MINISTERSTWO EDUKACJI NARODOWEJ
INSTYTUT INFORMATYKI UNIWERSYTETU WROCŁAWSKIEGO
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

IX OLIMPIADA INFORMATYCZNA
2001/2002

WARSZAWA, 2002

MINISTERSTWO EDUKACJI NARODOWEJ
INSTYTUT INFORMATYKI UNIWERSYTETU WROCŁAWSKIEGO
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

IX OLIMPIADA INFORMATYCZNA
2001/2002

WARSZAWA, 2002

Autorzy tekstów:

prof. dr hab. Zbigniew Czech
dr Piotr Chrząstowski-Wachtel
dr hab. Krzysztof Diks
dr hab. Wojciech Guzicki
dr Przemysława Kanarek
dr Marcin Kubica
mgr Marcin Mucha
Krzysztof Onak
Jakub Pawlewicz
mgr Marcin Sawicki
Marcin Stefaniak
dr Krzysztof Stencel
Tomasz Waleń
Paweł Wolff

Autorzy programów na dysku CD-ROM:

Jarosław Byrka
Wojciech Dudek
mgr Marcin Mucha
Krzysztof Onak
mgr Remigiusz Różycki
Marcin Stefaniak
Tomasz Waleń
Paweł Wolff

Opracowanie i redakcja:

dr hab. Krzysztof Diks
Krzysztof Onak

Skład: Krzysztof Onak

Pozycja dotowana przez Ministerstwo Edukacji Narodowej.

Druk książki został sfinansowany przez PROKOM SOFTWARE SA

ISBN 83-917700-0-1

Spis treści

Krzysztof Diks

Wstęp

Oddajemy do rąk czytelników sprawozdanie i rozwiązania zadań z IX Olimpiady Informatycznej. Od opublikowania sprawozdań z VIII Olimpiady we wrześniu ubiegłego roku, w naszym olimpijskim świecie wydarzyło się bardzo wiele. Zacznijmy od sukcesów reprezentantów Polski na arenie międzynarodowej.

W dniach od 23-go do 28-go kwietnia, w Wilnie na Litwie, odbyła się VIII Bałtycka Olimpiada Informatyczna. Wzięło w niej udział osiem krajów: Dania, Estonia, Finlandia, Litwa, Łotwa, Niemcy, Polska i Szwecja. Polska była reprezentowana przez szóstkę zawodników z czołówki tegorocznej edycji Olimpiady. Polacy spisali się znakomicie i zdobyli

Karol Cwalina i Marcin Michalski — złote medale,
Paweł Parys i Bartosz Walczak — srebrne medale,
Michał Jaszczyk i Piotr Stańczyk — brązowe medale.

Więcej o BOI'2002 można znaleźć w witrynie

`http://aldona.mii.lt/pms/olimp/english/boi2002/index.html`.

W dniach 30-ty czerwca – 6-ty lipca, w Koszycach na Słowacji, odbyła się IX Olimpiada Informatyczna Europy Centralnej (`http://cs.science.upjs.sk/ceoi/`). Wzięło w niej udział 8 krajów: Chorwacja, Czechy, Iran (gość), Niemcy, Polska, Rumunia, Słowacja, Węgry. Polska reprezentowana była przez Karola Cwalinę (medal srebrny), Marcina Michalskiego (medal brązowy), Piotra Stańczyka i Bartka Walczaka (medal srebrny).

Ważnym elementem przygotowań naszych zawodników do udziału w zawodach informatycznych są obozy naukowe dla finalistów Olimpiady. W tym roku zorganizowaliśmy dwa takie obozy.

Na początku czerwca w Warszawie odbył się doroczny obóz czesko-polsko-słowacki. Osiemnastu zawodników z Czech, Polski i Słowacji (po sześciu z każdego kraju) miało okazję nie tylko rozwiązywać zadanie, ale także bliżej się poznać, wymienić doświadczenia i zaprzyjaźnić.

W drugim tygodniu sierpnia młodzi finaliści IX Olimpiady spotkali się w Zakopanem na obozie naukowym im. Antoniego Kreczmara. Była okazja, żeby nie tylko rozwiązywać zadania, nauczyć się czegoś nowego, ale także pograć w piłkę i pochodzić po górach.

Olimpiada Informatyczna jest wielkim przedsięwzięciem organizacyjno-naukowo-dydaktycznym. Wymaga współpracy i zaangażowania wielu osób z różnych miejsc w całym kraju. Ich wspólny wysiłek przyczynił się do sukcesu tegorocznej Olimpiady. Wszystkim osobom zaangażowanym w prace IX Olimpiady Informatycznej składam tą drogą gorące podziękowania.

Prezentowana książeczka zawiera zadania wraz z rozwiązaniami z IX Olimpiady Informatycznej. Na dysku CD-ROM załączono programy wzorcowe i testy, które posłużyły do sprawdzenia rozwiązań zawodników. Przedstawiamy też zadania z tegorocznych olimpiad, bałtyckiej i Europy Centralnej. Wszystkim autorom materiałów zawartych w tym wydawnictwie serdecznie dziękuję. Mam nadzieję, że przedstawione materiały pozwolą na jeszcze lepsze przygotowywanie się do udziału w olimpiadach informatycznych, jak i posłużą doskonaleniu umiejętności algorytmiczno-programistycznych.

Krzysztof Diks
Warszawa, sierpień 2002 roku

Sprawozdanie z przebiegu IX Olimpiady Informatycznej 2001/2002

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

ORGANIZACJA ZAWODÓW

W roku szkolnym 2001/2002 odbyły się zawody IX Olimpiady Informatycznej. Olimpiada Informatyczna jest trójstopniowa. Rozwiązaniem każdego zadania zawodów I, II i III stopnia jest program napisany w jednym z ustalonych przez Komitet języków programowania lub plik z wynikami. Zawody I stopnia miały charakter otwartego konkursu przeprowadzonego dla uczniów wszystkich typów szkół młodzieżowych.

5 października 2001 r. rozesłano plakaty zawierające zasady organizacji zawodów I stopnia oraz zestaw 5 zadań konkursowych do 3240 szkół i zespołów szkół młodzieżowych ponadpodstawowych oraz do wszystkich kuratorów i koordynatorów edukacji informatycznej. Zawody I stopnia rozpoczęły się dnia 15 października 2001 r. Ostatecznym terminem nadsyłania prac konkursowych był 12 listopada 2001 roku.

Zawody II i III stopnia były dwudniowymi sesjami stacjonarnymi, poprzedzonymi jednodniowymi sesjami próbnymi. Zawody II stopnia odbyły się w pięciu okręgach: Warszawie, Wrocławiu, Toruniu, Katowicach i Krakowie oraz w Sopocie i Rzeszowie, w dniach 12–14.02.2002 r., natomiast zawody III stopnia odbyły się w ośrodku firmy Combidata Poland S.A. w Sopocie, w dniach 15–19.04.2002 r.

Uroczystość zakończenia IX Olimpiady Informatycznej odbyła się w dniu 19.04.2002 r. w Sali Posiedzeń Urzędu Miasta w Sopocie z udziałem wiceministra edukacji narodowej, pana prof. Tomasza Goban-Klasa.

SKŁAD OSOBOWY KOMITETÓW OLIMPIADY INFORMATYCZNEJ

Komitet Główny

przewodniczący:
dr hab. Krzysztof Diks, prof. UW (Uniwersytet Warszawski)
zastępcy przewodniczącego:
prof. dr hab. Maciej M. Sysło (Uniwersytet Wrocławski)
dr Andrzej Walat (OEIiZK)
sekretarz naukowy:
dr Marcin Kubica (Uniwersytet Warszawski)
kierownik Jury:
dr Krzysztof Stencel (Uniwersytet Warszawski)
kierownik organizacyjny:
Tadeusz Kuran (OEIiZK)
członkowie:
prof. dr hab. Zbigniew Czech (Politechnika Śląska)
mgr Jerzy Dałek (Ministerstwo Edukacji Narodowej)
dr Przemysława Kanarek (Uniwersytet Wrocławski)
dr hab. Krzysztof Loryś, prof. UWr (Uniwersytet Wrocławski)
prof. dr hab. Jan Madey (Uniwersytet Warszawski)
prof. dr hab. Wojciech Rytter (Uniwersytet Warszawski)
mgr Krzysztof J. Święcicki (Ministerstwo Edukacji Narodowej)
dr Maciej Ślusarek (Uniwersytet Jagielloński)
dr hab. inż. Stanisław Waligórski, prof. UW (Uniwersytet Warszawski)
dr Bolesław Wojdyło (Uniwersytet Mikołaja Kopernika w Toruniu)
sekretarz Komitetu Głównego:
Monika Kozłowska-Zając

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

Komitet Główny odbył 5 posiedzeń. 25 stycznia 2002 r. przeprowadzono seminarium przygotowujące organizację zawodów II stopnia.

Komitety okręgowe

Komitet Okręgowy w Warszawie
przewodniczący:
dr Wojciech Plandowski (Uniwersytet Warszawski)
członkowie:
dr Marcin Kubica (Uniwersytet Warszawski)
dr Adam Malinowski (Uniwersytet Warszawski)
dr Andrzej Walat (OEIiZK)

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

Komitet Okręgowy we Wrocławiu
przewodniczący:
dr hab. Krzysztof Loryś, prof. UWr (Uniwersytet Wrocławski)
zastępca przewodniczącego:
dr Helena Krupicka (Uniwersytet Wrocławski)
sekretarz:
inż. Maria Woźniak (Uniwersytet Wrocławski)
członkowie:
mgr Jacek Jagiełło (Uniwersytet Wrocławski)
dr Tomasz Jurdziński (Uniwersytet Wrocławski)
dr Przemysława Kanarek (Uniwersytet Wrocławski)
dr Witold Karczewski (Uniwersytet Wrocławski)

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

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

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

Górnośląski Komitet Okręgowy
przewodniczący:
prof. dr hab. Zbigniew Czech (Politechnika Śląska w Gliwicach)
zastępca przewodniczącego:
mgr inż. Sebastian Deorowicz (Politechnika Śląska w Gliwicach)
sekretarz:
mgr inż. Marcin Szołtysek (Politechnika Śląska w Gliwicach)
członkowie:
dr inż. Mariusz Boryczka (Uniwersytet Śląski w Sosnowcu)
mgr Wojciech Wieczorek (Uniwersytet Śląski w Sosnowcu)

Siedzibą Górnośląskiego Komitetu Okręgowego jest Politechnika Śląska w Gliwicach, ul. Akademicka 16.

Komitet Okręgowy w Krakowie
przewodniczący:
prof. dr hab. Paweł Idziak (Uniwersytet Jagielloński)
zastępca przewodniczącego:
dr Maciej Ślusarek (Uniwersytet Jagielloński)
sekretarz:
mgr Edward Szczypka (Uniwersytet Jagielloński)
członkowie:
mgr Henryk Białek (Kuratorium Oświaty w Krakowie)
dr inż. Janusz Majewski (Akademia Górniczo-Hutnicza w Krakowie)

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

Jury Olimpiady Informatycznej

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

Michał Adamaszek
Jarosław Byrka
mgr Krzysztof Ciebiera
Tomasz Czajka
Wojciech Dudek
Andrzej Gąsienica-Samek
mgr Łukasz Kowalik
Tomasz Malesiński
mgr Marcin Mucha
Krzysztof Onak
Arkadiusz Paterek
mgr Remigiusz Różycki
Rafał Rusin
mgr Marcin Sawicki
Piotr Sankowski
Krzysztof Sikora
Marcin Stefaniak
Tomasz Waleń
Paweł Wolff

ZAWODY I STOPNIA

W IX Olimpiadzie Informatycznej wzięło udział 1334 zawodników. Decyzją Komitetu Głównego Olimpiady do zawodów zostało dopuszczonych 42 uczniów gimnazjum.

- Gimnazjum nr 24 przy III L. O. w Gdyni: 6 uczniów
- Gimnazjum nr 50 w Bydgoszczy: 5
- Gimnazjum Akademickie w Toruniu: 3
- Gimnazjum nr 16 w Szczecinie: 3
- Gimnazjum w Ślemieniu: 3
- Gimnazjum nr 2 w Kielcach: 2
- Gimnazjum im. Boh. Westerplatte w Warszawie: 1
- Gimnazjum nr 1 w Gdyni: 1
- Gimnazjum nr 1 w Zambrowie: 1
- Gimnazjum nr 11 w Toruniu: 1
- Gimnazjum nr 11 w Rzeszowie: 1
- Gimnazjum nr 13 w Poznaniu: 1
- Gimnazjum nr 13 we Wrocławiu: 1
- Gimnazjum nr 16 w Warszawie: 1
- Gimnazjum nr 2 w Warszawie: 1
- Gimnazjum nr 2 w Brwinowie: 1
- Gimnazjum nr 34 w Łodzi: 1

- Gimnazjum nr 55 w Warszawie: 1
- Gimnazjum nr 6 w Rzeszowie: 1
- Gimnazjum nr 7 w Warszawie: 1
- Gimnazjum w Jasienicy: 1
- Publiczne Gimnazjum nr 3 w Kluczborku: 1
- II Prywatne Gimnazjum w Katowicach: 1
- Gimnazjum w Tarnowie Podgórnym: 1
- Katolickie Gimnazjum Społeczne w Biskupcu: 1
- Miejskie Gimnazjum nr 2 w Piekarach Śląskich: 1

Kolejność województw pod względem liczby uczestników była następująca:

mazowieckie	202
śląskie	159
małopolskie	141
pomorskie	140
kujawsko-pomorskie	100
dolnośląskie	89
wielkopolskie	82
łódzkie	71
podkarpackie	67
zachodniopomorskie	53
lubelskie	46
warmińsko-mazurskie	43
podlaskie	42
świętokrzyskie	39
lubuskie	31
opolskie	29

W zawodach I stopnia najliczniej reprezentowane były szkoły:

V L. O. im. A. Witkowskiego w Krakowie	43 uczniów
VIII L. O. im. A. Mickiewicza w Poznaniu	43
XIV L. O. im. S. Staszica w Warszawie	31
III L. O. im. Marynarki Wojennej RP w Gdyni	30
VI L. O. im. W. Sierpińskiego w Gdyni	16
IV L. O. im. T. Kościuszki w Toruniu	14
XIII L. O. im. L. Lisa-Kuli w Warszawie	13
XIV L. O. im. Polonii Belgijskiej we Wrocławiu	13
I L. O. im. A. Mickiewicza w Białymstoku	12
V L. O. im. S. Żeromskiego w Gdańsku	12
I L. O. im. S. Staszica w Lublinie	11
IV L. O. im. Kazimierza Wielkiego w Bydgoszczy	11
VI L. O. im. J. i J. Śniadeckich w Bydgoszczy	11
XXVII L. O. im. T. Czackiego w Warszawie	11
III L. O. im. A. Mickiewicza we Wrocławiu	10
L. O. im. Króla Władysława Jagiełły w Dębicy	10
XII L. O. im. S. Wyspiańskiego w Łodzi	9
Z. S. Energetycznych, XIII L. O. w Gdańsku	9
I L. O. im. B. Krzywoustego w Głogowie	8
I L. O. im. C. K. Norwida w Bydgoszczy	8
VIII L. O. im. M. Skłodowskiej-Curie w Katowicach	8
XX VIII L. O. im. J. Kochanowskiego w Warszawie	8
Z. S. O. nr 1 im. St. Dubois w Koszalinie	8
I L. O. im. B. Nowodworskiego w Krakowie	7
I L. O. im. W. Łukasińskiego w Dąbrowie Górniczej	7

II L. O. im. C. K. Norwida w Tychach	7
II L. O. im. M. Konopnickiej w Inowrocławiu	7
V L. O. w Bielsku-Białej	7
Gimnazjum nr 24 przy III L. O. w Gdyni	6
Gdyńskie Liceum Autorskie	6
I L. O. im. M. Kopernika w Gdańsku	6
I L. O. im. M. Kopernika w Łodzi	6
I L. O. im. Ruy Barbosa w Warszawie	6
I L. O. im. T. Kościuszki w Legnicy	6
I L. O. im. Ziemi Kujawskiej we Włocławku	6
II L. O. im. H. Kołłątaja w Wałbrzychu	6
II L. O. im. R. Traugutta w Częstochowie	6
Katolickie L. O. Ojców Pijarów w Krakowie	6
Techniczne Zakłady Naukowe w Częstochowie	6
V L. O. im. Ks. J. Poniatowskiego w Warszawie	6
V L. O. w Elblągu	6
VI L. O. im. J. Kochanowskiego w Radomiu	6
VIII L. O. im. Władysława IV w Warszawie	6
X L. O. im. Królowej Jadwigi w Warszawie	6
XIII L. O. w Szczecinie	6
Gimnazjum nr 50 w Bydgoszczy	5
I L. O. im. J. Śniadeckiego w Pabianicach	5
I L. O. im. M. Konopnickiej w Suwałkach	5
II L. O. im. A. Mickiewicza w Słupsku	5
II L. O. im. R. Traugutta w Częstochowie	5
II L. O. im. W. Wróblewskiego w Gliwicach	5
III L. O. im. Bohaterów Westerplatte w Gdańsku	5
III L. O. im. Św. Jana Kantego w Poznaniu	5
IV L. O. im. K. K. Baczyńskiego w Olkuszu	5
IX L. O. im. C. K. Norwida w Częstochowie	5
Katolickie Społeczne L. O. Przymierza Rodzin w Warszawie	5
L. O. im. K.E.N. w Stalowej Woli	5
Z. S. Elektronicznych i Technicznych w Olsztynie	5
Zespół Szkół Mechaniczno-Elektrycznych w Żywcu	5

Najliczniej reprezentowane były miasta:

Warszawa	144	Tychy	10
Kraków	88	Zielona Góra	10
Gdynia	66	Gorzów Wielkopolski	9
Poznań	55	Siedlce	9
Gdańsk	42	Włocławek	9
Bydgoszcz	39	Olkusz	8
Wrocław	37	Opole	8
Łódź	32	Pabianice	8
Częstochowa	27	Suwałki	8
Lublin	25	Legnica	7
Toruń	24	Piotrków Trybunalski	7
Szczecin	22	Sosnowiec	7
Katowice	20	Brodnica	6
Białystok	18	Mielec	6
Kielce	18	Nysa	6
Rzeszów	15	Słupsk	6
Gliwice	14	Wałbrzych	6
Bielsko Biała	13	Chrzanów	5
Olsztyn	13	Jastrzębie Zdrój	5
Elbląg	12	Ostrowiec Świętokrzyski	5
Inowrocław	12	Piła	5
Głogów	11	Sanok	5
Koszalin	11	Skarżysko-Kamienna	5
Dąbrowa Górnicza	10	Tarnowskie Góry	5

Dębica	10	Zgierz	5
Radom	10	Zgorzelec	5
Stalowa Wola	10	Żywiec	5
Tarnów	10		

Zawodnicy uczęszczali do następujących klas:

do klasy I	gimnazjum	2 zawodników
do klasy II	gimnazjum	12
do klasy III	gimnazjum	28
do klasy I	szkoły średniej	150
do klasy II	szkoły średniej	288
do klasy III	szkoły średniej	454
do klasy IV	szkoły średniej	359
do klasy V	szkoły średniej	34

7 zawodników nie podało informacji o klasie, do której uczęszczają.

W zawodach I stopnia zawodnicy mieli do rozwiązania pięć zadań: „Koleje", „Komiwojażer Bajtazar", „Superskoczek", „Wyspa" i „Zamek". Już po ogłoszeniu wyników wykryto błąd w oprogramowaniu sprawdzającym. W dniu 11 grudnia, w poprawionym środowisku testowym dokonano ponownego, dwukrotnego sprawdzenia wszystkich nadesłanych prac. Osiągnięto bardzo wysoką stabilność wyników. Sprawdzanie przeprowadzono na tych samych danych testowych co za pierwszym razem, na tych samych komputerach i przy tych samych limitach czasowych. W przypadku zadań SUM i ZAM wyniki nie zmieniły się. W przypadku zadań WYS, KOL i KOM zawodnicy odzyskiwali niesłusznie utracone punkty. Dodatkowo w przypadku zadania KOL okazało się, że w poprzednim sprawdzaniu niektóre błędne programy uzyskały dodatnią liczbę punktów za niepoprawne odpowiedzi. Ponowne sprawdzanie skorygowało także tę punktację. Po zapoznaniu się z procedurą i wynikami ponownego sprawdzania Komitet Główny Olimpiady Informatycznej postanowił:

1. Przyjąć wyniki ponownego sprawdzania jako oficjalne wyniki I stopnia IX Olimpiady Informatycznej.
2. Zakwalifikować do zawodów II stopnia tylko te osoby, które w ponownym sprawdzeniu uzyskały co najmniej 270 punktów.
3. Zaprosić na zawody II stopnia, poza konkursem i na koszt organizatorów, wszystkie te osoby, które w pierwszym sprawdzeniu uzyskały wynik co najmiej 270 punktów (co kwalifikowało je do zawodów II stopnia), jednak w powtórnym sprawdzeniu uzyskały mniej niż 270 punktów.

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

- Koleje

	liczba zawodników	czyli
100 pkt.	109	8,2%
99–75 pkt.	276	20,7%
74–50 pkt.	260	19,5%
49–1 pkt.	518	38,8%
0 pkt.	171	12,8%

- Komiwojażer Bajtazar

	liczba zawodników	czyli
100 pkt.	174	13,0%
99–75 pkt.	100	7,5%
74–50 pkt.	85	6,4%
49–1 pkt.	617	46,3%
0 pkt.	358	26,8%

- Superskoczek

	liczba zawodników	czyli
100 pkt.	61	4,5%
99–75 pkt.	105	7,9%
74–50 pkt.	81	6,1%
49–1 pkt.	181	13,6%
0 pkt.	906	67,9%

- Wyspa

	liczba zawodników	czyli
100 pkt.	331	24,8%
99–75 pkt.	87	6,6%
74–50 pkt.	276	20,7%
49–1 pkt.	450	33,7%
0 pkt.	190	14,2%

- Zamek

	liczba zawodników	czyli
100 pkt.	144	10,8%
99–75 pkt.	41	3,1%
74–50 pkt.	93	7,0%
49–1 pkt.	361	27,0%
0 pkt.	695	52,1%

W sumie za wszystkie 5 zadań:

SUMA	liczba zawodników	czyli
500 pkt.	20	1,5%
499–375 pkt.	146	11,0%
374–250 pkt.	183	13,7%
249–1 pkt.	930	69,7%
0 pkt.	55	4,1%

Wszyscy zawodnicy otrzymali informacje ze swoimi wynikami. Na stronie internetowej Olimpiady udostępnione były testy na podstawie których oceniano prace.

ZAWODY II STOPNIA

Do zawodów II stopnia zakwalifikowano 314 zawodników, którzy osiągnęli w zawodach I stopnia wynik nie mniejszy niż 270 pkt. Trzech zawodników nie zgłosiło się na zawody. W zawodach II stopnia uczestniczyło 311 zawodników.

Na zawody II stopnia zaproszono do udziału poza konkursem 27 zawodników, z których 15 uczestniczyło w zawodach.

Zawody II stopnia odbyły się w dniach 12–14 lutego 2002 r. w pięciu stałych okręgach oraz w Sopocie i Rzeszowie:

- w Toruniu — 35 zawodników z następujących województw:
  - kujawsko-pomorskie (23)
  - warmińsko-mazurskie (4)
  - podlaskie (7)
  - zachodniopomorskie (1)
- we Wrocławiu — 57 zawodników z następujących województw:
  - dolnośląskie (22)
  - lubuskie (1)
  - łódzkie (5)
  - opolskie (1)
  - wielkopolskie (21)
  - zachodniopomorskie (7)
- w Warszawie — 68 zawodników z następujących województw:
  - lubelskie (2)
  - łódzkie (3)
  - mazowieckie (58)

  - podlaskie (4)
  - warmińsko-mazurskie (1)
- w Krakowie — 51 zawodników z następujących województw:
  - małopolskie (51)
- w Katowicach — 26 zawodników z następujących województw:
  - małopolskie (1)
  - opolskie (2)
  - śląskie (21)
  - świętokrzyskie (2)
- w Sopocie — 42 zawodników z następujących województw:
  - kujawsko-pomorskie (1)
  - pomorskie (37)
  - zachodniopomorskie (4)
- w Rzeszowie — 32 zawodników z następujących województw:
  - lubelskie (6)
  - małopolskie (1)
  - podkarpackie (18)
  - śląskie (1)
  - świętokrzyskie (6)

W zawodach II stopnia najliczniej reprezentowane były szkoły:

V L.O. im. A. Witkowskiego w Krakowie	36 uczniów
III L.O. im. Marynarki Wojennej RP w Gdyni	20
VIII L.O. im. A. Mickiewicza w Poznaniu	17
XIV L.O. im. S. Staszica w Warszawie	12
VI L.O. im. J. i J. Śniadeckich w Bydgoszczy	10
VI L. O. im. W. Sierpińskiego w Gdyni	8
XIV L. O. im. Polonii Belgijskiej we Wrocławiu	7
XXVII L. O. im. T. Czackiego w Warszawie	7
I L. O. w Białymstoku	5
VI L. O. im. J. Kochanowskiego w Radomiu	5
I L. O. im. S. Staszica w Lublinie	4
V L. O. im. Ks. J. Poniatowskiego w Warszawie	4
VIII L. O. im. Władysława IV w Warszawie	4
Zespół Szkół Ogólnokształcących nr 2 w Wałbrzychu	4
I L. O. im. M. Kopernika w Krośnie	3
I L. O. im. K.E.N w Sanoku	3
II L. O. im. St. Staszica w Starachowicach	3
III L. O. im. A. Mickiewicza we Wrocławiu	3
IV L. O. im. T. Kościuszki w Toruniu	3
Katolickie L. O. Przymierza Rodzin w Warszawie	3
L L. O. im. Ruy Barbosa w Warszawie	3
L. O. im. kr. Władysława Jagiełły w Dębicy	3
V L. O. w Bielsku-Białej	3
VIII L. O. im. M. Skłodowskiej-Curie w Katowicach	3

Najliczniej reprezentowane były miasta:

Kraków	47 zawodników
Warszawa	47
Gdynia	31
Poznań	18

Bydgoszcz	13
Wrocław	13
Szczecin	7
Lublin	6
Białystok	5
Gdańsk	5
Radom	5
Katowice	4
Łódź	4
Rzeszów	4
Toruń	4
Wałbrzych	4
Bielsko-Biała	3
Częstochowa	3
Dębica	3
Inowrocław	3
Krosno	3
Nowy Sącz	3
Sanok	3
Starachowice	3

W dniu 12 lutego odbyła się sesja próbna, podczas której zawodnicy rozwiązywali nie liczące się do ogólnej klasyfikacji zadanie „Izolator". W dniach konkursowych zawodnicy rozwiązywali zadania: „Działka", „Kurort narciarski", „Protokoły" oraz „Wyliczanka", każde oceniane maksymalnie po 100 punktów. Podczas zawodów okręgowych Jury wykryło przypadek próby ściągania. Zawodnik posiadał przy sobie ściągawki i dwie nieoznakowane dyskietki. Komitet w wyniku głosowania jednogłośnie zdyskwalifikował zawodnika.

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

- Izolator (zadanie próbne)

	liczba zawodników	czyli
100 pkt.	226	72,7%
99–75 pkt.	7	2,3%
74–50 pkt.	43	13,8%
49–1 pkt.	21	6,7%
0 pkt.	14	3,5%

- Działka

	liczba zawodników	czyli
100 pkt.	2	0,6%
99–75 pkt.	13	4,2%
74–50 pkt.	13	4,2%
49–1 pkt.	157	50,5%
0 pkt.	126	40,5%

- Kurort narciarski

	liczba zawodników	czyli
100 pkt.	81	26,0%
99–75 pkt.	32	10,3%
74–50 pkt.	36	11,6%
49–1 pkt.	68	21,9%
0 pkt.	94	30,2%

- Protokoły

	liczba zawodników	czyli
100 pkt.	10	3,2%
99–75 pkt.	9	2,9%
74–50 pkt.	38	12,2%
49–1 pkt.	138	44,4%
0 pkt.	116	37,3%

- Wyliczanka

	liczba zawodników	czyli
100 pkt.	20	64,3%
99–75 pkt.	3	1,0%
74–50 pkt.	23	7,4%
49–1 pkt.	95	30,6%
0 pkt.	170	54,7%

W sumie za wszystkie 4 zadania:

SUMA	liczba zawodników	czyli
400 pkt.	0	0,0%
399–300 pkt.	7	2,2%
299–200 pkt.	34	10,9%
199–1 pkt.	235	75,6%
0 pkt.	35	11,3%

Zawodnikom przesłano informacje z wynikami zawodów.

ZAWODY III STOPNIA

Zawody III stopnia odbyły się w ośrodku firmy Combidata Poland S.A. w Sopocie w dniach od 15 do 19 kwietnia 2001 r.

W zawodach III stopnia wzięło udział 48 najlepszych uczestników zawodów II stopnia, którzy uzyskali wynik nie mniejszy niż 196 pkt. Jeden zawodnik uczestniczył poza konkursem. Zawodnicy pochodzili z następujących województw:

mazowieckie	15
małopolskie	9
pomorskie	5
kujawsko-pomorskie	3
dolnośląskie	3
podkarpackie	3
śląskie	3
zachodniopomorskie	2
świętokrzyskie	2
wielkopolskie	2
podlaskie	1

Niżej wymienione szkoły miały w finale więcej niż jednego zawodnika:

XIV L.O. im. St. Staszica w Warszawie	8 zawodników
V L. O. im. A. Witkowskiego w Krakowie	7
I L. O. im. M. Kopernika w Krośnie	2
III L. O. im. Marynarki Wojennej RP w Gdyni	2
VIII L. O. im. A. Mickiewicza w Poznaniu	2
XIV L. O. im. Polonii Belgijskiej we Wrocławiu	2

15 kwietnia odbyła się sesja próbna, podczas której zawodnicy rozwiązywali nie liczące się do ogólnej klasyfikacji zadanie „Minusy”. W dniach konkursowych zawodnicy rozwiązywali zadania: „Narciarze”, „Waga”, „Liczby B-gładkie”, „Nawiasy” i „Szyfr”, każde oceniane maksymalnie po 100 punktów.

