.IN. Test LAM0.IN był testem z treści zadania. Test LAM1.IN był prostym testem poprawnościowym, test LAM2.IN — prostym testem wydajnościowym, test LAM3.IN — testem losowym. Punkty w teście LAM4.IN tworzyły napis „Olimpiada Informatyczna". Pozostałe testy były trudnymi testami wydajnościowymi. Poniżej przedstawiamy schematy ułożenia punktów w poszczególnych testach.



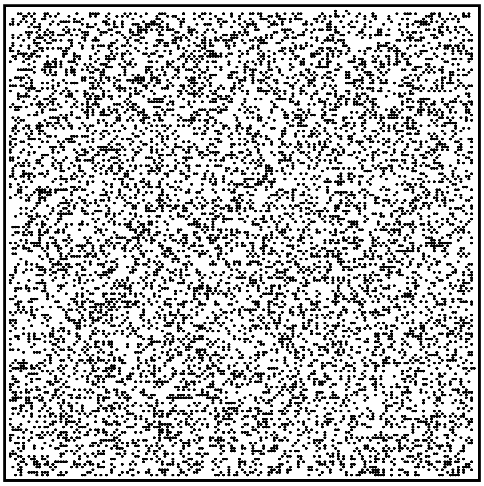
LAM0.IN $n = 6$



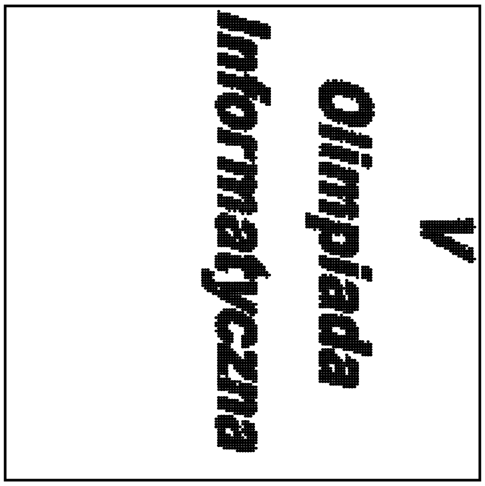
LAM1.IN $n = 15$



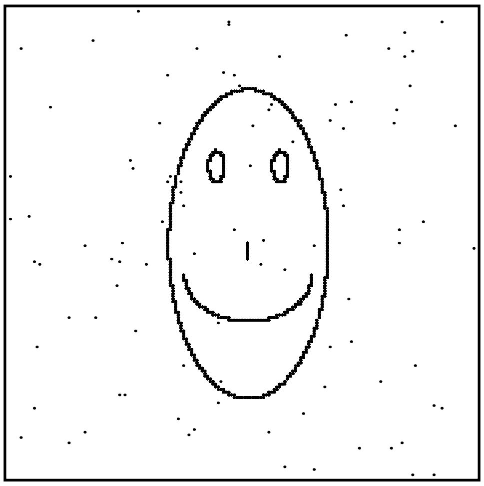
LAM2.IN $n = 5000$



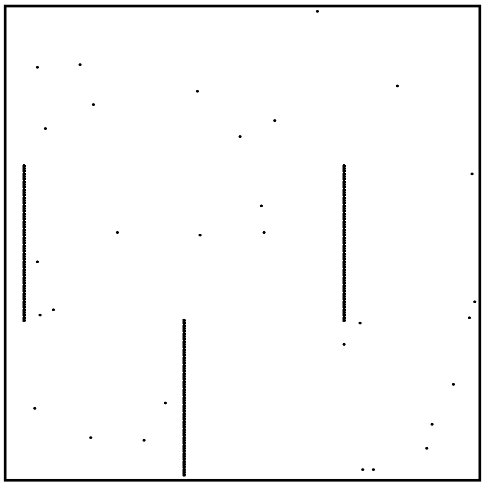
LAM3.IN $n = 10000$



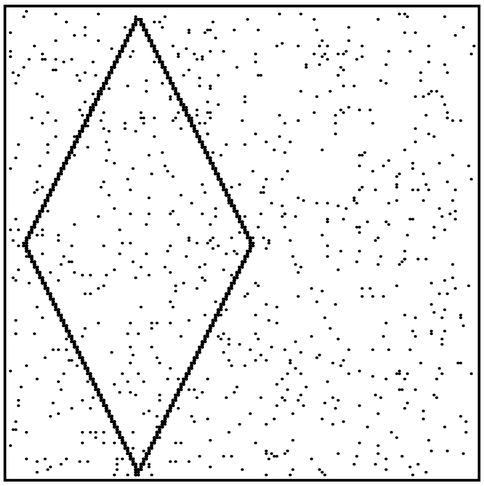
LAM4.IN $n = 15000$



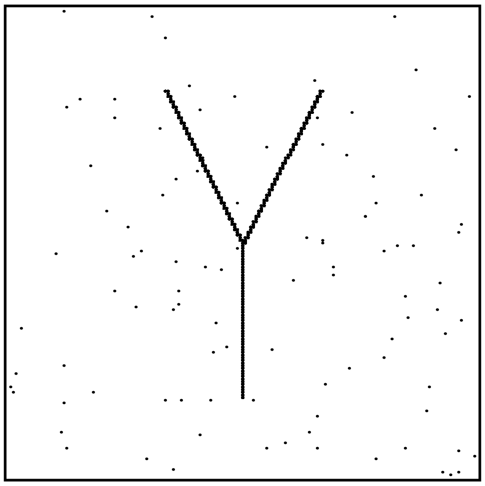
LAM5.IN $n = 17000$



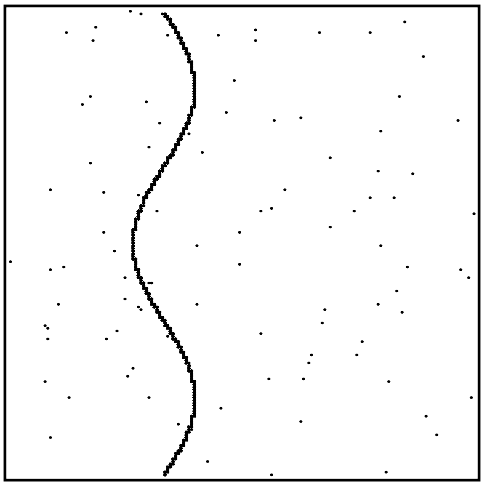
LAM6.IN $n = 30000$



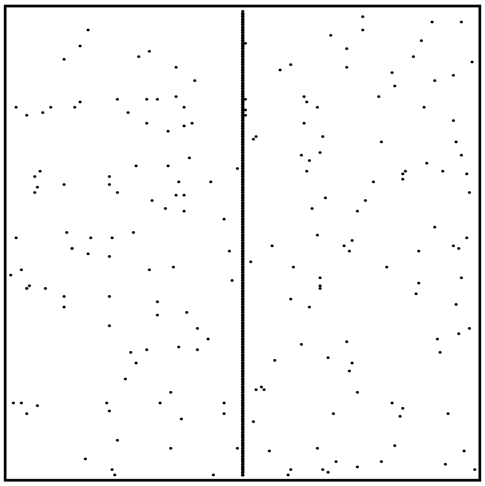
LAM7.IN $n = 30000$



LAM8.IN $n = 30000$



LAM9.IN $n = 30000$



LAM10.IN $n = 30000$

Prostokąty

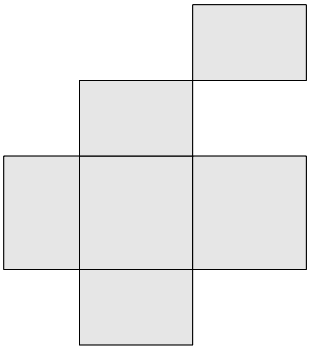
Na płaszczyźnie narysowano n prostokątów, których boki są równoległe do osi współrzędnych i wierzchołki mają obie współrzędne całkowite.

Przyjmujemy że:

- *każdy prostokąt jest blokiem,*
- *jeśli dwa różne bloki mają wspólny odcinek to tworzą one nowy blok, w przeciwnym przypadku mówimy, że są rozłączne.*

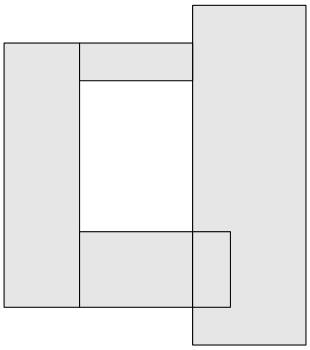
Przykłady

Prostokąty na rysunku a tworzą dwa rozłączne bloki.



Rys. a

Prostokąty na rysunku b tworzą jeden blok.



Rys. b

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego PRO.IN liczbę prostokątów oraz współrzędne ich wierzchołków;*
- *znajduje liczbę rozłącznych bloków utworzonych przez te prostokąty;*
- *zapisuje wynik w pliku tekstowym PRO.OUT.*

Wejście

W pierwszym wierszu pliku wejściowego PRO.IN znajduje się liczba całkowita n, $1 \leqslant n \leqslant 7000$. Jest to liczba prostokątów.

W następnych n wierszach są zapisane współrzędne wierzchołków prostokątów. Każdy prostokąt jest opisany za pomocą czterech liczb: współrzędnych x i y lewego dolnego wierzchołka oraz współrzędnych x i y prawego górnego wierzchołka. Są to liczby całkowite nieujemne nie większe niż 10000.

Wyjście

W pierwszym i jedynym wierszu pliku PRO.OUT należy zapisać jedną liczbę całkowitą: liczbę rozłącznych bloków utworzonych przez dane prostokąty.