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

- Minusy (zadanie próbne)

	liczba zawodników	czyli
100 pkt.	42	87,5%
99–75 pkt.	3	6,3%
74–50 pkt.	1	2,1%
49–1 pkt.	0	0,0%
0 pkt.	2	4,1%

- Liczby B-gładkie

	liczba zawodników	czyli
100 pkt.	0	0,0%
99–75 pkt.	1	2,1%
74–50 pkt.	0	0,0%
49–1 pkt.	44	91,7%
0 pkt.	3	6,2%

- Narciarze

	liczba zawodników	czyli
100 pkt.	3	6,2%
99–75 pkt.	1	2,1%
74–50 pkt.	5	10,4%
49–1 pkt.	20	41,7%
0 pkt.	19	39,6%

- Nawiasy

	liczba zawodników	czyli
100 pkt.	0	0,0%
99–75 pkt.	1	2,1%
74–50 pkt.	5	10,4%
49–1 pkt.	31	64,6%
0 pkt.	11	22,9%

- Szyfr

	liczba zawodników	czyli
100 pkt.	0	0,0%
99–75 pkt.	1	2,1%
74–50 pkt.	5	10,4%
49–1 pkt.	31	64,6%
0 pkt.	11	22,9%

- Waga

	liczba zawodników	czyli
100 pkt.	6	12,5%
99–75 pkt.	4	8,3%
74–50 pkt.	2	4,2%
49–1 pkt.	15	31,2%
0 pkt.	21	43,8%

W sumie za wszystkie 5 zadań:

SUMA	liczba zawodników	czyli
500 pkt.	0	0,0%
499–375 pkt.	1	2,1%
374–250 pkt.	5	10,4%
249–1 pkt.	42	87,5%
0 pkt.	0	0,0%

W dniu 19 kwietnia 2002 roku, w Sali Posiedzeń Urzędu Miasta w Sopocie, ogłoszono wyniki finału IX Olimpiady Informatycznej 2001/2002 i rozdano nagrody ufundowane przez: PROKOM Software S.A., Ogólnopolską Fundację Edukacji Komputerowej, Wydawnictwa Naukowo-Techniczne i Olimpiadę Informatyczną. Laureaci I, II i III miejsca otrzymali, odpowiednio, złote, srebrne i brązowe medale. Poniżej zestawiono listę wszystkich laureatów i wyróżnionych finalistów:

(1) Paweł Parys, L. O. im. St. Staszica w Tarnowskich Górach, laureat I miejsca, 452 pkt. (komputer — PROKOM; książka, roczny abonament na książki — WNT; upominek — OFEK)

(2) Bartosz Walczak, V L. O. im. A. Witkowskiego w Krakowie, laureat I miejsca, 364 pkt. (komputer — PROKOM; książka — WNT; upominek — OFEK)

(3) Karol Cwalina, XIV L. O. im. St. Staszica w Warszawie, laureat II miejsca, 296 pkt. (komputer — PROKOM; książka — WNT; upominek — OFEK)

(4) Marcin Michalski, III L. O. im. Marynarki Wojennej RP, laureat II miejsca, 269 pkt. (komputer — PROKOM; książka — WNT; upominek — OFEK)

(5) Marcin Pilipczuk, XIV L. O. im. St. Staszica w Warszawie, laureat II miejsca, 265 pkt. (drukarka laserowa — PROKOM; książka — WNT; upominek — OFEK)

(6) Piotr Stańczyk, XIV L. O. im. St. Staszica w Warszawie, laureat II miejsca, 262 pkt. (drukarka laserowa — PROKOM; książka — WNT; upominek — OFEK)

(7) Michał Jaszczyk, XIII L. O. w Szczecinie, laureat III miejsca, 249 pkt. (drukarka laserowa — PROKOM; książka — WNT; upominek — OFEK)

(8) Wojciech Matyjewicz, III L. O. w Tarnowie, laureat III miejsca, 239 pkt. (drukarka laserowa — PROKOM; książka — WNT; upominek — OFEK)

(9) Tomasz Wawrzyniak, XIV L. O. im. Polonii Belgijskiej we Wrocławiu, laureat III miejsca, 209 pkt. (drukarka laserowa — PROKOM; książka — WNT; upominek — OFEK)

(10) Szymon Acedański, VIII L. O. w Katowicach, laureat III miejsca, 208 pkt. (drukarka laserowa — PROKOM; książka — WNT; upominek — OFEK)

(11) Jan Wróbel, V L. O. im. A. Witkowskiego w Krakowie, laureat III miejsca, 203 pkt. (drukarka laserowa — PROKOM; książka — WNT; upominek — OFEK)

(12) Arkadiusz Pawlik, V L. O. im. A. Witkowskiego w Krakowie, finalista z wyróżnieniem, 174 pkt. (drukarka atramentowa — PROKOM; książka — WNT; upominek — OFEK)

(13) Marek Żylak, I L. O. im. B. Prusa w Siedlcach, finalista z wyróżnieniem, 171 pkt. (drukarka atramentowa — PROKOM; książka — WNT; upominek — OFEK)

(14) Piotr Tabor, XIV L. O. im. St. Staszica w Warszawie, finalista z wyróżnieniem, 167 pkt. (drukarka atramentowa — PROKOM; książka — WNT; upominek — OFEK)

(15) Marcin Jurkowski, VI L. O. im. J. i J. Śniadeckich w Bydgoszczy, finalista z wyróżnieniem, 166 pkt. (drukarka atramentowa — PROKOM; książka — WNT; upominek — OFEK)

(16) Mateusz Stefek, L. O. im. A. Osuchowskiego w Cieszynie, finalista z wyróżnieniem, 163 pkt. (drukarka atramentowa — PROKOM; książka — WNT; upominek — OFEK)

(17) Bartłomiej Romański, XIV L. O. im. St. Staszica w Warszawie, finalista z wyróżnieniem, 154 pkt. (drukarka atramentowa — PROKOM; książka — WNT; upominek — OFEK)

(18) Marcin Wielgus, I L. O. im. B. Nowodworskiego w Krakowie, finalista z wyróżnieniem, 153 pkt. (drukarka atramentowa — PROKOM; książka — WNT; upominek — OFEK)

(19) Jan Prażuch, V L. O. im. A. Witkowskiego w Krakowie, finalista z wyróżnieniem, 149 pkt. (drukarka atramentowa — PROKOM; książka — WNT; upominek — OFEK)

Pozostali finaliści otrzymali książkę ufundowaną przez WNT i upominek ufundowany przez OFEK. Oto ich lista w kolejności alfabetycznej:

- Tomasz Batkiewicz, IV L. O. im. C. K. Norwida w Ostrowcu Świętokrzyskim
- Piotr Cerobski, XIV L. O. im. St. Staszica w Warszawie
- Andrzej Chodor, IV L. O. im. Hanki Sawickiej
- Przemysław Dębiak, IX L. O. w Gdańsku
- Robert Dyczkowski, XIV L. O. im. St. Staszica w Warszawie
- Przemysław Dzierżak, VI L. O. w Gdyni
- Mateusz Goryca, VI L. O. im. J. Kochanowskiego w Radomiu
- Paweł Grajewski, I L. O. w Suwałkach

- Nhat Viet Ha, LXXX L. O. im. L. Staffa w Warszawie
- Tomasz Idziaszek, XLI L. O. im. L. Lelewela w Warszawie
- Jan Kaczmarczyk, V L. O. im. A. Witkowskiego w Krakowie
- Piotr Karłowicz, Niepubliczne L. O. nr 27 w Starej Miłośnie
- Tomasz Kazimierczuk, I L. O. im. M. Kopernika w Krośnie
- Marek Kirejczyk, I L. O. im. T. Reytana w Warszawie
- Marcin Kosieradzki, V L. O. im. J. Poniatowskiego w Warszawie
- Marcin Koszów, I L. O. w Głogowie
- Łukasz Krygier, III L. O. im. B. Lindego w Toruniu
- Łukasz Pobereżnik, V L. O. im. A. Witkowskiego w Krakowie
- Robert Rychcicki, X L. O. w Szczecinie
- Wojciech Samotij, XIV L. O. im. Polonii Belgijskiej we Wrocławiu
- Dawid Sieradzki, VIII L. O. im. A. Mickiewicza w Poznaniu
- Sebastian Stafiej, II L. O. im. M. Konopnickiej w Inowrocławiu
- Jakub Suder, V L. O. im. A. Witkowskiego w Krakowie
- Wiktor Tomczak, I L. O. im. M. Kopernika w Gdańsku
- Szymon Wąsik, VIII L. O. im. A. Mickiewicza w Poznaniu
- Piotr Włodarczyk, XIV L. O. im. St. Staszica w Warszawie
- Rafał Wojtak, Zespół Szkół Elektronicznych w Rzeszowie
- Jarosław Wrona, I L. O. im. M. Kopernika w Krośnie
- Marcin Zawadzki, III L. O. im. Marynarki Wojennej RP w Gdyni

Ogłoszono komunikat o powołaniu:

- reprezentacji Polski na Międzynarodową Olimpiadę Informatyczną oraz Olimpiadę Informatyczną Centralnej Europy w składzie:

  (1) Paweł Parys
  (2) Bartosz Walczak
  (3) Karol Cwalina
  (4) Marcin Michalski

  zawodnikami rezerwowymi zostali:

  (5) Marcin Pilipczuk
  (6) Piotr Stańczyk

- reprezentacji Polski na Bałtycką Olimpiadę Informatyczną w składzie:

  (1) Paweł Parys
  (2) Bartosz Walczak
  (3) Karol Cwalina
  (4) Marcin Michalski
  (5) Marcin Pilipczuk
  (6) Piotr Stańczyk

  zawodnikami rezerwowymi zostali:

(7) Michał Jaszczyk
(8) Wojciech Matyjewicz

- na obóz czesko-polsko-słowacki: reprezentacja (wraz z rezerwowymi) na Międzynarodową Olimpiadę Informatyczną,
- na obóz rozwojowo-treningowy im. A. Kreczmara dla finalistów Olimpiady Informatycznej: reprezentanci na Międzynarodową Olimpiadę Informatyczną, laureaci i finaliści Olimpiady, z pominięciem zawodników z ostatnich klas szkół średnich.

Sekretariat Olimpiady wystawił łącznie 48 zaświadczenia o zakwalifikowaniu do zawodów III stopnia celem przedłożenia dyrekcji szkoły.

Sekretariat wystawił łącznie 18 zaświadczeń o uzyskaniu tytułu laureata i 30 zaświadczeń o uzyskaniu tytułu finalisty IX Olimpiady Informatycznej celem przedłożenia władzom szkół wyższych.

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

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

- Andrzej Dyrek (Uniwersytet Jagielloński, V L. O. im. A. Witkowskiego w Krakowie)
  - Bartosz Walczak (laureat I miejsca)
  - Jan Wróbel (laureat III miejsca)
  - Arkadiusz Pawlik (finalista z wyróżnieniem)
  - Jan Prażuch (finalista z wyróżnieniem)
  - Jan Kaczmarczyk (finalista)
  - Łukasz Pobereżnik (finalista)
  - Jakub Suder (finalista)
- Andrzej Gąsienica-Samek (student Uniwersytetu Warszawskiego)
  - Karol Cwalina (laureat II miejsca)
  - Piotr Stańczyk (laureat II miejsca)
  - Bartłomiej Romański (finalista z wyróżnieniem)
  - Marek Żylak (finalista z wyróżnieniem)
  - Piotr Cerobski (finalista)
  - Robert Dyczkowski (finalista)
  - Nhat Viet Ha (finalista)
  - Tomasz Idziaszek (finalista)
  - Piotr Karłowicz (finalista)
  - Marek Kirejczyk (finalista)
- Ryszard Szubartowski (III L. O. im. Marynarki Wojennej RP w Gdyni)
  - Marcin Michalski (laureat II miejsca)
  - Przemysław Dzierżak (finalista)
  - Marcin Zawadzki (finalista)
- Janina Matyjewicz (Gimnazjum nr 7 w Tarnowie)
  - Wojciech Matyjewicz (laureat III miejsca)
- Aleksy Schubert (XIV L. O. im. St. Staszica w Warszawie)
  - Marcin Pilipczuk (laureat III miejsca)
  - Piotr Tabor (finalista z wyróżnieniem)
- Michał Szuman (XIII L. O. w Szczecinie)
  - Michał Jaszczyk (laureat III miejsca)

- Michał Baczyński (Uniwersytet Śląski w Katowicach)
  - Szymon Acedański (laureat III miejsca)
- Alicja Wawrzyniak (Uniwersytet Wrocławski)
  - Tomasz Wawrzyniak (laureat III miejsca)
- Jan Chróścicki (I L. O. im. B. Prusa w Siedlcach)
  - Marek Żylak (finalista z wyróżnieniem)
- Witold Jarnicki (V L. O. im. A. Witkowskiego w Krakowie)
  - Arkadiusz Pawlik (finalista z wyróżnieniem)
- Małgorzata Jurkowska (Akademia Medyczna w Bydgoszczy)
  - Marcin Jurkowski (finalista z wyróżnieniem)
- Maria Wielgus (Akademickie Centrum Komputerowe Cyfronet AGH w Krakowie)
  - Marcin Wielgus (finalista z wyróżnieniem)
- Ewa Batkiewicz (IV L. O. im. C. K. Norwida w Ostrowcu Świętokrzyskim)
  - Tomasz Batkiewicz (finalista)
- Michał Bednarczyk (Huta Szkła „Irena” w Inowrocławiu)
  - Sebastian Stafiej (finalista)
- Leszek Chodor (PŚK w Kielcach)
  - Andrzej Chodor (finalista)
- Bernadetta Grajewska (Rejonowy Urząd Poczty w Suwałkach)
  - Paweł Grajewski (finalista)
- Lucyna Kukła (I L. O. w Głogowie)
  - Marcin Koszów (finalista)
- Bogna Lubańska (V L. O. im. J. Poniatowskiego w Warszawie)
  - Marcin Kosieradzki (finalista)
- Mirosław Modzelewski (Politechnika Radomska, Katedra Informatyki)
  - Mateusz Goryca (finalista)
- Jolanta Nowak (IV L. O. im. H. Sawickiej w Kielcach)
  - Andrzej Chodor (finalista)
- Bartosz Nowierski (student Politechniki Poznańskiej)
  - Dawid Sieradzki (finalista)
  - Szymon Wąsik (finalista)
- Andrzej Piotrowski (I L. O. im. M. Kopernika w Krośnie)
  - Jarosław Wrona (finalista)
- Małgorzata Rostkowska (XIV L. O. im. S. Staszica w Warszawie)
  - Piotr Włodarczyk (finalista)
- Andrzej Ruszała (I L. O. im. M. Kopernika w Krośnie)
  - Tomasz Kazimierczuk (finalista)

- Romuald Rychcicki (Szczecin)
  - Robert Rychcicki (finalista)
- Dominik Samotij (student Akademii Medycznej we Wrocławiu)
  - Wojciech Samotij (finalista)
- Hanna Stachera (XIV L. O. im. St. Staszica w Warszawie)
  - Piotr Cerobski (finalista)
- Iwona Waszkiewicz (VI L. O. im. J. i J. Śniadeckich w Bydgoszczy)
  - Marcin Jurkowski (finalista)
- Danuta Zaremba (III L. O. im. S. B. Lindego w Toruniu)
  - Łukasz Krygier (finalista)

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

W III etapie został zorganizowany towarzyszący Olimpiadzie Internetowy Konkurs Informatyczny „Pogromcy Algorytmów". Uczestnik Konkursu, uczeń szkoły średniej z najlepszym wynikiem, został zaproszony do uczestniczenia w obozie rozwojowo-treningowym im. A. Kreczmara dla finalistów Olimpiady Informatycznej.

Warszawa, 3 czerwca 2002 roku

Regulamin Olimpiady Informatycznej

§1 WSTĘP

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

§2 CELE OLIMPIADY INFORMATYCZNEJ

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

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

(3) Stymulowanie aktywności poznawczej młodzieży informatycznie uzdolnionej.

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

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

(6) Wyłanianie reprezentacji Rzeczypospolitej Polskiej na Międzynarodową Olimpiadę Informatyczną, Olimpiadę Informatyczną Centralnej Europy i inne międzynarodowe zawody informatyczne.

§3 ORGANIZACJA OLIMPIADY

(1) Olimpiadę przeprowadza Komitet Główny Olimpiady Informatycznej.

(2) Olimpiada Informatyczna jest trójstopniowa.

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

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

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

(6) Integralną częścią rozwiązania zadań zawodów I, II i III stopnia jest program napisany w języku programowania i środowisku, wybranym z listy języków i środowisk ustalanej przez Komitet Główny corocznie przed rozpoczęciem zawodów i ogłaszanej w „Zasadach organizacji zawodów".

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

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

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

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

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

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

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

§4 KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

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

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

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

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

(5) Komitet powołuje i rozwiązuje komitety okręgowe Olimpiady.

(6) Komitet:

- (a) opracowuje szczegółowe „Zasady organizacji zawodów”, które są ogłaszane razem z treścią zadań zawodów I stopnia Olimpiady,
- (b) powołuje i odwołuje członków Jury Olimpiady, które jest odpowiedzialne za sprawdzenie zadań,
- (c) udziela wyjaśnień w sprawach dotyczących Olimpiady,
- (d) ustala listy laureatów i wyróżnionych uczestników oraz kolejność lokat,
- (e) przyznaje uprawnienia i nagrody rzeczowe wyróżniającym się uczestnikom Olimpiady,
- (f) ustala kryteria wyłaniania uczestników uprawnionych do startu w Międzynarodowej Olimpiadzie Informatycznej, Olimpiadzie Informatycznej Centralnej Europy i innych międzynarodowych zawodach informatycznych, publikuje je w „Zasadach organizacji zawodów”, oraz ustala ostateczną listę reprezentacji.

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

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

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

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

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

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

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

(14) Przewodniczący:

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

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

(a) zadania Olimpiady,
(b) rozwiązania zadań Olimpiady przez okres 2 lat,
(c) rejestr wydanych zaświadczeń i dyplomów laureatów,
(d) listy laureatów i ich nauczycieli,
(e) dokumentację statystyczną i finansową.

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

§5 KOMITETY OKRĘGOWE

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

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

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

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

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

§6 PRZEBIEG OLIMPIADY

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

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

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

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

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

§7 UPRAWNIENIA I NAGRODY

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

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

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

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

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

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

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

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

§8 FINANSOWANIE OLIMPIADY

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

§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.

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

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

Warszawa, 17 września 2001 roku

Zasady organizacji zawodów w roku szkolnym 2001/2002

Olimpiada Informatyczna jest organizowana przy współudziale firmy PROKOM Software S.A. Podstawowym aktem prawnym dotyczącym Olimpiady jest Regulamin Olimpiady Informatycznej, którego pełny tekst znajduje się w kuratoriach. Poniższe zasady są uzupełnieniem tego Regulaminu, zawierającym szczegółowe postanowienia Komitetu Głównego Olimpiady Informatycznej o jej organizacji w roku szkolnym 2001/2002.

§1 WSTĘP

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

§2 ORGANIZACJA OLIMPIADY

(1) Olimpiadę przeprowadza Komitet Główny Olimpiady Informatycznej.

(2) Olimpiada Informatyczna jest trójstopniowa.

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

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

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

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

(7) Do zawodów II stopnia zostanie zakwalifikowanych 280 uczestników, których rozwiązania zadań I stopnia zostaną ocenione najwyżej; do zawodów III stopnia — 40 uczestników, których rozwiązania zadań II stopnia zostaną ocenione najwyżej. Komitet Główny może zmienić podane liczby zakwalifikowanych uczestników co najwyżej o 20%.

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

(9) Terminarz zawodów:

zawody I stopnia — 15.10–12.11.2001 r.
  ogłoszenie wyników:
    w witrynie Olimpiady — 8.12.2001 r.
    pocztą — 21.12.2001 r.
zawody II stopnia — 12–14.02.2002 r.
  ogłoszenie wyników:
    w witrynie Olimpiady — 23.02.2002 r.
    pocztą — 7.03.2002 r.
zawody III stopnia — 15-19.04.2002 r.

§3 WYMAGANIA DOTYCZĄCE ROZWIĄZAŃ ZADAŃ ZAWODÓW I STOPNIA

(1) Zawody I stopnia polegają na samodzielnym rozwiązywaniu zadań eliminacyjnych (niekoniecznie wszystkich) i przesłaniu rozwiązań do Olimpiady Informatycznej. Możliwe są tylko dwa sposoby przesyłania:

- poprzez witrynę Olimpiady o adresie: `www.oi.pjwstk.waw.pl` do godziny 12.00 (w południe) dnia 12 listopada 2001r. Olimpiada nie ponosi odpowiedzialności za brak możliwości przekazania rozwiązań przez witrynę w sytuacji nadmiernego obciążenia lub awarii serwisu. Odbiór przesyłki zostanie potwierdzony przez Olimpiadę zwrotnym listem elektronicznym (prosimy o zachowanie tego listu). Brak potwierdzenia może oznaczać, że rozwiązanie nie zostało poprawnie zarejestrowane. W tym przypadku zawodnik powinien nadać swoje rozwiązanie przesyłką poleconą za pośrednictwem zwykłej poczty. Szczegóły dotyczące sposobu postępowania przy przekazywaniu zadań i związanej z tym rejestracji będą dokładnie podane w witrynie.
- pocztą, przesyłką poleconą, pod adresem:

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

  w nieprzekraczalnym terminie nadania do 12 listopada 2001 r. (decyduje data stempla pocztowego). Prosimy o zachowanie dowodu nadania przesyłki.

Rozwiązania dostarczane w inny sposób nie będą przyjmowane. W przypadku jednoczesnego zgłoszenia rozwiązania przez Internet i listem poleconym, ocenie podlega jedynie rozwiązanie wysłane listem poleconym. W takim przypadku konieczne jest również podanie w dokumencie zgłoszeniowym identyfikatora użytkownika użytego do zgłoszenia rozwiazań przez Internet (patrz pkt 10).

(2) Ocena rozwiązania zadania jest określana na podstawie wyników testowania programu i uwzględnia poprawność oraz efektywność metody rozwiązania użytej w programie.

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

(4) Rozwiązanie każdego zadania składa się z programu (tylko jednego) w postaci źródłowej i skompilowanej; imię i nazwisko uczestnika musi być podane w komentarzu na początku każdego programu.

(5) 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`

(6) Opcje kompilatora powinny być częścią tekstu programu.

(7) Program powinien odczytywać plik wejściowy z bieżącego katalogu i zapisywać plik wyjściowy również do bieżącego katalogu.

(8) Program nie powinien oczekiwać na jakąkolwiek czynność, np. naciśnięcie klawisza, ruch myszą, wpisanie liczby lub litery.

(9) Dane testowe są bezbłędne, zgodne z warunkami zadania i podaną specyfikacją wejścia.

(10) Uczestnik przysyła:

- jedną dyskietkę (jeden plik skompresowany metodą ZIP w przypadku korzystania z witryny), w formacie FAT (standard dla komputerów PC) zawierającą:
  - spis zawartości dyskietki oraz dane osobowe zawodnika w pliku nazwanym `SPIS.TRC`,
  - do każdego rozwiązanego zadania — programy w postaci źródłowej i skompilowanej,
  - dyskietka (plik ZIP) nie powinna zawierać żadnych podkatalogów,
- wypełniony dokument zgłoszeniowy (dostępny jako załącznik lub w witrynie internetowej Olimpiady).

(11) Poprzez witrynę o adresie `www.oi.pjwstk.waw.pl` można uzyskać odpowiedzi na pytania dotyczące Olimpiady. Pytania należy przysyłać na adres: `olimpiada@oeiizk.waw.pl`. Komitet Główny może nie udzielić odpowiedzi na pytanie z ważnych przyczyn, m.in. gdy jest ono niejednoznaczne lub dotyczy sposobu rozwiązania zadania. Prosimy wszystkich uczestników Olimpiady o regularne zapoznawanie się z ukazującymi się odpowiedziami.

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

§4 UPRAWNIENIA I NAGRODY

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

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

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

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

(5) Komitet Główny ustala skład reprezentacji Polski na XIV Międzynarodową Olimpiadę Informatyczną w 2002 roku na podstawie wyników zawodów III stopnia i regulaminu tej Olimpiady Międzynarodowej. Szczegółowe zasady zostaną podane po otrzymaniu formalnego zaproszenia na XIV Międzynarodową Olimpiadę Informatyczną.

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

(7) Wyznaczeni przez Komitet Główny reprezentanci Polski na olimpiady międzynarodowe oraz finaliści, którzy nie są w ostatniej programowo klasie swojej szkoły, zostaną zaproszeni do nieodpłatnego udziału w III Obozie Naukowo-Treningowym im. Antoniego Kreczmara, który odbędzie się w okresie wakacji 2002 roku.

(8) Komitet Główny może przyznawać finalistom i laureatom nagrody, a także stypendia ufundowane przez osoby prawne lub fizyczne.

§5 PRZEPISY KOŃCOWE

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

(2) Komitet Główny Olimpiady Informatycznej zawiadamia wszystkich uczestników zawodów I i II stopnia o ich wynikach. Uczestnicy zawodów I stopnia, którzy prześlą rozwiązania jedynie przez Internet zostaną zawiadomieni pocztą elektroniczną, a poprzez witrynę Olimpiady będą mogli zapoznać się ze szczegółowym raportem ze sprawdzania ich rozwiązań. Pozostali zawodnicy otrzymają tę samą informację w terminie późniejszym zwykłą pocztą.

(3) Każdy uczestnik, który przeszedł do zawodów wyższego stopnia oraz dyrektor jego szkoły otrzymują informację o miejscu i terminie następnych zawodów.

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

Witryna Olimpiady: `www.oi.pjwstk.waw.pl`

Zawody I stopnia

Zawody I stopnia — opracowania zadań

Tomasz Waleń
Treść zadania, Opracowanie

Wojciech Dudek
Program

Koleje

Bajtockie Koleje Państwowe postanowiły pójść z duchem czasu i wprowadzić do swojej oferty połączenie InterCity. Ze względu na brak sprawnych lokomotyw, czystych wagonów i prostych torów można było uruchomić tylko jedno takie połączenie. Kolejną przeszkodą okazał się brak informatycznego systemu rezerwacji miejsc. Napisanie głównej części tego systemu jest Twoim zadaniem.

Dla uproszczenia przyjmujemy, że połączenie InterCity przebiega przez n miast ponumerowanych kolejno od 1 do n (miasto na początku trasy ma numer 1, a na końcu n). W pociągu jest m miejsc i między żadnymi dwiema kolejnymi stacjami nie można przewieźć większej liczby pasażerów.

System informatyczny ma przyjmować kolejne zgłoszenia i stwierdzać, czy można je zrealizować. Zgłoszenie jest akceptowane, gdy na danym odcinku trasy w pociągu jest wystarczająca liczba wolnych miejsc, w przeciwnym przypadku zgłoszenie jest odrzucane. Nie jest możliwe częściowe zaakceptowanie zgłoszenia, np. na części trasy albo dla mniejszej liczby pasażerów. Po zaakceptowaniu zgłoszenia uaktualniany jest stan wolnych miejsc w pociągu. Zgłoszenia przetwarzane są jedno po drugim w kolejności nadchodzenia.

Zadanie

Napisz program, który:

- *wczyta z pliku wejściowego* `kol.in` *opis połączenia oraz listę zgłoszonych rezerwacji,*
- *obliczy, które zgłoszenia zostaną przyjęte, a które odrzucone,*
- *zapisze do pliku wyjściowego* `kol.out` *odpowiedzi na wszystkie zgłoszenia.*

Wejście

W pierwszym wierszu pliku tekstowego `kol.in` *znajdują się trzy liczby całkowite n, m i z ($1 \leqslant n \leqslant 60\,000$, $1 \leqslant m \leqslant 60\,000$, $1 \leqslant z \leqslant 60\,000$) pooddzielane pojedynczymi odstępami i oznaczające odpowiednio: liczbę miast na trasie, liczbę miejsc w pociągu i liczbę zgłoszeń. W kolejnych z wierszach opisane są kolejne zgłoszenia. W wierszu o numerze $i+1$ opisane jest i-te zgłoszenie. Zapisane są w nim trzy liczby całkowite p, k i l ($1 \leqslant p < k \leqslant n$, $1 \leqslant l \leqslant m$) pooddzielane pojedynczymi odstępami i oznaczające odpowiednio: numer stacji początkowej, numer stacji docelowej i wymaganą liczbę miejsc.*

Wyjście

Twój program powinien zapisać w pliku tekstowym `kol.out` *z wierszy. W i-tym wierszu powinien zostać zapisany dokładnie jeden znak:*

- `T` *— jeśli i-te zapytanie zostało zaakceptowane,*
- `N` *— w przeciwnym przypadku.*

Przykład

Dla pliku wejściowego `kol.in`*:*

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

poprawną odpowiedzią jest plik wyjściowy `kol.out`*:*

```
T
T
N
N
```

Uwaga: *W treści zadania uwzględniono drobne uproszczenia dokonane w czasie trwania I etapu.*

Rozwiązanie

Zadanie polega na zasymulowaniu działania systemu rezerwacji miejsc w Bajtockich Kolejach Państwowych. Wymaga to zaprojektowania struktury danych utrzymującej informacje o wolnych miejscach po przetworzeniu dotychczasowych zgłoszeń. Struktura ta powinna udostępniać następujące operacje:

- *Inicjalizuj*(n, m) — przygotowanie stuktury i zapamiętanie, że na każdym jednostkowym odcinku trasy można przewieźć m pasażerów,
- *sprawdźZgłoszenie*(p, k, l) — sprawdzenie, czy dane zgłoszenie może zostać przyjęte (tzn. czy na podanej trasie z p do k jest co najmniej l wolnych miejsc),
- *przyjmijZgłoszenie*(p, k, l) — przyjęcie zgłoszenia i zarezerwowanie l miejsc na podanej trasie.

Pseudokod rozwiązania zadania, wykorzystujący wspomnianą strukturę, ma następującą postać:

```
Inicjalizuj(n, m);
for i := 1 to z do
  if sprawdźZgłoszenie(P[i], K[i], L[i])) then begin
    przyjmijZgłoszenie(P[i], K[i], L[i]);
    zapiszOdpowiedź(T);
  end else
    zapiszOdpowiedź(N);
```

(n — liczba stacji na trasie, z — liczba zgłoszeń, m — liczba miejsc w pociągu, tablice P, K i L — dane o zgłoszeniach, odpowiednio, numer stacji początkowej, numer stacji końcowej i żądana liczba miejsc).

Najprostszą implementacją stuktury danych umożliwiającej wykonywanie wspomnianych operacji jest tablica zawierająca informacje o liczbie zajętych miejsc na odcinkach trasy pomiędzy kolejnymi stacjami:

```
var S : array[1..MAX_N − 1] of integer;
```

gdzie $S[i]$ zawiera informację o liczbie zajętych miejsc na trasie $(i, i+1)$. Inicjalizacja struktury jest bardzo prosta:

```
procedure Inicjalizuj(n, m);
var i : integer;
begin
  for i := 1 to n − 1 do
    S[i]:=0;
end;
```

Równie proste są operacje *sprawdźZgłoszenie* i *przyjmijZgłoszenie* — wystarczy sprawdzenie, czy na podanym odcinku liczba miejsc jest większa bądź równa żądanej w zgłoszeniu, oraz odpowiednia modyfikacja stanu zajętych miejsc w przypadku dokonania rezerwacji.

```
function sprawdźZgłoszenie(p, k, l) : bool;
var i : integer;
begin
  for i := p to k − 1 do
    if S[i] + l ⩽ m then return false;
  return true;
end;
```

```
procedure przyjmijZgłoszenie(p, k, l);
begin
  for i := p to k − 1 do
    S[i] := S[i] + l;
end;
```

Powyższa implementacja co prawda spełnia wszystkie wymogi funkcjonalności, lecz pesymistyczny czas działania poszczególnych operacji nie jest zadowalający:

- *Inicjalizuj* — $O(n)$,
- *sprawdźZgłoszenie* — $O(n)$,
- *przyjmijZgłoszenie* — $O(n)$.

Tak więc całkowity czas działania rozwiązania przy użyciu takiej struktury danych będzie wynosił $O(nz)$, co w przypadku maksymalnych danych dla zadania ($n = 60\,000$, $z = 60\,000$) nie jest akceptowalne.

Rozwiązanie wzorcowe

Aby dla określonych w zadaniu rozmiarów danych wejściowych rozwiązanie działało w rozsądnym czasie, operacje *sprawdźZgłoszenie* i *przyjmijZgłoszenie* powinny zostać efektywnie zaimplementowane — na przykład dobrze byłoby, żeby ich pesymistyczny czas wykonania wynosił $O(\log n)$.

Rozwiązanie wzorcowe używa struktury danych bardzo zbliżonej do tej, która została wykorzystana w rozwiązaniu zadania „Kopalnia złota” ([8]).

Struktura składa się ze zrównoważonego drzewa binarnego o $n-1$ liściach. Każdy z liści odpowiada za odcinek trasy o jednostkowej długości: liść o nr. 1 — $(1,2)$, liść o nr. 2 — $(2,3)$, itd. Natomiast wierzchołki wewnętrzne drzewa odpowiadają za odcinki trasy będące sumami odcinków jednostkowych z ich poddrzew. Tak więc korzeń odpowiada za całą trasę $(1,n)$.

W każdym wierzchołku przechowywane są następujące informacje:

- *lewy*, *prawy* — wskaźniki do korzeni poddrzew, odpowiednio, lewego i prawego,
- *liści_w_lewym* — liczba liści w lewym poddrzewie,
- *max* — maksymalne obciążenie na trasie reprezentowanej przez poddrzewo o korzeniu w tym wierzchołku otrzymane w wyniku realizacji wszystkich dotychczasowych zgłoszeń,
- *obciążenie* — obciążenie przypisane do wierzchołka.

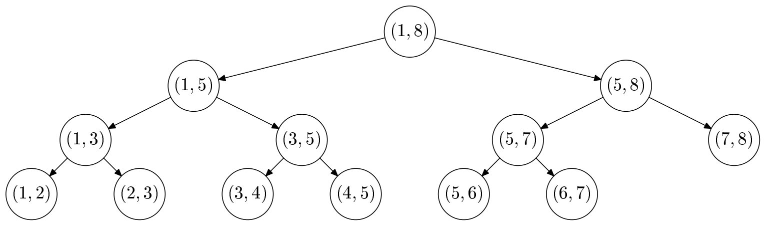
Przechowywanie informacji o tym, za jaki dokładnie przedział odpowiada dany wierzchołek nie jest konieczne. Tę informację można obliczyć w trakcie obchodzenia drzewa. Zachowany jest następujący niezmiennik: liczba miejsc zarezerwowanych na przedziale jednostkowym (po rozważeniu wszystkich przyjętych rezerwacji) jest równa sumie wartości pól *obciążenie* we wszystkich wierzchołkach na ścieżce od korzenia do wierzchołka reprezentującego ten przedział jednostkowy (włącznie).

Inicjalizacja

Przygotowanie struktury jest trochę bardziej skomplikowane niż w poprzednim przypadku. Konieczne jest zbudowanie zrównoważonego drzewa binarnego o odpowiedniej liczbie liści. Aby zbudować drzewo o n liściach wystarczy rekurencyjnie zbudować poddrzewa o odpowiednio $n-\lfloor \frac{n}{2} \rfloor$ i $\lfloor \frac{n}{2} \rfloor$ liściach, a następnie zwrócić drzewo zawierające obydwa poddrzewa.

```
function BudujDrzewo(ile_liści : longint) : PTDrzewo;
var
  e : PTDrzewo;
begin
  if ile_liści = 0 then
    return nil;
  else begin
    New (e);
    with e↑ do begin
      max := 0;
      obciążenie := 0;
      if ile_liści > 1 then begin
        liści_w_lewym := ile_liści − (ile_liści div 2);
        lewy := BudujDrzewo (liści_w_lewym);
        prawy := BudujDrzewo (ile_liści - liści_w_lewym);
      end else begin
        liści_w_lewym := 0;
        lewy := nil;
        prawy := nil;
      end;
    end;
    return e;
  end;
end;
```

Dla połączenia o długości $n = 8$ drzewo ma następującą postać:



Rys. 1. Przykładowe drzewo dla $n = 8$. W nawiasach zaznaczono przedziały, którym odpowiadają poszczególne wierzchołki.