Przykład

Dla pliku tekstowego PRO.IN:

```
9
0 3 2 6
4 5 5 7
4 2 6 4
2 0 3 2
5 3 6 4
3 2 5 3
1 4 4 7
0 0 1 4
0 0 4 1
```

poprawnym rozwiązaniem jest plik wyjściowy PRO.OUT:

```
2
```

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

UZUPEŁNIENIE TREŚCI ZADANIA

Podczas trwania zawodów treść zadania uzupełniono następującym stwierdzeniem:

Każdy prostokąt ma dodatnie pole.

ROZWIĄZANIE

Zadanie można rozwiązać na kilka sposobów. Na początku omówimy rozwiązania polegające na przeszukiwaniu grafów. Zacznijmy od opisania grafu. Wierzchołkami będą podane prostokąty. Powiemy, że dwa prostokąty są połączone krawędzią, jeśli mają wspólny odcinek (tzn. gdy tworzą blok). Każdy graf rozpada się na tzw. *składowe*. Składową grafu nazywamy taki jego maksymalny podgraf, w którym każde dwa wierzchołki można połączyć ścieżką. Zauważmy, że dwa prostokąty wchodzą w skład tego samego bloku, jeśli w tak skonstruowanym grafie istnieje między nimi ścieżką. Zatem szukana liczba bloków jest równa liczbie składowych skonstruowanego przez nas grafu. Tę liczbę składowych można wyznaczyć za pomocą procedur przeszukiwania grafów.

Jedną z metod przeszukiwania grafów jest metoda przeszukiwania grafów w głąb (ang. *Depth First Search*, w skrócie *DFS*). Polega ona na tym, że najpierw zaznaczamy w specjalnej tablicy, że żadnego wierzchołka dotychczas nie odwiedziliśmy, a następnie rozpoczynamy przeszukiwanie. Wybieramy wierzchołek, którego dotychczas nie odwiedziliśmy, zaznaczamy, że został już odwiedzony i wyruszamy z niego do któregokolwiek jego nie odwiedzonego sąsiada. Zaznaczamy, że tego sąsiada odwiedziliśmy i podążamy dalej. Jeśli dojdziemy do wierzchołka, którego wszystkich sąsiadów już odwiedziliśmy, musimy się cofnąć i poszukać nowego wierzchołka, którego nie odwiedziliśmy dotychczas. Robimy to tak długo, aż odwiedzimy wszystkie wierzchołki dostępne z wierzchołka początkowego. Wtedy zapisujemy „na koncie" jedną składową i szukamy wierzchołka, którego nie odwiedziliśmy. Zaznaczamy, że został odwiedzony i ponawiamy przeszukiwanie. Po jego zakończeniu zaznaczamy „na koncie" drugą składową i powtarzamy to postępowanie dotąd, aż odwiedzimy wszystkie wierzchołki. Najprościej można zaimplementować ten algorytm za pomocą następujacej procedury rekurencyjnej:

```
procedure DFS (v : Wierzcholek);
  var
    w : Wierzchołek;
  begin
    Odwiedzony[v] := true;
    for w in Sasiedzi[v] do
      if not Odwiedzony[w] then DFS(w)
  end;
procedure SzukajDFS (var liczba : integer);
  var
    v : Wierzcholek;
  begin
    forall v do Odwiedzony[v] := false;
```

```
    liczba := 0;
    forall v do
      if not Odwiedzony[v] then begin
        DFS(v);
        liczba := liczba + 1
      end
  end;
```

Można również przeszukiwać graf wszerz (ang. *Breadth First Search*, w skrócie *BFS*). Przede wszystkim tworzymy kolejkę — listę, do której dopisujemy kolejne elementy z jednej strony i pobieramy z drugiej, tak jak to się dzieje w kolejce do sklepu (Starsi jeszcze pamiętają to zjawisko!). Do kolejki pustej dopisujemy pierwszy wierzchołek. Teraz zaczynamy przeszukiwanie. Robimy to w ten sposób, że wybieramy pierwszy wierzchołek, zaznaczamy, że został już odwiedzony i dołączamy do kolejki tych jego sąsiadów, którzy nie zostali jeszcze odwiedzeni. Potem pobieramy pierwszy wierzchołek z kolejki i dołączamy do kolejki wszystkich nieodwiedzonych sąsiadów tego wierzchołka itd. W chwili, gdy kolejka stanie się pusta, odwiedzone zostaną wszystkie wierzchołki znajdujące się w tej samej składowej co wierzchołek początkowy. Zapisujemy „na koncie” jedną składową, wybieramy dowolny nieodwiedzony wierzchołek i powtarzamy procedurę. Można ten sposób przeszukiwania zapisać jak następuje:

```
procedure BFS(v : Wierzcholek);
  var
    K : Kolejka;
    u, w : Wierzcholek;
  begin
    K := PUSTA;
    Odwiedzony[v] := true;
    Dopisz(v, K);
    while K ≠ PUSTA do begin
      u := Pobierz(K);
      for w in Sasiedzi[u] do
        if not Odwiedzony[w] then begin
          Odwiedzony[w] := true;
          Dopisz(w, K)
        end
    end
  end;
procedure SzukajBFS(var liczba : integer);
  var
    v : Wierzcholek;
  begin
  for v do Odwiedzony[v] := false;
    liczba := 0;
    forall v do
      if not Odwiedzony[v] then begin
        BFS(v);
```

```
            liczba := liczba + 1
        end
    end;
```

Oczywiście procedura *Dopisz*(v, K) dopisuje wierzchołek v na końcu kolejki K, a funkcja *Pobierz*(K) zwraca pierwszy wierzchołek kolejki i usuwa go z niej. Zauważmy, że przedstawiona wyżej procedura *SzukajBFS* niewiele różni się od procedury *SzukajDFS*.

W obu metodach przeszukiwania problemem jest zbudowanie grafu w pamięci. Przy maksymalnej liczbie 7000 prostokątów liczba krawędzi może sięgać nawet 49 milionów! Będzie tak, na przykład wtedy, gdy wszystkie prostokąty będą identyczne. Oczywiście tak dużego grafu nie da się zbudować w pamięci operacyjnej i program próbujący to zrobić zakończy się błędem. Budowanie grafu w pamięci operacyjnej nie jest jednak konieczne. Sąsiadów danego wierzchołka szukamy przecież na bieżąco, przeszukując wszystkie wierzchołki. Zauważmy, że nigdzie nie wykorzystujemy krawędzi grafu w inny sposób, niż do stwierdzenia, czy rozważany wierzchołek jest sąsiadem wierzchołka, którym aktualnie się zajmujemy. Taki sposób implementacji wymaga już znacznie mniej pamięci, choć jego czas działania pozostanie taki sam. Ten sposób implementacji, w którym budujemy graf w pamięci operacyjnej, wymaga czasu rzędu n^2 i takiego samego rzędu pamięci, drugi, w którym graf budujemy „na bieżąco", takiego samego czasu, ale pamięci tylko rzędu n. Jest to znaczna poprawa; taki program będzie mógł pomyślnie zakończyć działanie nawet dla największych spodziewanych danych.

Następna metoda rozwiązania zadania polega na wykorzystaniu nowej struktury danych, tzw. zbiorów rozłącznych. Na początku wczytujemy do pamięci dane wszystkich prostokątów. Następnie tworzymy tablicę S typu **array**[1..*MaxLiczbaProstokatow*] **of integer** i całą ją wypełniamy liczbą -1. Wartość $S[i] = -1$ oznacza, że i-ty prostokąt stanowi samodzielny blok. Na początku przyjmujemy, że tak jest. Potem zbadamy wszystkie możliwe połączenia bloków. Przypuśćmy więc, że znajdziemy pierwsze połączenie bloków: niech prostokąt drugi stanowi jeden blok z prostokątem piątym. Wtedy zmienimy jedną wartość: albo położymy $S[2]{:=}5$, albo $S[5]{:=}2$. Chwilowo jest to obojętne. W każdym przypadku tylko jeden prostokąt ma przypisaną liczbę -1, drugi zaś ma przypisany **numer** tego pierwszego prostokąta. Ten prostokąt, który ma przypisaną liczbę -1, nazwiemy **głową** bloku. Przyjmijmy dla ustalenia uwagi, że głową tego bloku jest prostokąt o numerze 2. Wtedy $S[2] = -1$ oraz $S[5] = 2$. Niech następnie okaże się, że prostokąty piąty i czwarty tworzą jeden blok (na razie nie zajmujemy się kolejnością, w jakiej wykrywamy te połączenia mniejszych bloków w większe; do tej kwestii jeszcze powrócimy). Wtedy szukamy głów obu bloków, w których są te dwa prostokąty. Dla prostokąta czwartego jest to łatwe: równość $S[4] = -1$ świadczy o tym, że jest on głową bloku, którego jest częścią. Dla prostokąta piątego jest trudniej: najpierw stwierdzamy, że $S[5] = 2$, co znaczy, iż prostokąt piąty jest w tym samym bloku co prostokąt drugi. Teraz równość $S[2] = -1$ pokazuje, że głową tego bloku jest prostokąt drugi. Wreszcie łączymy te dwa bloki. Możemy znów uczynić to na dwa sposoby: albo przypiszemy wartość $S[2] := 4$, czyniąc czwarty prostokąt głową tego bloku, albo przypiszemy $S[4] := 2$, dołączając czwarty prostokąt do bloku, którego głową jest nadal prostokąt drugi. Oba rozwiązania wydają się być równoważne.

Po przebadaniu wszystkich możliwych połączeń bloków, tzn. po sprawdzeniu, czy i-ty prostokąt graniczy z j-tym prostokątem dla każdej pary różnych liczb $\{i, j\}$, w tablicy S będziemy mieli tyle liczb -1, ile jest bloków.

Zastanówmy się teraz, jakie czynności będą wykonywane w czasie uaktualniania tablicy S. Jedną z tych czynności będzie operacja *Find*(i), która prostokątowi o numerze i przypisze numer głowy bloku, którego częścią jest i-ty prostokąt. Druga czynność *Union*(i,j) polega na łączeniu w jeden dwóch bloków, których głowy mają numery i oraz j. Ta druga czynność jest łatwa: możemy albo przypisać $S[i]{:=}j$, albo $S[j]{:=}i$. Pierwsza jest realizowana za pomocą procedury:

```
function Find(i : integer) : integer;
  var
    j : integer;
  begin
    j := S[i];
    while j > 0 do begin
      i := j;
      j := S[i]
    end;
    Find := i
  end;
```

Nadszedł moment, w którym warto się zastanowić nad czasem wykonania poszczególnych czynności. Procedura *Union* jest wykonywana w stałym czasie. Procedura *Find* jest wykonywana w czasie proporcjonalnym do długości drogi (w tablicy S) przebywanej od i do głowy bloku. Okazuje się, że ta droga nie będzie wydłużała się zbytnio podczas wykonywania procedury *Union*, jeśli będziemy zawsze „podczepiać” mniejszy blok do głowy większego, przy czym przez mniejszy rozumiemy ten, w którym najdłuższa droga do głowy jest krótsza niż w drugim bloku. Niech zatem ujemna wartość $S[i]$ oznacza liczbę przeciwną do długości najdłuższej drogi do głowy w bloku, którego głową jest i-ty prostokąt. Na początku więc oczywiście $S[i] = -1$ dla wszystkich i. Potem procedura *Union* przybierze postać:

```
procedure Union(i, j : integer);
  begin
    if S[j] < S[i] then
      S[i] := j
    else begin
      if S[j] = S[i] then
        {wydłużamy drogę o 1}
        S[i] := S[i] - 1;
      S[j] := i
  end;
```

Wreszcie powiedzmy słowo o kolejności sprawdzania. Ponieważ trzeba sprawdzić wszystkie pary, więc możemy to zrobić za pomocą podwójnej pętli:

```
for i:=1 to LiczbaProstokatow do begin
  S[i] := −1;
  for j:=1 to i − 1 do
    Union(Find(i), Find(j))
end
```

Po wykonaniu tej pętli wystarczy zliczyć wszystkie głowy, tzn. wystarczy policzyć ile liczb ujemnych występuje w tablicy S. Ten sposób implementacji jest dość szybki (jest rzędu $n^2 \log n$) i pomimo, że wydaje się być wolniejszy od poprzednich (o czynnik $\log n$), daje szybko działające programy — między innymi dzięki swej prostocie. Warto jeszcze wspomnieć o tym, że podczas wykonywania procedury *Find* możemy zmodyfikować wszystkie wartości tablicy S na drodze od i do głowy, przypisując każdemu podczepionemu prostokątowi numer głowy bloku. To bardzo przyspieszy następne wyszukiwania i w praktyce da program nawet nieco szybszy od programów wykorzystujących metody teorii grafów.

INNE ROZWIĄZANIA

Warto tu zauważyć, że istnieje jeszcze inne rozwiązanie zadania, działające w czasie bliskim $O(n \log n)$. Nie będziemy go jednak opisywać ze względu na duży stopień zaawansowania wykorzystanych w nim technik i struktur danych. Z tych samych przyczyn tego rozwiązania nie brano pod uwagę przy ustalaniu czasów odcięcia dla testów.

TESTY

Oprócz przykładu z treści zadania, zastosowano 10 testów. Test PRO1.IN miał na celu sprawdzenie poprawności w sytuacji, w której pewne prostokąty stykały się tylko rogami lub fragmentami brzegów. Test PRO2.IN to „szachownica” wymiaru 10×10. Testy PRO3.IN–PRO6.IN obejmowały stosunkowo nieduże układy różnie rozmieszczonych prostokątów. Testy PRO7.IN–PRO10.IN obejmowały układy duże, po ok. 5000 prostokątów.

IX Międzynarodowa Olimpiada Informatyczna Kapsztad, grudzień 1997

teksty zadań

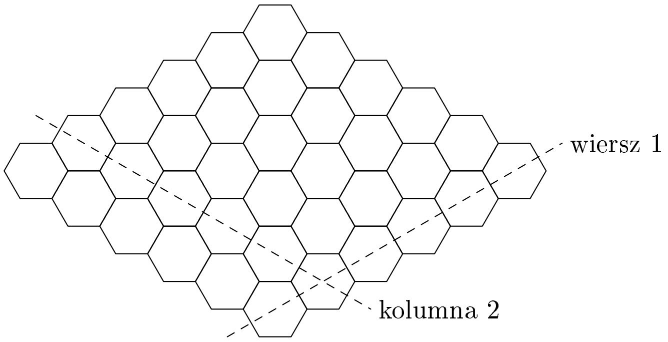
Krzysztof Diks, Andrzej Walat
przekład

Gra Hex

Celem gry dla pierwszego gracza jest ustawienie swoich pionów na planszy w taki sposób, aby „związać” kolumnę 1 z kolumną N.

Reguły gry

Hex *jest strategiczną grą dwuosobową rozgrywaną na planszy w kształcie rombu o rozmiarach $N \times N$. Pola na planszy są sześciokątami foremnymi. Rysunek przedstawia planszę dla $N = 6$.*



(1) *Graczami są Twój program i biblioteka testująca.*

(2) *Twój program zawsze wykonuje pierwszy ruch.*

(3) *Gracze na zmianę umieszczają swoje piony na planszy.*

(4) *Piona można postawić tylko na wolnym polu.*

(5) *Dwa pola są sąsiednie, jeśli mają wspólną krawędź.*

(6) *Piony tego samego gracza na sąsiednich polach sa „związane”.*

(7) *Relacja wiązania jest przechodnia (i przemienna): jeśli pion1 jest związany z pionem2 i pion2 jest związany z pionem3, to pion 1 jest związany z pionem3.*

Zadanie

- *Napisz program grający w Hex.*
- *Celem gracza 1 (Twojego programu) jest związanie swojego piona w kolumnie 1 ze swoim pionem w kolumnie N.*
- *Drugi gracz (program testujący) próbuje związać swój pion z wiersza 1 ze swoim pionem z wiersza N.*

- *Jeśli Twój program gra optymalnie, to zawsze wygra.*

Wejście i wyjście

Twój program nie może ani czytać z żadnego pliku, ani pisac do żadnego pliku. Nie może czytać z klawiatury, ani wypisywać niczego na ekran. Wszystkie dane wejściowe będzie otrzymywał za pomocą funkcji z biblioteki HexLib. Ta biblioteka utworzy plik HEX.OUT, którego zawartość powinieneś zignorować.

Na początku Twojej rozgrywki z komputerem stan planszy przedstawia stan gry (pewne pola planszy mogą być zajęte), którą Twój program może jeszcze wygrać. Twój program musi korzystać z funkcji `GetMax` *i* `LookAtBoard`*, by ustalić stan gry. Na początku rozgrywki liczby pionów obu graczy na planszy są takie same.*

Ograniczenia

(1) *Rozmiar planszy (N) będzie zawsze należał do przedziału od 1 do 20 (włącznie).*

(2) *Twój program może potrzebować do 200 ruchów, by zakończyć grę. Cała rozgrywka musi się zakończyć w 40 sekund. Gwarantuje się, że program testujący wykorzysta co najwyżej 20 sekund.*

Biblioteka

Gotową bibliotekę HexLib musisz skonsolidować ze swoim kodem. Dla każdego języka programowania, w katalogu zadania znajduje się przykładowy plik (`TESTHEX.CPP`*,* `TESTHEX.C`*,* `TESTHEX.PAS` *i* `TESTHEX.BAS`*) pokazujący, jak to zrobić.*

Biblioteka HexLib zawiera następujące funkcje i procedury (odpowiednio: Pascal, C/C++ i Basic):

```
function LookAtBoard (row, column: integer): integer;
int LookAtBoard (int row, int column);
declare function LookAtBoard cdecl (byval x as integer, byval y
as integer)
```

Zwraca

- -1*, jeśli row* < 1*, lub row* $> N$*, lub column* < 1*, lub column* $> N$*,*
- *0, jeśli nie ma piona na tej pozycji,*
- *1, jeśli pion na danej pozycji należy do Twojego programu (gracz 1),*
- *2, jeśli pion na danej pozycji należy do programu testującego (gracz 2).*

```
procedure PutHex (row, column: integer);
void PutHex (int row, int column);
declare sub PutHex cdecl (byval x as integer, byval y as integer)
```

Umieszcza Twojego piona na wskazanej pozycji, jeśli jest wolna.

```
function GameIsOver: integer;
int GameIsOver (void);
declare function GameIsOver cdecl ()
```

Zwraca jedną z następujących wartości

- *0 — gra się nie zakończyła,*
- *1 — na każdym polu planszy stoi pion,*
- *2 — Twój program wygrał,*
- *3 — program testujący wygrał.*

```
procedure MakeLibMove;
void MakeLibMove(void);
declare sub MakeLibMove cdecl ()
```

Umożliwia programowi testującemu obliczenie następnego ruchu i umieszczenie swojego piona na planszy. Zmiana na planszy będzie pokazywana przez `LookAtBoard` *i inne funkcje.*

```
function GetRow: integer;
int GetRow (void);
declare function GetRow cdecl ()
```

Zwraca numer wiersza, w którym program testujący umieścił piona, lub -1, jeśli żaden pion nie został jeszcze postawiony. Ta funkcja zawsze zwraca tę samą wartość, aż do ponownego wywołania `MakeLibMove`.

```
function GetColumn: integer;
int GetColumn (void);
declare function GetColumn cdecl ()
```

Zwraca numer kolumny, w której program testujący umieścił piona, lub -1, jeśli żaden pion nie został jeszcze postawiony. Ta funkcja zawsze zwraca tę samą wartość, aż do ponownego wywołania `MakeLibMove`.

```
function GetMax: integer;
int GetMax (void);
declare function GetMax cdecl ()
```

Zwraca rozmiar planszy N.