Sprawdzanie zgłoszenia

Wierzchołki drzewa wyznaczają *przedziały elementarne*. Nie jest ich zbyt dużo, dokładnie $2n - 3$, jednak umożliwiają przedstawienie dowolnej trasy z przedziału $(1, n)$ przy użyciu co najwyżej $O(\log n)$ przedziałów elementarnych. Efektywność implementacji poszczególnych operacji na takiej strukturze wynika z tego, że zamiast operować na przedziałach jednostkowej długości, można wszystkie operacje sprawdzania/przyjmowania zgłoszeń wykonywać na przedziałach elementarnych.

Funkcja *sprawdźZgłoszenie* ma następujący schemat:

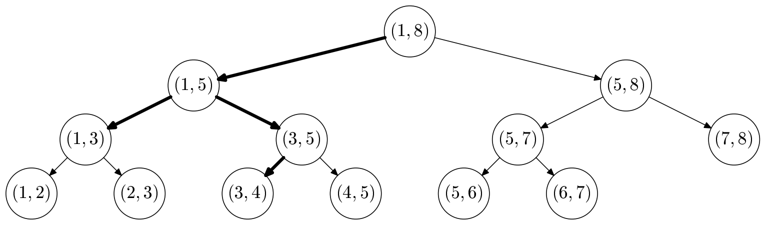
```
function sprawdźZgłoszenie(p, k, l) : bool;
begin
  ZnajdźŚcieżki(p, k − 1, s_l, s_p);
  if (ObcMax(s_l, s_p) + l ⩽ m) then return true;
  return false;
end;
```

Używane są w niej dwie pomocnicze funkcje:

- *ZnajdźŚcieżki* — Szukanie ścieżek w drzewie do wierzchołków reprezentujących najdłuższe przedziały elementarne zawarte na początku i na końcu przedziału (p, k) odpowiednio. Dodatkowo obie ścieżki są wstępnie przetwarzane, np. znajdowany jest najgłębiej położony wspólny wierzchołek na obu ścieżkach.
- *ObcMax* — Wyznaczenie maksymalnego obciążenia (tzn. maksymalnej liczby zajętych miejsc na odcinku jednostkowym w tym przedziale) dla przedziału opisującego zlecenie.

```
procedure ZnajdźŚcieżki(pocz, kon : longint; var s_l, s_p : TŚcieżka);
var
  e : PTDrzewo;
  i : longint;
begin
  { Zbudowanie ścieżki s_l prowadzącej od korzenia }
  { do liścia (pocz, pocz + 1) }
  e := drzewo; i := 1; s_l.wierz[1] := e;
  while (e↑.liści_w_lewym ≠ 0) do begin
    if pocz ⩽ e↑.liści_w_lewym then
      e := e↑.lewy;
    else begin
      pocz := pocz − e↑.liści_w_lewym;
      e := e↑.prawy;
    end;
    Inc(i); s_l.wierz[i] := e;
  end;
  s_l.długość := i;
  { Zbudowanie ścieżki s_p prowadzącej od korzenia }
  { do liścia (kon, kon + 1) }
  ...
  { Znajdowanie ostatniego wspólnego miejsca na ścieżkach s_l i s_p }
  i := 1;
  while (i ⩽ s_l.długość) and (i ⩽ s_p.długość) and
        (s_l.wierz[i] = s_p.wierz[i]) do
    Inc(i);
  s_l.rozdz := i − 1; s_p.rozdz := i − 1;
```

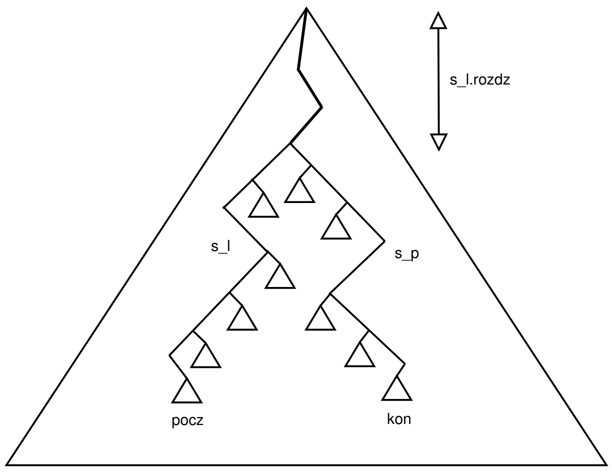
```
{ kompresja ścieżek }
for i := s_l.długość downto s_l.rozdz + 1 do
  if s_l.wierz[i] ≠ s_l.wierz[i − 1]↑.lewy then break;
s_l.długość := i;
{ symetryczna kompresja dla s_p }
...
end;
```



Rys. 2. Ścieżki znalezione przez *ZnajdźŚcieżki* dla $pocz = 1$ i $kon = 4$.

Mając dane *s_l* i *s_p* można podać zbiór przedziałów elementarnych, z których składa się trasa z *pocz* do *kon*:

- jeśli $s\_l.długość = s\_l.rozdz$, to jest to $\{(pocz, pocz + 1)\}$ — zbiór składający się z przedziału reprezentowanego przez ostatni wierzchołek na ścieżce *s_l*,
- jeśli $s\_l.długość > s\_l.rozdz$, to przedział $(pocz, kon)$ można przedstawić jako sumę przedziałów reprezentowanych przez wierzchołki będące na końcach *s_l* lub *s_p*, przedziały odpowiadające prawym synom wierzchołków $s\_l[s\_l.rozdz + 1..s\_l.długość - 1]$, które są całkowicie zawarte w $(pocz, kon)$ oraz przedziały odpowiadające lewym synom wierzchołków $s\_p[s\_p.rozdz + 1..s\_p.długość - 1]$, które są całkowicie zawarte w $(pocz, kon)$ (zobacz rys. 3).



Rys. 3. Ścieżki z zaznaczonymi wierzchołkami reprezentującymi poszukiwane przedziały elementarne.

Funkcja *ObcMax* analizuje zbiór przedziałów elementarnych i zwraca maksymalne znalezione obciążenie. Dla danego przedziału elementarnego (odpowiadającego wierzchołkowi v), maksymalne obciążenia jest równe sumie $v.max$ i sumarycznemu obciążeniu (pola *obciążenie*) wierzchołków od v (z wyłączeniem v) do korzenia. Na przykład, dla rys. 2 obciążenie wierzchołka $(1, 3)$ jest równe:

$$Obc(1, 3) = (1, 3).max + (1, 5).obciążenie + (1, 8).obciążenie$$

Kod procedury *ObcMax* ma następującą postać:

```
function ObcMax(var s_l, s_p : TŚcieżka) : longint;
var
  c, obc_max_l : longint;
  i : longint;
begin
  { obliczenie maksymalnego obciążenia dla s_l }
  c := s_l.wierz[s_l.długość]↑.max;
  for i := s_l.długość − 1 downto s_l.rozdz + 1 do begin
```

```
    if (s_l.wierz[i]↑.prawy ≠ s_l.wierz[i + 1]) then
      c := Max(s_l.wierz[i]↑.prawy↑.max, c);
    c := c + s_l.wierz[i]↑.obciążenie;
  end;
  for i := Min(s_l.rozdz, s_l.długość − 1) downto 1 do
    c := c + s_l.wierz[i]↑.obciążenie;
  obc_max_l := c;
  { analogiczne obliczenia dla s_p }
  c := s_p.wierz[s_p.długość]↑.max;
  for i := s_p.długość − 1 downto s_p.rozdz + 1 do begin
    if s_p.wierz[i]↑.lewy ≠ s_p.wierz[i + 1] then
      c := Max(s_p.wierz[i]↑.lewy↑.max, c);
    c := c + s_p.wierz[i]↑.obciążenie;
  end;
  for i := Min(s_p.rozdz, s_p.długość − 1) downto 1 do
    c := c + s_p.wierz[i]↑.obciążenie;
  return Max(c, obc_max_l);
end;
```

Czas wykonania procedur *ZnajdźŚcieżki* i *ObcMax* zależy liniowo od długości znalezionych ścieżek, a ponieważ wszystkie operacje wykonywane są na drzewie o maksymalnej głębokości $O(\log n)$, to i taki jest czas wykonania powyższych operacji. Oczywiście czas wykonania funkcji *SprawdźZgłoszenie* również wynosi $O(\log n)$.

Przyjmowanie zgłoszenia

Dodawanie zgłoszenia jest bardzo podobne do funkcji *ObcMax*, jednak tu zamiast obliczania obciążeń dla przedziałów elementarnych, konieczne jest dodanie dla nich obciążenia. Konieczne jest również poprawienie wartości pola *max* dla węzłów leżących na ścieżkach od węzłów odpowiadających przedziałom elementarnym do korzenia.

Kod procedury *Obciąż* ma następującą postać:

```
procedure Obciąż(var s_l, s_p : TŚcieżka; o_ile : longint);
var
  i : longint;
begin
  if (s_l.rozdz = s_l.długość) and
    (s_p.rozdz = s_p.długość) then begin
    Inc(s_l.wierz[s_l.długość]↑.obciążenie, o_ile);
    Inc(s_l.wierz[s_l.długość]↑.max, o_ile);
    for i := s_l.rozdz − 1 downto 1 do
        s_l.wierz[i]↑.max := Max(s_l.wierz[i]↑.prawy↑.max,
                                  s_l.wierz[i]↑.lewy↑.max)
                              + s_l.wierz[i]↑.obciążenie;
  end else begin
    Inc(s_l.wierz[s_l.długość]↑.obciążenie, o_ile);
    Inc(s_l.wierz[s_l.długość]↑.max, o_ile);
    for i := s_l.długość − 1 downto s_l.rozdz + 1 do begin
      if s_l.wierz[i + 1] ≠ s_l.wierz[i]↑.prawy then begin
        Inc(s_l.wierz[i]↑.prawy↑.obciążenie, o_ile);
        Inc(s_l.wierz[i]↑.prawy↑.max, o_ile);
      end;
      s_l.wierz[i]↑.max := Max(s_l.wierz[i]↑.prawy↑.max,
                                s_l.wierz[i]↑.lewy↑.max)
                            + s_l.wierz[i]↑.obciążenie;
    end;
    { Analogiczne obliczenia dla s_p }
    Inc(s_p.wierz[s_p.długość]↑.obciążenie, o_ile);
    Inc(s_p.wierz[s_p.długość]↑.max, o_ile);
    for i := s_p.długość − 1 downto s_p.rozdz + 1 do begin
      if s_p.wierz[i + 1] ≠ s_p.wierz[i]↑.lewy then begin
        Inc(s_p.wierz[i]↑.lewy↑.obciążenie, o_ile);
        Inc(s_p.wierz[i]↑.lewy↑.max, o_ile);
```

```
      end;
      s_p.wierz[i]↑.max := Max(s_p.wierz[i]↑.prawy↑.max,
                               s_p.wierz[i]↑.lewy↑.max)
                           + s_p.wierz[i]↑.obciążenie;
    end;
    { poprawienie pola max na ścieżce od s_l.wierz[s_l.rozdz] do korzenia }
    for i := s_l.rozdz downto 1 do
      s_l.wierz[i]↑.max := Max(s_l.wierz[i]↑.prawy↑.max,
                               s_l.wierz[i]↑.lewy↑.max)
                           + s_l.wierz[i]↑.obciążenie;
  end;
end;
```

Tak jak poprzednio, czas wykonania procedury *Obciąż* zależy liniowo od długości ścieżek, co daje czas $O(\log n)$.

Całkowity czas wykonania programu wynosi zatem $O(n + z\log n)$.

Testy

Rozwiązania testowane były przy użyciu zestawu 12 testów:

- `kol1.in` — $n = 10$, $z = 10$;
- `kol2.in` — $n = 300$, $z = 50$;
- `kol3.in` — $n = 350$, $z = 500$, małe wartości l;
- `kol4.in` — $n = 30000$, $z = 1000$, zgłoszenia obejmujące krótkie trasy;
- `kol5.in` — $n = 60000$, $z = 60000$, $m = 60000$, zgłoszenia obejmujące krótkie trasy;
- `kol6.in` — $n = 30000$, $z = 999$, $m = 30000$, zgłoszenia obejmujące długie trasy;
- `kol7.in` — $n = 10000$, $z = 10000$;
- `kol8.in` — $n = 10000$, $z = 10000$, pełne trasy, wszystkie z odpowiedzią `T`;
- `kol9.in` — $n = 15001$, $z = 15000$;
- `kol10.in` — $n = 10000$, $z = 20000$, $m = 600$;
- `kol11.in` — $n = 60000$, $z = 6000$, $m = 60000$;
- `kol12.in` — $n = 60000$, $z = 60000$, $m = 60000$.

Tomasz Waleń
Treść zadania, Opracowanie

Tomasz Waleń
Program

Komiwojażer Bajtazar

Komiwojażer Bajtazar ciężko pracuje podróżując po Bajtocji. W dawnych czasach komiwojażerowie sami mogli wybierać miasta, które chcieli odwiedzić, i kolejność, w jakiej to czynili, jednak te czasy minęły już bezpowrotnie. Z chwilą utworzenia Centralnego Urzędu d/s Kontroli Komiwojażerów każdy komiwojażer otrzymuje z Urzędu listę miast, które może odwiedzić, i kolejność, w jakiej powinien to uczynić. Jak to zazwyczaj bywa z centralnymi urzędami, narzucona kolejność odwiedzania miast nie ma zbyt dużo wspólnego z optymalną. Przed wyruszeniem w trasę Bajtazar chciałby przynajmniej dowiedzieć się, ile czasu zajmie mu odwiedzenie wszystkich miast — obliczenie tego jest Twoim zadaniem.

Miasta w Bajtocji są ponumerowane od 1 do n. Numer 1 ma stolica Bajtocji, z niej właśnie rozpoczyna podróż Bajtazar. Miasta połączone są siecią dróg dwukierunkowych. Podróż między dwoma miastami bezpośrednio połączonymi drogą zawsze zajmuje 1 jednostkę czasu. Ze stolicy można dotrzeć do wszystkich pozostałych miast Bajtocji. Jednak sieć dróg została zaprojektowana bardzo oszczędnie, stąd drogi nigdy nie tworzą cykli.

Zadanie

Napisz program, który:

- *wczyta z pliku wejściowego* `kom.in` *opis sieci dróg Bajtocji oraz listę miast, które musi odwiedzić Bajtazar,*
- *obliczy sumaryczny czas podróży Bajtazara,*
- *zapisze go w pliku wyjściowym* `kom.out`.

Wejście

W pierwszym wierszu pliku tekstowego `kom.in` *zapisana jest jedna liczba całkowita n równa liczbie miast w Bajtocji, $1 \leqslant n \leqslant 30\,000$. W kolejnych $n-1$ wierszach opisana jest sieć dróg — w każdym z tych wierszy są zapisane dwie liczby całkowite a i b ($1 \leqslant a, b \leqslant n$, $a \neq b$), oznaczające, że miasta a i b połączone są bezpośrednio drogą. W wierszu o numerze $n+1$ zapisana jest jedna liczb całkowita m równa liczbie miast, które powinien odwiedzić Bajtazar, $1 \leqslant m \leqslant 5\,000$. W następnych m wierszach zapisano numery kolejnych miast na trasie podróży Bajtazara — po jednej liczbie w wierszu.*

Wyjście

W pierwszym i jedynym wierszu pliku tekstowego `kom.out` *powinna zostać zapisana jedna liczba całkowita równa łącznemu czasowi podróży Bajtazara.*

Przykład

Dla pliku wejściowego `kom.in`*:*

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

poprawną odpowiedzią jest plik wyjściowy `kom.out`*:*

```
7
```

Rozwiązanie

Zauważmy, że sieć dróg w Bajtocji tworzy drzewo (las spójny bez cykli). Zadanie polega na obliczeniu sumarycznej długości trasy pokonywanej przez komiwojażera. Rozwiązanie zadania (w najbardziej ogólnej formie) ma postać:

```
{ Komiwojażer rozpoczyna podróż w stolicy Bajtocji }
suma := odległość(1, trasa[1]);
for i := 1 to m − 1 do
    suma := suma + odległość(trasa[i], trasa[i+1]);
{ Zapisz jako wynik: suma }
```

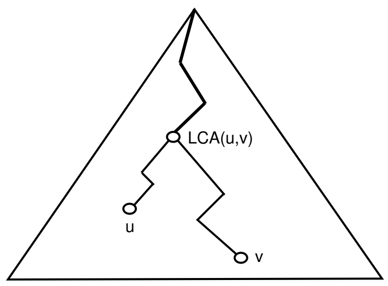
gdzie *trasa* jest tablicą zawierająca kolejne miasta na trasie podróży Bajtazara, a funkcja *odległość* zwraca odległość (w liczbie krawędzi) pomiędzy zadaną parą miast (wierzchołków w drzewie).

W zależności od sposobu obliczania funkcji *odległość* można otrzymać różne rozwiązania:

- Najprostszym sposobem jest wyznaczanie odległości przy pomocy przeszukiwania wszerz drzewa połączeń. Jest to bardzo proste w zapisie rozwiązanie, jednak pesymistyczny czas obliczania odległości wynosi $\Theta(n)$, co dla tego zadania okazuje się zbyt kosztowne.

- W dalszej części opisane jest rozwiązanie, które wykorzystuje informację o tym, że graf połączeń jest drzewem. Przed rozpoczęciem odpowiadania na pytania o odległości, wstępnie przetwarzane jest drzewo połączeń tak, by ułatwić późniejszą pracę.

Przydatne będzie pojęcie *najniższego wspólnego przodka* (ang. lowest common ancestor) wierzchołków w drzewie z korzeniem. Najniższym wspólnym przodkiem wierzchołków u i v, w skrócie $\mathrm{LCA}(u,v)$, nazywamy najbardziej odległy od korzenia wierzchołek drzewa, który jest jednocześnie przodkiem u (leży na ścieżce od u do korzenia) i v.



Rys. 1. Najniższy Wspólny Przodek wierzchołków u i v.

Pojęcie LCA jest bardzo pomocne przy liczeniu odległości między wierzchołkami w dowolnym drzewie. Wystarczy ukorzenić drzewo w dowolnym wierzchołku (w naszym przypadku będzie to stolica). Wówczas

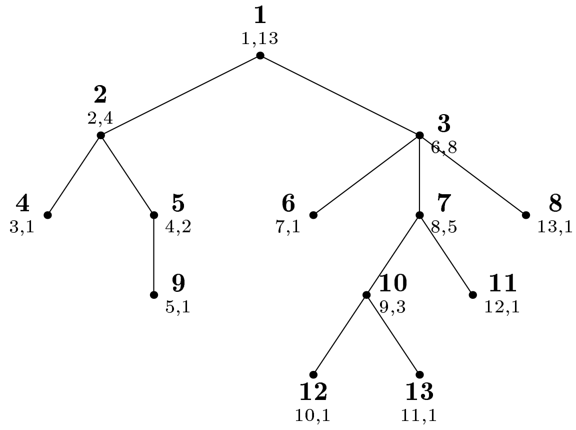
$$\mathit{odległość}(u,v) = \mathit{poziom}(u) + \mathit{poziom}(v) - 2 \cdot \mathit{poziom}(\mathrm{LCA}(u,v)),$$

gdzie $\mathit{poziom}(u)$ oznacza odległość wierzchołka u od korzenia. Obliczenie wartości *poziom* dla wszystkich wierzchołków jest proste. Można to zrobić za pomocą przeszukiwania wszerz.

Do dalszych obliczeń potrzebna będzie również umiejętność szybkiego odpowiadania na pytania typu: czy wierzchołek u jest potomkiem v? Jeśli odpowiednio przygotuje się drzewo, to okazuje się, że można będzie udzielać odpowiedzi na takie pytania w czasie $O(1)$. Można to zrobić w następujący sposób:

1. Dla każdego wierzchołka v obliczamy $\mathit{rozmiar}(v)$ — rozmiar poddrzewa o korzeniu w v.

2. Numerujemy wierzchołki w porządku *preorder* (tzn. najpierw numer zostaje nadany wierzchołkowi, potem rekurencyjnie numerowane są poddrzewa zawierające synów wierzchołka).



Rys. 2. Przykładowe drzewo z obliczonymi numerami preorder i rozmiarami poddrzew.

wierzchołek	1	2	3	4	5	6	7	8	9	10	11	12	13
nr_pre	1	2	6	3	4	7	8	13	5	9	12	10	11
rozmiar	13	4	8	1	2	1	5	1	1	3	1	1	1

3. Numeracja preorder ma następującą własność: dla wierzchołka v wszyscy jego potomkowie mają numery z zakresu $nr\_pre(v) \ldots nr\_pre(v) + rozmiar(v) - 1$ (v jest sam swoim potomkiem). Zatem sprawdzenie, czy u jest potomkiem v, sprowadza się do sprawdzenia następującego warunku:

```
function potomek(u, v : integer) : boolean;
begin
  return (nr_pre[u] ⩾ nr_pre[v])
           and (nr_pre[u] < nr_pre[v] + rozmiar[v]);
end;
```

Za pomocą funkcji *potomek* można obliczyć LCA(u, v), np. przeszukując ścieżkę od u do korzenia w celu znalezienia najniższego wierzchołka, który będzie wspólnym przodkiem u i v:

```
function lca(u, v : integer) : integer;
var i : integer;
begin
   if potomek(u, v) then return v;
   else if potomek(v, u)) then return u;
   else begin
      i := ojciec[u];
      while i > 0 do begin
         if potomek(v, i) then return i;
         i := ojciec[i];
      end;
   end;
end;
```

Jednak takie postępowanie dalej wymaga w pesymistycznym przypadku czasu $O(n)$ dla pojedynczego zapytania. Konieczne są więc pewne usprawnienia. Dla każdego wierzchołka należy obliczyć wierzchołki odległe o $2^0, 2^1, 2^2, \ldots$ w kierunku korzenia. Zdefiniujmy tablicę $p[i, v]$:

- $p[i, v]$ — wierzchołek u leżący na ścieżce pomiędzy v i korzeniem, taki że $odległość(u, v) = 2^i$; jeśli taki wierzchołek nie istnieje, to $p[i, v] = korzeń$.

Dla drzewa z rysunku 2 tablica p ma następującą postać:

v	1	2	3	4	5	6	7	8	9	10	11	12	13
$p[0, v]$	1	1	1	2	2	3	3	3	5	7	7	10	10
$p[1, v]$	1	1	1	1	1	1	1	1	2	3	3	7	7
$p[2, v]$	1	1	1	1	1	1	1	1	1	1	1	1	1

Tablicę p można bardzo prosto obliczyć korzystając z własności

$$p[i,j] = p[i-1, p[i-1,j]]$$

dla $i > 0$.

Dzięki tablicy p można obliczyć LCA w sposób bardzo zbliżony do wyszukiwania binarnego:

```
1: function lca(u, v : integer) : integer;
2: begin
3:    if potomek(u, v) then return v;
4:    else if potomek(v,u) then return u;
5:    else begin
6:       i := u;
7:       j := ⌈log₂ n⌉;
8:       while j ⩾ 0 do begin
9:          if potomek(v, p[j,i])) then j := j − 1;
10:         else i := p[j,i];
11:      end;
12:      return p[0,i];
13:   end;
14: end;
```

Niezmiennikiem pętli **while** jest warunek, że i nie jest przodkiem v oraz $p[j+1,i]$ jest przodkiem zarówno u, jak i v. Instrukcje z linii 9–10 zachowują niezmiennik. Po zakończeniu pętli wiadomo, że i nie jest przodkiem v, jednak $p[0,i] = ojciec[i]$ jest, stąd $p[0,i]$ jest poszukiwanym wierzchołkiem.

Pozostaje jeszcze uzasadnić, że pętla **while** zawsze się kończy. Jeśli w pętli wykonywana jest instrukcja $i := p[j,i]$, to w następnym kroku wartość j zostanie zmniejszona. Oznaczmy nową wartość i przez i'. Z niezmiennika wynika, że $potomek(v, p[j+1,i]) = \mathbf{true}$, a z wcześniejszych rozważań mamy $p[j+1,i] = p[j,p[j,i]]$, stąd $p[j+1,i] = p[j,i']$, a zatem w następnym obrocie pętli warunek $potomek(v, p[j,i'])$ będzie prawdziwy i zostanie wykonana instrukcja $j := j-1$. Liczba obrotów pętli **while** jest ograniczona przez $2(\lceil \log_2 n \rceil + 1)$.

Obliczenie tablic *pre_nr*, *poziom*, *rozmiar* wymaga czasu $O(n)$, natomiast obliczenie tablicy p (ze względu na jej rozmiar) wymaga czasu $O(n \log n)$. Dzięki wstępnemu przetworzeniu drzewa późniejsze odpowiedzi na pytania o odległości wymagają tym razem jedynie czasu $O(\log n)$, stąd całe rozwiązanie ma złożoność: $O(n \log n + m \log n)$.

Zadanie to można rozwiązać, choć jest to zaskakujące, w czasie $O(n+m)$. Rozwiązanie opiera się na bardzo sprytnym obliczaniu *najniższego wspólnego przodka*. Opis sposobu postępowania można znaleźć w pracy B. Schieber'a i U. Vishkin'a [24].

Testy

Rozwiązania testowane były przy użyciu zestawu 11 testów:

- `kom1.in` — test poprawnościowy z $n = 1$ i $m = 1$;
- `kom2.in` — $n = 5$, $m = 5$;
- `kom3.in` — $n = 10$, $m = 5$;
- `kom4.in` — $n = 10$, $m = 5$;
- `kom5.in` — $n = 1000$, $m = 500$;
- `kom6.in` — $n = 3000$, $m = 1000$;
- `kom7.in` — $n = 5000$, $m = 5000$;
- `kom8.in` — $n = 10000$, $m = 3000$;
- `kom9.in` — $n = 15000$, $m = 5000$;
- `kom10.in` — $n = 30000$, $m = 5000$;
- `kom11.in` — $n = 30000$, $m = 5000$.

Krzysztof Onak
Treść zadania, Opracowanie

Krzysztof Onak
Program

Superskoczek

Na nieskończonej szachownicy znajduje się superskoczek, który może wykonywać różnego rodzaju ruchy. Każdy pojedynczy ruch jest określony za pomocą dwóch liczb całkowitych — pierwsza mówi o ile kolumn (w prawo w przypadku liczby dodatniej lub w lewo w przypadku ujemnej), a druga o ile wierszy (do przodu w przypadku liczby dodatniej lub do tyłu w przypadku ujemnej) przesuwa się skoczek wykonując ruch.

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `sup.in` *zestawy danych opisujące różne superskoczki,*
- *dla każdego superskoczka stwierdzi, czy za pomocą swoich ruchów może on dotrzeć do każdego pola na planszy,*
- *zapisze wynik do pliku tekstowego* `sup.out`.

Wejście

W pierwszym wierszu pliku tekstowego `sup.in` *znajduje się jedna liczba całkowita k określająca liczbę zestawów danych, $1 \leqslant k \leqslant 100$. Po niej następuje k zestawów danych. W pierwszym wierszu każdego z nich pojawia się liczba całkowita n będąca liczbą rodzajów ruchów, które może wykonywać superskoczek, $1 \leqslant n \leqslant 100$. Każdy z kolejnych n wierszy zestawu danych zawiera dwie liczby całkowite p i q oddzielone pojedynczym odstępem, opisujące ruch, $-100 \leqslant p, q \leqslant 100$.*

Wyjście

Plik tekstowy `sup.out` *powinien składać się z k wierszy. Wiersz i-ty powinien zawierać jedno słowo* `TAK`*, jeśli superskoczek opisany w i-tym zestawie danych może dotrzeć do każdego pola na planszy, a słowo* `NIE` *w przeciwnym przypadku.*

Przykład

Dla pliku wejściowego `sup.in`*:*

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

poprawną odpowiedzią jest plik wyjściowy `sup.out`*:*

```
TAK
NIE
```

Rozwiązanie

Podstawowe pojęcia

Zauważmy, że ruch superskoczka to para: przesunięcie w prawo lub w lewo oraz do przodu lub w tył. Od tej pory ruchy superskoczka będziemy utożsamiać z dwuwymiarowymi wektorami o całkowitych współrzędnych, czyli uporządkowanymi parami liczb całkowitych. Jeśli skoczek może wykonać ruch o x $(-x)$ kolumn w prawo (w lewo) i y $(-y)$ wierszy do przodu (w tył), to temu ruchowi odpowiada wektor $[x; y]$.

Wektory można dodawać, a ich dodawanie zdefiniowane jest jako dodawanie odpowiednich współrzędnych, tzn.

$$[x_1; y_1] + [x_2; y_2] = [x_1 + x_2; y_1 + y_2].$$

Dodawanie wektorów odpowiada w naszym przypadku składaniu ruchów skoczka. Podobnie możemy mnożyć wektory przez skalary, w naszym przypadku przez liczby całkowite. Dla liczb całkowitych a, x i y mamy

$$a \cdot [x; y] = [x; y] \cdot a = [ax; ay].$$

Niech $\mathcal{U}$ będzie zbiorem ruchów, jakie może wykonywać superskoczek. Zauważmy, że możemy dołożyć do tego zbioru, bez zmiany odpowiedzi na pytanie będące treścią zadania, dowolny wektor będący złożeniem (sumą) skończonej liczby wektorów, które już w tym zbiorze były. Dla zbioru ruchów $\mathcal{U}$ zdefinujemy $\bar{\mathcal{U}}$ (domknięcie $\mathcal{U}$) jako zbiór wszystkich skończonych sum ruchów ze zbioru $\mathcal{U}$. Możemy zapisać kilka prostych własności operacji domknięcia:

- $\mathcal{U} \subseteq \bar{\mathcal{U}}$
- $\bar{\mathcal{U}} = \bar{\bar{\mathcal{U}}}$
- Jeśli $\mathcal{U}_1 \subseteq \mathcal{U}_2$, to $\bar{\mathcal{U}}_1 \subseteq \bar{\mathcal{U}}_2$.

Zupełność

Zbiór ruchów $\mathcal{U}$ będziemy nazywali *zupełnym*, jeśli pozwala on dotrzeć do wszystkich pól nieskończonej szachownicy, tzn. gdy

$$\bar{\mathcal{U}} = \{[x; y] \mid x \text{ i } y \text{ są liczbami całkowitymi}\}.$$

Jak nietrudno zauważyć, celem zadania jest znalezienie i zaimplementowanie szybkiego algorytmu sprawdzania, czy zbiór jest zupełny.

Odwracalność

Powiemy, że zbiór ruchów $\mathcal{U}$ jest *odwracalny*, jeżeli dla każdego wektora $\nu \in \mathcal{U}$ mamy $-\nu \in \bar{\mathcal{U}}$. Mówiąc prościej, własność ta oznacza, że jeśli przejdziemy skoczkiem z jednego pola na inne, to zawsze możemy za pomocą pewnego zestawu ruchów wrócić, czyli wszystkie nasze poczynania są odwracalne.

Osiowość

Zdefinujemy teraz warunek wystarczający do tego, aby zbiór wektorów był odwracalny, a jednocześnie okaże się, że będzie to warunek konieczny dla zupełności. Co to oznacza? Otóż jeśli okaże się, że warunek ten nie będzie spełniony, to na pewno zbiór ruchów nie będzie zupełny. Natomiast jeśli będzie spełniony, to będziemy mogli w dalszych rozważaniach korzystać z założenia o odwracalności ruchów.

Zapowiedziany wcześniej warunek brzmi:

> Do zbioru $\bar{\mathcal{U}}$ należą wektory postaci $[a; 0]$, $[-b; 0]$, $[0; c]$ i $[0; -d]$ dla pewnych dodatnich liczb całkowitych a, b, c i d.

Zbiory ruchów spełniające tę własność będziemy nazywać *osiowymi*.

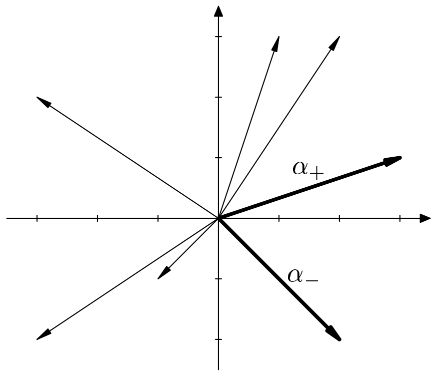
Jeśli zbiór wektorów jest zupełny, to wówczas jest osiowy w sposób oczywisty, ponieważ jego domknięcie zawiera wektory $[1; 0]$, $[-1; 0]$, $[0; 1]$ i $[0; -1]$.

Pozostaje udowodnić, że jeśli zbiór wektorów jest osiowy, to jest odwracalny. Weźmy dowolny zbiór osiowy $\mathcal{U}$ oraz wektor $\nu \in \mathcal{U}$. Niech $\alpha = [a; 0]$, $\beta = [-b; 0]$, $\gamma = [0; c]$ i $\delta = [0; -d]$ będą wektorami ze zbioru $\bar{\mathcal{U}}$, jak w warunku osiowości. Niech $\nu = [x; y]$. Rozważmy najpierw przypadek $x > 0$ i $y = 0$, czyli wektor należący do dodatniej półosi odciętych. Wówczas mamy $(b-1)\nu + x\beta = [(b-1)x - bx; 0] = -\nu$, czyli $-\nu \in \bar{\mathcal{U}}$. Teraz niech $x > 0$ i $y > 0$, a zatem ν jest wektorem z pierwszej ćwiartki. Wówczas $(bd-1)\nu + dx\beta + by\delta = [(bd-1)x - bdx; (bd-1)y - bdy] = -\nu$. Analogicznie postępujemy w pozostałych przypadkach. Tak więc osiowość jest warunkiem wystarczającym dla odwracalności.

Algorytm rozstrzygania osiowości

Zajmiemy się teraz problemem stwierdzania osiowości zbioru ruchów $\mathcal{U}$. Dokładniej pokażemy, jak stwierdzić istnienie w zbiorze $\bar{\mathcal{U}}$ wektora $[a; 0]$, takiego jak w definicji osiowości. Sprawdzenie istnienia pozostałych wektorów wykonuje się analogicznie.

Z niezerowych wektorów ze zbioru $\mathcal{U}$ tworzymy dwa zbiory wektorów. Jeden zawiera wektory o nieujemnej drugiej współrzędnej, a drugi o niedodatniej. Jeśli co najmniej jeden z tych zbiorów jest pusty, to odpowiedź na pytanie o osiowość jest negatywna. Każdy taki wektor tworzy z dodatnią półosią odciętych dwa kąty. Pod uwagę będziemy brali ten o mniejszej mierze. Teraz z każdego ze zbiorów wybieramy wektor o minimalnym kącie tworzonym z półdodatnią osią odciętych. Oznaczmy te wektory zgodnie z intuicją przez α_+ i α_-. W szczególnym przypadku może się nawet okazać, że $\alpha_+ = \alpha_-$.



Na rysunku wyróżniono wektory tworzące minimalne kąty z dodatnią półosią odciętych.

Udowodnimy teraz, że jeśli istnieje w zbiorze $\bar{\mathcal{U}}$ wektor o żądanej własności, to składając α_+ i α_- odpowiednią liczbę razy też otrzymamy wektor o tej własności. Niech $\alpha_+ = [p_+; q_+]$ i $\alpha_- = [p_-; q_-]$. Jeśli $q_+ = 0$ lub $q_- = 0$, to sytuacja jest oczywista. Przyjmijmy więc, że $q_+ > 0$ i $q_- < 0$. Niech

$$[a; 0] = w_1\beta_1 + w_2\beta_2 + \cdots + w_k\beta_k,$$

gdzie $a > 0$, $\beta_1, \beta_2, \ldots, \beta_k \in \mathcal{U}$, a w_1, w_2, $\ldots$, w_k są dodatnimi całkowitymi współczynnikami. Możemy także przyjąć, że żaden wektor β_i nie jest postaci $[b; 0]$ dla pewnego całkowitego b. Gdyby bowiem $b > 0$, oznaczałoby to, że $q_+ = q_- = 0$, a ten przypadek już odrzuciliśmy. Natomiast w przeciwnym przypadku, tzn. $b \leqslant 0$, możemy usunąc składnik odpowiadający temu wektorowi z sumy wektorów, a nadal suma ta będzie miała oczekiwaną przez nas postać. Przypuśćmy, że dla pewnego ustalonego i mamy $\beta_i \neq \alpha_+$ i $\beta_i \neq \alpha_-$ oraz $\beta_i = [b_1, b_2]$, gdzie $b_2 \neq 0$. Bez utraty ogólności możemy przyjąć, że $b_2 > 0$ i $i = 1$. Pomnóżmy obie strony równości podanej powyżej przez $q_+ > 0$. Otrzymujemy wówczas

$$[aq_+; 0] = w_1[b_1q_+; b_2q_+] + q_+w_2\beta_2 + \cdots + q_+w_k\beta_k.$$

Zauważmy, że mamy prostą nierówność na cotangensach kątów $\frac{p_+}{q_+} \geqslant \frac{b_1}{b_2}$, a stąd $b_2p_+ \geqslant b_1q_+$. Dokonując małej zmiany dostaniemy

$$[a'; 0] = w_1[b_2p_+; b_2q_+] + q_+w_2\beta_2 + \cdots + q_+w_k\beta_k$$

lub zapisując inaczej

$$[a'; 0] = b_2w_1\alpha_+ + q_+w_2\beta_2 + \cdots + q_+w_k\beta_k,$$

gdzie $a' \geqslant aq_+ > 0$. W ten sposób wyeliminowaliśmy jeden z wektorów różnych od α_+ i α_-. Postępując tak dalej możemy usunąć wszystkie takie wektory, otrzymując wektor o żądanej postaci jako złożenie wektorów α_+ i α_-.

Przypuśćmy teraz, że mamy już dane wektory α_+ i α_- i chcemy stwierdzić, czy za ich pomocą możemy utworzyć wektor $[a; 0]$, gdzie $a > 0$. Jeśli $q_+ = 0$, to odpowiedź jest pozytywna wtedy i tylko wtedy, gdy $p_+ > 0$. Tak samo jest, gdy $q_- = 0$. Gdy żaden z tych przypadków nie zachodzi, to bierzemy m wektorów α_+ i n wektorów α_-, gdzie m i n są dodatnimi liczbami całkowitymi takimi, aby

$$[a; 0] = m\alpha_+ + n\alpha_-.$$

Tak będzie wtedy i tylko wtedy, gdy $mq_+ + nq_- = 0$, np. jeśli $m = -q_-$ i $n = q_+$. Wystarczy teraz sprawdzić, czy otrzymane w ten sposób a jest dodatnie. Jeśli tak, to sprawa jest oczywista. Jeśli nie, to wybierając inne m i n też otrzymamy niedodatnie a.

Pozostaje wciąż otwarty problem jak wybrać wektory α_+ i α_-. Podstawową trudnością tutaj jest stwierdzenie, który z dwóch danych wektorów tworzy z dodatnią półosią odciętych mniejszy kąt. Można to uczynić na wiele sposobów, na przykład skorzystać z własności odpowiednich funkcji trygonometrycznych, iloczynu wektorowego bądź skalaranego. Nie jest to już duży problem dla ucznia szkoły średniej, dlatego nie zostanie tutaj dokładnie opisany. Przykładowe jego rozwiązanie można znaleźć w [14].

Algorytm rozstrzygania zupełności

Mając dany zbiór $\mathcal{U}$ możliwych ruchów sprawdzamy najpierw, czy jest osiowy. Jeśli nie, to nie jest też zupełny. Natomiast jeśli jest osiowy, to możemy korzystać z odwracalności. Dlatego mówiąc o ruchu α będziemy przyjmować teraz, że możemy wykonywać także ruch postaci $-\alpha$. Pokażemy, że dla każdego zbioru ruchów $\mathcal{U}$ istnieją dwa ruchy, za pomocą których można dojść do tych samych pól nieskończonej szachownicy. Zakładamy tutaj, że Czytelnik jest obeznany z podstawowymi własnościami funkcji NWD i NWW (największy wspólny dzielnik i najmniejsza wspólna wielokrotność). Przyjmujemy także założenie, że $\mathrm{NWD}(0,0) = 0$.

Zakładamy, że mamy dwa wektory $\alpha = [a;0]$ i $\beta = [b;c]$ rozpinające pewną siatkę ruchów i dokładamy ruch $\gamma = [d;e]$. Dodatkowo załóżmy, że jeśli $c = 0$, to $b = 0$. Zastanówmy się, jak zmieniają się nasze możliwości poruszania się wzdłuż osi odciętych. Jeśli $e = 0$, to minimalne przesunięcie, jakie możemy wykonać, będzie wynosiło $\alpha' = [\mathrm{NWD}(a,d);0]$. W przeciwnym przypadku niech $c \neq 0$. Wówczas za pomocą złożenia pewnej liczby wektorów α i β dostaniemy wektor zawarty w osi odciętych. Przyjmijmy bez utraty ogólności, że $c > 0$ i $e > 0$. Jeśli tak nie jest, to zamiast danego wektora możemy wziąć przeciwny do niego. Najmniejsze przesunięcie, jakie możemy otrzymać za pomocą β i γ tym przypadku, to $\frac{\mathrm{NWW}(c,e)}{c}\beta - \frac{\mathrm{NWW}(c,e)}{e}\gamma$. Wszystkie pozostałe są całkowitymi wielokrotnościami tego podstawowego przesunięcia. Ostatecznie minimalne przesunięcie wzdłuż osi odciętych będzie można opisać za pomocą wektora $\alpha' = [\mathrm{NWD}(a, \frac{\mathrm{NWW}(c,e)}{c}b - \frac{\mathrm{NWW}(c,e)}{e}d);0]$.

Jeśli $c = 0$ i $e = 0$, to przyjmujemy $\beta' = [0;0]$. W przeciwnym przypadku istnieją całkowite x i y takie, że $\mathrm{NWD}(c,e) = cx + ey$. Biorąc wektor $\beta' = x\beta + y\gamma$ dostajemy najmniejsze możliwe przesunięcie w pionie plus jakieś odchylenie w poziomie. Dodatkowo korzystając z α' możemy zmniejszyć to odchylenie.

Mieliśmy dany na początku zbiór $\mathcal{X} = \{\alpha, \beta, \gamma, -\alpha, -\beta, -\gamma\}$, a otrzymaliśmy zbiór $\mathcal{X}' = \{\alpha', \beta' - \alpha', -\beta'\}$. Chcemy pokazać, że $\bar{\mathcal{X}} = \bar{\mathcal{X}}'$, czyli że jeden zbiór jest zupełny wtedy i tylko wtedy, gdy zupełny jest drugi. Wektory ze zbioru $\mathcal{X}'$ otrzymaliśmy z wektorów ze zbioru $\mathcal{X}$, a zatem $\mathcal{X}' \subseteq \bar{\mathcal{X}}$ i $\bar{\mathcal{X}}' \subseteq \bar{\mathcal{X}}$. Weźmy dowolny wektor ze zbioru $\mathcal{X}$. Wygenerujemy go za pomocą wektorów ze zbioru $\mathcal{X}'$. Niech $\delta = [f;g] \in \mathcal{X}$ oraz przyjmijmy, że $\alpha' = [a';0]$ oraz $\beta' = [b';c']$. Jeśli $g \neq 0$, to c' jest różne od zera i dzieli g, wówczas przyjmijmy, że $x = -\frac{g}{c'}$. Jeśli $g = 0$, wówczas niech $x = 0$. Mamy wtedy $\delta + x\beta' = [h;0]$, dla pewnego całkowitego h. Ponadto a' dzieli wówczas h, bo w przeciwnym przypadku dałoby się wygenerować mniejsze przesunięcie wzdłuż osi odciętych, co przeczyłoby wcześniejszym rozważaniom, a zatem δ można wygenerować za pomocą wektorów ze zbioru $\mathcal{X}'$. To dowodzi, że zachodzi drugie zawieranie i ostatecznie $\bar{\mathcal{X}} = \bar{\mathcal{X}}'$.

Zauważmy, że korzystając z opisanych tutaj metod możemy wygenerować dla odwracalnego zbioru wektorów $\mathcal{U}$ dwa wektory, które generują taki sam zbiór ruchów, o ile przyjmiemy, że możemy korzystać także z wektorów przeciwnych. Startujemy od dwóch wektorów zerowych i dodajemy kolejne wektory ze zbioru $\mathcal{U}$. Jeśli na końcu mamy $\alpha = [a;0]$ i $\beta = [b;c]$, to $\mathcal{U}$ jest zupełny wtedy i tylko wtedy, gdy a i c należą do zbioru $\{-1,1\}$. Daje to ostatecznie metodę sprawdzania, czy $\mathcal{U}$ jest zupełny.

Implementacja i program wzorcowy

Powyżej została opisana metoda rozstrzygania, czy zbiór ruchów jest zupełny. Wymaga ona starannej implementacji tak, aby nie przeoczyć żadnego przypadku.

W programie wzorcowym `sup.cpp` zdefiniowana została klasa wektorów (`Wektor`) wraz z podstawowymi operacjami na jej obiektach. Dzięki temu można bardziej abstrakcyjnie i łatwiej implementować wszystkie części programu operujące na wektorach. Do rozstrzygania, czy zbiór jest osiowy używamy czterech obiektów klasy `DojscieNaPolos`, którym przekazujemy zbiory wektorów, a one odpowiadają na pytanie, czy można dojść za pomocą tych wektorów na określoną półoś. Pojedynczy obiekt tej klasy stwierdza, czy można dojść na dodatnią część osi odciętych. Dlatego do czterech obiektów przekazujemy wektory z początkowego zbioru ruchów obrócone odpowiedni o 0, 90, 180 i 270 stopni. Obiekt klasy `DojscieNaPolos` korzysta z dwóch obiektów klasy `NajbardziejWPrawo`, które wyznaczają niezerowe wektory tworzące najmniejszy kąt z dodatnią półosią odciętych. Równocześnie ze sprawdzaniem osiowości przeprowadzamy w programie sprawdzenie zupełności przy założeniu odwracalności. Służy do tego obiekt klasy `Siatka`, który otrzymuje kolejne wektory, tworzy ich dwuwektorową bazę i ponadto potrafi odpowiadać, czy zbiór wektorów jest zupełny. Odpowiedź jest pozytywna wtedy i tylko wtedy, gdy można dojść do każdej półosi (osiowość) i obiekt klasy `Siatka` stwierdzi zupełność przy założeniu odwracalności.

Do obliczania najmniejszego wspólnego dzielnika a i b, liczb x i y takich, że

$$\mathrm{NWD}(a,b) = ax + by,$$

oraz najmniejszej wspólnej wielokrotności a i b wykorzystujemy w programie rozszerzony algorytm Euklidesa. Jego opis można znaleźć na przykład w [14][1].

Niech n będzie liczbą wektorów w początkowym zbiorze, a d niech będzie ograniczeniem rozmiaru wektorów w zbiorze ruchów. Sprawdzenie osiowości zabiera programowi wzorcowemu czas $O(n)$. Natomiast sprawdzenie

[1] Algorytm Euklidesa został szerzej omówiony także przez Jakuba Pawlewicza w opracowaniu do zadania „Wyliczanka" (str. 67) — Red.

zupełności, przy założeniu odwracalności, wymaga czasu rzędu $O(n \log d)$. Zatem czas działania programu dla pojedynczego zestawu danych szacuje się przez $O(n \log d)$. Ponieważ d jest w przypadku tego zadania niewielką liczbą, więc możemy przyjąć, że $\log d$ jest ograniczony przez stałą, a wówczas program działa w linowym czasie.

Inne rozwiązania

Istnieją podobne do przedstawionej powyżej sposoby rozwiązania tego zadania, które działają niestety w gorszym czasie. Można także zaimplementować nieefektywnie powyższe rozwiązanie.

Inne podejście do rozwiązania zadania, to sprawdzenie, czy poruszając się skoczkiem po ograniczonym obszarze uda się wygenerować kilka wektorów, o których zbiorze wiemy, że jest zupełny. Mogą to być na przykład $[1; 0]$, $[0; 1]$ i $[-1; -1]$. Dowód zupełności jest w tym przypadku banalny. Trudności mogą się tutaj pojawić przy dowodzie, że taki obszar wystarczy, aby te wektory uzyskać zawsze, gdy jest to możliwe.

Kolejne — raczej częściowe — rozwiązanie problemu, to nagromadzenie dużej wiedzy o przypadkach, gdy rozwiązanie problemu jest oczywiste. Oto przykłady warunków implikujących, że zbiór wektorów nie jest zupełny:

- Dla wszystkich wektorów pierwsza współrzędna jest nieujemna.
- NWD jednej ze współrzędnych wszystkich wektorów jest różny od 1.

Implementujemy sprawdzanie kilku takich warunków. Jeśli choć jeden z nich jest spełniony, to odpowiadamy zgodnie z prawdą, że zbiór nie jest zupełny, a jeśli żaden nie jest spełniony, to odpowiedź brzmi „prawdopodobnie zupełny", ale niepewność nie znajduje oczywiście w żaden sposób odbicia w formacie pliku wyjściowego. Rzecz jasna można tworzyć wiele wariacji tego rozwiązania, ale bardzo trudno ocenić ich skuteczność.

Testy

Programy uczestników były oceniane za pomocą 10 plików wejściowych. Zestawy ruchów były w większości przypadków wygenerowane losowo. Jednocześnie starano się, aby w każdym pliku w około połowie przypadków odpowiedź była negatywna.

Piotr Chrząstowski Treść zadania, Opracowanie
Tomasz Waleń Program

Wyspa

W Bajtlandii postanowiono zorganizować mecz pomiędzy dwiema zwaśnionymi drużynami piłkarskimi: Linuksowcami i Mikromiękkimi. Ponieważ jednak kibice obu drużyn znani są z wzajemnej głębokiej antypatii, należy ich ulokować w miastach możliwie najdalej od siebie oddalonych i pozwolić na oglądanie meczu tylko w telewizji. Bajtlandia jest wyspą, a wszystkie jej miasta leżą na wybrzeżu. Wzdłuż brzegów wyspy biegnie dwukierunkowa autostrada, która łączy wszystkie miasta. Z każdego miasta do każdego innego można dojechać na dwa sposoby: w kierunku zgodnym z ruchem wskazówek zegara i w kierunku przeciwnym. Długość krótszej z tych dróg jest odległością między miastami.

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `wys.in` *opis wyspy,*
- *obliczy maksymalną odległość, na jaką mogą zostać odseparowani kibice,*
- *zapisze wynik w pliku tekstowym* `wys.out`.

Wejście

W pierwszym wierszu pliku tekstowego `wys.in` *zapisana jest jedna dodatnia liczba całkowita* n, $2 \leqslant n \leqslant 50\,000$, *oznaczająca liczbę miast znajdujących się na wyspie. W kolejnych* n *wierszach zapisano długości odcinków autostrady pomiędzy sąsiednimi miastami. Każdy z tych wierszy zawiera jedną dodatnią liczbę całkowitą. W wierszu o numerze* $i+1$ *zapisana jest długość odcinka autostrady pomiędzy miastem o numerze* i, *a miastem o numerze* $i+1$, *natomiast w wierszu o numerze* $n+1$ *zapisana jest długość drogi pomiędzy miastem* n *a 1. Całkowita długość autostrady nie przekracza 1 000 000 000.*

Wyjście

Pierwszy i jedyny wiersz pliku tekstowego `wys.out` *powinien zawierać jedną liczbę całkowitą oznaczającą maksymalną odległość, na jaką mogą zostać odseparowani kibice.*

Przykład

Dla pliku wejściowego `wys.in`*:*

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

poprawną odpowiedzią jest plik wyjściowy `wys.out`*:*

```
7
```

Rozwiązanie

Zadanie „Wyspa" okazało się zadaniem dość prostym, niesprawiającym przesadnych problemów. Rozwiązanie najprostsze, polegające na zbadaniu wszystkich możliwych par miast, znajdowało się w miarę szybko. Jednak, jak to zwykle bywa, najprostsze rozwiązanie nie wystarczało do satysfakcjonującego zaliczenia zadania. Łatwo bowiem można było tu użyć algorytmów za mało efektywnych, aby uzyskać znaczącą liczbę punktów.

Przyjmijmy, że miasta są ponumerowane zgodnie z ruchem wskazówek zegara, i załóżmy, że długości kolejnych odcinków autostrady umieszczone są w tablicy $D[1..n]$. Wartość $D[i]$ oznacza długość odcinka autostrady między miastem i, a $i+1$ dla $1 \leqslant i < n$, zaś $D[n]$ oznacza długość odcinka autostrady między miastem n a miastem 1, wszystkie brane zgodnie z ruchem wskazówek zegara. Należy tu zwrócić uwagę na to, że wartości w tablicy D niekoniecznie są odległościami między miastami. Będzie tak tylko wtedy, jeśli długość najdłuższego odcinka drogi między kolejnymi miastami nie przekracza połowy długości całej pętli. W takim przypadku odpowiedzią na całe zadanie jest odległość między miastami będącymi końcami tego odcinka. Sprawdzenie, czy tak przypadkiem nie jest, może stanowić etap wstępny związany z wypełnieniem tablicy D. Gdyby okazało się, że faktycznie jeden z odcinków jest dłuższy od połowy obwodu, to kończymy program podając jako wynik długość obwodu minus długość tego odcinka. W przeciwnym razie przechodzimy do dalszych obliczeń.

Rozwiązanie sześcienne

Rozwiązanie najprymitywniejsze polegało na wykonaniu pętli przebiegającej wszystkie pary (i, k) dla $1 \leqslant i < k \leqslant n$ i dla każdej takiej pary obliczenie, jak długa jest droga między miastami o numerach i oraz k. Ze względu na to, że autostrada jest dwukierunkowa, trzeba było sprawdzić, w którą stronę jest krócej: czy od x_i do x_k, czy z x_k do x_i przez x_1. Przykładową pętlę wykonującą to zadanie można sobie wyobrazić tak:

```
rekord := 0;
  { największa znaleziona do tej pory odległość między miastami }
for i := 1 to n − 1 do
  for k := i + 1 to n do
    begin
      s1 := 0; { suma kilometrów między i a k }
      s2 := 0; { suma kilometrów między k a i }
      for j := i to k − 1 do s1 := s + D[j];
        { od i do k (zgodnie z ruchem wskazówek zegara) }
      for j := k to n do  s2 := s2 + D[j];
        { od k do 1 }
      for j := 1 to i − 1 do s2 := s2 + D[j];
        { od 1 do i }
      if s1 > s2 then s := s2 else s := s1;
        { wybieramy mniejszą z odległości }
      if rekord < s then rekord := s;
        { i poprawiamy rekord, jeśli trzeba }
    end;
{ rekord jest poszukiwaną odpowiedzią }
```

Łatwo można zauważyć, że złożoność tego algorytmu wynosi $O(n^3)$, a nieco dokładniej około $\frac{1}{2}n^3$ dodawań. Zewnętrzne dwie pętle względem i oraz k wykonujemy dla połowy par (i, k), zaś wewnętrzne pętle względem j przebiegają cały odcinek $[1..n]$. Można tę złożoność łatwo zmniejszyć około trzykrotnie przez wyeliminowanie dwóch ostatnich pętli: w końcu jeśli odejmiemy od długości całego obwodu długość odcinka autostrady między i, a k, to otrzymamy długość w drugą stronę. Wystarczy więc obliczoną wartość s_1 porównać z $A - s_1$, gdzie A jest obwodem wyspy, czyli zawczasu (np. przy wczytywaniu danych do D) policzoną długością całej autostrady wokół wyspy.

Niestety! Zważywszy, że n może sięgać 50000, musimy uznać to rozwiązanie za niewystarczające: $\frac{1}{6}50000^3$ przekracza $2 \cdot 10^{13}$, czyli 20 bilionów. Wykonanie stosownych obliczeń na naszych komputerach zajęłoby wiele godzin. Zdecydowanie za dużo, aby myśleć o sukcesie w olimpiadzie.

Rozwiązanie kwadratowe

Zauważmy, że stosunkowo prosto możemy się pozbyć jednej pętli. Wystarczy, jeśli wraz z generowaniem kolejnej pary (i, k) skorzystamy z wyniku, który uzyskaliśmy dla poprzedniej pary $(i, k-1)$, dla $k > i+1$. Oto stosowny fragment kodu:

```
rekord := 0;
  { największa znaleziona do tej pory odległość między miastami }
for i := 1 to n − 1 do begin
  s1 := 0;
    { obliczaną długość drogi między i oraz k }
    { musimy inicjalizować na 0 dla każdego i }
  for k := i + 1 to n do begin
    s1 := s1 + D[k − 1];
      { s1 to liczba kilometrów między i a k }
      { zgodnie z ruchem wskazówek zegara }
    s2 := A − s1;
    if s1 > s2 then s := s2 else s := s1;
      { wybieramy mniejszą z odległości }
```

```
      if rekord < s then rekord := s;
         { i poprawiamy rekord, jeśli trzeba }
   end;
end;
{ rekord jest poszukiwaną odpowiedzią }
```

Jest już znacznie lepiej: uniknęliśmy wewnętrznej pętli. Złożoność zmalała nam do $\frac{1}{2}n^2$, czyli dla $n = 50000$ do „zaledwie” nieco ponad miliarda obrotów wewnętrznej pętli. Niestety jest to nadal zbyt dużo, jak na stosowane w olimpiadzie limity odcięcia.

Rozwiązanie liniowe

Okazuje się, że zadanie to można zrobić stosując sprytny algorytm, który łatwiej jest wymyśleć, niż udowodnić jego poprawność. Żeby zilustrować pomysł algorytmu wyobraźmy sobie, że sporządziliśmy taśmę mającą tyle jednostek długości, ile kilometrów liczy sobie autostrada, a następnie zaznaczywszy na niej miasta w odpowiedniej odległości od siebie, skleiliśmy ją tworząc okrąg. Teraz nasze zadanie polega na znalezieniu takiej pary miast x, y, aby odpowiadające im punkty okręgu X, Y były położone możliwie blisko średnicy okręgu, a ściślej, żeby kąt XOY był możliwie bliski 180 stopniom.

Idea jest następująca. Będziemy używali dwóch wskaźników: x i y, oznaczających miasto pierwsze i drugie z badanej pary ($x \leqslant y$). Następnie, w zależności od tego, czy droga XY jest dłuższa od połowy obwodu, czy krótsza, albo przejdziemy do przodu ze wskaźnikiem x albo y. Po każdym ruchu sprawdzimy, czy nie pobiliśmy rekordu.

Nasz algorytm wygląda więc następująco:

```
rekord := A;
   { tym razem szukamy najmniejszego odchylenia od półokręgu; }
   { A — długość autostrady, }
   { wartość A została wyznaczona w czasie wczytywania danych }
s := 0;
x := 1; y := 1;
   { s oznacza długość drogi xy zgodnie ze wskazówkami zegara }
pół_obwodu := A / 2;
while y ⩽ n do
   { gdy y dojdzie do n + 1 będzie to oznaczało, }
   { że każde miasto było brane pod uwagę }
   begin
      if s < pół_obwodu then
         begin
            s := s + D[y]
            y := y + 1; { dorzucamy nowy odcinek drogi na końcu }
         end
      else
         begin
            s := s − D[x];
            x := x + 1;  { ucinamy początkowy odcinek drogi }
         end;
      t := |pół_obwodu − s|;
      if t < rekord then rekord := t;
   end;
wynik := pół_obwodu − rekord;
```

Poruszamy się zatem po okręgu przechodząc albo z jednego albo z drugiego końca do przodu i po każdym takim przejściu badamy odległość między końcami. Twierdzimy, że postępując w ten sposób nie zgubimy rozwiązania, czyli, że jeśli p, q są poszukiwanymi, najdalej oddalonymi od siebie, punktami ($p < q$), to w którymś momencie algorytmu mieliśmy $x = p$, $y = q$.

Postarajmy się to udowodnić. Załóżmy przeciwnie, że nigdy się tak nie zdarzyło, aby zaszło $p = x$, $q = y$ jednocześnie. Zmienna y w pewnym momencie osiągnie wartość q, gdyż została zainicjalizowana na 1 i za pomocą dodawania jedynki dojdzie do wartości $n + 1$, więc musi przejść przez q. Pokażemy, że w czasie, kiedy zmienna y miała wartość q, zmienna x musiała przyjąć wartość p. Rozważmy dwa przypadki.

Przypadek pierwszy: zmienna y przyjmuje wartość q zanim zmienna x przyjęła wartość p. W tym przypadku w momencie, w którym zmienna y przyjęła wartość q, antypodalny (po drugiej stronie średnicy) względem q

punkt q'[1] jest większy od x. Wtedy jednak nasz algorytm wymusi na zmiennej x dotarcie do punktu p. Długość łuku xq przekracza bowiem pół obwodu i zgodnie z algorytmem, będziemy posuwali x do przodu. Niezależnie od tego, czy punkt p leży na prawo czy na lewo od q', zmienna x przyjmie w końcu wartość p, zanim zmienna y przesunie się za q.

Przypadek drugi: zmienna x przyjmuje wartość p zanim zmienna y dotrze do q. Zastanówmy się więc, co się stało gdy zmienna x po raz pierwszy osiągnęła wartość p? Przypadek, gdy antypodalny punkt p' był mniejszy od y, mamy już przebadany. Załóżmy zatem, że antypodalny punkt p' jest większy od y. W tej sytuacji jednak odcinek xy jest krótszy niż pół obwodu, więc zmienna y zacznie posuwać się do przodu aż dojdzie do wartości q.

Mamy więc sytuację, że albo w momencie przyjęcia wartości q przez zmienną y wartość zmiennej x jest na tyle mała, że będzie musiała dojść do p zanim y ruszy się z miejsca, albo też w momencie, gdy zmienna x przyjmuje wartość p, zmienna y ma na tyle małą wartość, że zostaje zmuszona do marszu naprzód i przyjęcia wartości q.

Zauważmy jeszcze, że w oczywisty sposób algorytm zawsze się zatrzyma: Wartość $x+y$ jest nieujemna i nie przekracza $2n$, a jednocześnie zwiększa się w każdym kroku o jeden. Liczba kroków musi być więc skończona.

Rzecz jasna, można było przerwać wykonywanie algorytmu, gdyby tylko rekord osiągnął zero (znaleźlibyśmy średnicę), ale wymagałoby to sprawdzenia dodatkowego warunku, więc nie zawsze byśmy wygrali.

Zadanie to cieszyło się największą popularnością wśród uczestników pierwszego etapu i było głównym dostarczycielem punktów. Sporo rozwiązań uzyskało maksymalną liczbę 100 punktów. Przedstawione rozwiązanie wymaga zadeklarowania tablicy do przechowywania długości poszczególnych odcinków autostrady, więc wymaga pamięci o liniowym rozmiarze. Można zrobić to w pamięci stałej, ale wymaga to ponad dwukrotnego przeczytania danych: raz żeby wyznaczyć długość obwodu, a następnie żeby czytając otwarty dwukrotnie plik posuwać się po nim wskaźnikami.

Testy

Testy obejmowały zarówno „duże jak i małe przypadki” i zawierały odcinki przekraczające pół obwodu, jak i nieprzekraczające. Szczególnym przypadkiem była też minimalna konfiguracja zawierająca tylko dwa miasta.

- `wys1a.in` — $n = 10$, mały test z jedną bardzo dużą odległością — 99999980
- `wys1b.in` — $n = 2$, przypadek brzegowy, tylko dwa miasta
- `wys2a.in` — $n = 11$, wszystkie odległości między miastami są równe 1
- `wys2b.in` — $n = 21$, prosty test poprawnościowy
- `wys3a.in` — $n = 7$, na przemian występujące odległości 5 i 9
- `wys3b.in` — $n = 50$, mały test losowy
- `wys4.in` — $n = 100$, test losowy
- `wys5.in` — $n = 500$, test losowy
- `wys6.in` — $n = 5000$, test z odległościami będącymi kolejnymi liczbami naturalnymi
- `wys7.in` — $n = 10000$, test z jedną bardzo dużą odległością (pozostałe są bardzo małe)
- `wys8.in` — $n = 20000$, duży test losowy z bardzo dużymi odległościami
- `wys9.in` — $n = 50000$, duży test losowy z dużymi odległościami
- `wys10.in` — $n = 50000$, duży test losowy z małymi odległościami

Pary testów (1a,1b), (2a,2b), (3a,3b) były zgrupowane.

[1] Nie przeszkadza nam to, że q' może nie być całkowite w przypadku, gdy długość obwodu jest liczbą nieparzystą.

Zbigniew J. Czech	Zbigniew J. Czech, Paweł Wolff	Paweł Wolff
Treść zadania	Opracowanie	Program

Zamek

Megachip IV Wspaniały, król Bajtocji, zamierza wydać za mąż swą urodziwą córkę, księżniczkę Adę. Zapytał ją, jakiego męża chciałaby mieć. Księżniczka odpowiedziała, że jej przyszły małżonek powinien być mądry, a także ani skąpy, ani rozrzutny. Zamyślił się król nad tym, jakim to próbom powinni być poddani kandydaci na męża, aby mógł wybrać dla swej córki najlepszego z nich. Po długich dumaniach stwierdził, że najlepiej posłuży do tego zamek, który kazał był wybudować ku uciesze mieszkańców Bajtocji. Zamek składa się z dużej liczby komnat, w których zgromadzono bogactwa królestwa. Komnaty te, połączone korytarzami, mogą być zwiedzane przez poddanych w celu podziwiania wystawionych w nich wspaniałości. Za zwiedzenie komnaty trzeba uiścić pewną liczbę bajtków (bajtek jest jednostką pieniężną Bajtocji). Zwiedzanie zamku rozpoczyna się od komnaty wejściowej.