Punktacja

- *Jeśli Twój program wygra, zdobywasz całą pulę punktów za rozgrywkę.*
- *Jeśli Twój program przegra, zdobywasz 20% puli punktów za rozgrywkę.*
- *Jeśli Twój program zatrzyma się przed zakończeniem gry lub przekroczy limit czasu, otrzymujesz 0 punktów.*

Krzysztof Diks, Andrzej Walat
przekład

Rozpoznawanie znakow

Ten problem wymaga napisania programu rozpoznającego znaki.

Szczegóły

Każdy idealny obraz znaku składa sie z 20-tu wierszy, po 20 cyfr '0' lub '1' w każdym wierszu. Spójrz na układ obrazów znaków w pliku na rys. 1a.

Plik FONT.DAT zawiera reprezentacje 27 idealnych obrazów znaków w następującym porządku:

␣abcdefghijklmnopqrstuvwxyz

gdzie ␣ reprezentuje znak odstępu.

Plik IMAGE.DAT zawiera jeden lub więcej, być może zakłóconych, obrazów znaków. Obraz znaku może być zakłócony na następujące sposoby:

- *co najwyżej jeden wiersz może być zdublowany (zdublowane wiersze występują jeden po drugim),*
- *co najwyżej jeden wiersz może zostać opuszczony,*
- *pewne zera mogły zostać zamienione na jedynki,*
- *pewne jedynki mogły zostać zamienione na zera.*

W żadnym obrazie znaku nie występuje jednocześnie zdublowanie i opuszczenie wiersza. W każdym obrazie znaku w testach liczba przekłamanych zer i jedynek nie przekracza 30%.

W przypadku zdublowania wiersza, w każdym z obu wierszy mogą wystąpić przekłamania znaków, niekoniecznie w tym samym miejscu.

Zadanie

Napisz program rozpoznający (na podstawie wzorców czcionek zapisanych w pliku FONT.DAT) ciąg złożony z jednego lub więcej znaków w obrazie zapisanym w pliku IMAGE.DAT.

Rozpoznając znaki w obrazie dobieraj je w taki sposób, żeby było możliwe przekształcenie ciągu odpowiadających im wzorców czcionek na dany (zakłócony) obraz za pomocą minimalnej łącznej liczby przestawień cyfr '1' lub '0'. Możesz przy tym przyjąć najbardziej korzystne założenie dotyczące dublowania i opuszczania wierszy. W przypadku zdublowania wierszy licz zakłócenia tylko w mniej zniekształconym wierszu. Dobrze napisany program powinien rozpoznać wszystkie znaki w danych testowych. Istnieje jedno najlepsze rozwiązanie dla każdego zestawu danych testowych.

Poprawne rozwiązanie wykorzysta dokładnie wszystkie dane zapisane w pliku IMAGE.DAT.

Wejście

Oba pliki danych zaczynają się od liczby całkowitej N ($19 \leqslant N \leqslant 1200$), która określa liczbę następnych wierszy pliku:

```
N
(znak-1)(znak-2)(znak-3)...(znak20)
(znak-1)(znak-2)(znak-3)...(znak20)
(znak-1)(znak-2)(znak-3)...(znak20)
(znak-1)(znak-2)(znak-3)...(znak20)
(znak-1)(znak-2)(znak-3)...(znak20)
(znak-1)(znak-2)(znak-3)...(znak20)
(znak-1)(znak-2)(znak-3)...(znak20)
(znak-1)(znak-2)(znak-3)...(znak20)
(znak-1)(znak-2)(znak-3)...(znak20)
(znak-1)(znak-2)(znak-3)...(znak20)
...
```

Każdy z tych wierszy zawiera 20 znaków. Nie ma żadnych odstępów oddzielających zera i jedynki.

Plik FONT.DAT zawiera wzorce czcionek i zawsze ma 541 wierszy. W każdym teście jego zawartość może być inna.

Punktacja

Punkty będą procentem rozpoznanych znaków.

Wyjście

Twój program musi utworzyć plik IMAGE.OUT, zawierający jeden napis złożony z rozpoznanych znaków. Powinien on mieć format jednego wiersza tekstu ASCII. W pliku wyjściowym nie powinno być żadnych separatorów. Jeśli Twój program nie może rozpoznać jakiegoś znaku, powinien na odpowiedniej pozycji wypisać „?”.

Przykładowe pliki danych: przedstawiono niekompletny przykład pokazujący początek pliku FONT.DAT (spacja i `a`*) oraz przykład pliku IMAGE.DAT (zakłócone* `a`*).*

FONT.DAT

```
540
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000011100000000000
00000111110110000000
00001111110011000000
00001110001100100000
00001100001100010000
00001100000100010000
00000100000100010000
00000010000000110000
00000001000001110000
00001111111111110000
00001111111111110000
00001111111111000000
00001000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
```

Rys. 1a

IMAGE.DAT

```
19
00000000000000000000
00000000000000000000
00000000000000000000
00000011100000000000
00100111011011000000
00001111110011000000
00001110001100100000
00001100001100010000
00001100000100010000
00000100000100010000
00000010000000110000
00001110111111110000
00001111111111110000
00001111111111000000
00001000010000000000
00000000000000000000
00000000000001000000
00000000000000000000
00000000000000000000
```

Rys. 1b

Przykładowe wyjście:

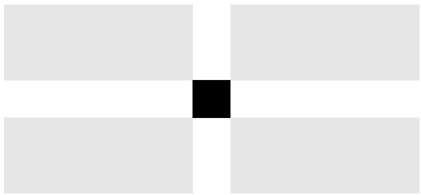
IMAGE.OUT	**Objaśnienie**
a	*Rozpoznano pojedynczy znak* `a`

Krzysztof Diks, Andrzej Walat
przekład

Opisywanie mapy

Jesteś pomocnikiem kartografa i otrzymałeś zadanie naniesienia nazw miast na nowej mapie.

Mapa jest siatką 1000×1000 pól. Każde miasto zajmuje jedno pole na mapie. Nazwy miast należy umieścić na mapie w prostokątnych ramkach utworzonych z pól siatki. Taką ramkę nazywamy etykietą.



Rys. 1 Miasto i cztery dozwolone położenia etykiety

Rozmieszczenie etykiet musi spełniać następujące warunki:

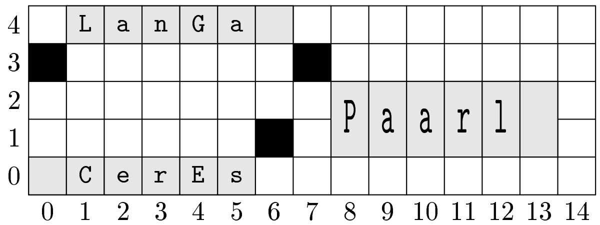
(1) Etykieta miasta musi zajmować jedną z czterech pozycji wskazanych na rysunku 1.

(2) Etykiety nie mogą na siebie nachodzić.

(3) Etykiety nie mogą nachodzić na pola reprezentujące miasta.

(4) Etykiety nie mogą wychodzić poza mapę.

Każda etykieta zawiera wszystkie litery nazwy miasta plus pojedynczą spację. Dla każdej nazwy miasta dana jest szerokość i wysokość tworzacych ją znaków; pojedyncza spacja ma taki sam rozmiar.



Rys. 2 Wycinek mapy

Kolumny siatki są ponumerowane od lewej do prawej kolejnymi liczbami, poczynając od 0. Wiersze są ponumerowane od dołu do góry, również poczynając od 0.

Rysunek 2 przedstawia lewy dolny róg mapy z miastami Langa na pozycji (0,3), Ceres na pozycji (6,1) i Pearl na pozycji (7,3). Wszystkie etykiety są umieszczone prawidłowo, ale nie jest to jedyne prawidłowe ich rozmieszczenie.

Zadanie

Napisz program, który wczytuje pozycje miast na mapie wraz z wielkościami liter i nazwami tych miast. Twój program powinien rozmieścić nazwy na mapie zgodnie z podanymi warunkami i wypisać pozycje etykiet.

Wejście

Pierwszy wiersz pliku wejściowego zawiera liczbę miast na mapie (N). W każdym z kolejnych N wierszy są zapisane następujące dane: pozycja miasta na mapie – numer kolumny (X) i numer wiersza (Y); rozmiar liter tworzących nazwę miasta — szerokość (W) i wysokość (H), które są liczbami całkowitymi; nazwa miasta. Nazwy miast są pojedynczymi słowami. Liczba miast jest nie większa niż 1000. Nazwa miasta nie może mieć więcej niż 200 znaków.

Przykładowe wejście:

MAPS.DAT	**Objaśnienie:**
3	$N = 3$
0 3 1 1 Langa	$X = 0,\ Y = 3,\ W = 1,\ H = 1$
6 1 1 1 Ceres	
7 3 1 2 Paarl	

Wyjście

Twój program powinien zapisać N wierszy w pliku wyjściowym. Każdy wiersz odpowiada jednemu miastu z pliku wejściowego i musi zawierać: numer kolumny i numer wiersza górnego lewego pola etykiety tego miasta. Jeśli Twój program nie może ulokować etykiety, to musi wypisać parę liczb -1 -1. Porządek wierszy w pliku wyjściowym odpowiada porządkowi miast w pliku wejściowym.