Król wręczył sakiewkę każdemu kandydatowi na męża księżniczki. W każdej sakiewce była taka sama liczba bajtków. Poprosił każdego kandydata, aby ten wybrał taką drogę, która pozwoli, poczynając od komnaty wejściowej, odwiedzić pewną liczbę komnat zamku oraz zakończyć zwiedzanie w komnacie, w której przebywa księżniczka, i wydać przy tym dokładnie kwotę, jaka była w sakiewce. Rozrzutni kandydaci, którzy wydawali po drodze zbyt dużo, nie docierali do komnaty księżniczki, skąpi natomiast zjawiali się z niepustą sakiewką i księżniczka wyprawiała ich w dalszą drogę celem opróżnienia sakiewki.

Niestety do dziś żadnemu z kandydatów nie udało się sprostać zadaniu króla, a księżniczka Ada wciąż z utęsknieniem czeka na swój ideał. Może Ty staniesz w szranki pisząc program, który pomoże biednej księżniczce?

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `zam.in` *opis zamku, numer komnaty, w której znajduje się księżniczka, i kwotę w sakiewce,*
- *wyznaczy ciąg komnat, przez które należy kolejno przejść, aby dojść z komnaty wejściowej do komnaty, w której znajduje się księżniczka, i wydać dokładnie całą zawartość sakiewki,*
- *zapisze znalezioną drogę w pliku tekstowym* `zam.out`.

Możesz założyć, że dla danych testowych taka droga zawsze istnieje. Jeśli istnieje wiele takich dróg, to Twój program powinien wyznaczyć dowolną z nich.

Wejście

W pierwszym wierszu pliku tekstowego `zam.in` *zapisanych jest pięć dodatnich liczb całkowitych* n, m, w, k, s, $1 \leqslant n \leqslant 100$, $1 \leqslant m \leqslant 4950$, $1 \leqslant w, k \leqslant n$, $1 \leqslant s \leqslant 1000$, *pooddzielanych pojedynczymi odstępami. Liczba* n *jest równa liczbie komnat, a* m *liczbie korytarzy. Komnaty są ponumerowane od 1 do* n. *Liczba* w *jest numerem komnaty wejściowej, a* k *numerem komnaty, w której znajduje się księżniczka. Liczba* s *określa liczbę bajtków w sakiewce. W drugim wierszu zapisanych jest* n *dodatnich liczb całkowitych* $o_1, o_2, \ldots, o_n$, $1 \leqslant o_i \leqslant 1000$, *pooddzielanych pojedynczymi odstępami. Liczba* o_i *jest równa opłacie za (każdorazowy) wstęp do komnaty nr* i. *W kolejnych* m *wierszach zapisane są po dwie dodatnie liczby całkowite* x, y, $x \neq y$, $1 \leqslant x, y, \leqslant n$, *oddzielone pojedynczym odstępem. Każda taka para* x, y *określa, że komnaty o numerach* x *i* y *są połączone korytarzem.*

Wyjście

Twój program powinien w pierwszym (i jedynym) wierszu pliku `zam.out` *zapisać ciąg dodatnich liczb całkowitych, pooddzielanych pojedynczymi odstępami, określający numery kolejnych komnat, przez które należy przejść, aby dojść z komnaty wejściowej do komnaty, w której znajduje się księżniczka, i wydać dokładnie całą zawartość sakiewki.*

Przykład

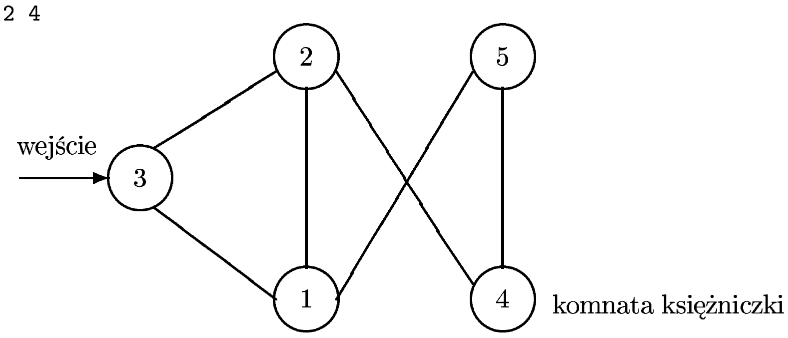
Dla pliku wejściowego `zam.in`:

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

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

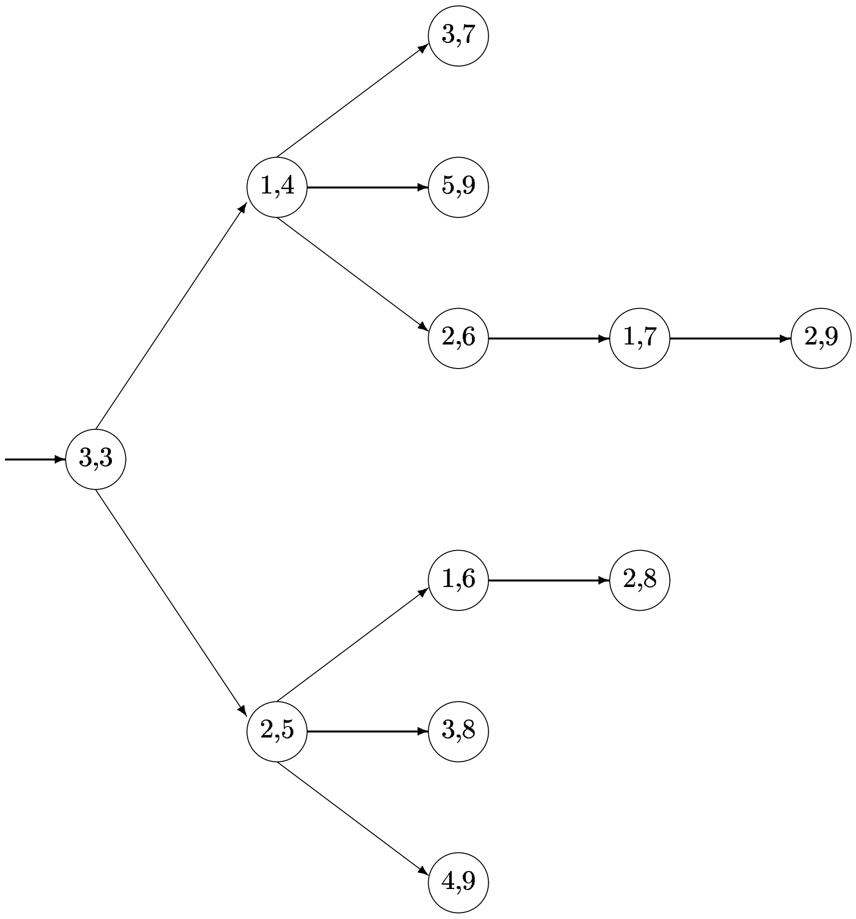
poprawną odpowiedzią jest plik wyjściowy `zam.out`*:*

```
3 2 4
```



Rozwiązanie

Niech graf nieskierowany $G = (V, E)$ reprezentuje komnaty i korytarze zamku, natomiast funkcja wagi $o: V \to \mathbf{N}$ niech określa opłaty, jakie należy uiszczać przy odwiedzaniu komnat. Zadanie polega na znalezieniu takiej drogi w grafie G, że suma wag związanych z poszczególnymi wierzchołkami na tej drodze wynosi dokładnie s. Rozważmy graf skierowany $G' = (V', E')$, w którym $V' = V \times \{1, 2, \ldots, s\}$, natomiast łuki ze zbioru E' łączą tylko takie dwa wierzchołki (v, a) i (u, b), że w grafie G jest krawędź między wierzchołkami v i u, oraz $b = a + o(u)$. Graf G' dla przykładowego grafu zamku z treści zadania przedstawiono na rys. 1. Na rysunku tym pierwszy element pary (x, y) opisującej wierzchołek jest numerem wierzchołka w grafie G, zaś drugi element jest liczbą bajtków jaką wydatkowano po dojściu do wierzchołka x.



Rys. 1. Graf G', w którym sprawdzamy, czy istnieje droga pomiędzy wierzchołkami $(3, 3)$ i $(4, 9)$ (wierzchołki izolowane grafu pominięto).

Łatwo jest zauważyć, że droga w grafie G' między pewnymi wierzchołkami (v, a) i (u, b) odpowiada drodze w grafie G między wierzchołkami v i u, po przebyciu której wydanych zostanie b bajtków, o ile do momentu odwiedzenia wierzchołka v (włącznie) wydano a bajtków. W związku z tym, wystarczy znaleźć drogę w grafie G', łączącą wierzchołek $(w, o(w))$ z (k, s). W rozwiązaniu wzorcowym `zam.pas` użyto do tego celu algorytmu przeszukiwania grafu wszerz, w którym dla każdego odwiedzanego wierzchołka pamiętany jest jego poprzednik w drzewie przeszukiwania, co umożliwia odtworzenie szukanej ścieżki „od końca". Na uwagę zasługuje fakt, że nie jest potrzebne reprezentowanie w pamięci całego zbioru E', gdyż na podstawie zbioru E można efektywnie (na potrzeby opisywanego algorytmu) odtworzyć, jakie krawędzie zawiera graf G'. Koszt czasowy powyższego algorytmu jest $O((n+m)s)$, a pamięciowy — $O(ns+m)$).

Rozwiązania nieoptymalne

W pliku `zam1.pas` znajduje się implementacja algorytmu, w którym systematycznie przeszukuje się wszystkie możliwe drogi łączące komnaty w i k, których waga nie przekracza s. W przypadku znalezienia drogi o wadze dokładnie równej s algorytm kończy działanie. Jego oczekiwana złożoność czasowa jest wykładnicza względem liczby komnat, n. Propozycja rozwiązania heurystycznego znajduje się w pliku `zam2.pas`. W tym rozwiązaniu próba znalezienia odpowiedniej drogi odbywa się w dwóch krokach. W pierwszym, przez losowe błądzenie w grafie, znajdowana jest droga $(w = d_1,\ d_2,\ \ldots,\ d_a = k)$ łącząca komnaty w i k, której koszt nie przekracza s. (Jeśli podczas takiego błądzenia koszt przebytej drogi przekroczy s zanim zostanie osiągnięta komnata k, to błądzenie zostaje ponowione.) Natomiast w drugim kroku, o ile znaleziona droga ma koszt mniejszy niż s, losowana jest komnata d_l na tejże drodze, z której znów w wyniku błądzenia losowego zostanie podjęta próba znalezienia takiej drogi $(d_l = p_1, p_2, \ldots, p_b)$, że waga drogi

$$(w = d_1,\ d_2,\ \ldots,\ d_l = p_1,\ p_2,\ \ldots,\ p_{b-1},\ p_b,\ p_{b-1},\ \ldots,\ p_2,\ p_1 = d_l,\ d_{l+1},\ \ldots,\ d_a = k)$$

wynosi dokładnie s. Jeśli próba ta się nie powiedzie, opisywany tu krok drugi jest powtarzany 100 razy (arbitralnie dobrana liczba powtórzeń). Jeśli po tych stu powtórzeniach kroku drugiego nie zostanie osiągnięty cel, cały proces powtarzany jest od początku, tzn. jeszcze raz przeprowadzany jest krok pierwszy i co najwyżej sto razy krok drugi, aż do znalezienia rozwiązania.

Testy

Przygotowanych zostało 10 testów. Można je podzielić na następujące kategorie:

- małe testy poprawnościowe (testy 1–3);
- testy średniej wielkości (testy 4–6);
- duże testy wydajnościowe (testy 7–10).

Rozwiązanie `zam1.pas` przechodzi w sensownym czasie tylko przez testy małe, natomiast `zam2.pas` — dodatkowo przez test numer 7. Poniżej znajduje się opis wszystkich testów:

- `zam1.in` — mały test, $n = 3$;
- `zam2.in` — mały test, $n = 4$;
- `zam3.in` — mały test losowy, $n = 8$;
- `zam4.in` — średni test, $n = 20$, graf zamku jest losowy, wagi wierzchołków grafu i liczba s są tak dobrane, aby nie było zbyt wielu poprawnych dróg (eliminuje to rozwiązania heurystyczne);
- `zam5.in` — średni test, $n = 15$, graf zamku jest losowy, gęsty, wagi o_i i liczba s, jak wyżej;
- `zam6.in` — średni test, $n = 25$, graf dość gęsty, liczby o_i, s oraz sama struktura grafu powodują, że istnieje dokładnie jedna poprawna droga;
- `zam7.in` — duży test, $n = 100$, graf dość gęsty, losowy, losowe wartości o_i powodują, że istnieje wiele poprawnych dróg — rozwiązania heurystyczne mają szansę przechodzić ten test;
- `zam8.in` — duży test, $n = 100$, graf losowy, rzadki, dalej podobnie jak w przypadku testu numer 4;
- `zam9.in` — duży test, $n = 100$, graf losowy, gęsty, dalej podobnie jak w przypadku testu numer 4;
- `zam10.in` — duży test, $n = 98$, analogiczny do testu numer 6.

Zawody II stopnia

Zawody II stopnia — opracowania zadań

Zbigniew J. Czech	**Zbigniew J. Czech, Marcin Mucha**	**Marcin Mucha**
Treść zadania	Opracowanie	Program

Izolator

Firma Izomax produkuje wielowarstwowe izolatory cieplne. Każda z i warstw, $i = 1, 2, \ldots, n$, cechuje się dodatnim współczynnikiem izolacji a_i. Warstwy są ponumerowane zgodnie z kierunkiem ucieczki ciepła.

$$\textit{ciepło} \;\rightarrow\; ||a_1|a_2|\ldots|a_i|a_{i+1}|\ldots|a_n|| \;\rightarrow$$

Współczynnik izolacji całego izolatora, A, określony jest sumą współczynników izolacji jego warstw. Ponadto współczynnik A rośnie, jeśli po warstwie o niższym współczynniku izolacji występuje warstwa o wyższym współczynniku, zgodnie z wzorem:

$$A = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n-1} max(0, a_{i+1} - a_i)$$

Na przykład, współczynnik izolacji izolatora o postaci:

$$\rightarrow \; ||5|4|1|7|| \; \rightarrow$$

wynosi $A = (5 + 4 + 1 + 7) + (7 - 1) = 23$.

Zadanie

Napisz program, który dla zadanych współczynników izolacji warstw $a_1, a_2, \ldots, a_n$ wyznacza największy możliwy współczynnik izolacji A.

Wejście

W pierwszym wierszu pliku tekstowego `izo.in` *zapisana jest liczba warstw n, $1 \leqslant n \leqslant 100\,000$. W kolejnych n wierszach zapisane są współczynniki $a_1, a_2, \ldots, a_n$, po jednym w każdym wierszu. Współczynniki te są liczbami całkowitymi i spełniają nierówności $1 \leqslant a_i \leqslant 10\,000$.*

Wyjście

W pierwszym i jedynym wierszu pliku tekstowego `izo.out` *Twój program powinien zapisać jedną liczbę całkowitą równą największej możliwej wartości współczynnika izolacji A izolatora zbudowanego z warstw o podanych współczynnikach, ułożonych w odpowiedniej kolejności.*

Przykład

Dla pliku wejściowego `izo.in`*:*

```
4
5
4
1
7
```

poprawną odpowiedzią jest plik wyjściowy `izo.out`*:*

```
24
```

Rozwiązanie

Pokażemy, że istnieje optymalne ułożenie warstw, w którym nie występuje ciąg trzech kolejnych warstw o rosnących współczynnikach izolacji. W tym celu startujemy z dowolnego optymalnego ułożenia. Jeśli istnieje w nim podciąg a_i, a_{i+1}, a_{i+2} taki, że $a_i < a_{i+1} < a_{i+2}$, to zamieniamy warstwy o współczynnikach a_{i+1} i a_{i+2}. Łatwo zauważyć, że w wyniku tej zamiany współczynnik izolacji całego izolatora A nie maleje. Nie można takich zamian wykonywać w nieskończoność, bo za każdym razem zmniejszamy wartość wyrażenia $\sum_{i=1}^{n} i \cdot a_i$ (jest ono ograniczone od dołu). A więc w pewnym momencie dostaniemy ciąg bez rosnących podciągów długości 3. Drugi człon wzoru na łączny współczynnik izolacji A pochodzi od co najwyżej $\lfloor n/2 \rfloor$ rozłącznych par warstw.

Największą wartość współczynnik A osiąga wtedy, gdy pierwsze wyrazy tych par to najmniejsze elementy w liczbie $\lfloor n/2 \rfloor$, a drugie wyrazy to największe elementy ciągu w liczbie $\lfloor n/2 \rfloor$. Widać więc, że aby rozwiązać zadanie, wystarczy znać medianę ciągu. Elementy mniejsze od mediany wchodzą do drugiego członu ze znakiem minus, a elementy większe od mediany ze znakiem plus. Ze względu na ograniczenia danych najwygodniej jest zastosować zliczanie, a potem w jednym przejściu obliczyć wartość współczynnika izolacji. Rozwiązanie to zostało zaimplementowane w programie `izo.pas`. Ograniczenia na dane testowe zostały dobrane w taki sposób, aby nie trzeba było implementować liniowego algorytmu wyznaczania mediany.

Rozwiązanie nieoptymalne

Rozwiązanie nieoptymalne polega na sortowaniu danych za pomocą algorytmu QuickSort, a następnie obliczeniu współczynnika izolacji całego izolatora przez zsumowanie wartości współczynników izolacji $a_{\lfloor n/2 \rfloor+1} \ldots a_n$ ze znakiem plus, zaś współczynników $a_1 \ldots a_{\lfloor n/2 \rfloor}$ ze znakiem minus. Rozwiązanie takie zostało zaimplementowane w programie `izo2.pas`.

Testy

Przygotowanych zostało 10 testów. Wszystkie testy są losowe, różnią się rozmiarami. Rozwiązanie optymalne i rozwiązanie z algorytmem QuickSort są w praktyce nie do odróżnienia pod względem szybkości działania, nawet jeśli zwiększymy znacząco rozmiar danych. Oto lista przygotowanych testów:

- `izo1.in` — $n = 5$;
- `izo2.in` — $n = 50$;
- `izo3.in` — $n = 501$;
- `izo4.in` — $n = 1000$;
- `izo5.in` — $n = 5001$;
- `izo6.in` — $n = 10000$;
- `izo7.in` — $n = 50001$;
- `izo8.in` — $n = 99999$;
- `izo9.in` — $n = 100000$;
- `izo10.in` — $n = 100000$.

Krzysztof Onak
Jakub Pawlewicz — Treść zadania
Jakub Pawlewicz — Opracowanie
Krzysztof Onak — Program

Działka

Dane jest pole w kształcie kwadratu o boku długości n. Pole jest podzielone na n^2 kwadratów o boku długości 1. Każdy kwadrat jest albo użytkowy, albo nieużytkowy. Na polu wyznaczamy działkę. Ma ona kształt prostokąta i może się składać wyłącznie z kwadratów użytkowych. Powierzchnia działki jest równa polu odpowiadającego jej prostokąta. Szukamy działki o jak największej powierzchni.

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `dzi.in` *opis pola,*
- *wyznaczy działkę o największej powierzchni,*
- *zapisze w pliku tekstowym* `dzi.out` *powierzchnię wyznaczonej działki.*

Wejście

W pierwszym wierszu pliku tekstowego `dzi.in` *znajduje się jedna liczba całkowita n, $1 \leqslant n \leqslant 2000$. W kolejnych n wierszach opisane są kwadraty tworzące kolejne rzędy pola. Każdy z tych wierszy zawiera n liczb, 0 lub 1, pooddzielanych pojedynczymi odstępami; opisują one kolejne kwadraty w rzędzie — 0 oznacza kwadrat użytkowy, a 1 nieużytkowy.*

Wyjście

Twój program powinien zapisać w pierwszym i jedynym wierszu pliku tekstowego `dzi.out` *jedną liczbę całkowitą — największą powierzchnię działki. W przypadku, gdy wszystkie kwadraty są nieużytkowe i nie ma żadnej działki, Twój program powinien dać odpowiedź 0.*

Przykład

Dla pliku wejściowego `dzi.in`*:*

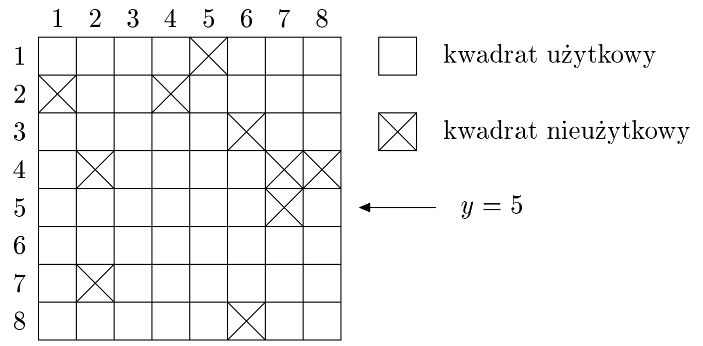
```
5
0 1 0 1 0
0 0 0 0 0
0 0 0 0 1
1 0 0 0 0
0 1 0 0 0
```

poprawną odpowiedzią jest plik wyjściowy `dzi.out`*:*

```
9
```

Rozwiązanie

Wejście przetwarzamy wiersz po wierszu. Za każdym razem aktualizujemy tablicę głębokości $G[0..n+1]$. Przypuśćmy, że znajdujemy się w wierszu o numerze y. Wartość $G[i]$, dla $1 \leqslant i \leqslant n$, mówi nam, jak daleko wstecz sięgają w kolumnie i-tej kwadraty użytkowe — w tym przypadku są to kwadraty o współrzędnych (i,y), $(i,y-1)$, $\ldots$, $(i,y-G[i]+1)$, gdzie pierwsza współrzędna to numer kolumny, a druga to numer wiersza, ponadto kwadrat $(i,y-G[i])$ jest nieużytkowy. Przyjmujemy oczywiście, że 0 oznacza, że już (i,y) jest nieużytkowy. Ponadto $G[0]=G[n+1]=0$ oraz kwadraty poza planszą są nieużytkowe. Nietrudno napisać algorytm działający w czasie $O(n^2)$ i w pamięci $O(n)$, który obliczy kolejno tablicę G dla każdego wiersza pola.



Przykładowe pole dla $n = 8$ i $y = 5$.

Wartości tablicy głębokości na przykładowym polu dla $y = 5$ wynoszą $G[0] = 0$, $G[1] = 3$, $G[2] = 1$, $G[3] = 5$, $G[4] = 3$, $G[5] = 4$, $G[6] = 2$, $G[7] = 0$, $G[8] = 1$, $G[9] = 0$.

Pokażemy teraz jak obliczyć maksymalne pole prostokąta użytkowego o jednym boku w wierszu o numerze y, a drugim w jakimś wcześniejszym wierszu, mając daną tablicę G dla tego właśnie wiersza y. Wykorzystamy do tego celu stos, który będzie przechowywał pary postaci (k, g), gdzie k będzie numerem kolumny startowej, a g wysokością prostokąta. Będziemy przetwarzali kolumny w kolejności numerów od 0 do $n+1$. Po przetworzeniu kolumny numer i stos będzie zawierał maksymalne w sensie zawierania działki o prawym dolnym kwadracie w wierszu y i kolumnie i. Niech S będzie tablicą reprezentującą zawartość stosu. Zachowany będzie następujący niezmiennik. Niech $i < j$, $(k_i, g_i) = S[i]$, $(k_j, g_j) = S[j]$, wtedy $k_i < k_j$ i $g_i < g_j$. Przetwarzanie wiersza zaczynamy z pustym stosem. Przy dojściu do kolumny numer i wykonujemy następujące czynności. Ściągamy ze stosu wszystkie pary postaci (k, g), dla których $g > G[i]$. Przy okazji dostajemy pola $(i-k)g$ działek, które są kandydatami do największej działki. Jeśli teraz stos jest pusty lub na szczycie stosu znajduje się para (k', g'), gdzie $g' < G[i]$, to wstawiamy na stos parę $(k'', G[i])$, gdzie k'' jest wartością k ostatniej zdjętej pary z S lub i, jeśli nic nie zostało zdjęte ze stosu. Wyjątkiem jest sytuacja, kiedy $G[i] = 0$, wtedy nie wstawiamy pary $(k'', G[i])$. Rozważając kolumnę $n+1$ zdejmiemy oczywiście wszystkie prostokąty ze stosu. Nietrudno wykazać, że algorytm jest poprawny, gdyż rozszerza prostokąty o zadanej głębokości tak długo, jak to tylko możliwe i zrzuca je ze stosu dopiero wtedy, gdy nie ma wyboru. Wówczas ewentualnie obcinana jest wysokość ostatniego zdejmowanego prostokąta do wysokości, która pozwala na jego przedłużenie w prawo.

Formalny opis powyższego algorytmu zawiera procedura *PrzetwórzWiersz*.

```
procedure PrzetwórzWiersz;
begin
  Init(S); { Inicjacja stosu }
  for i := 1 to n + 1 do
  begin
    był_Pop := false;
    if not Empty(S) then (k,g) := Top(S);
    while not Empty(S) and g > G[i] do
    begin
      { Kandydatem jest działka [k,i − 1] × [y − G[i] + 1,y] }
      Kandydat((i − k) ∗ G[i]);
        { (i − k) ∗ G[i] jest polem działki; wykonanie procedury Kandydat }
        { powoduje zapamiętanie największego pola z pól }
        { dotychczas odkrytych działek }
      (k',g') := Pop(S); był_Pop := true;
      if not Empty(S) then (k,g) := Top(S);
    end;
    if (Empty(S) and (G[i] > 0)) or (g < G[i]) do
      if był_Pop then Push(S,(k',G[i])) else Push(S,(i,G[i]));
  end;
end;
```

Algorytm działa w czasie i pamięci $O(n)$. Przykładowe wykonanie procedury *PrzetwórzWiersz* zamieszczone jest w poniższej tabeli. Działka $[x_1, x_2] \times [y_1, y_2]$ oznacza prostokąt, którego lewy-górny róg znajduje się w kolumnie x_1 i wierszu y_1, a prawy-dolny róg znajduje się w kolumnie x_2 i wierszu y_2.

	$G[i]$	Kandydaci	Stos
Inicjacja			pusty
$i = 1$	3		(1,3)
$i = 2$	1	$[1,1] \times [3,5]$	(1,1)
$i = 3$	5		(1,1) (3,5)
$i = 4$	3	$[3,3] \times [1,5]$	(1,1) (3,3)
$i = 5$	4		(1,1) (3,3) (5,4)
$i = 6$	2	$[5,5] \times [2,5]$ $[3,5] \times [3,5]$	(1,1) (3,2)
$i = 7$	0	$[3,6] \times [4,5]$ $[1,6] \times [5,5]$	pusty
$i = 8$	1		(8,1)
$i = 9$	0	$[8,8] \times [5,5]$	pusty

Ponieważ przetwarzamy n wierszy, więc złożoność czasowa całego algorytmu wyniesie $O(n^2)$, a pamięciowa $O(n)$. Testy były konstruowane z myślą o preferowaniu właśnie takiego rozwiązania.

Testy

Zadanie było testowane na zestawie 15 danych testowych.

Testy o numerach 1 - 6 zawierają niewielką liczbę losowo wybranych kwadratów nieużytkowych, a pozostałe są użytkowe.

Testy o numerach 7 - 9 zostały utworzone w następujący sposób. Wpierw wypełniamy całe pole kwadratami nieużytkowymi. Następnie wybieramy losowo pewną liczbę figur ograniczonych hiperbolami o równaniach $(x-x_0)(y-y_0) = \pm a^2$ i wypełniamy je kwadratami użytkowymi. Stałe dodatnie x_0, y_0, a są wybierane losowo. Figury te mają tę własność, że każdy wpisany prostokąt ma pole równe a^2. Dodatkowo brzeg tych figur został losowo postrzępiony.

W testach o numerach 10 - 12 cały kwadrat użytkowy poprzecinany jest nieużytkowymi prostymi przebiegającymi pod kątem 45 stopni do osi przez płaszczyznę.

W testach o numerach 13 - 15 cały kwadrat jest użytkowy poza kwadratem (bez wnętrza) obróconym o kąt 45 stopni i wpisanym w pole.

- `dzi1.in` — $n = 10$, losowy
- `dzi2.in` — $n = 100$, losowy
- `dzi3.in` — $n = 500$, losowy
- `dzi4.in` — $n = 1000$, losowy
- `dzi5.in` — $n = 2000$, losowy
- `dzi6.in` — $n = 2000$, losowy
- `dzi7.in` — $n = 100$, hiperbole
- `dzi8.in` — $n = 2000$, hiperbole
- `dzi9.in` — $n = 2000$, hiperbole
- `dzi10.in` — $n = 1000$, proste
- `dzi11.in` — $n = 1876$, proste
- `dzi12.in` — $n = 2000$, proste
- `dzi13.in` — $n = 1000$, wpisany kwadrat
- `dzi14.in` — $n = 1500$, wpisany kwadrat
- `dzi15.in` — $n = 2000$, wpisany kwadrat

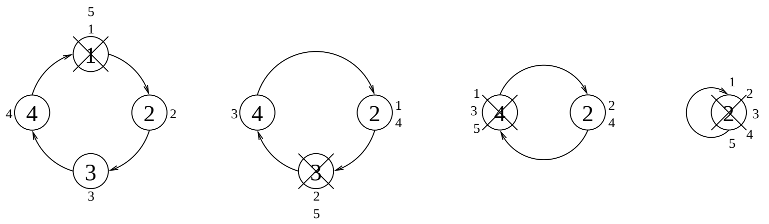
Jakub Pawlewicz
Treść zadania, Rozwiązanie autorskie

Krzysztof Onak
Rozwiązanie wzorcowe, Program

Wyliczanka

Dzieci ustawiły się w kółko i bawią się w wyliczankę. Dzieci są ponumerowane od 1 do n w ten sposób, że (dla $i = 1, 2, \ldots, n-1$) na lewo od dziecka nr i stoi dziecko nr $i+1$, a na lewo od dziecka nr n stoi dziecko nr 1. Dziecko, „na które wypadnie" w wyliczance, wypada z kółka. Wyliczanka jest powtarzana, aż nikt nie zostanie w kółku. Zasady wyliczanki są następujące:

- *Pierwszą wyliczankę zaczyna dziecko nr 1. Każdą kolejną wyliczankę zaczyna dziecko stojące na lewo od dziecka, które ostatnio wypadło z kółka.*
- *Wyliczanka za każdym razem składa się z k sylab. Dziecko, które zaczyna wyliczankę, mówi pierwszą jej sylabę; dziecko stojące na lewo od niego mówi drugą sylabę, kolejne dziecko trzecią itd. Dziecko, które mówi ostatnią sylabę wyliczanki, wypada z kółka.*



Przykładowa wyliczanka dla $n = 4$ i $k = 5$.

Obserwujemy dzieci bawiące się w wyliczankę i widzimy, w jakiej kolejności wypadają one z kółka. Na podstawie tej informacji próbujemy odgadnąć, z ilu sylab składa się wyliczanka.

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `wyl.in` *opis kolejności, w jakiej dzieci wypadały z kółka,*
- *wyznaczy najmniejszą dodatnią liczbę k, dla której dzieci bawiąc się w k-sylabową wyliczankę będą wypadać z kółka w zadanej kolejności, lub stwierdzi, że takie k nie istnieje,*
- *zapisze w pliku tekstowym* `wyl.out` *wyznaczoną liczbę k lub słowo* `NIE` *w przypadku, gdy takie k nie istnieje.*

Wejście

W pierwszym wierszu pliku tekstowego `wyl.in` *znajduje się jedna dodatnia liczba całkowita n, $2 \leqslant n \leqslant 20$. W drugim wierszu znajduje się n liczb całkowitych pooddzielanych pojedynczymi odstępami — i-ta liczba określa, jako które z kolei dziecko nr i wypadło z kółka.*

Wyjście

Twój program powinien zapisać w pierwszym i jedynym wierszu pliku tekstowego `wyl.out` *jedną liczbę całkowitą: najmniejszą liczbę (k) sylab, jakie może mieć wyliczanka, lub jedno słowo* `NIE`*, jeśli taka liczba nie istnieje.*

Przykład

Dla pliku wejściowego `wyl.in`*:*

```
4
1 4 2 3
```

poprawną odpowiedzią jest plik wyjściowy `wyl.out`*:*

```
5
```

Rozwiązanie autorskie

Rozwiązanie zadania opiera się na prostych faktach z teorii liczb i pojęć takich jak kongruencja, największy wspólny dzielnik, najmniejsza wspólna wielokrotność. Definicje tych pojęć przedstawiamy poniżej.

Definicja 1 (Kongruencje) *Niech m będzie dodatnią liczbą całkowitą. Mówimy, że liczba a* **przystaje do** *liczby b* **modulo** *m wtedy i tylko wtedy, gdy $m \mid a - b$ (m dzieli całkowicie $a - b$), co zapisujemy*

$$a \equiv b \,(\mathrm{mod}\ m) \Leftrightarrow m \mid a - b.$$

Resztę z dzielenia a przez m oznaczamy przez $a \bmod m$.

Przy tych oznaczeniach zachodzą oczywiste związki. Jeśli $r = a \bmod m$, to $0 \leqslant r < m$ oraz $a \equiv r (\mathrm{mod}\ m)$. Jeśli $a \equiv b (\mathrm{mod}\ m)$, to $a \bmod m = b \bmod m$.

Definicja 2 (NWD) Największym wspólnym dzielnikiem *liczb całkowitych a i b jest największa dodatnia liczba całkowita $d = \mathrm{NWD}(a,b)$ taka, że $d \mid a$ i $d \mid b$.*

Definicja 3 (NWW) Najmniejszą wspólną wielokrotnością *liczb całkowitych a i b jest najmniejsza dodatnia liczba całkowita $e = \mathrm{NWW}(a,b)$ taka, że $a \mid e$ i $b \mid e$.*

NWD i NWW można w analogiczny sposób zdefiniować dla większej liczby parametrów niż 2.

Problem przedstawiony w zadaniu „Wyliczanka” można sprowadzić do układu kongruencji postaci:

$$k \equiv b_i (\mathrm{mod}\ i) \qquad \text{dla } i = 1, 2, \ldots, n. \tag{1}$$

Robimy to przez symulowanie kolejnych wyliczanek. Pierwszą wyliczankę zaczynamy od dziecka o numerze 1. Na podstawie danych wejściowych znamy numer dziecka, które jako pierwsze wypadło z kółka. Możemy zatem określić, o ile dzieci na lewo znajduję się dziecko, które wypada z kółka. Jest to jedna z liczb $0, 1, \ldots, n-1$. Oznaczmy tę liczbę przez b_n. W kółku jest n dzieci. Odliczając kolejne sylaby wyliczanki wielokrotnie możemy wracać do dziecka o numerze jeden, a następnie po odliczaniu b_n sylab zatrzymać się na dziecku, które powinno jako pierwsze wypaść z kółka. Wnioskujemy zatem, że k jest postaci $an + b_n$, czyli $k \bmod n = b_n$. Wyrzucamy to dziecko z kółka i zapamiętujemy, od którego dziecka zaczyna się następna wyliczanka. Z każdą następna wyliczanką postępujemy podobnie. W i-tej wyliczance jest $n - i + 1$ dzieci w kółku. Wiemy, od którego dziecka zaczyna się aktualna wyliczanka. Znamy też numer dziecka, które jako i-te w kolejności wypadło z kółka. Możemy zatem określić, o ile dzieci na lewo znajduje się dziecko, które ma wypaść z kółka. Jest to jedna z liczb $0, 1, \ldots, n - i$ i oznaczamy ją przez b_{n-i+1}. W ten sposób uzyskamy, że $k \bmod (n - i + 1) = b_{n-i+1}$. Stosując to rozumowanie dla $i = 1, 2, \ldots, n$ dostaniemy wszystkie kongruencje układu (1).

Naszym celem teraz jest znalezienie rozwiązania układu (1) lub stwierdzenie, że nie ma on rozwiązań. W tym celu będzie nam potrzebnych parę faktów.

Fakt 4 *Niech a i b będą dodatnimi liczbami całkowitymi i niech $\mathrm{NWD}(a,b) = d$. Wtedy istnieją liczby całkowite α i β takie, że*

$$a\alpha + b\beta = d.$$

Dowód Przedstawimy algorytm Euklidesa znajdowania największego wspólnego dzielnika d oraz znajdowania liczb α i β. Następnie udowodnimy jego poprawność.

Załóżmy, że $a \geqslant b$. Poniższy pseudokod wylicza wartości d, α i β. W komentarzach napisane są niezmienniki, które są zachowane w danej linii.

```
procedure Euklides(a, b, var d, var α, var β);
begin
  r1 := a; r2 := b; s1 := 1; s2 := 0; t1 := 0; t2 := 1;
  while r2 > 0 do
  begin { NWD(a,b) = NWD(r1,r2), t1a + s1b = r1, t2a + s2b = r2 }
    q := r1 div r2; r3 := r1 mod r2;
    { r1 = qr2 + r3, 0 ⩽ r3 < r2 i NWD(r1,r2) = NWD(r2,r3) }
    s3 := s1 − qs2; t3 := t1 − qt2;
    { t3a + s3b = r3 }
    r1 := r2; r2 := r3; s1 := s2; s2 := s3; t1 := t2; t2 := t3;
  end;
  { r2 = 0 }
  d := r1;
```

14: α := t_1; β := s_1;
15: **end**;

Niezmiennik pętli **while** w linii 5 jest w oczywisty sposób prawdziwy przy pierwszym obrocie pętli. Należy udowodnić, że niezmiennik ten będzie zachowany po wykonaniu instrukcji z wnętrza pętli.

Pierwsze dwie równości z linii 7 są prawdziwe. Trzecia równość wynika z następujących spostrzeżeń. Jeśli $d \mid r_1$ i $d \mid r_2$ to $d \mid r_1 - qr_2 = r_3$ oraz jeśli $d \mid r_2$ i $d \mid r_3$ to $d \mid qr_2 + r_3 = r_1$, zatem $\mathrm{NWD}(r_1, r_2) \mid \mathrm{NWD}(r_2, r_3)$ oraz $\mathrm{NWD}(r_2, r_3) \mid \mathrm{NWD}(r_1, r_2)$.

Udowodnimy równość z linii 9.

$$t_3a + s_3b = (s_1 - qs_2)a + (t_1 - qt_2)b = (s_1a + t_1b) - q(s_2a + t_2b) = r_1 - qr_2 = r_3.$$

Po wykonaniu instrukcji przypisania w linii 10, niezmiennik pętli z linii 5 zostaje zachowany, a ponadto r_2 maleje. Ponieważ r_2 jest zawsze nieujemne, więc pętla **while** się zakończy i będą spełnione równości z linii 5 oraz 12. Zatem

$$\mathrm{NWD}(a, b) = \mathrm{NWD}(r_1, r_2) = \mathrm{NWD}(r_1, 0) = r_1 = d,$$

$$d = r_1 = t_1a + s_1b = \alpha a + \beta b.$$

$\square$

Twierdzenie 5 *Układ kongruencji*

$$\begin{cases} x \equiv a \pmod{dp} \\ x \equiv b \pmod{dq} \end{cases}, \tag{2}$$

gdzie $\mathrm{NWD}(p, q) = 1$, *ma rozwiązanie wtedy i tylko wtedy, gdy*

$$a \equiv b \pmod{d}. \tag{3}$$

Jeśli układ (2) *ma rozwiązanie, to wszystkie rozwiązania tego układu spełniają kongruencję*

$$x \equiv a\alpha q + b\beta p \pmod{dpq}, \tag{4}$$

gdzie α *i* β *to takie liczby całkowite, że* $\alpha p + \beta q = 1$.

Dowód Załóżmy, że układ (2) ma rozwiązanie. Z (2) mamy $dp \mid x - a$ oraz $dq \mid x - b$, skąd $d \mid x - a$ oraz $d \mid x - b$. Wynika stąd, że $d \mid (x - b) - (x - a) = a - b$, a zatem $a \equiv b (\mathrm{mod}\ d)$.

Załóżmy teraz, że spełnione jest (3). Szukamy rozwiązania (2). Muszą być spełnione warunki $dp \mid x - a$ oraz $dq \mid x - b$, skąd $dpq \mid q(x - a)$ oraz $dpq \mid p(x - b)$. Zatem dla dowolnych całkowitych α i β zachodzi

$$dpq \mid \beta q(x - a) + \alpha p(x - b). \tag{5}$$

Niech α i β będą takie, że $\alpha p + \beta q = 1$. Z faktu 4 wynika, że możemy znaleźć takie α i β. Wtedy (5) przekształci się do

$$dpq \mid x(\beta q + \alpha p) - a\beta q - b\alpha p = x - a\beta q - b\alpha p. \tag{6}$$

Zatem x musi spełniać kongruencję

$$x \equiv a\beta q + b\alpha p \pmod{dpq}. \tag{7}$$

Pozostaje udowodnić, że wszystkie x spełniające kongruencję (7) są rozwiązaniami układu (2).

Należy sprawdzić, czy $dp \mid x - a$ oraz $dq \mid x - b$ pod warunkiem, że zachodzi (6). Przekształcamy

$$a\beta q + b\alpha p - a = a(\beta q - 1) + b\alpha p = a(\beta q - (\alpha p + \beta q)) + b\alpha p = -a\alpha p + b\alpha p = (b - a)\alpha p.$$

Z (6) mamy $dpq \mid (x - a) - (a\beta q + b\alpha p - a)$. Zatem $dpq \mid (x - a) - (b - a)\alpha p$, czyli

$$dp \mid (x - a) - (b - a)\alpha p. \tag{8}$$

Z założenia (3) mamy, że $d \mid b - a$, zatem $dp \mid (b - a)\alpha p$, skąd z (8) mamy $dp \mid x - a$. W analogiczny sposób pokazujemy, że $dq \mid x - b$. $\square$

Jako wniosek z twierdzenia 5 przedstawimy procedurę rozwiązywania układu kongruencji

$$\begin{cases} x \equiv a \pmod{A} \\ x \equiv b \pmod{B} \end{cases}. \tag{9}$$

Rozwiązanie polega na przedstawieniu A i B w postaci $A = dp$ i $B = dq$. Mamy $d = \mathrm{NWD}(A, B)$ oraz $p = A/d$ i $q = B/d$, wtedy $\mathrm{NWD}(p, q) = 1$. Oznaczmy $C = dpq$. Wartość dpq jest też $\mathrm{NWW}(A, B)$, czyli zachodzi $C = \mathrm{NWW}(A, B)$. Jeśli układ (9) ma rozwiązanie, to wszystkie rozwiązania x będą miały postać

$x \equiv c (\mathrm{mod}\ C)$, gdzie c można wyznaczyć używając twierdzenia 5. Poniższa funkcja *Układ2* sprawdza istnienie rozwiązania oraz wyznacza liczby c i C.

```
function Układ2(a, A, b, B, var c, var C) : boolean;
begin
  Euklides(A, B, d, α, β);
  if a mod d = b mod d then
  begin
    p := A div d; q := B div d;
    C := dpq; c := aβq + bαp;
    return true;
  end
  else return false;
end;
```

Komentarza wymaga fakt, że $\alpha p + \beta q = 1$. Procedura *Euklides* zwróci wartości α, β, d takie, że $a\alpha + b\beta = d$. Wstawiając $a = dp$, $b = dq$ i dzieląc obie strony przez d otrzymujemy żądaną równość.

Rozwiązanie układu (1) możemy uzyskać stosując $n-1$ krotnie funkcję *Układ2*. Iteracyjnie znajdujemy rozwiązanie układu U_j składającego się z j kongruencji

$$x \equiv b_i \quad (\mathrm{mod}\ i) \text{ dla } 1 \leqslant i \leqslant j,$$

po kolei dla $j = 1, 2, \ldots, n$. Jeśli mamy rozwiązanie $x \equiv c_{j-1} (\mathrm{mod}\ C_{j-1})$ układu U_{j-1}, to rozwiązanie $x \equiv c_j (\mathrm{mod}\ C_j)$ układu U_j znajdujemy rozwiązując układ

$$\begin{cases} x \equiv c_{j-1} & (\mathrm{mod}\ C_{j-1}) \\ x \equiv b_j & (\mathrm{mod}\ j) \end{cases}.$$

Rozumowanie to przedstawiamy w postaci pseudokodu funkcji *RozwiążUkład*.

```
function RozwiążUkład(n, b₁, b₂, ..., bₙ, var c, var C) : boolean;
begin
  c := b₁; C := 1;
  for j := 2 to n do
  begin
    if not Układ2(c, C, bⱼ, j, c', C') then
      return false;
    c := c' mod C'; C := C';
  end;
  return true;
end;
```

Jeśli w wyniku powyższej funkcji dostaniemy rozwiązanie układu (1), to wszystkie rozwiązania będą miały postać

$$x \equiv c \quad (\mathrm{mod}\ C), \tag{10}$$

gdzie

$$C = \mathrm{NWW}(n, \ldots, \mathrm{NWW}(3, \mathrm{NWW}(2, 1)) \ldots) = \mathrm{NWW}(1, 2, \ldots, n).$$

Przykładowo dla $n = 20$ będzie $C = 2^4 \cdot 3^2 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 = 232792560 < 2^{28}$. Zatem wyniki będą mieścić się w liczbach 28-bitowych. Poszukiwaną liczbą k jest po prostu najmniejsza dodatnia liczba całkowita spełniająca kongruencję (10), czyli liczba c z jednym wyjątkiem, a mianowicie jeśli c wynosi 0, to wtedy odpowiedzią jest C.

Rozwiązanie wzorcowe

Okazuje się, że można to zadanie rozwiązać korzystając z prostszego repertuaru metod. Załóżmy na razie, że istnieje liczba k szukana w zadaniu. Tak jak poprzednio, w pierwszym kroku wyznaczamy układ kongruencji (1), jakie musi spełniać rozwiązanie.

Zauważmy, że niektóre z kongruencji mogą być niepotrzebne do wyznaczenia liczby k. Jeśli na przykład zachodzi w zbiorze liczb całkowitych $a_1 \equiv a_2 (\mathrm{mod}\ m_1 m_2)$, gdzie m_1 i m_2 są dodatnie, to $a_1 \equiv a_2 (\mathrm{mod}\ m_1)$. Wynika to stąd, że jeżeli $m_1 m_2$ dzieli $a_1 - a_2$, to tym bardziej m_1 dzieli $a_1 - a_2$. Ponadto otrzymujemy następujący prosty wniosek z twierdzenia 5: jeśli mamy dodatnie m_1 i m_2, które są dodatkowo względnie pierwsze, tzn.

NWD$(m_1, m_2) = 1$, to na podstawie kongruencji $x \equiv a_1 (\bmod\ m_1)$ i $x \equiv a_2 (\bmod\ m_2)$ możemy wyznaczyć jednoznacznie (a właściwie z dokładnością modulo $m_1 m_2$) takie c, że $x \equiv c (\bmod\ m_1 m_2)$. Nietrudno wysnuć stąd wniosek, że jeżeli rozwiązanie istnieje, to do jego wyznaczenia wystarczą kongruencje modulo największe potęgi liczb pierwszych. Na przykład dla $n = 10$ bierzemy kongruencje modulo 5, 7, 8 i 9. Oznaczmy przez c_1, c_2, ..., c_q liczby, modulo które kongruencje będziemy teraz rozważali.

Oto algorytm, dzięki któremu znajdziemy kandydata na rozwiązanie:

```
1: c := 0;
2: C := 1;
3: for i := 1 to q do begin
4:   { niezmiennik: dla każdego j ∈ {1,...,i−1} zachodzi c ≡ b_{c_j} (mod c_j) }
5:   while c ≢ b_{c_i} (mod c_i) do
6:     c := c + C;
7:   C := C * c_i;
8: end;
9: { dla każdego j ∈ {1,...,q} zachodzi c ≡ b_{c_j} (mod c_j) }
```

Pozostaje udowodnić poprawność działania algorytmu. Przypomnijmy sobie, że wartości c_j są parami względnie pierwsze. Przypuśćmy, że jesteśmy wewnątrz pętli z ustaloną wartością i. Zauważmy, że jeśli $i > 1$, to C jest z jednej strony iloczynem dotychczas rozważonych wartości c_j, a z drugiej strony równocześnie ich najmniejszą wspólną wielokrotnością. Jeżeli natomiast $i = 1$, to $C = 1$. Niezależnie od tego, który przypadek zachodzi w pętli **while** w wierszach 5–6, C jest minimalną dodatnią wartością, której dodanie do c nie spowoduje, że dla $j < i$ przestaną zachodzić kongruencje $c \equiv b_{c_j} (\bmod\ c_j)$. Pętla zakończy działanie po co najwyżej c_i jej wykonaniach. Stanie się tak, ponieważ dodawanie C do c generuje wszystkie możliwe reszty modulo c_i. Mamy bowiem, gdy dochodzimy do wspomnianej pętli,

$$\{c \bmod c_i, (c + C) \bmod c_i, \ldots, (c + (c_i - 1)C) \bmod c_i\} = \{0, 1, \ldots, c_i - 1\}.$$

Gdyby tak nie było, tzn. gdyby reszty modulo powtórzyły się wcześniej niż po dodaniu C co najmniej c_i razy, to oznaczałoby, że najmniejsza wspólna wielokrotność liczb C i c_i, których największy wspólny dzielnik jest równy 1, jest mniejsza od $c_i C$, co już prawdą na pewno nie jest. Ponadto, gdy pętla z wierszy 5–6 się zatrzyma, to będzie spełniona kongruencja, którą dokładamy do dotychczas spełnionych. To kończy dowód poprawności przedstawionego algorytmu.

Otrzymujemy ostatecznie z dokładnością modulo $c_1 c_2 \ldots c_q$ kandydata na rozwiązanie układu kongruencji (1) — jest nim wartość c po wykonaniu powyższego algorytmu. Do tej pory zakładaliśmy, że rozwiązanie układu istnieje. Sprawdzamy teraz zatem, czy uzyskane c spełnia wszystkie kongruecje. Jeśli nie, to zadanie nie ma rozwiązania, a jeśli tak, to $k \equiv c (\bmod\ c_1 c_2 \ldots c_q)$ i szukane k otrzymujemy wybierając najmniejszą dodatnią wartość, która spełnia kongruencję.

Program wzorcowy `wyl.pas` stanowi implementację właśnie tego rozwiązania i działa w czasie $O(n^2)$.

Testy

Zadanie testowane było na podstawie 12 testów, z czego pary testów (2a,2b) oraz (7a,7b) zostały zgrupowane.

- `wyl1.in` — $n = 7$, $k = 420$
- `wyl2a.in` — $n = 10$, odpowiedź `NIE`
- `wyl2b.in` — $n = 13$, $k = 174236$
- `wyl3.in` — $n = 14$, $k = 360360$
- `wyl4.in` — $n = 15$, $k = 223123$
- `wyl5.in` — $n = 19$, $k = 232792560$
- `wyl6.in` — $n = 17$, $k = 7172328$
- `wyl7a.in` — $n = 20$, odpowiedź `NIE`
- `wyl7b.in` — $n = 18$, $k = 6123123$
- `wyl8.in` — $n = 20$, $k = 73121593$
- `wyl9.in` — $n = 19$, $k = 160849000$
- `wyl10.in` — $n = 20$, $k = 211433423$

Marcin Sawicki
Treść zadania, Opracowanie

Krzysztof Onak
Program

Kurort narciarski

W Bajtogórach znajduje się kurort narciarski Bajtyrk słynący z narciarskich tras biegowych. Są one bardzo malownicze, gdyż część z nich jest położona wysoko w górach lub w trudno dostępnych miejscach. Użytkownicy tras często korzystają z wyciągów ułatwiających dotarcie do niektórych z nich. Każdy wyciąg i każda trasa zaczyna się i kończy na określonej polanie. Trasy narciarskie nie mogą się przecinać, ale mogą przebiegać naturalnymi skalnymi tunelami i estakadami.

Trasy narciarskie mogą być jednokierunkowe lub dwukierunkowe. Podobnie, niektóre wyciągi (kolejki linowe) mogą być jedno lub dwukierunkowe.

Za korzystanie z wyciągów płaci się kartą magnetyczną. Karty kupuje się w kasach. Każda karta zawiera określoną liczbę punktów. Skorzystanie z każdego z wyciągów wiąże się z utratą pewnej liczby punktów zależnej od wyciągu. Niestety kasy nie zwracają pieniędzy za niewykorzystane punkty.

Bajtoni jest dziś na nartach ostatni dzień. Została mu jedna karta z pewną liczbą punktów, które chciałby wykorzystać do maksimum. Możesz założyć, że ta liczba punktów wystarczy na powrót do Bajtyrku.

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `kur.in` *opis sieci tras i wyciągów oraz informacje o położeniu Bajtoniego i liczbie punktów na jego karcie,*
- *obliczy, z jaką najmniejszą liczbą punktów na karcie Bajtoni może dziś wrócić na dół, do Bajtyrku,*
- *zapisze wynik w pliku tekstowym* `kur.out`*.*

Wejście

W pierwszym wierszu pliku tekstowego `kur.in` *zapisane są dwie dodatnie liczby całkowite* n *i* n', $1 \leqslant n' < n \leqslant 1\,000$, *oddzielone pojedynczym odstępem, oznaczające odpowiednio liczbę wszystkich polan oraz liczbę tych polan, które znajdują się na dole w samym Bajtyrku (są to polany o numerach od 1 do* n' *włącznie).*

W drugim wierszu zapisana jest jedna dodatnia liczba całkowita k, $1 \leqslant k \leqslant 5\,000$, *równa łącznej liczbie wszystkich tras narciarskich. Każdy z kolejnych* k *wierszy zawiera po dwie różne dodatnie liczby całkowite, pooddzielane pojedynczymi odstępami,* $1 \leqslant p_1 \neq p_2 \leqslant n$. *Liczby te oznaczają numery polan, początkowej i końcowej, danej trasy narciarskiej. Trasy dwukierunkowe są tu liczone dwukrotnie i reprezentowane w pliku wejściowym przez dwa (niekoniecznie kolejne) wiersze (postaci „*p_1 p_2*” i „*p_2 p_1*”).*

W $k+3$*-cim wierszu zapisana jest jedna dodatnia liczba całkowita* m, $1 \leqslant m \leqslant 300$, *równa liczbie wszystkich wyciągów. W kolejnych* m *wierszach opisane są wyciągi. W każdym z tych wierszy zapisane są trzy dodatnie liczby całkowite* q_1, q_2 *i* r, *pooddzielane pojedynczymi odstępami. Liczby* q_1 *i* q_2 *oznaczają odpowiednio numer polany, na której wyciąg się zaczyna, i numer polany, na której się kończy,* $1 \leqslant q_1 \neq q_2 \leqslant n$. *Liczba* r *jest równa liczbie punktów, które trzeba zapłacić za przejazd wyciągiem,* $1 \leqslant r \leqslant 1000$. *Wyciągi dwukierunkowe (kolejki linowe) są tu liczone dwukrotnie i reprezentowane w pliku wejściowym przez dwa (niekoniecznie kolejne) wiersze (postaci „*q_1 q_2 r_1*” i „*q_2 q_1 r_2*”). Ceny przejazdu wyciągiem w jedną i w drugą stronę mogą być różne.*

W ostatnim wierszu zapisane są dwie dodatnie liczby całkowite b *i* s, *oddzielone pojedynczym odstępem,* $1 \leqslant b \leqslant n$, $1 \leqslant s \leqslant 2\,000$. *Pierwsza z nich oznacza numer polany, na której znajduje się Bajtoni, a druga jest równa liczbie punktów na jego ostatniej karcie magnetycznej.*

Wyjście

Twój program powinien zapisać w pierwszym (i jedynym) wierszu pliku tekstowego `kur.out` *jedną liczbę całkowitą, równą najmniejszej możliwej liczbie punktów, z jaką Bajtoni może wrócić do Bajtyrku.*

Przykład

Dla pliku wejściowego `kur.in`*:*

```
5 2
6
3 2
```

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

poprawną odpowiedzią jest plik wyjściowy `kur.out`*:*

```
1
```

Najprostsze rozwiązanie

Na pierwszy rzut oka mogłoby się wydawać, że zadanie to jest podobne do zadania szukania najkrótszej (czyli najtańszej) drogi w grafie, tyle tylko, że trzeba poodwracać nierówności. Chwila namysłu pozwala jednak dostrzec różnice: przykładowo, przy szukaniu najkrótszej drogi nigdy nie opłaca się wracać w to samo miejsce. Natomiast Bajtoni często może potrzebować zjeżdżać i wjeżdżać po kilka razy tą samą trasą (na przykład, gdyby miał do dyspozycji tylko jeden wyciąg i jedną trasę zjazdową wzdłuż tego wyciągu, jeździłby w kółko aż by mu zabrakło punktów na karcie).

Gdzie zatem szukać natchnienia do rozwiązania tego zadania? Przypomina ono trochę problem plecakowy, który polega na tym, że musimy zapakować plecak o danej pojemności wybranymi przedmiotami tak, by jak najmniej miejsca pozostało niewykorzystanego. Zadanie to rozwiązuje się metodą programowania dynamicznego. Różnica jest taka, że możliwości wyboru wyciągu nie są u Bajtoniego niezależne od poprzednio dokonanych wyborów, ogranicza go nie tylko liczba punktów na karcie, ale też jego bieżące położenie na stoku. Różnica ta nie przeszkadza jednak wykorzystać tego skojarzenia przy rozwiązywaniu naszego zadania — rozwiązanie wzorcowe, opisane w następnym punkcie, blisko nawiązuje do algorytmu dla problemu plecakowego.

Tymczasem proponuję, byśmy najpierw zajęli się rozwiązaniem, które wydaje mi się prostsze do zrozumienia, a przy tym jest równie efektywne czasowo, co rozwiązanie firmowe, choć ma większą złożoność pamięciową. Inspiracją do tego rozwiązania jest zaś pewne pojęcie z mechaniki.

Chodzi o pojęcie przestrzeni fazowej, bardzo przez fizyków lubiane. Wszyscy wiemy, co to znaczy, że w danej chwili cząstka zajmuje jakieś miejsce w przestrzeni. Wiemy też, co to znaczy, że jej pęd jest jakimś wektorem. Jeśli zaczepimy ten wektor w środku układu współrzędnych, to jego koniec też będzie punktem w pewnej przestrzeni (w której odległości nie mierzy się co prawda w metrach, ale w kilogramach razy metr na sekundę). Jeśli teraz pomnożymy kartezjańsko te dwie przestrzenie, otrzymamy pewną przestrzeń sześciowymiarową, w której każdy punkt wyznacza jednocześnie położenie i pęd cząstki. Czyli zamiast mówić: cząstka ma położenie $[1\mathrm{m}, 2\mathrm{m}, -3\mathrm{m}]$ i pęd $[4\frac{\mathrm{kg\,m}}{\mathrm{s}}, 3\frac{\mathrm{kg\,m}}{\mathrm{s}}, -2\frac{\mathrm{kg\,m}}{\mathrm{s}}]$, mówimy: cząstka w przestrzeni fazowej ma położenie

$$[1\mathrm{m}, 2\mathrm{m}, -3\mathrm{m}, 4\tfrac{\mathrm{kg\,m}}{\mathrm{s}}, 3\tfrac{\mathrm{kg\,m}}{\mathrm{s}}, -2\tfrac{\mathrm{kg\,m}}{\mathrm{s}}]$$

Masło maślane, ale fizycy twierdzą, że jest to bardzo wygodne...

Tymczasem wyobraźmy sobie, że Bajtoni nie tylko jeździ od polany do polany, ale porusza się w przestrzeni fazowej, w której jedną osią jest jego położenie na stoku, drugą zaś liczba impulsów na karcie. Ponieważ mamy n polan i początkowo s impulsów, więc Bajtoni albo jest w trasie, albo zajmuje jeden z $n(s+1)$ wyróżnionych punktów w naszej przestrzeni fazowej. Natomiast zarówno trasy biegowe, jak i wyciągi są krawędziami, łączącymi te punkty — trasy biegowe łączą punkty o tej samej drugiej współrzędnej, zaś wyciągi prowadzą rzecz jasna w kierunku zmniejszającej się drugiej współrzędnej.

Otrzymaliśmy w ten sposób graf, który ma co najwyżej $n(s+1)$ wierzchołków i co najwyżej $(m+k)(s+1)$ krawędzi (bo tak samo, jak każdą polanę reprezentuje w przestrzeni fazowej wiele punktów dla różnych stanów karty, tak samo wiele krawędzi reprezentuje ten sam wyciąg lub trasę, pokonywaną z różnym zapasem impulsów). Wiadomo, że wyznaczenie wszystkich wierzchołków osiągalnych w takim grafie z naszego wierzchołka początkowego (polana nr b, stan karty s) można wykonać standardowym algorytmem przeszukiwania grafu wszerz lub w głąb w czasie $O(n(s+1)+(m+k)(s+1)) = O((n+m+k)s)$ i w pamięci o rozmiarze $O(ns)$. Wystarczy zatem, że podczas takiego przeszukiwania, ilekroć natrafimy na polanę o numerze mniejszym lub równym n' (czyli znajdującą się w Bajtyrku), sprawdzimy aktualny stan karty i porównamy go z najmniejszym uzyskanym dotychczas, by wyznaczyć liczbę będącą rozwiązaniem naszego zadania.

Rozwiązanie wzorcowe

Zauważmy, że nasze rozwiązanie zadziałałoby równie dobrze, jeśli za skorzystanie z niektórych wyciągów w ramach promocji ładowano by Bajtoniemu kartę dodatkowymi impulsami, pod warunkiem, że nie wolno by mu było przekroczyć w ten sposób początkowej liczby s impulsów. Nie można tego powiedzieć o rozwiązaniu wzorcowym, które ma jednak tę zaletę, że jego złożoność pamięciowa wynosi tylko $O(ms+n)$. Aby uzyskać tę oszczędność, poczyńmy kilka obserwacji:

- Skoro nigdy nie poruszamy się w naszej przestrzeni fazowej w kierunku wyższej liczby impulsów na karcie, to możemy z góry założyć, że graf będziemy zwiedzać w kolejności od najwyższej liczby punktów do najniższej. Nazwijmy każdy ze zbiorów wierzchołków grafu, odpowiadający ustalonej liczbie impulsów, *warstwą*.
- Aby zwiedzić całkowicie jedną warstwę, potrzebujemy wykonać przeszukiwanie grafu o rozmiarze tylko n wierzchołków i k krawędzi.
- Zwiedzając warstwę, zapamiętujemy oczywiście, w które miejsca można z niej przejść do warstw położonych niżej. Jednak przejście do niższej warstwy możliwe jest tylko za pomocą wyciągu, więc zbiór wierzchołków grafu, pozostających do zwiedzenia w niższych warstwach, ma rozmiar ograniczony przez ms. Zbioru tego nie możemy trzymać w zwykłej kolejce, bo potrzebujemy móc łatwo wybrać z niego wszystkie wierzchołki z danej warstwy. Możemy go natomiast przechowywać w tablicy kolejek, indeksowanej liczbą impulsów na karcie, lub (jak w rozwiązaniu firmowym) po prostu założyć dwuwymiarową tablicę wartości logicznych $przejazd[0..s-1, 1..m]$, w której wartość *prawda* w elemencie $przejazd[i,j]$ oznacza, że można dotrzeć na polanę na końcu wyciągu j, mając i impulsów na karcie.

Podsumowując, złożoność czasowa przeszukiwania jednej warstwy wynosi $O(n+k+m)$ kroków (gdyż poruszanie się w obrębie tej warstwy to $O(n+k)$ kroków, zaś „odhaczanie" w tablicy *przejazd* możliwych wyjść z tej warstwy to $O(m)$ kroków). Po pomnożeniu tego przez liczbę warstw dostajemy to samo ograniczenie co poprzednio, $O((n+k+m)s)$. Jednak z pamięcią jest lepiej — tablica *przejazd* zajmuje $O(ms)$ komórek pamięci, zaś na przeszukiwanie każdej z warstw wystarczy nam $O(n)$ komórek, zatem łącznie potrzebujemy na obliczenia tylko pamięci o rozmiarze $O(ms+n)$ (nie uwzględniając oczywiście $O(k)$ komórek, niezbędnych do przechowywania danych wejściowych).

Proszę teraz przypomnieć sobie skojarzenie z plecakiem. Rzeczywiście, można powiedzieć, że tablicę *przejazd* wyznaczamy metodą programowania dynamicznego, podobnie jak w rozwiązaniu problemu plecakowego, tyle tylko, że ma ona jeden wymiar więcej.

Dalsze optymalizacje

Można zauważyć, że co prawda w naszym zadaniu $m \leqslant 300$ oraz $n \leqslant 1000$, czyli dla danych o maksymalnym rozmiarze zachodzi nierówność $m < n$, jednak nie jest to wcale regułą, mogą się zdarzyć dane, gdzie np. $n = 30$ i $m = 250$, wtedy wcześniejsze rozwiązanie może być bardziej oszczędne. Aby uniknąć tego kłopotu, należałoby indeksować tablicę *przejazd* nie numerami wyciągów, lecz skompresowanymi numerami tych polan, na których kończą się wyciągi (przykładowo, jeśli wyciągi nr 10, 15 i 17 kończyłyby się na tej samej polanie, to wszystkim im odpowiadałaby ta sama kolumna tabeli $przejazd[\cdot, 3]$). Wówczas łatwo poprawiamy naszą złożoność pamięciową do $O(\min(m,n)s+n)$. Taki zabieg nie ma jednak praktycznego sensu w naszym wypadku, gdy $ms \leqslant 6 \cdot 10^5$.