Przykładowe wyjście:

MAPS.OUT	**Objaśnienie:**
1 4	*etykieta miasta Langa na pozycji $(1, 4)$*
0 0	*etykieta miasta Ceres na pozycji $(0, 0)$*
8 2	*etykieta miasta Paarl na pozycji $(8, 2)$*

Punktacja

Dla każdego zestawu danych testowych:

- *Wynik punktowy będzie procentem liczby prawidłowo rozmieszczonych etykiet w stosunku do wzorcowego rozwiązania organizatorów.*
- *Minimalna liczba punktów to 0%, maksymalna 100%.*
- *Jeśli rozmieszczenie jakiejkolwiek etykiety narusza ograniczenia, Twój program otrzyma 0 pkt.*
- *Jeśli etykiety nie odpowiadają miastom, Twój program otrzyma 0 pkt.*

Krzysztof Diks, Andrzej Walat
przekład

Wyprawa na Marsa

W ramach przygotowań do misji kosmicznej planuje się, że na powierzchni Marsa wyląduje pojemnik zawierający pewną liczbę pojazdów marsjańskich.

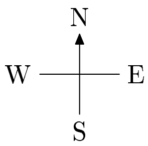
Wszystkie te pojazdy wyruszą z miejsca lądowania pojemnika i będą się kierowały do nadajnika, który wyląduje w pobliżu. Pojazdy przemieszczając się w kierunku nadajnika mają za zadanie pobierać próbki skał marsjańskich. Każda próbka może być pobrana tylko raz, przez pierwszy pojazd, który ją napotka. Inne pojazdy mogą przejeżdżać przez to samo miejsce.

Pojazdy nie mogą wjeżdżać na niebezpieczne obszary.

Pojazd marsjański jest skonstruowany w taki sposób, że może poruszać się tylko w dwóch kierunkach — południowym lub wschodnim, a jego trasa musi przebiegać przez pola prostokątnej siatki, od miejsca wylądowania pojemnika do miejsca, w którym znajduje się nadajnik. Na tym samym polu może się w tej samej chwili znajdować wiele pojazdów.

pojemnik →	0	0	0	0	0	0	0	0	0	0	wiersz 1
	0	0	0	0	0	1	1	0	0	0	
	0	0	0	1	0	2	0	0	0	0	
	1	1	0	1	2	0	0	0	0	1	
	0	1	0	0	2	0	1	1	0	0	
	0	1	0	1	0	0	1	1	0	0	
	0	1	2	0	0	0	0	1	0	0	
	0	0	0	0	0	0	0	0	0	0	wiersz 8

nadajnik



UWAGA: Jeśli pojazd nie może wykonać dozwolonego ruchu zanim osiągnie nadajnik, wszystkie pobrane przez niego próbki są stracone bezpowrotnie.

Zadanie

Zaplanuj ruchy pojazdów tak, aby uzyskać jak najwięcej punktów, maksymalizując liczbę zebranych próbek skalnych oraz liczbę pojazdów marsjańskich, które dotrą do nadajnika.

Wejście

Powierzchnię planety pomiędzy pojemnikiem i nadajnikiem reprezentuje siatka o rozmiarach $P \times Q$, na której pojemnik zajmuje zawsze pole $(1,1)$, a nadajnik — pole (P,Q). Rodzaj pola określa się następująco:

- *Obszar czysty: 0*
- *Obszar niebezpieczny: 1*
- *Próbka skały: 2*

Plik wejściowy zawiera następujące dane:

```
liczba_pojazdów_w_pojemniku
P
Q
(X1 Y1) (X2 Y1) (X3 Y1)... (XP Y1)
(X1 Y2) (X2 Y2) (X3 Y2)... (XP Y2)
(X1 Y3) (X2 Y3) (X3 Y3)... (XP Y3)
...
(X1 YQ) (X2 YQ) (X3 YQ)... (XP YQ)
```

P i Q to rozmiary siatki (liczba kolumn i liczba wierszy) nie większe niż 128, liczba_pojazdów_w_pojemniku jest liczbą całkowitą mniejszą od 1000, każdy z Q wierszy reprezentuje jeden wiersz siatki. Przykładowe wejście:

MARS.DAT	**Objaśnienie:**
2	*Liczba pojazdów*
10	*P*
8	*Q*
0 0 0 0 0 0 0 0 0 0	*Wiersz 1*
0 0 0 0 0 1 1 0 0 0	*Wiersz 2*
0 0 0 1 0 2 0 0 0 0	*Wiersz 3*
1 1 0 1 2 0 0 0 0 1	*Wiersz 4*
0 1 0 0 2 0 1 1 0 0	*Wiersz 5*
0 1 0 1 0 0 1 1 0 0	*Wiersz 6*
0 1 2 0 0 0 0 1 0 0	*Wiersz 7*
0 0 0 0 0 0 0 0 0 0	*Wiersz 8*

Wyjście

Kolejne wiersze opisują ruchy pojazdów w kierunku nadajnika. Każdy wiersz składa się z numeru pojazdu i cyfry 0 lub 1, gdzie 0 oznacza ruch na południe, a 1 na wschód. Przykładowe wyjście:

MARS.OUT	**Objaśnienie:**
1 1	*pojazd 1 rusza na wschód*
1 0	*pojazd 1 rusza na południe*
2 1	*pojazd 2 rusza na wschód*
2 0	*pojazd 2 rusza na południe*
1 1	*...*
1 1	
2 0	
2 1	
2 0	
2 0	
2 0	

2 0
1 1
1 0
1 0
1 0
1 0
1 0
1 0
2 0
2 1
1 1
1 1
1 1
1 1
1 1
2 1
2 1
2 1
2 1
2 1
2 1

Do nadajnika dotarły oba pojazdy dostarczając 3 próbki skał. Za takie rozwiązanie otrzymałbyś maksymalną liczbę punktów (100%).

Punktacja

Punkty zostaną obliczone na podstawie liczby próbek zebranych i dostarczonych do nadajnika z uwzględnieniem liczby pojazdów, które dotarły do nadajnika i liczby tych, które nie dotarły.

- *Każdy nielegalny ruch unieważnia rozwiązanie testu i daje 0 punktów za test. Nielegalny ruch występuje wtedy, gdy pojazd wjeżdża na niebezpieczne pole lub opuszcza siatkę.*
- *Wynik = (liczba próbek dostarczonych do nadajnika + liczba pojazdów, które dotarły do nadajnika − liczba pojazdów, które nie dotarły do nadajnika) procentowo w stosunku do maksymalnej liczby punktów.*
- *Maksymalnie można zdobyć 100%, a minimalnie 0%.*

Krzysztof Diks, Andrzej Walat
przekład

Baza kontenerowa

Kompania Neptune Cargo zarządza magazynem kontenerów. Magazyn przyjmuje kontenery na przechowanie, a następnie je wydaje.

Kontenery przybywają do magazynu o równych godzinach. Pozostają w magazynie całkowitą liczbę godzin. Dokumenty przybywające z kontenerem określają deklarowany czas jego wydania. Pierwszy kontener przybywa o godz. 1. Zlecenie wydania kontenera może nastąpić w dowolnym czasie różniącym się od deklarowanego nie więcej niż 5 godz.

W tym zadaniu czas jest odmierzany w godzinach — w rosnących, całkowitych liczbach dodatnich, nie większych niż 150.

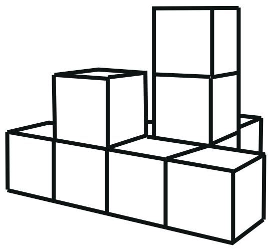
Dźwig, który może swobodnie poruszać się nad całą powierzchnią magazynu, umieszcza kontenery w magazynie, usuwa je z magazynu, a czasami przemieszcza.

Zadanie

Napisz program oparty na dobrej strategii przyjmowania, przemieszczania i wydawania kontenerów. Dobra strategia minimalizuje liczbę ruchów dźwigu.

Magazyn ma kształt prostopadłościanu. Jego długość (X), szerokość (Y) i wysokość (Z) są dostępne dla programu.

Każdy kontener jest sześcianem o rozmiarach $1 \times 1 \times 1$. Kontenery mogą być umieszczane na podłodze lub w stertach jeden na drugim. Dźwig może przemieścić tylko kontener znajdujący się na wierzchu.



Przemieszczenie kontenera z jednego miejsca na inne jest traktowane jako jeden ruch dźwigu. Wszystkie ruchy dźwigu mają miejsce pomiędzy pobraniami i wydaniami kontenerów. Czas ruchu jest zaniedbywalny.

Gdy magazyn jest pełny, Twój program musi odmówić przyjęcia nowego kontenera. Twój program może się stać mniej efektywny lub być niezdolny do działania, gdy magazyn jest prawie pełny. Twój program ma prawo w dowolnej chwili odmówić przyjęcia kontenera.

WEJŚCIE

Twój program powinien współpracować interakcyjnie z modułem symulacyjnym, dostarczającym dane, na które Twój program musi reagować, podejmując odpowiednie akcje i wysyłając komunikaty. W momencie rozpoczęcia działania programu magazyn jest pusty.

Gdy będziesz testował program, moduł dostarczy sensownych danych dla ustalonego zestawu danych o małym rozmiarze. Każdy kontener jest jednoznacznie identyfikowany za pomocą dodatniej liczby całkowitej. Twój program może w każdej chwili wywołać następujące funkcje i procedury:

```
int GetX();
function GetX: integer;
DECLARE FUNCTION GetX CDECL ()
```

Zwraca długość magazynu (integer).

```
int GetY();
function GetY: integer;
DECLARE FUNCTION GetY CDECL ()
```

Zwraca szerokość magazynu (integer).

```
int GetZ();
function GetZ: integer;
DECLARE FUNCTION GetZ CDECL ()
```

Zwraca wysokość magazynu (integer).