Przy podanych ograniczeniach na rozmiar danych wejściowych, dalsze optymalizowanie czasu rozwiązania również nie wydaje się celowe. Jednak gdyby liczba wyciągów była znacznie mniejsza od liczby polan i tras (konkretnie, gdyby $m^2 \ll k$), zaś liczba punktów s była większa od m, można by zamiast przeszukiwać w każdej warstwie cały graf, policzyć w nim raz wszystkie możliwe przejazdy łączone trasami biegowymi pomiędzy polanami, na których zaczynają się lub kończą wyciągi (których to polan jest oczywiście nie więcej niż $2m$), a później rozważać już tylko zredukowany graf o $\leqslant 2m$ wierzchołkach. Koszt takiego algorytmu wyniósłby $O(m(n+k))$ na wyznaczenie tras łączonych plus $O(m^2s)$ na właściwe obliczenia. Daje to w sumie $O(m(n+k+ms))$, co jest istotnie mniejsze od $(n+k+m)s$, gdy $m \ll s$ i jednocześnie $m^2 \ll k$.

Testy

Zadanie testowane było na zestawie 10 danych testowych.

Nr	typ testu	n	k	m	s	wynik
1	test losowy	10	15	8	200	5
2	test losowy	40	1	300	500	44
3	liniopętla	259	163	145	1000	50
4	warstwowiec	143	536	63	300	216
5	kaktusowiec	481	266	234	1234	11
6	test losowy	50	2000	300	2000	26
7	liniopętla	812	800	111	1900	40
8	kaktusowiec	879	624	266	1410	2
9	test losowy	1000	5000	300	2000	0
10	warstwowiec	908	4681	274	1999	1184

Paru słów komentarza wymaga zastosowana tu terminologia:

Liniopętla Na początku jest tworzona skierowana linia, na której znajdują się wierzchołki. Do każdego z wierzchołków jest doczepiana „pętla” właściwie będąca skierowanym cyklem. Dla każdej krawędzi jest losowane, czy będzie ona wyciągiem czy zwykłą trasą. Nasz bohater rozpoczyna i kończy podróż na końcach skonstruowanej linii.

Warstwowiec Tworzonych jest kilka grup wierzchołków, dla każdej z nich losuje się wewnętrzne trasy jak w przypadku testów losowych. Każda grupa stanowi *warstwę*. Dla ustalonej kolejności warstw losuje się wyciągi pomiędzy kolejnymi warstwami w określonym kierunku. Bajtonii rozpoczyna i kończy wędrówkę w skrajnych warstwach.

Kaktusowiec Tworzony jest kaktus, a każdy cykl prosty (z jakich jest zbudowany kaktus) zostaje zorientowany. Następnie losuje się dla każdej krawędzi, czy jest ona wyciągiem czy trasą. Polany, startowa i docelowe, są wybierane losowo.

Marcin Stefaniak **Tomasz Waleń**
Treść zadania, Opracowanie Program

Protokoły

Pan Półsieć pracuje w firmie telekomunikacyjnej Bajkotel i jest projektantem protokołów sieciowych. Obecnie zajmuje się on protokołem umożliwiającym przesyłanie danych z jednego komputera do drugiego za pomocą kabla nowej generacji. Kablem takim można przesyłać sygnał o k różnych poziomach napięcia, przy czym napięcie to może się zmieniać co $1/n$ sekundy ($1/n$ sekundy, w trakcie której napięcie musi być stałe, nazywamy **impulsem***). Dane przesyłane są w postaci paczek obejmujących m kolejnych impulsów (czyli przesłanie jednej paczki zajmuje m/n sekund).*

Ze względów technicznych, w obrębie każdej paczki napięcie nie może być stałe, lecz co jakiś czas musi się zmieniać. Mówiąc ściślej, nie można przesyłać paczek danych zawierających l kolejnych impulsów o takim samym poziomie napięcia.

Jeżeli protokół umożliwia przesłanie x różnych paczek, to mówimy, że w jednej paczce możemy zakodować $\log_2 x$ bitów informacji. Pan Półsieć zastanawia się, ile bitów informacji można przesłać maksymalnie w ciągu jednej sekundy.

Przykład

Załóżmy, że kablem można przesyłać sygnał o 2 różnych poziomach napięcia ($k = 2$), które oznaczamy 0 i 1. Jeżeli napięcie może się zmieniać 20 razy na sekundę ($n = 20$), paczki obejmują po 4 impulsy ($m = 4$) i w obrębie każdej paczki żadne 3 kolejne impulsy nie mogą mieć takiego samego napięcia ($l = 3$), to nie można przesyłać paczek: 0000, 0001, 1000, 1111, 1110, 0111. Można natomiast przesyłać paczki: 0010, 0011, 0100, 0110, 0101, 1101, 1100, 1011, 1001 i 1010. Ponieważ można przesyłać 10 różnych rodzajów paczek, więc w każdej paczce można zakodować $\log_2 10$ bitów informacji. W ciągu sekundy można przesłać $20/4 = 5$ paczek, czyli $5 \cdot \log_2 10 \approx 16.6096$ bitów informacji.

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `pro.in` *liczby k, n, m i l opisujące protokół,*
- *obliczy maksymalną liczbę bitów informacji, jakie można przesłać w ciągu sekundy,*
- *zapisze w pliku tekstowym* `pro.out` *obliczoną liczbę bitów zaokrągloną w dół do najbliższej liczby całkowitej.*

Wejście

W pierwszym wierszu pliku tekstowego `pro.in` *zapisane są cztery liczby całkowite, pooddzielane pojedynczymi odstępami:*

- *liczba poziomów napięcia k ($2 \leqslant k \leqslant 10$),*
- *częstotliwość impulsów n ($1 \leqslant n \leqslant 1000$),*
- *rozmiar paczki danych m ($1 \leqslant m \leqslant 100$),*
- *liczba l kolejnych impulsów w paczce, w obrębie których musi nastąpić zmiana napięcia ($2 \leqslant l \leqslant m$).*

Dodatkowo przyjmujemy, że $\frac{n}{m}$ jest liczbą całowitą mniejszą lub równą 10.

Wyjście

Twój program powinien zapisać w pierwszym i jedynym wierszu pliku tekstowego `pro.out` *jedną liczbę całkowitą: maksymalną liczbę bitów, jakie można przesłać w ciągu sekundy, zaokrągloną w dół do najbliższej liczby całkowitej.*

Przykład

Dla pliku wejściowego `pro.in`*:*
```
2 20 4 3
```
poprawną odpowiedzią jest plik wyjściowy `pro.out`*:*
```
16
```

Rozwiązanie

Podobnie jak to w treści zadania, oznaczmy przez k liczbę poziomów napięcia, a przez L (ponieważ małe l myli się z jedynką 1) długość niepoprawnego ciągu kolejnych identycznych impulsów.

Wzór rekurencyjny

Interesuje nas liczba prawidłowych paczek o długości m. Znajdziemy wzór rekurencyjny, który umożliwia jej policzenie.

Oznaczmy szukaną liczbę jako $Ile(m)$. Dla małych m

- $Ile(m) = k^m = k \cdot Ile(m-1)$ dla $1 \leqslant m < L$,
- $Ile(L) = k \cdot Ile(L-1) - k$

Pierwszy wzór jest oczywisty, ponieważ w paczkach krótszych niż L nie może zajść niepożądane nam zjawisko zbyt długiej serii identycznych impulsów. Drugi wzór jest prawdziwy, ponieważ istnieje dokładnie k błędnych paczek długości L. Są to po prostu ciągi impulsów o tym samym poziomie napięcia — jednym spośród k możliwych.

Zastanówmy się, jak policzyć $Ile(m)$ dla $m > L$. Zauważmy, że wszystkie poprawne paczki długości m dają się wygenerować z paczek o długości o jeden mniejszej przez dostawienie końcowego impulsu. Jest tak, ponieważ fragment poprawnej paczki też jest poprawną paczką, w szczególności wszystkie impulsy prócz ostatniego muszą tworzyć poprawną paczkę. Zatem

$$Ile(m) = k \cdot Ile(m-1) - Niepoprawne(m)$$

gdzie $Niepoprawne(m)$ oznacza liczbę niepoprawnych paczek długości m, których początki były poprawne. Ale błąd w takich paczkach może być spowodowany tylko przez niedopuszczalną sekwencję impulsów pod koniec paczki! Zatem są to poprawne paczki długości $m-L$ z doklejonym ciągiem L jednakowych poziomów napięcia. Poniższy schemat ilustruje postać niepoprawnych paczek:

$$\begin{array}{cccccc|c}
\cdots & 1 & 0 & 0 & \cdots & 0 & 0 \\
\cdots & 1 & 2 & 2 & \cdots & 2 & 2 \\
 & \vdots & & & & & \\
\cdots & 1 & (k-1) & (k-1) & \cdots & (k-1) & (k-1) \\
 & \vdots & & & & & \\
\cdots & (k-1) & 0 & 0 & \cdots & 0 & 0 \\
 & \vdots & & & & & \\
\cdots & (k-1) & (k-2) & (k-2) & \cdots & (k-2) & (k-2)
\end{array}$$

Zabroniony ciąg L impulsów wybrany może zostać zawsze na dokładnie $k-1$ sposobów, jako że nie może przyjąć tego poziomu, którym się kończy poprzedzająca go paczka długości $m-L$. Czyli błędne paczki możemy otrzymać na $(k-1) \cdot Ile(m-L)$ sposobów. Ostatecznie

- $Ile(m) = k \cdot Ile(m-1) - (k-1) \cdot Ile(m-L)$ dla $m > L$

Otrzymaliśmy wzór rekurencyjny pozwalający nam obliczyć $Ile(m)$ na podstawie wartości $Ile(i)$ dla mniejszych $i < m$. Wyliczając w pętli kolejne wartości $Ile(i)$ dla $i = 1, 2, \ldots, m$, możemy łatwo wyznaczyć interesujące nas $Ile(m)$, stosując m razy nasz wzór. Co więcej, wystarczy nam pamiętać tylko L ostatnich wartości, na czym możemy oszczędzać pamięć. W każdym kroku pętli wykonujemy stałą liczbę dodawań i mnożeń, więc łącznie kosztuje to nas $O(m)$ dodawań i mnożeń.

Są to jednak duże liczby. Liczba $Ile(m) = \Theta(k^m)$, czyli liczba cyfr $Ile(m)$ jest rzędu $m \log k$. Obie operacje arytmetyczne, jakie wykonujemy na dużych liczbach — mnożenie przez małą liczbę (rzędu k) i odejmowanie — wykonuje się w czasie rzędu liczby cyfr. Ostatecznie złożoność czasowa policzenia $Ile(m)$ wynosi $O(m^2 \log k)$, a złożoność pamięciowa $O(Lm \log k)$.

Obliczenie przepustowości protokołu

Ostatecznym celem jest policzenie liczby bitów, które można przesłać w ciągu sekundy, wyrażonej wzorem

$$\frac{n}{m} \log_2 Ile(m).$$

Można zapytać, skąd w tym wzorze logarytm i do tego jeszcze dwójkowy? Otóż jeden bit informacji – taka jest definicja – to 2 różne stany. Układ k bitów pozwala na zakodowanie 2^k różnych informacji. Na odwrót, gdy mamy możliwość przesłania 2^k różnych wiadomości, to odpowiada to k bitom informacji. Gdy wyjdziemy poza liczby całkowite, x różnych wiadomości odpowiada $\log_2 x$ bitom informacji. Ma to sens - jeśli bowiem możemy wysyłać 3-stanowe komunikaty, to odpowiada to $\log_2 3 \approx 1.58$ bitom informacji. Wprawdzie za pomocą jednego takiego komunikatu prześlemy najwyżej 1 bit, ale już za pomocą dwóch można przesłać aż 3 bity (mamy $3 \times 3 = 9$ możliwości). Ale przecież $\log_2 3 + \log_2 3 \approx 3.17 > 3$, czyli się zgadza.

Zadanie wymagało dokładnego wyliczenia liczby bitów i rozwiązania, które korzystały z obliczeń przybliżonych nie przechodziły wszystkich testów.

Aby precyzyjnie obliczyć wynik, korzystamy z prawa $\log b^a = a \log b$, dzięki czemu

$$\frac{n}{m} \log_2 \mathit{Ile}(m) = \log_2 \mathit{Ile}(m)^{\frac{n}{m}}.$$

Ale w treści zadania wystąpiło założenie, że $\frac{n}{m}$ jest liczbą całkowitą mniejszą lub równą 10. Zatem, policzywszy $\mathit{Ile}(m)$, możemy podnieść tą liczbę do niedużej potęgi (kilka mnożeń dużych liczb, przybliżony koszt czasowy $O(m^2)$).

Na koniec, logarytm przy podstawie 2 to po prostu liczba bitów (długość) danej liczby. Jeśli reprezentujemy duże liczby w układzie dwójkowym to sprawdzenie tego jest banalne, jeśli inaczej – nic strasznego, wystarczy obliczać kolejne potęgi dwójki i porównywać z daną liczbą.

Na marginesie

Interesujący wariant tego zadania otrzymamy, gdy zapytamy się o przepustowość tego kanału dla dostatecznie dużych m. Można się spodziewać, że ta wartość się stabilizuje, ściślej mówiąc – istnieje granica

$$p = \lim_{m \to \infty} \frac{n}{m} \log_2 \mathit{Ile}(m)$$

W rzeczy samej, można pokazać, że

$$\frac{n}{m+1} \log_2 \mathit{Ile}(m) < p < \frac{n}{m} \log_2 \mathit{Ile}(m)$$

i oba te ciągi monotonicznie zbiegają do p. Na ograniczenie dolne można spojrzeć tak, że jeśli po każdej poprawnej m-paczce będziemy wstawiać jeden nieznaczący impuls dla zapewnienia poprawności, to możemy otrzymać dowolnie długi poprawny ciąg impulsów; jednak będziemy przy tym tracili na przekazywanej informacji, bo używamy nieznaczącego bitu.

Rozważmy teraz przepustowość kanału bez ograniczeń, wynosi ona $p_1 = n \log_2 k$. Jest tak, że $p = \lambda p_1$ dla pewnej stałej λ niezależnej od n i m. Stała λ zależy tylko od L, k, a w jaki sposób – to już temat na inne opracowanie.

Przy takim zadaniu nieuniknione wydają się obliczenia przybliżone na liczbach zmiennoprzecinkowych. W takim przypadku zadania należy formułować w następujący sposób: *„program ma policzyć wartość x z dokładnością do ε i wypisać ją zaokrągloną w dół do najbliższej liczby całkowitej”* (może się wtedy zdarzyć, że będzie więcej niż jedna poprawna odpowiedź). Albo nawet prościej: *„możesz założyć, że x różni się od najbliższej liczby całkowitej o więcej niż ε”.* Oczywiście rozwiązując takie zadanie należy zadbać o to, aby program radził sobie z błędami zaokrągleń.

Testy

Testy zostały tak dobrane, by w większości wypadków obliczenia na liczbach rzeczywistych dawały błędne odpowiedzi. Zadanie testowane było na zestawie 14 danych testowych.

- `pro1.in` — $k = 2$, $n = 20$, $m = 10$, $l = 3$, wynik: 14
- `pro2.in` — $k = 2$, $n = 10$, $m = 10$, $l = 10$, wynik: 9
- `pro3.in` — $k = 2$, $n = 15$, $m = 5$, $l = 2$, wynik: 3
- `pro4.in` — $k = 2$, $n = 184$, $m = 92$, $l = 88$, wynik: 183
- `pro5.in` — $k = 8$, $n = 100$, $m = 100$, $l = 65$, wynik: 299
- `pro6.in` — $k = 8$, $n = 240$, $m = 80$, $l = 60$, wynik: 719
- `pro7.in` — $k = 8$, $n = 498$, $m = 83$, $l = 64$, wynik: 1493

- `pro8.in` — $k = 8$, $n = 792$, $m = 88$, $l = 45$, wynik: 2375
- `pro9.in` — $k = 10$, $n = 891$, $m = 99$, $l = 35$, wynik: 2959
- `pro10.in` — $k = 10$, $n = 1000$, $m = 100$, $l = 10$, wynik: 3321
- `pro11.in` — $k = 8$, $n = 8$, $m = 4$, $l = 2$, wynik: 22
- `pro12.in` — $k = 2$, $n = 2$, $m = 2$, $l = 2$, wynik: 1
- `pro13.in` — $k = 7$, $n = 7$, $m = 7$, $l = 7$, wynik: 19
- `pro14.in` — $k = 7$, $n = 33$, $m = 11$, $l = 9$, wynik: 92

Zawody III stopnia

Zawody III stopnia — opracowania zadań

Piotr Chrząstowski
Wojciech Guzicki
Treść zadania

Wojciech Guzicki
Opracowanie

Jarosław Byrka
Program

Minusy

Działanie odejmowania nie jest łączne, np. $(5-2)-1=2$ *, natomiast* $5-(2-1)=4$*, a zatem* $(5-2)-1\neq 5-(2-1)$*. Wynika stąd, że wartość wyrażenia postaci* $5-2-1$ *zależy od kolejności wykonywania odejmowań. Przy braku nawiasów przyjmuje się, że wykonujemy działania od lewej strony, czyli wyrażenie* $5-2-1$ *oznacza* $(5-2)-1$*.*

Mamy dane wyrażenie postaci

$$x_1 \pm x_2 \pm \cdots \pm x_n,$$

gdzie $\pm$ *oznacza* $+$ *(plus) lub* $-$ *(minus), a* x_1*,* x_2*, …,* x_n *oznaczają (parami) różne zmienne. Chcemy w wyrażeniu postaci*

$$x_1 - x_2 - \cdots - x_n$$

tak rozmieścić nawiasy, aby uzyskać wyrażenie równoważne danemu.

Dla przykładu chcąc uzyskać wyrażenie równoważne wyrażeniu

$$x_1 - x_2 - x_3 + x_4 + x_5 - x_6 + x_7$$

możemy w

$$x_1 - x_2 - x_3 - x_4 - x_5 - x_6 - x_7$$

rozmieścić nawiasy na przykład tak

$$((x_1 - x_2) - (x_3 - x_4 - x_5)) - (x_6 - x_7).$$

Uwaga: *Nawiasowania, w których nawiasy nie otaczają żadnej zmiennej lub otaczają tylko jedną zmienną, są niedopuszczalne.*

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `min.in` *opis danego wyrażenia postaci* $x_1 \pm x_2 \pm \cdots \pm x_n$*,*
- *wyznaczy, w jaki sposób można powstawiać nawiasy do wyrażenia* $x_1 - x_2 - \cdots - x_n$*, tak aby uzyskać wyrażenie równoważne danemu; można powstawiać co najwyżej* $n-1$ *par nawiasów,*
- *opisze ten sposób w pliku tekstowym* `min.out`*.*

Wejście

W pierwszym wierszu pliku wejściowego `min.in` *zapisana jest jedna liczba całkowita* n*,* $2 \leqslant n \leqslant 1\,000\,000$*. Jest to liczba zmiennych w danym wyrażeniu. W każdym z kolejnych* $n-1$ *wierszy jest zapisany jeden znak,* `+` *lub* `-`*. W* i*-tym wierszu zapisany jest znak występujący w danym wyrażeniu między* x_{i-1} *i* x_i*.*

Wyjście

Twój program powinien zapisać w pierwszym wierszu pliku wyjściowego `min.out` *szukany sposób wstawienia nawiasów do wyrażenia* $x_1 - x_2 - \cdots - x_n$*. Należy zapisać tylko nawiasy i minusy (bez odstępów między nimi), pomijając zmienne* x_1*,* x_2*, …,* x_n*.*

Możesz założyć, że dla danych testowych zawsze istnieje rozwiązanie. Jeśli istnieje wiele możliwych rozwiązań, Twój program powinien zapisać jedno z nich.

Przykład

Dla pliku wejściowego `min.in`*:*

```
7
-
-
+
+
-
+
```

poprawną odpowiedzią jest plik wyjściowy `min.out`*:*

```
((-)-(--))-(-)
```

Rozwiązanie

W rozwiązaniu zadania „Kod” z VII Olimpiady Informatycznej ([7]) zostało wspomniane następujące zadanie:

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

$$\begin{gathered}(x_1 \circ x_2) \circ (x_3 \circ x_4),\\ ((x_1 \circ x_2) \circ x_3) \circ x_4,\\ x_1 \circ (x_2 \circ (x_3 \circ x_4)),\\ (x_1 \circ (x_2 \circ x_3)) \circ x_4,\\ x_1 \circ ((x_2 \circ x_3) \circ x_4).\end{gathered}$$

Liczba różnych sposobów rozstawienia nawiasów jest tzw. liczbą Catalana C_{n-1}. Liczby te można obliczyć ze wzoru:

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

Początkowe liczby Catalana są równe odpowiednio:

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

Znamy również wzór rekurencyjny

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

Powróćmy do wspomnianego zadania. Jeśli działanie $\circ$ jest łączne, tzn. spełnia warunek

$$a \circ (b \circ c) = (a \circ b) \circ c$$

dla dowolnych a, b i c, to niezależnie od sposobu rozstawienia nawiasów otrzymamy zawsze ten sam wynik. Jeśli zaś działanie $\circ$ jest niełączne, to różne sposoby rozstawienia nawiasów mogą prowadzić do różnych wyników (przy zachowaniu tej samej kolejności argumentów $x_1, x_2, \ldots, x_n$). Jednak możemy otrzymać co najwyżej C_{n-1} różnych wyników.

Przykładem działania niełącznego w zbiorze liczb rzeczywistych jest działanie odejmowania. Mamy na przykład:

$$2-(3-5)=4 \quad \text{oraz} \quad (2-3)-5=-6.$$

Możemy zatem zadać naturalne pytanie o to, czy różne sposoby rozstawienia nawiasów w wyrażeniu postaci

$$x_1 - x_2 - x_3 - \cdots - x_n$$

mogą prowadzić do C_{n-1} różnych wyników. Okazuje się jednak, że nie. Popatrzmy na przykład:

$$x_1 - (x_2 - (x_3 - x_4)) = (x_1 - (x_2 - x_3)) - x_4 = x_1 - x_2 + x_3 - x_4.$$

Zatem już dla $n = 4$ dwa różne sposoby rozstawienia nawiasów prowadzą do tego samego wyniku, a więc liczba możliwych wyników jest mniejsza od C_{n-1}.

Zauważmy, że jeśli rozstawimy nawiasy w dowolny sposób, a następnie otworzymy je kolejno, to otrzymamy wyrażenie postaci

$$x_1 - x_2 \pm x_3 \pm x_4 \pm \cdots \pm x_n,$$

gdzie każdy znak $\pm$ jest jednym ze znaków $+$ lub $-$. Wynika stąd, że możemy otrzymać co najwyżej 2^{n-2} różnych wyników. Udowodnimy następujące twierdzenie:

Twierdzenie Dla każdego wyrażenia postaci

$$x_1 - x_2 \pm x_3 \pm x_4 \pm \cdots \pm x_n \qquad (*)$$

(gdzie każdy znak $\pm$ jest jednym ze znaków $+$ lub $-$) istnieje takie rozstawienie nawiasów w wyrażeniu

$$x_1 - x_2 - x_3 - x_4 - \cdots - x_n,$$

że po otwarciu wszystkich nawiasów otrzymamy wyrażenie $(*)$.

Dowód Szukany sposób rozstawienia nawiasów otrzymamy w trzech krokach.

1. Każdy maksymalny blok postaci
$$x_i + x_{i+1} + \cdots + x_j$$
ujmiemy w nawias, zamieniając wszystkie znaki $+$ na znaki $-$; należy przy tym zwrócić uwagę na to, że ten blok jest poprzedzony znakiem $-$, co wynika z maksymalności.

   Otrzymujemy w ten sposób wyrażenie postaci
$$s_1 - s_2 - s_3 - \cdots - s_k,$$
gdzie $s_1, s_2, \ldots, s_k$ są albo pojedynczymi zmiennymi, albo blokami postaci
$$(x_i - x_{i+1} - \cdots - x_j).$$

2. Każdy blok postaci
$$(x_i - x_{i+1} - x_{i+2} - \cdots - x_j)$$
zastępujemy wyrażeniem
$$(((\ldots(x_i - x_{i+1}) - x_{i+2}) - \cdots - x_{j-1}) - x_j).$$

3. Całe wyrażenie
$$s_1 - s_2 - s_3 - \cdots - s_k,$$
w którym bloki $s_2, \ldots, s_k$ zostały już zastąpione odpowiednimi wyrażeniami tak jak w punkcie 2, przekształcamy do postaci
$$((\ldots(x_1 - s_2) - s_3) - \cdots - s_{k-1}) - s_k.$$

   To kończy dowód. □

Wniosek Dla działania dodawania istnieje dokładnie 2^{n-2} sposobów rozstawienia nawiasów w wyrażeniu

$$x_1 - x_2 - x_3 - x_4 - \cdots - x_n,$$

prowadzących do różnych wyrażeń algebraicznych.

Warto zwrócić uwagę na fakt, że dla $n \geqslant 4$ mamy nierówność

$$C_{n-1} > 2^{n-2}.$$

W zadaniu „Minusy” należało dokonać kroku 1, a więc po prostu zastąpić wszystkie bloki postaci

$$x_i + x_{i+1} + \cdots + x_j$$

blokami postaci

$$(x_i - x_{i+1} - \cdots - x_j).$$

W tym celu należy każdą grupę znaków $+ + \cdots +$ zastąpić grupą znaków $- - \cdots -$ tej samej długości, ujmując ją w nawias. Program wzorcowy działa następująco:

1. Wczytuje znak $-$ i wypisuje znak $-$.
2. Wczytuje kolejne znaki i dla każdego znaku oprócz pierwszego:
   - jeśli ten znak jest jest różny od poprzedniego, to wypisuje nawias otwierający lub zamykający (otwierający dla zmiany $-+$ i zamykający dla zmiany $+-$);
   - wypisuje znak $-$.
3. Jeśli ostatnim znakiem był znak $+$, to wypisuje nawias zamykający.

Inny sposób polega na tym, że po wczytaniu każdego znaku $-$ zapisujemy w pliku wyjściowym znak $-$, a po wczytaniu znaku $+$ zaczynamy zliczać kolejne znaki $+$; gdy zakończy się seria k znaków $+$, wypisujemy nawias otwierający, następnie k znaków $-$, i wreszcie nawias zamykający.

Na zakończenie zilustrujemy trzy kroki dowodu twierdzenia na przykładzie wyrażenia

$$x_1 - x_2 + x_3 - x_4 - x_5 + x_6 + x_7 - x_8 + x_9.$$

Najpierw przekształcamy je do postaci

$$x_1 - (x_2 - x_3) - x_4 - (x_5 - x_6 - x_7) - (x_8 - x_9).$$

Następnie wyrażenie $x_5 - x_6 - x_7$ zastępujemy wyrażeniem $(x_5 - x_6) - x_7$. Otrzymujemy zatem

$$x_1 - (x_2 - x_3) - x_4 - ((x_5 - x_6) - x_7) - (x_8 - x_9).$$

Wreszcie dostajemy

$$(((x_1 - (x_2 - x_3)) - x_4) - ((x_5 - x_6) - x_7)) - (x_8 - x_9).$$

Zadanie to prowokuje natychmiast do zadania następującego pytania: na ile różnych sposobów można rozstawić nawiasy w wyrażeniu

$$x_1 - x_2 - x_3 - x_4 - \cdots - x_n,$$

by otrzymać z góry zadane wyrażenie postaci

$$x_1 - x_2 \pm x_3 \pm x_4 \pm \cdots \pm x_n?$$

Pytanie to jest treścią zadania „Nawiasy”.

Testy

Jeden test zawierał dane z przykładu. Jeden składał się z jednego minusa. Pięć testów zawierało ciągi na przemian występujących minusów i plusów różnych długości (od 100 do 999999). Jeden test składał się z 999999 minusów. Wreszcie cztery testy zawierały losowe ciągi plusów i minusów różnych długości (od 1001 do 999999).

Piotr Chrząstowski
Treść zadania, Opracowanie

Remigiusz Różycki
Program

Narciarze

Na południowym stoku Bajtogóry znajduje się szereg tras narciarskich oraz jeden wyciąg. Wszystkie trasy prowadzą od górnej do dolnej stacji wyciągu. Każdego ranka grupa pracowników wyciągu sprawdza stan tras. Razem wjeżdżają na górną stację, po czym każdy zjeżdża do dolnej stacji wybraną trasą. Każdy pracownik zjeżdża tylko raz. Trasy pracowników mogą się częściowo pokrywać. Trasa sprawdzana przez każdego z nich cały czas prowadzi w dół.

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

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `nar.in` *mapę tras narciarskich,*
- *wyznaczy minimalną liczbę pracowników, którzy są w stanie sprawdzić wszystkie przecinki między polanami,*
- *zapisze wynik do pliku tekstowego* `nar.out`.

Wejście

W pierwszym wierszu pliku wejściowego `nar.in` *znajduje się jedna liczba całkowita n równa liczbie polan, $2 \leqslant n \leqslant 5\,000$. Polany są ponumerowane od 1 do n.*

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

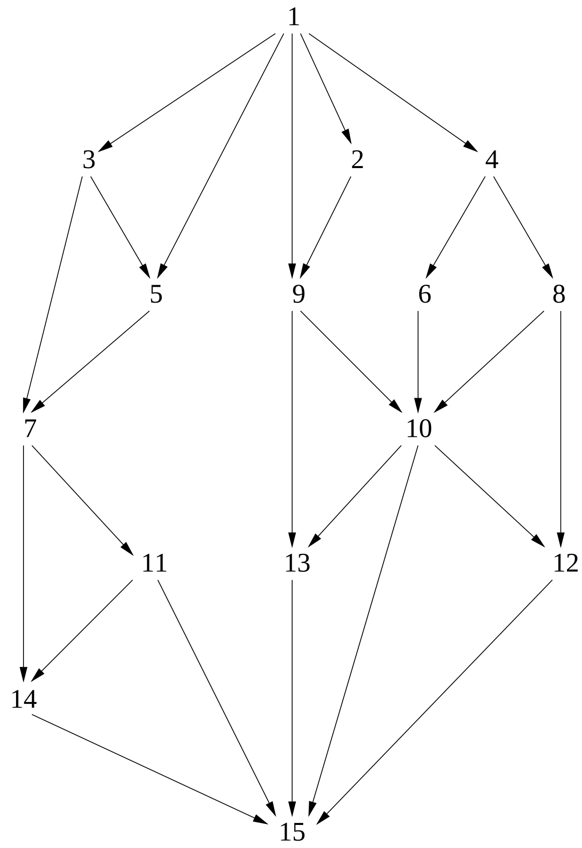
Wyjście

W pierwszym i jedynym wierszu pliku wyjściowego `nar.out` *powinna znajdować się dokładnie jedna liczba całkowita — minimalna liczba pracowników, którzy są w stanie sprawdzić wszystkie przecinki.*

Przykład

Dla pliku wejściowego `nar.in`*:*

```
15
5 3 5 9 2 4
1 9
2 7 5
2 6 8
1 7
1 10
2 14 11
2 10 12
2 13 10
3 13 15 12
2 14 15
1 15
1 15
1 15
```



poprawną odpowiedzią jest plik wyjściowy `nar.out`*:*

```
8
```

Omówienie zadania

Zadanie o narciarzach do tego stopnia przypominało treścią zadanie z I etapu sprzed 2 lat, że w czasie trwania konkursu zdarzały się pytania, jak to jest możliwe, żeby na olimpiadzie powtórzyło się zadanie. Oczywiście nie jest to możliwe i było to zupełnie inne zadanie, choć faktycznie miało identyczny tytuł i podobną scenerię: opisany był stok narciarski z jedną kolejką i wieloma trasami prowadzącymi przecinkami leśnymi i spotykającymi się na polanach. Przypomnijmy, że wtedy chodziło o maksymalną liczbę tras, które nie miały wspólnych odcinków i umożliwiały bezpieczny trening. Tutaj chodziło o co innego: o minimalną liczbę tras, które pokrywały cały graf tras narciarskich.

I tu i tam mieliśmy do czynienia z bardzo specyficznym rodzajem grafu. Jako że opisywał on trasy narciarskie na stoku, był on planarny (co gwarantował brak tuneli i estakad). Dodatkowo ścieżki w tym grafie nie miały cykli. W końcu graf miał dwa wyróżnione węzły: górną i dolną stację kolejki. Z górnej stacji można było dojechać do każdego węzła w grafie i jednocześnie z każdego węzła można było dojechać do dolnej stacji. Nazwijmy każdy taki graf *grafem narciarskim*. Skoro już dwukrotnie powtórzył się na olimpiadzie ten typ grafów, to zasługuje na swoją nazwę.

Zauważmy, że każdy graf narciarski ma „dualny” graf krawędziowy, który możemy zdefiniować następująco. Węzłami grafu dualnego są krawędzie grafu oryginalnego. Krawędzie w grafie dualnym definiujemy między e_1 i e_2, jeśli z przecinki e_1 wyjeżdża się na polanę, z której można wyjechać przecinką e_2. Czyli sąsiednimi są te krawędzie, które mogą wystąpić jedna za drugą na drodze pewnego zjazdu z góry na dół.