X, Y, Z *nie przekraczają 32.*

Następujące funkcje dostarczają informacji o sekwencji zdarzeń (przybywanie i wydawanie kontenerów). Kontenery przybywają o równych godzinach, zlecenia wydania są przyjmowane w trakcie godzin. Tak więc dla celów odmierzania czasu każde przybycie kontenera oznacza minięcie jednej godziny.

```
int GetNextContainer();
function GetNextContainer: integer;
DECLARE FUNCTION GetNextContainer CDECL ()
```

Zwraca numer kolejnego kontenera, który ma być przyjęty lub wydany. Jeśli nie ma więcej kontenerów do przyjęcia lub wydania, zwraca 0, co oznacza, że Twój program powinien zakończyć działanie, nawet gdy magazyn jest niepusty.

```
int GetNextAction();
function GetNextAction: integer;
DECLARE FUNCTION GetNextAction CDECL ()
```

Zwraca liczbę całkowitą wyznaczającą akcję, którą trzeba podjąć: 1 oznacza przyjęcie kontenera, 2 — wydanie kontenera.

```
int GetNextStorageTime();
function GetNextStorageTime: integer;
DECLARE FUNCTION GetNextStorageTime CDECL ()
```

Zwraca deklarowany czas wydania kontenera w godzinach (liczony od początku). Ta wartość ma służyć programowi dla celów planowania; rzeczywiste zlecenie wydania kontenera może nastąpić w dowolnym czasie różniącym się od deklarowanego co najwyżej o 5 godzin. Ta funkcja daje sensowną wartość tylko wtedy, gdy `GetNextAction` *zwraca 1.*

Kolejność wywoływania tych trzech funkcji jest dowolna. Kolejne wywołania `GetNextContainer`*,* `GetNextAction` *i* `GetNextStorageTime` *dają zawsze informację o tym samym kontenerze, aż do momentu przyjęcia, wydania lub odmowy przyjęcia kontenera; po czym, te funkcje dają informację o następnym kontenerze.*

Wyjście

Gdy Twój program uzyska wszystkie potrzebne informacje o kolejnym kontenerze, powinien skorzystać z nastepujacyh funkcji symulujących operacje magazynowe.

```
int MoveContainer(int x1, int y1, int x2, int y2);
function MoveContainer(x1, y1, x2, y2: integer): integer;
DECLARE FUNCTION MoveContainer CDECL (BYVAL x1 AS INTEGER, BYVAL
y1 AS INTEGER, BYVAL x2 AS INTEGER, BYVAL y2 AS INTEGER)
```

Przenieś kontener z wierzchu sterty $x1, y1$ na wierzch sterty $x2, y2$. Zwraca 1, jeśli ta akcja jest dozwolona, 0 gdy jest niedopuszczalna.

```
void RefuseContainer();
procedure RefuseContainer;
DECLARE SUB RefuseContainer CDECL ()
```

Odmowa przyjęcia kontenera.

```
void StoreArrivingContainer(int x, int y);
procedure StoreArrivingContainer(x, y: integer);
DECLARE SUB StoreArrivingContainer CDECL (BYVAL x AS INTEGER,
BYVAL y AS INTEGER)
```

Przyjmuje kontener i umieszcza na wierzchu sterty w pozycji x, y.

```
void RemoveContainer(int x, int y);
procedure RemoveContainer(x, y: integer);
DECLARE SUB RemoveContainer CDECL (BYVAL x AS INTEGER, BYVAL y AS
INTEGER)
```

Wydaje z magazynu kontener znajdujący się na wierzchu sterty x, y.

Jeśli Twój program nie może wykonać zleconej akcji, powinien zakończyc działanie. Niedozwolone ruchy są ignorowane przez moduł i nie mają wpływu na symulację, ani na punktację.

Twój program nie powinien tworzyć żadnego pliku. Moduł współdziałający z Twoim programem tworzy plik akcji, który będzie podstawą oceny.

Kolejność akcji

Twój program powinien otrzymać informację o zleceniu dotyczącym kontenera. Następnie, jeśli trzeba, powinien przemieścić kontenery w magazynie i przyjąć kontener, wydać kontener lub odmówić wykonania zlecenia.

Biblioteka

Gotową bibliotekę StackLib musisz skonsolidować ze swoim kodem. Standardowe biblioteki C i C++ zawierają tę bibliotekę i będzie ona automatycznie konsolidowana z Twoim programem, jeśli włączysz odpowiedni plik nagłówkowy. Przykładowe pliki źródłowe znajdują się w katalogu zadania. Są to `TESTSTK.BAS`, `TESTSTK.PAS`, `TESTSTK.CPP` *i* `TESTSTK.C`.

Punktacja

Program będzie testowany na kilku zestawach danych i w każdym przypadku punkty będą przyznawane na podstawie porównania z najlepszym znanym nam programem, z uwzględnieniem następujacych wskaźników:

- *Łączna liczba ruchów dźwigu.*
- *Kara 5 ruchów za każdą odmowę przyjęcia kontenera.*
- *Kara 5 ruchów za każdy kontener, który nie był przechowywany w magazynie (bo np. program zakończył w normalny sposób swoje działanie przed wykonaniem całego zadania).*
- *Suma punktów będzie obliczona względem najlepszego znanego rozwiązania.*
- *Jeśli program wykona dwa razy więcej operacji niż potrzeba, otrzyma 0 pkt.*
- *Minimalny wynik to 0%, a maksymalny to 100%.*

Krzysztof Diks, Andrzej Walat
przekład

Żarłoczne iShongololo

iShongololo *to zuluska nazwa krocionoga. Jest to długi, błyszczący, czarny owad, mający wiele nóg.*

iShongololo żywi się jadalnymi „owocami”. Na potrzeby tego zadania przyjmujemy, że mają one kształt prostopadłościanu o całkowitoliczbowych rozmiarach L (length — długość), W (width — szerokość) i H (height — wysokość).

Zadanie

Napisz program, który maksymalizuje liczbę bloków zjadanych przez iShongololo bez naruszenia ustalonych ograniczeń. Program musi wypisywać czynności wykonywane przez iShongololo, gdy drąży swoją ścieżkę zjadając owoc.

Na początku iShongololo znajduje się na zewnątrz owocu. Pierwszym blokiem, który musi zjeść jest 1, 1, 1. Następnie musi wejść do tego bloku. iShongololo kończy, gdy nie może już zgodnie z regułami zjeść żadnego bloku, ani wykonać ruchu.

Ograniczenia

(1) iShongololo zajmuje dokładnie jeden pusty blok.

(2) iShongololo zjada na raz jeden pełny blok.

(3) iShongololo nie może wykonać ruchu na pozycję, na której już był (tzn. zawrócić lub przeciąć swoją ścieżkę).

(4) iShongololo nie może wejść do pełnego (nie zjedzonego) bloku lub wyjść na zewnątrz owocu.

(5) iShongololo może wykonać ruch tylko do bloku o przyległej ścianie; może też zjeść blok o przyłeglej ścianie, ale tylko taki, który nie ma wspólnej ściany z żadnym innym pustym blokiem.

Wejście

Na wejściu Twój program otrzyma trzy liczby całkowite — długość (L), szerokość (W) i wysokość (H) prostopadłościanu.

Każda z liczb L, W, H jest podana w osobnym wierszu i należy do przedziału od 1 do 32 (włącznie). Przykładowe wejście:

TOXIC.DAT	**Objaśnienie:**
2	*długość — 2*
3	*szerokość — 2*
2	*wysokość — 2*

Wyjście

Plik wyjściowy składa się z wierszy rozpoczynających się od litery `E` *(eat — zjada) lub* `M` *(move — wchodzi do), po której występują 3 liczby całkowite reprezentujące współrzędne bloku (odpowiednio L, W, H), który iShongololo zjada lub do którego wchodzi.*

Oto przykład poprawnego pliku wyjściowego (być może nie reprezentującego rozwiązania optymalnego) dla przykładowych danych.

TOXIC.OUT	**Objaśnienie:**
E 1 1 1	*Zjada blok 1 1 1*
M 1 1 1	*Wchodzi do bloku 1 1 1*
E 2 1 1	*Zjada blok 2 1 1*
E 1 1 2	*Zjada blok 1 1 2*
E 1 2 1	*Zjada blok 1 2 1*
M 1 2 1	*Wchodzi do bloku 1 2 1*
E 1 3 1	*Zjada blok 1 3 1*
M 1 3 1	*Wchodzi do bloku 1 3 1*
E 2 3 1	*Zjada blok 2 3 1*
E 1 3 2	*Zjada blok 1 3 2*
M 1 3 2	*Wchodzi do bloku 1 3 2*

Punktacja

- *Jeśli iShongololo naruszy ograniczenia, otrzymasz 0 punktów za test.*
- *Ogólna liczba punktów wyraża procentowo stosunek liczby zjedzonych bloków do najlepszego znanego rozwiązania.*
- *Nie możesz otrzymać więcej niż 100%.*

IV Olimpiada Informatyczna Centralnej Europy Nowy Sącz, lipiec 1997

teksty zadań

Krzysztof Diks **Przemysława Kanarek, Marcin Kubica**
autor zadania przekład

Na drugi brzeg

Mieszkańcy Bajtocji uwielbiają wszelkie sporty, które oprócz sprawności fizycznej wymagają również sprytu i inteligencji. Jedną z takich dyscyplin jest przeprawianie się przez Hex-Dunajec — najszerszą rzekę Bajtocji. W poprzek Hex-Dunajca, od lewego brzegu do prawego, ustawionych jest n pali oznaczonych kolejno liczbami $1, \ldots, n$. Najbliżej lewego brzegu jest pal nr 1, a najbliżej prawego pal nr n. Zawodnik musi przeprawić się z lewego brzegu na prawy skacząc po palach.

Przeprawa rozpoczyna się w chwili 0. Zawodnik startuje z lewego brzegu rzeki i ma za zadanie osiągnąć prawy brzeg w jak najkrótszym czasie. W każdej chwili, każdy z pali jest wynurzony albo zanurzony. W każdej chwili zawodnik może stać na brzegu lub na palu. Na palu można stać jedynie wówczas, gdy jest on wynurzony.

W chwili 0 wszystkie pale są zanurzone. Dalsze zachowanie każdego pola opisują związane z nim dwie liczby całkowite a i b. Począwszy od chwili 1 przez a jednostek czasu pal pozostaje wynurzony. Przez kolejnych b jednostek czasu pal jest zanurzony. Następnie ponownie wynurza się na a jednostek czasu, potem zanurza się na b jednostek, itd. Np. jeżeli dla pewnego pala $a = 2$ i $b = 3$, to pal jest zanurzony w chwili 0, wynurza się na chwile 1 i 2, zanurza się na chwile 3, 4 i 5 itd.

Jeżeli w chwili t, zawodnik znajduje się w pewnym miejscu (na palu wynurzonym w chwili t lub na brzegu), to w chwili $t + 1$ może znaleźć się w dowolnym miejscu (na palu wynurzonym w chwili $t + 1$ lub na brzegu) odległym o co najwyżej 5 pali. Np. stojąc w chwili t na palu nr 5, w chwili $t + 1$ zawodnik może znaleźć się na lewym brzegu lub na dowolnym z pali 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 (oczywiście, o ile pal jest wynurzony).

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego RIV.IN liczbę bloków danych (każdy blok zawiera kompletny opis jednego przykładu zadania);*
- *dla każdego bloku:*
  - *wczytuje liczbę pali oraz opis ich zachowania;*
  - *oblicza najkrótszy możliwy czas potrzebny do przeprawienia się na prawy brzeg, o ile przeprawa jest możliwa;*
  - *zapisuje wynik w pliku tekstowym RIV.OUT.*

Wejście

Pierwszy wiersz pliku RIV.IN zawiera liczbę bloków danych x, $1 \leqslant x \leqslant 5$. Kolejne wiersze zawierają x bloków danych. Pierwszy blok rozpoczyna się w drugiej linii pliku wejściowego, a kolejne bloki następują bezpośrednio po sobie.

Pierwszy wiersz każdego bloku zawiera liczbę pali n, $5 < n \leqslant 1000$. W każdym z kolejnych n wierszy bloku znajdują się dwie liczby całkowite a i b ($1 \leqslant a, b \leqslant 5$) oddzielone pojedynczą spacją. Zachowanie i-tego pala opisują liczby z $(i+1)$-go wiersza bloku, $1 \leqslant i \leqslant n$.

Wyjście

Dla k-tego bloku danych, $1 \leqslant k \leqslant x$, Twój program powinien zapisywać wynik w k-tym wierszu pliku tekstowego RIV.OUT. Jeśli można przeprawić się przez rzekę, to wynikiem jest liczba całkowita będąca minimalną liczbą jednostek czasu koniecznych do przeprawy. Jeśli nie można przeprawić się przez rzekę, to wynikiem jest słowo NO.

Przykład

Dla pliku tekstowego RIV.IN:

```
2
10
1 1
1 1
1 1
1 1
1 1
1 1
1 1
1 1
1 1
1 1
10
1 1
1 1
1 1
1 1
2 1
1 1
1 1
1 1
1 1
1 1
```

prawidłowym rozwiązaniem jest plik tekstowy RIV.OUT;

```
NO
4
```

Krzysztof Diks
autor zadania

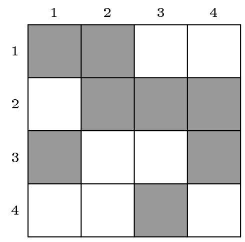
Przemysława Kanarek, Marcin Kubica
przekład

Zawody Strzeleckie

Witamy na Dorocznych Zawodach Strzeleckich Bajtocji. Każdy z zawodników strzela do tarczy w kształcie prostokąta. Tarcza jest podzielona na r wierszy i c kolumn, czyli składa się z $r \times c$ kwadratowych pól. Każde z pól tarczy jest zamalowane na czarno lub biało. Wiadomo, że w każdej kolumnie tarczy są dokładnie dwa pola białe i $r-2$ czarne. Wiersze tarczy numerujemy od góry w dół kolejnymi liczbami naturalnymi $1, \ldots, r$. Kolumny tarczy numerujemy od lewej do prawej liczbami $1, \ldots, c$. Każdy z zawodników strzela serię złożoną z c strzałów. Seria c strzałów jest poprawna jeżeli w jej trakcie trafiono dokładnie po jednym białym polu w każdej kolumnie i nie pozostał żaden wiersz, w którym nie byłoby trafionego białego pola. Pomóż zawodnikowi znaleźć poprawną serię strzałów, o ile taka seria istnieje.

Przykład

Przyjmijmy, że tarcza wygląda jak na poniższym rysunku:



Poprawna seria strzałów, to np. trafienia w pola znajdujące się w wierszach 2, 3, 1, 4 odpowiednio w kolejnych kolumnach 1, 2, 3, 4.

Zadanie

Napisz program, który:

- *wczytuje liczbę bloków danych z pliku tekstowego SHO.IN; każdy blok zawiera kompletny opis jednej tarczy;*
- *następnie dla każdego bloku*
  - *wczytuje rozmiar tarczy i rozkład białych pól na tarczy;*
  - *sprawdza, czy istnieje poprawna seria strzałów dla tej tarczy; jeśli tak, to znajduje jedną z nich;*
  - *zapisuje wynik w pliku tekstowym SHO.OUT;*

Wejście

W pierwszym wierszu pliku danych SHO.IN podana jest liczba bloków danych (opisów tarcz) x, $1 \leqslant x \leqslant 5$. W kolejnych wierszach opisanych jest x bloków. Opis pierwszego bloku rozpoczyna się w drugim wierszu pliku wejściowego, a opis każdego kolejnego bloku rozpoczyna się w wierszu bezpośrednio następującym po zakończeniu opisu poprzedniego bloku.

Opis każdego bloku (tarczy) rozpoczyna się od wiersza zawierającego dwie liczby całkowite r i c ($2 \leqslant r \leqslant c \leqslant 1000$) oddzielone pojedynczym odstępem. Liczby te oznaczają odpowiednio liczbę wierszy i kolumn tarczy. Każdy z kolejnych c wierszy bloku (opisu tarczy) zawiera po dwie liczby oddzielone pojedynczym odstępem. Liczby w $(i+1)$-szym wierszu bloku ($1 \leqslant i \leqslant c$) oznaczają pozycje (tzn. numery wierszy na planszy) białych pól w i-tej kolumnie tarczy.

Wynik

Dla i-tego bloku, $1 \leqslant i \leqslant x$, Twój program powinien zapisać w i-tym wierszu pliku wynikowego SHO.OUT ciąg c numerów wierszy (oddzielonych pojedynczymi spacjami) tworzących opis poprawnej serii strzałów w białe pola kolejnych kolumn $1, 2, \ldots, c$. Jeżeli dla podanej tarczy poprawna seria nie istnieje, w i-tym wierszu pliku SHO.OUT należy wpisać słowo NO.

Przykład

Dla pliku tekstowego SHO.IN:

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

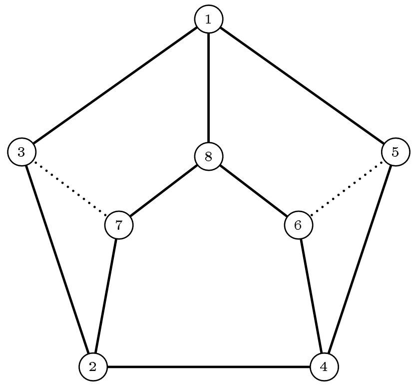
jedną z poprawnych odpowiedzi jest plik tekstowy SHO.OUT:

```
2 3 1 4
NO
```

Krzysztof Diks **Przemysława Kanarek, Marcin Kubica**
autor zadania przekład

Jaskinie

W Bajtocji jest wiele jaskiń. Oto mapa jednej z nich:



W Bajtocji każda z jaskiń ma następujące cechy:

- *wszystkie komnaty i korytarze leżą na tym samym poziomie,*
- *korytarze nie przecinają się,*
- *część komnat leży na obwodzie jaskini — nazywamy je* komnatami zewnętrznymi*,*
- *wszystkie pozostałe komnaty, leżące wewnątrz, nazywamy* komnatami wewnętrznymi*,*
- *wejście do jaskini prowadzi do jednej z zewnętrznych komnat,*
- *z każdej komnaty wychodzą dokładnie trzy korytarze, prowadzące do trzech różnych innych komnat; jeśli komnata jest zewnętrzna, to dwa spośród korytarzy prowadzą do sąsiednich komnat zewnętrznych na obwodzie jaskini, a jeden do komnaty wewnętrznej,*
- *korytarze łączące zewnętrzne komnaty nazywamy* korytarzami zewnętrznymi*, a wszystkie pozostałe* korytarzami wewnętrznymi*,*
- *z każdej komnaty można dojść do każdej innej komnaty korzystając jedynie z wewnętrznych korytarzy, ale jeśli dodatkowo wymagamy by przez każdy korytarz przechodzić co najwyżej raz, to można to zrobić tylko w jeden sposób,*
- *nie wszystkie korytarze są jednakowo łatwe do pokonania — dzielimy je na dwie kategorie: łatwe i trudne.*

Postanowiono udostępnić wszystkie jaskinie dla zwiedzających. Aby zapewnić płynny i bezpieczny przepływ zwiedzających, w każdej z jaskiń należy ustalić trasę zwiedzania, która przechodzi dokładnie raz przez każdą komnatę.

Jedynym wyjątkiem od tej zasady jest komnata wejściowa, od której zwiedzanie się zaczyna i na której się kończy (tzn. zwiedzający przechodzą przez tę komnatę dokładnie dwa *razy). Trasa zwiedzania powinna być przeznaczona dla przeciętnego turysty i zawierać jak najmniej trudnych korytarzy.*

Przykład

Przyjrzyjmy się jaskini przedstawionej na rysunku. Komnata wejściowa ma numer 1. Trudne korytarze są zaznaczone linią przerywaną. Trasa 1, 5, 4, 6, 8, 7, 2, 3 nie przechodzi przez żaden z trudnych korytarzy. Trasa zwiedzania kończy się oczywiście w komnacie nr 1, chociaż pomijamy tę jedynkę na końcu opisu trasy.

Zadanie

Napisz program, który:

- *wczytuje opis jaskini z pliku tekstowego CAV.IN;*
- *znajduje trasę zwiedzania, która zaczyna i kończy się w komnacie wejściowej, przechodzi przez każdą komnatę dokładnie raz oraz prowadzi przez jak najmniejszą liczbę trudnych korytarzy;*
- *zapisuje wynik w pliku tekstowym CAV.OUT.*

Wejście

W pierwszym wierszu pliku tekstowego CAV.IN znajdują się dwie liczby całkowite n, k (oddzielone pojedynczym odstępem). Liczba n, $3 < n \leqslant 500$, to liczba wszystkich komnat w jaskini, a k, $k \geqslant 3$, to liczba komnat zewnętrznych. Komnaty są ponumerowane od 1 do n. Komnata wejściowa ma numer 1. Komnaty zewnętrzne mają numery $1, 2, \ldots, k$, aczkolwiek nie muszą leżeć na obwodzie jaskini w tej kolejności. Kolejne $3n/2$ wiersze pliku zawierają opisy korytarzy. Opis każdego korytarza składa się z trzech liczb całkowitych a, b, c, (oddzielonych pojedynczymi spacjami). Liczby a i b są numerami komnat, które łączy korytarz. Liczba c jest równa 0 lub 1 — 0 oznacza, że korytarz jest łatwy, a 1, że trudny.

Wyjście

Twój program powinien zapisać w pierwszym wierszu pliku tekstowego CAV.OUT ciąg n liczb całkowitych, pooddzielanych pojedynczymi odstępami. Pierwszą liczbą powinno być 1 (numer komnaty wejściowej), a kolejnymi $n-1$ liczbami powinny być numery kolejnych komnat trasy zwiedzania.

Przykład

Dla tekstowego pliku wejściowego CAV.IN:

```
8 5
1 3 0
3 2 0
7 3 1
7 2 0
8 7 0
1 8 0
6 8 0
```

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

jednym z poprawnych wyników jest następujący plik tekstowy CAV.OUT:

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

P. Chrząstowski-Wachtel, W. Guzicki — autorzy zadania
P. Kanarek, M. Kubica — przekład

Przedziały liczb całkowitych

Przedział liczb całkowitych [a..b], gdzie $a < b$, to zbiór wszystkich kolejnych liczb całkowitych rozpoczynających się od a i kończących na b.

Zadanie

Napisz program, który:

- *wczytuje liczbę przedziałów i ich opisy z pliku tekstowego INT.IN;*
- *znajduje minimalną liczbę elementów zbioru zawierającego co najmniej dwie różne liczby całkowite z każdego z przedziałów;*
- *zapisuje wynik w pliku tekstowym INT.OUT.*

Wejście

Pierwszy wiersz pliku tekstowego INT.IN zawiera liczbę przedziałów n, $1 \leqslant n \leqslant 10000$. Każdy z kolejnych n wierszy zawiera dwie liczby całkowite a i b, $0 \leqslant a < b \leqslant 10000$, oddzielone pojedynczą spacją. Liczby te oznaczają początek i koniec przedziału.

Wyjście

Twój program powinien zapisać w pierwszym wierszu pliku tekstowego INT.OUT jedną liczbę całkowitą; ma to być minimalna liczba elementów zbioru zawierającego co najmniej dwie różne liczby z każdego z przedziałów.

Przykład

Dla pliku tekstowego INT.IN

```
4
3 6
2 4
0 2
4 7
```

prawidłowym rozwiązaniem jest plik tekstowy INT.OUT:

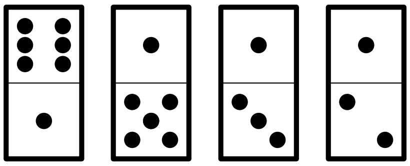
```
4
```

Krzysztof Diks
autor zadania

Marcin Kubica
przekład

Domino

Kostki domino to niewielkie prostokątne klocki. Na każdym klocku są dwa kwadratowe pola, a każde pole jest puste, lub znajduje się na nim od 1 do 6 oczek. Na stole ułożono rząd kostek domino, np.:



Liczba oczek w górnym rzędzie pól wynosi $6 + 1 + 1 + 1 = 9$, *a w dolnym* $1 + 5 + 3 + 2 = 11$.

Różnica (wartość bezwzględna z różnicy) liczb oczek w górnym i dolnym rzędzie wynosi 2.

Każdą kostkę domino można obrócić o 180 stopni (oczywiście kostka musi pozostać odwrócona oczkami do góry).

Jaka jest najmniejsza liczba kostek, które trzeba obrócić, aby różnica (jw. wartość bezwzględna z różnicy) liczb oczek w górnym i dolnym rzędzie była minimalna?

W sytuacji przedstawionej na powyższym rysunku wystarczy obrócić skrajnie prawą kostkę, aby zmniejszyć różnicę do 0.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego DOM.IN liczbę kostek domino i ich opis,*
- *wylicza najmniejszą liczbę kostek, które trzeba obrócić, aby różnica liczb oczek w górnym i dolnym rzędzie była minimalna,*
- *wypisuje wynik do pliku tekstowego DOM.OUT.*

Wejście

Pierwszy wiersz pliku tekstowego DOM.IN zawiera liczbę całkowitą n, $1 \leqslant n \leqslant 1000$. *Jest to liczba kostek domino ułożonych na stole.*

W każdym z n *kolejnych wierszy zapisane są po dwie liczby całkowite* a *i* b, $0 \leqslant a, b \leqslant 6$, *oddzielone pojedynczym odstępem. Liczby* a *i* b *zapisane w* $(i+1)$*-szym wierszu pliku wejściowego,* $1 \leqslant i \leqslant n$, *opisują* i*-tą kostkę — odpowiednio liczbę oczek na jego górnym i dolnym polu.*

Wyjście

Twój program powinien wypisać dokładnie jedną liczbę całkowitą, w pierwszym wierszu pliku tekstowego DOM.OUT. Powinna to być najmniejsza liczba kostek, które trzeba obrócić, aby różnica liczb oczek w górnym i dolnym rzędzie była minimalna.

PRZYKŁAD

Dla pliku tekstowego DOM.IN:

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

poprawnym wynikiem jest plik tekstowy DOM.OUT:

```
1
```

Wojciech Rytter
autor zadania

Przemysława Kanarek, Marcin Kubica
przekład

Liczby szesnastkowe

Podstawą systemu szesnastkowego jest liczba 16. Do zapisu liczb w tym systemie używamy 16 cyfr, które oznaczamy $0, 1, \ldots, 9, A, B, C, D, E, F$*. Wielkie litery* $A, \ldots, F$ *oznaczają odpowiednio* $0, \ldots, 15$*. Na przykład wartość liczby szesnastkowej* $CF2$ *wynosi w systemie dziesiętnym* $12 \cdot 16^2 + 15 \cdot 16 + 2 = 3314$*. Niech* X *oznacza zbiór wszystkich dodatnich liczb całkowitych, których szesnastkowa reprezentacja ma najwyżej 8 cyfr. Ponadto żadna z cyfr w reprezentacji nie powtarza się i najbardziej znacząc cyfra jest różna od zera. Największym elementem w* X *jest liczba o reprezentacji szesnastkowej* $FEDCBA98$*, drugą co do wielkości liczbą jest* $FEDCBA97$*, trzecią* $FEDCBA96$*, itd.*

Zadanie

Napisz program, który:

- *czyta dodatnią liczbę całkowitą n z pliku tekstowego HEX.IN, n jest nie większe niż liczba elementów w zbiorze* X*;*
- *znajduje* n*-ty co do wielkości (licząc od największego) element zbioru* X*;*
- *zapisuje wynik w pliku tekstowym* $HEX.OUT$*.*

Wejście

Pierwszy wiersz pliku HEX.IN zawiera liczbę n zapisaną w systemie dziesiętnym.

Wyjście

Twój program powienien wypisać w systemie szesnastkowym w pierwszym wierszu pliku tekstowego $HEX.OUT$ n*-ty co do wielkości (licząc od największego) element zbioru* X*.*

Przykład

Dla pliku tekstowego HEX.IN:

```
11
```

prawidłowym rozwiązaniem jest plik tekstowy $HEX.OUT$*:*

```
FEDCBA87
```

Literatura

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

[1] *I Olimpiada Informatyczna 1993/1994*, Warszawa–Wrocław 1994

[2] *II Olimpiada Informatyczna 1994/1995*, Warszawa–Wrocław 1995

[3] *III Olimpiada Informatyczna 1995/1996*, Warszawa–Wrocław 1996

[4] *IV Olimpiada Informatyczna 1996/1997*, Warszawa 1997

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

[6] A. V. Aho, J. D. Ullman *Principles of Compiler Design*, Addison-Wesley, 1977

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[23] W. M. Waite, G. Goos *Konstrukcja kompilatorów*, WNT, Warszawa 1989

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

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

Niniejsza publikacja stanowi wyczerpujące źródło informacji o zawodach V Olimpiady Informatycznej, przeprowadzonych w roku szkolnym 1997/1998. Książka zawiera zarówno informacje dotyczące organizacji zawodów, regulaminu oraz klasyfikacji, jak i treści oraz wyczerpujące omówienia zadań Olimpiady.

V 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 83–906301–4–1