Oznaczmy nasz oryginalny graf narciarski przez N, a dualny do niego graf krawędziowy przez N^d.

Rozważmy teraz relację w grafie dualnym N^d zdefiniowaną między węzłami tego grafu (czyli przecinkami leśnymi na trasach) następująco:

$$e_1 \leq e_2 \Leftrightarrow \text{istnieje w } N^d \text{ ścieżka z } e_1 \text{ do } e_2,$$

Jest to zatem relacja wyrażająca fakt *poprzedzania*, czy bycia wyżej. Przecinka e_1 poprzedza przecinkę e_2, jeśli z e_1 można dojechać do e_2. Relacja ta ma następujące własności:

- jest zwrotna, czyli $\forall e \in E : e \leq e$;
- jest antysymetryczna, czyli $\forall e_1, e_2 \in E : e_1 \leq e_2 \wedge e_2 \leq e_1 \Rightarrow e_1 = e_2$;

- jest przechodnia, $\forall e_1, e_2, e_3 \in E : e_1 \leq e_2 \wedge e_2 \leq e_3 \Rightarrow e_1 \leq e_3$.

Każda relacja spełniająca te własności nazywa się relacją *częściowego porządku*. Relacje częściowego porządku pełnią ważną rolę w informatyce. Umożliwiają one klasyfikację danych i od lat są przedmiotem rozległych badań.

Mówimy, że zbiór A jest częściowo uporządkowany, jeśli została w nim określona relacja częściowego porządku $\leq$. Taki uporządkowany zbiór oznaczamy jako parę $\langle A, \leq \rangle$. Ważne jest zawsze, żeby wiedzieć, o jaki porządek nam chodzi, bowiem zbiory można porządkować na wiele sposobów. W tym sensie nasz zbiór przecinek leśnych został właśnie częściowo uporządkowany przez relację $\leq$.

Jeśli między dwoma elementami zbioru częściowo uporządkowanego zachodzi relacja $a \leq b$, to mówimy, że a jest *niewiększe* od b, a b *niemniejsze* od a. Wprowadzimy jeszcze dwa pojęcia, które pozwolą nam na sformułowanie twierdzenia kluczowego dla rozwiązania naszego zadania.

W zadanym zbiorze A uporządkowanym częściowo przez relację $\leq$, zbiór $\{a_1, \ldots, a_n\}$ nazwiemy *łańcuchem*, jeśli $a_1 \leq a_2$, $a_2 \leq a_3$, $\ldots$, $a_{n-1} \leq a_n$[1]. Odnosząc to do naszego przypadku, łańcuchami będą na przykład wszystkie możliwe zestawy przecinek leśnych tworzące spójne odcinki tras narciarskich. Powiemy, że łańcuch jest *maksymalny*, jeśli nie da się go powiększyć. W naszym przypadku maksymalne łańcuchy będą odpowiadały trasom, które zaczynają się przy górnej stacji kolejki, a kończą przy dolnej. Dla danego łańcucha $L = \{a_1, \ldots, a_n\}$, w którym $a_1 \leq a_2 \leq \ldots \leq a_n$, element a_1 nazwiemy elementem najmniejszym, zaś element a_n elementem największym w tym łańcuchu. Wszystkie te pojęcia można łatwo odnieść do zbiorów nieskończonych: wtedy czasami łańcuch może nie mieć elementu najmniejszego bądź największego, ale w naszym przypadku, kiedy mamy do czynienia ze skończonymi grafami, każdy łańcuch ma zarówno element najmniejszy, jak i największy.

W pewnym sensie dualnym pojęciem jest pojęcie *antyłańcucha*. Zbiór $B = \{b_1, \ldots, b_n\}$ nazwiemy *antyłańcuchem*, jeśli żadne dwa elementy tego zbioru nie są ze sobą w relacji $\leq$. Podobnie, antyłańcuch B nazwiemy maksymalnym, jeśli dla każdego $x \in A \setminus B$ zbiór $B \cup \{x\}$ nie jest antyłańcuchem.

Niech A_1, $\ldots$, A_m będzie rodziną podzbiorów zbioru A. Powiemy, że pokrywa ona zbiór A, jeśli $\bigcup_{k=1}^{m} A_k = A$. Zauważmy, że przejazd narciarski z góry na dół odpowiada wybraniu pewnego maksymalnego łańcucha w grafie N^d. Zadanie nasze sprowadza się zatem do znalezienia takiej rodziny maksymalnych łańcuchów, która pokrywa zbiór wierzchołków grafu dualnego N^d i która zawiera możliwie mało elementów.

Z pomocą przychodzi nam następujące twierdzenie Dilwortha, odkryte w 1950 roku i opisane m.in. w znakomitej książce W. Marka i W. Lipskiego pt. „Analiza kombinatoryczna” ([21]).

Twierdzenie *W dowolnym skończonym zbiorze częściowo uporządkowanym $\langle P, \leq \rangle$ maksymalna liczność antyłańcucha jest równa minimalnej liczbie łańcuchów, które pokrywają zbiór P.*

Dowód Dowód przytaczamy za cytowaną książką. Załóżmy, że maksymalna liczność antyłańcucha w zbiorze P wynosi m. Zauważmy najpierw, że trzeba co najmniej m łańcuchów, aby pokryć zbiór P; każdy bowiem łańcuch może zawierać co najwyżej jeden element antyłańcucha. Pokażemy, że wystarcza m łańcuchów, aby pokryć zbiór P.

Stosujemy indukcję względem liczności zbioru P. Dla $|P| = 1$ twierdzenie zachodzi w oczywisty sposób. Jedyny antyłańcuch maksymalny ma 1 element i tyle samo wynosi minimalna liczba łańcuchów, które ten zbiór pokrywają. Załóżmy teraz, że teza zachodzi dla wszystkich $k < n = |P|$. Rozważmy dowolny łańcuch maksymalny L oraz dowolny antyłańcuch A o największej możliwej liczności m. Jeżeli usuniemy ze zbioru P wszystkie elementy z L, a w otrzymanym zbiorze $P \setminus L$ zignorujemy te elementy relacji, które dotyczyły L, to mamy dwa przypadki. Albo długość najdłuższego antyłańcucha, oznaczmy go przez A', wynosi teraz $m - 1$ i wtedy na mocy założenia indukcyjnego zbiór $P \setminus L$ da się pokryć za pomocą $m - 1$ łańcuchów, co włącznie z łańcuchem L stanowi m-elementowe pokrycie zbioru P (koniec dowodu), albo długość A' jest nadal równa m (mniej nie może być, bo każdy łańcuch z każdym antyłańcuchem może mieć co najwyżej jeden wspólny element).

Zauważmy, że A' jest maksymalnym antyłańcuchem nie tylko w $P \setminus L$, ale także w P, a ponadto $A' \cap L = \emptyset$. Określmy dwa zbiory: $D = \{x \in P : \exists a \in A' : a \leq x\}$ oraz $G = \{x \in P : \exists a \in A' : x \leq a\}$. Zbiór D jest to zbiór tych elementów, które nie leżą powyżej antyłańcucha A', zaś G, to zbiór elementów nieleżących poniżej A'. Rozważmy element minimalny c łańcucha L (położony najwyżej). Nie może on należeć do D, gdyż inaczej można by łańcuch L powiększyć o pewien element $a \in A'$, $a < c$ — taki element znaleźlibyśmy w naszym antyłańcuchu — przecząc maksymalności L. (W tym przypadku $a < c$ oznacza, że $a \leq c$ i $a \neq c$.) Tak samo zbiór G nie zawiera elementu maksymalnego łańcucha L. Zatem $|D| < |P|$ i $|G| < |P|$ i na mocy założenia indukcyjnego istnieją rozkłady tych zbiorów na łańcuchy: $G = G_1 \cup \cdots \cup G_m$, $D = D_1 \cup \cdots \cup D_m$, bo antyłańcuch A w P jest również antyłańcuchem w każdym z tych zbiorów ($A \subseteq D$ i $A \subseteq G$).

Przyjmijmy bez utraty ogólności, że $a_i \in G_i \cap D_i$ dla każdego $1 \leq i \leq m$. Zauważmy, że $P = D \cup G$, gdyż inaczej mając element x nieporównywalny z żadnym a_i moglibyśmy antyłańcuch A powiększyć o x, przecząc założeniu o jego maksymalności. Zatem $P = (D_1 \cup G_1) \cup \cdots \cup (D_m \cup G_m)$ jest rozkładem zbioru P na m łańcuchów. To kończy dowód. □

[1] Przyjmujemy tu, że elementy a_1, $\ldots$, a_n są parami różne i że $n \geqslant 1$.

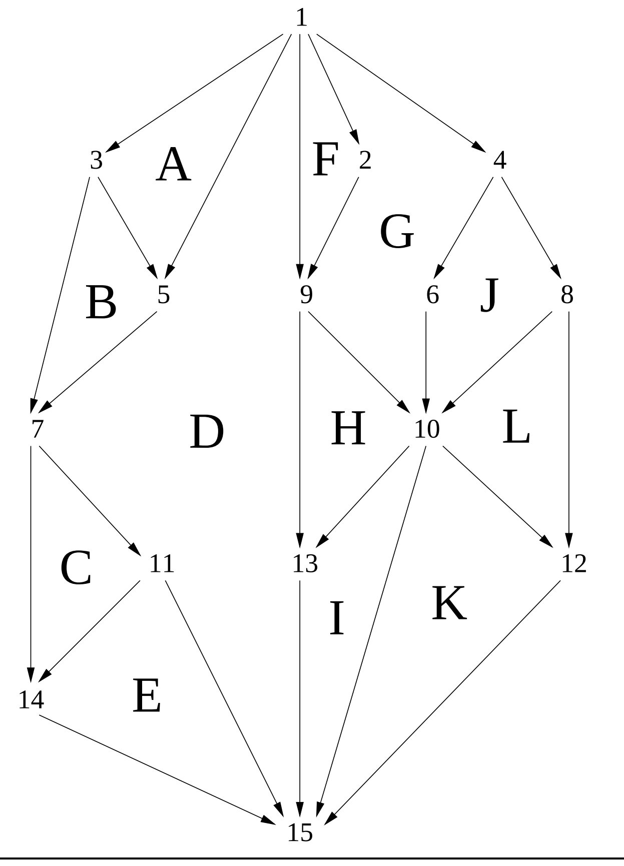
Mamy zatem udowodniony podstawowy fakt wiążący liczbę zjazdów z górnej stacji pokrywających cały graf z maksymalnym antyłańcuchem w grafie, w którym przecinki są połączone ze sobą, jeśli istnieje między nimi polana.

I tu niespodzianka! Okazuje się, że problem wyznaczenia liczności maksymalnego antyłańcucha w skończonym grafie opisującym porządek częściowy jest problemem trudnym obliczeniowo: konkretnie jest to problem NP-zupełny. Oznacza to, że nie są znane rozwiązujące go algorytmy o złożoności mniejszej niż wykładnicza. Gdybyśmy zatem nie mieli do czynienia z grafem narciarskim, a z dowolnym grafem częściowego porządku, to dla danych rozmiaru sięgającego już paruset wierzchołków bylibyśmy praktycznie bezradni, mogąc liczyć co najwyżej na szczęście, że uda nam się trafić rozwiązanie spośród wykładniczej liczby kandydatów, jakimi byłyby wszystkie możliwe antyłańcuchy maksymalne.

Dopiero tutaj z pomocą przychodzi nam planarność grafu narciarskiego. W grafach planarnych bowiem liczność maksymalnego łańcucha można wyznaczyć w koszcie liniowym ze względu na liczbę węzłów grafu. Oto algorytm.

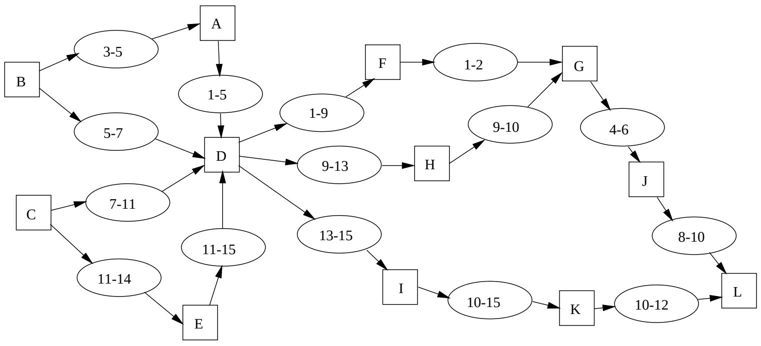
Najpierw zauważmy, że przecinki dzielą powierzchnię góry na obszary lasu, widoczne gołym okiem, gdy spojrzy się na rysunek. Są to te fragmenty lasu, między którymi istnieje połączenie nie wymagające przekroczenia żadnej trasy narciarskiej. W przykładowym rysunku są to na przykład obszary ograniczone kolejno przecinkami wiodącymi (czasem pod górę) między węzłami 1, 2, 9, 10, 6, 4, 1 lub 9, 10, 13, 9. Wewnątrz przykładowego grafu mamy ich 12. Takie obszary płaszczyzny dla grafu planarnego nazywa się jego ścianami. Zauważmy teraz, że każdy maksymalny antyłańcuch przecinek leśnych z grafu N^d składa się z takich przecinek, które stanowią krawędzie kolejnych, sąsiadujących ze sobą ścian. Gdyby bowiem gdzieś była przerwa, to znaczyłoby, że zapominając o jakiejś krawędzi dobrowolnie zrezygnowaliśmy z maksymalności antyłańcucha. Umówmy się, że będziemy zatem przecinali krawędzie z zachodu na wschód: każda spada choć trochę w dół, więc to pojęcie jest jednoznaczne. Łatwo można taki ciąg krawędzi znaleźć w naszym przykładowym grafie, np. kolejno krawędzie 1-3, 1-5, 9-13, 9-10, 6-10, 8-10, 8-12 wyznaczają maksymalny antyłańcuch, choć nie jest on najliczniejszy. Najdłuższy antyłańcuch będzie odpowiadał najdłuższej ścieżce w tym grafie, a jego długość będzie o 2 większa niż liczba krawędzi na tej ścieżce: do przecinanych w trakcie trawersowania krawędzi trzeba dorzucić dwie skrajne, stanowiące wejście do najbardziej zachodniej ściany w tej ścieżce i wyjście z najbardziej wschodniej. Te krawędzie też wchodzą do antyłańcucha.

Jak zrobić to automatycznie? Zacznijmy od skonstruowania grafu ścian S dla oryginalnego grafu narciarskiego N. Węzłami S będą ściany grafu N, a krawędzie będą łączyły te ściany, które są rozdzielone przecinką z zachodu na wschód. Na rysunku 2 przedstawiony jest graf ścian dla przykładu z treści zadania. Węzły grafu oznaczone są symbolami ścian, a krawędzie grafu poetykietowane są numerami węzłów będących końcami krawędzi oryginalnego grafu.



Rys. 1. Przykładowy graf z wpisanymi symbolami ścian.

Zauważmy, że w grafie tym mamy pewną dowolność: równie dobrze ściany F i G mogłaby rozdzielać krawędź $2-9$. Aby nie komplikować opisu wybraliśmy dowolną z krawędzi — na pewno spośród dwóch krawędzi rozdzielających dwie ściany tylko jedna może wejść do maksymalnego antyłańcucha i wybór, która to ma być, jest dowolny.



Rys. 2. Graf ścian dla stoku z treści zadania.

Wystarczy teraz znaleźć długość najdłuższej ścieżki w tym grafie i dodać do niej 2. W tym celu można wykorzystać na przykład następującą metodę: najpierw ściany posortować topologicznie, czyli ustawić je w tablicy jedna za drugą tak, aby zachować porządek wynikający z grafu ścian, a następnie przy jednokrotnym przejściu tablicy ścian aktualizować numery odpowiadające maksymalnej długości drogi od pewnej ściany początkowej. Dla naszego grafu maksymalne wartości będą odpowiadały czterom równie dobrym ścieżkom: BADFGJL, BADHGJL, CEDFGJL oraz CEDHGJL, każda o 6 krawędziach.

Zastanówmy się nad złożonością naszego algorytmu. Graf ścian można utworzyć w czasie liniowym ze względu na liczbę krawędzi grafu N. Sortowanie topogiczne też wymaga liniowego czasu. Ponieważ w grafach planarnych liczba krawędzi i ścian jest liniowa ze względu na liczbę wierzchołków (wynika to z twierdzenia Eulera), więc złożoność całego algorytmu jest liniowa w stosunku do liczby wierzchołków oryginalnego grafu N.

Można się oczywiście zżymać, że do rozwiązania tego zadania nie potrzeba wcale było przeprowadzać nieoczywistego przecież dowodu twierdzenia Dilwortha. „Wystarczyło” mieć odpowiednio dobrą intuicję i „wyczuć” je, a następnie zaprogramować opisany algorytm. To prawda, ale jakoś tak jest, że takie rzeczy znają, albo sami wymyślają, właśnie ci, którzy tego typu dowody są w stanie przeprowadzić.

A pomysł zadania przyszedł mi do głowy, gdy stałem przed schematem tras narciarskich pewnego atrakcyjnego ośrodka i zastanawiałem się, ile to razy będę musiał wjechać na górę, żeby poznać cały teren. Oczywiście jedna osoba musi wjechać tyle razy, ilu co najmniej objeżdżaczy musi wsiąść rano do kolejki. Faktem jest, że ośrodek ten zawierał nieco więcej niż jedną kolejkę, ale dałem spokój z dalszym utrudnianiem zadania. Jak się okazało, słusznie. W końcu jedynie trzy osoby uzyskały maksa.

Testy

Programy zawodników były oceniane za pomocą zestawu 10 testów wygenerowanych losowo przy różnych stopniach rozgałęzienia wierzchołków.

Krzysztof Onak — Treść zadania, Opracowanie
Marcin Mucha — Program

Waga

Mamy do dyspozycji wagę szalkową. Waga znajduje się w stanie równowagi wtedy i tylko wtedy, gdy obie szalki są puste lub gdy sumy mas odważników na obu szalkach są takie same. W danym zbiorze odważników należy znaleźć takie dwa rozłączne podzbiory, aby po położeniu wszystkich odważników z jednego z nich na jednej szalce, natomiast wszystkich odważników z drugiego zbioru na drugiej szalce, waga znalazła się w stanie równowagi. Ponadto spośród wszystkich układów odważników, dla których waga jest w równowadze, należy wybrać ten, który zawiera odważnik o maksymalnej możliwej masie. W przypadku wielu takich rozwiązań trzeba podać tylko jedno z nich.

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `wag.in` *opis dostępnych odważników,*
- *wyznaczy dwa rozłączne zbiory odważników spełniające warunki zadania,*
- *zapisze wynik do pliku tekstowego* `wag.out`*.*

Wejście

W pierwszym wierszu pliku tekstowego `wag.in` *znajduje się jedna liczba całkowita n, $2 \leqslant n \leqslant 1\,000$, równa liczbie dostępnych odważników. W każdym z następnych n wierszy znajduje się po jednej dodatniej liczbie całkowitej równej masie jednego odważnika ze zbioru dostępnych odważników. Można założyć, że łączna masa odważników nie przekracza $50\,000$.*

Wyjście

W pierwszym wierszu pliku wyjściowego `wag.out` *należy zapisać dwie nieujemne liczby całkowite p i q oddzielone pojedynczym odstępem i oznaczające, odpowiednio, liczbę odważników w pierwszym i w drugim zbiorze. W drugim wierszu powinno się znaleźć p liczb całkowitych oddzielonych pojedynczymi odstępami. Powinny to być masy odważników z pierwszego zbioru. Natomiast w trzecim wierszu powinno zostać wypisanych q liczb całkowitych oddzielonych pojedynczymi odstępami i równych masom odważników w drugim zbiorze.*

Przykład

Dla pliku wejściowego `wag.in`*:*

```
4
1
1
2
7
```

poprawną odpowiedzią jest plik wyjściowy `wag.out`*:*

```
2 1
1 1
2
```

Rozwiązanie

Ponumerujmy odważniki, którymi dysponujemy, od 1 do n. Niech $m_1, m_2, \ldots, m_n$ będą ich masami, a s niech będzie ich łączną masą. Oczywiście mówiąc dalej o masie zbioru odważników będziemy mieli na myśli sumę mas odważników należących do tego zbioru. Możemy przyjąć, że $m_1 \leqslant m_2 \leqslant \ldots \leqslant m_n$. Będziemy budowali strukturę danych, która będzie dawała nam szybko odpowiedź na pytanie zadane w treści zadania dla coraz większego zbioru odważników. Zaczniemy od zbioru pustego, a potem będziemy dokładali kolejno odważniki o masach $m_1, m_2, \ldots, m_n$, czyli w kolejności niemalejących mas.

Przypuśćmy, że dysponujemy tablicą $T[1..s]$. Jeśli pozycja pod indeksem k jest pusta (w programie może to być wyróżniona wartość), to oznacza, że nie ma podzbioru dotychczas rozważonego zbioru odważników o masie

k. W przeciwnym przypadku $T[k]$ mówi nam, że otrzymaliśmy podzbiór o masie k dokładając odważnik o masie $T[k]$ do wcześniejszego podzbioru odważników. Mając taką tablicę możemy łatwo i szybko dowiedzieć się, jakich odważników użyć, aby uzyskać podzbiór o masie k. Oto funkcja wydobywająca tę informację z tablicy T:

```
function podaj_listę_mas(T, k);
begin
  lista := pusta_lista;
  i := k;
  while i ≠ 0 do begin
    lista ⊕ T[i];
      { Operator ⊕ dorzuca na koniec listy podany element. }
    i := i − T[i];
  end;
  return lista;
end;
```

Jej poprawność wynika z prostej obserwacji, że jeśli za pomocą odważnika $T[k]$ otrzymaliśmy podzbiór o masie k, to wcześniej za pomocą innych odważników otrzymaliśmy podzbiór o masie $k-T[k]$, dla którego powtarzamy rozumowanie. Zatrzymujemy się dopiero, gdy zejdziemy do podzbioru o masie 0, czyli pustego. Zauważmy, że w naszym przypadku — dokładamy odważniki w niemalejącej wielkości mas — otrzymujemy z wykonania powyższej funkcji listę odważników o nierosnącej wielkości mas.

Rozszerzmy teraz tablicę T o dodatkową pozycję o indeksie 0. Tablicę T inicjujemy w ten sposób, że wszystkie pozycje o dodatnich indeksach są puste, a $T[0]$ zawiera cokolwiek (nie ważne co), ale nie jest puste. Oznacza to, że na początku jedyną możliwą do otrzymania masą jest 0. Przypuśćmy teraz, że rozpatrzyliśmy już jakiś podzbiór odważników i dokładamy nowy o masie m_i. Aktualizacja tablicy T może wyglądać następująco:

```
for j := s downto 0 do
  if not pusty(T[j]) then begin
    { Istniał podzbiór o masie j }
    { we wcześniejszym zbiorze odważników. }
    if pusty(T[j + m_i]) then begin
      { Istnieje podzbiór o nowej masie — aktualizacja T. }
      T[j + m_i] := m_i;
    end else begin
      { Otrzymaliśmy nowy podzbiór o masie j + m_i. }
      { Za chwilę to wykorzystamy. Oznaczmy to miejsce (∗). }
    end;
  end;
```

Jeśli znajdziemy się w $(*)$, to oznacza, że istnieją dwa rożne podzbiory dotychczas rozważonego zbioru odważników, które mają taką samą masę i jeden z nich zawiera odważnik o numerze i, a drugi nie. Ponadto jeśli istnieją takie zbiory, to trafimy do $(*)$. Możemy usunąć odważniki, które należą do obu zbiorów i otrzymać dwa rozłączne niepuste zbiory o tych samych masach.

Treścią zadania jest podanie spośród układów, w których waga znajduje się w równowadze takiego, w którym użyto odważnika o największej możliwej masie. Dokładając odważniki w niemalejącej wielkości mas możemy się dowiedzieć zatem, jaki jest ten odważnik o największej możliwej masie, gdyż jest to po prostu ostatni odważnik (przyjmijmy, że jego numer to p), dla którego znajdziemy się w punkcie $(*)$ w pewnym obrocie pętli podczas aktualizacji tablicy T. Możemy korzystając z wcześniejszej funkcji zdobyć listę mas odważników w obu zbiorach. Jeśli trafiliśmy do $(*)$, gdy $j = q$, to jeden zbiór zawiera odważniki o masach *podaj_listę_mas*$(T, q + m_p)$, a drugi odważnik o masie m_p i odważniki o masach *podaj_listę_mas*(T, q). Wartości p i q możemy zapamiętać, gdy znajdziemy się w $(*)$.

Zatem pozostaje usunąć powtórzenia, ale okazuje się, że można tego uniknąć. Wystarczy bowiem brać dla danego odważnika najmniejsze j, dla którego trafiamy do punktu $(*)$. Nie mamy wówczas powtórzeń, bo gdybyśmy mieli, to trafilibyśmy potem z jeszcze mniejszym j do $(*)$ rozważając ten sam odważnik (j byłoby wówczas zmniejszone o masę powtarzającego się odważnika). Wystarczy wstawić więc w miejsce $(*)$ następujący fragment:

```
p := i;
q := j;
```

a potem po prostu wypisać (bez usuwania czegokolwiek) masy odważników w obu zbiorach.

Implementacja

Program implementujemy zgodnie z powyższymi wskazówkami. Na początku sortujemy masy odważników. Możemy użyć nawet algorytmu o złożoności kwadratowej, bo nie zmieni to tym razem ogólnej złożoności programu. Potem dokładamy odważniki w niemalejącej kolejności, a na końcu wypisujemy masy zbiorów odważników odpowiadające sytuacji, w której po raz ostatni znaleźliśmy się w $(*)$. Musimy także uwzględnić sytuację, którą na razie pomijaliśmy milczeniem, a mianowicie, gdy ani razu nie trafimy do miejsca $(*)$. Oznacza to, że jedyny układ, w którym mamy stan równowagi na wadze, to puste obie szalki, a zatem puste zbiory odważników. Tak stanie się na przykład, gdy dysponujemy jedynie dwoma odważnikami o masach odpowiednio 1 i 2. W takim przypadku musimy wypisać odpowiednie rozwiązanie.

Dodatkowo możemy nieco przyśpieszyć działanie programu zauważając, że nie mają znaczenia pozycje w T znajdujące się pod indeksami większymi niż $\frac{s}{2}$.

Dostajemy ostatecznie program o złożoności czasowej $O(ns)$ i pamięciowej $O(s)$. Takie rozwiązanie było wymagane od uczestników.

Dalsze udoskonalenia

Okazuje się możemy udoskonalić nasz algorytm uzyskując czas działania rzędu $O(\min(n, \sqrt{s})s)$. Niech m będzie liczbą różnych mas odważników. Tym razem będziemy aktualizowali tablicę T w co najwyżej m przebiegach, a nie n jak to miało miejsce poprzednio. Najwięcej zyskamy oczywiście, gdy m będzie znacząco mniejsze od n.

Zauważmy najpierw, że jedyny interesujący z punktu widzenia zadania układ, w którym na jednej i drugiej szalce znajduje się odważnik o tej samej masie to taki, w którym oba odważniki są największymi użytymi, bo w przeciwnym przypadku moglibyśmy usunąć je z szalek nic z punktu widzenia zadania nie tracąc. Ponadto jeśli już są największe, to wystarczy rozpatrzyć jedynie układ, w którym tylko one znajdują się na szalkach wagi.

Przejdźmy już teraz do naszego algorytmu. Tym razem na początku musimy stwierdzić, ile mamy odważników danego rodzaju. Potem rozpatrujemy najpierw pary odważników o tej samej masie, jeśli dysponujemy takimi. Zapamiętujemy najlepsze uzyskane w ten sposób rozwiązanie.

Teraz wystarczy zgodnie ze spostrzeżeniem rozpatrzyć jedynie układy, w których nie występują odważniki o takich samych masach na obu szalkach. Będziemy teraz w jednym przebiegu aktualizowali tablicę T o użycie wszystkich odważników o danej masie. Stworzymy w tym celu dodatkową tablicę V o takich samych wymiarach, jak T. Będzie nam ona mówiła, ile wykorzystaliśmy odważników o przetwarzanej aktualnie masie, aby uzyskać podzbiór o danej, nowej masie. Oto gotowy algorytm:

```
zainicjujT(T); { Inicjacja taka jak poprzednio. }
najodważnik := 0;
najmasa := 0;
for dostępne (masa,ile), gdzie masa w rosnącej kolejności do begin
  for i := 0 to s do
    V[i] := 0;
  for i := 0 to s do
    if not puste(T[i]) and V[i] ≠ ile then begin
      if puste(T[i + masa]) then begin
        { Istnieje podzbiór o nowej masie — aktualizacja T. }
        T[i + masa] := masa;
        V[i + masa] := V[i] + 1;
      end else begin
        { Otrzymaliśmy nowy podzbiór o masie i + masa. }
        if najodważnik < masa then begin
          najodważnik := masa;
          najmasa := i + masa;
        end;
      end;
    end;
end;
```

Otrzymujemy rozwiązanie, w którym masy odważników z jednego zbioru to *podaj_listę_mas*(T, *najmasa*), a z drugiego to *najodważnik* i *podaj_listę_mas*(T, *najmasa* − *najodważnik*), o ile *najodważnik* $\neq 0$ — bo wówczas znaleźliśmy jakieś rozwiązania. Musimy porównać to rozwiązanie (bez odważników o tych samych masach po obu stronach) z rozpatrzonym wcześniej przypadkiem (o ile też coś znaleźliśmy), gdy biorąc dwa odważniki o tych samych masach stawialiśmy je na przeciwnych szalkach, i wybrać to lepsze, czyli to, w którym użyto większego maksymalnego odważnika.

Podczas implementacji możemy, tak jak poprzednio, zauważyć, że część tablicy T (a zatem i V) o indeksach większych od $\frac{s}{2}$ nie wnosi nic do rozwiązania zadania.

Pozostaje oszacować złożoność tego rozwiązania. Wymagania pamięciowe są nieco większe, ale nadal jest to $O(s)$. Złożoność czasowa, którą uzyskujemy, to $O(ms)$. Wiemy, że $m \leqslant n$. Dla danego s spróbujmy oszacować maksymalną możliwą wartość m. Musimy brać odważniki o jak najmniejszych masach, a zatem $1, 2, \ldots$, dopóki nie przekroczymy s. Mamy zatem

$$1 + 2 + \cdots + m \leqslant s,$$

i ze wzoru na sumę wyrazów ciągu arytmetycznego

$$\frac{m(m+1)}{2} \leqslant s.$$

Przekształcamy dalej:

$$m^2 + m \leqslant 2s,$$
$$m^2 \leqslant 2s,$$
$$m \leqslant \sqrt{2s}.$$

A zatem m jest rzędu $O(\min(n, \sqrt{s}))$ i ostatecznie możemy napisać, że złożoność czasowa algorytmu wynosi $O(\min(n, \sqrt{s})s)$.

Testy

Programy zawodników były oceniane za pomocą zestawu 16 testów. Większość z nich została dobrana w taki sposób, aby nie istniało proste rozwiązanie.

- `wag1.in` — $n = 13$
- `wag2.in` — $n = 50$
- `wag3.in` — $n = 99$
- `wag4.in` — $n = 199$
- `wag5.in` — $n = 399$
- `wag6.in` — $n = 799$
- `wag7.in` — $n = 1000$
- `wag8.in` — $n = 1000$
- `wag9.in` — $n = 1000$
- `wag10.in` — $n = 1000$
- `wag11.in` — $n = 200$
- `wag12.in` — $n = 500$
- `wag13.in` — $n = 1000$
- `wag14.in` — $n = 200$
- `wag15.in` — $n = 500$
- `wag16.in` — $n = 1000$

Wojciech Guzicki
Treść zadania

Wojciech Guzicki
Przemysława Kanarek
Opracowanie

Marcin Stefaniak
Program

Liczby B-gładkie

Niech B będzie dodatnią liczbą całkowitą. Liczbę naturalną n nazwiemy B-gładką, jeśli w jej rozkładzie na czynniki pierwsze nie występują liczby pierwsze większe od B. Równoważnie możemy powiedzieć, że liczbę n nazywamy B-gładką, gdy można przedstawić ją jako iloczyn liczb dodatnich całkowitych mniejszych bądź równych B.

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `lic.in` *trzy dodatnie liczby całkowite n, m oraz B,*
- *wyznaczy liczbę wszystkich liczb B-gładkich w przedziale $[n, n+m]$ (włącznie),*
- *zapisze wynik w pliku tekstowym* `lic.out`.

Wejście

W pierwszym wierszu pliku tekstowego zapisano trzy liczby całkowite n, m i B, pooddzielane pojedynczymi odstępami, $1 \leqslant n \leqslant 2\,000\,000\,000$, $1 \leqslant m \leqslant 100\,000\,000$, $1 \leqslant B \leqslant 1\,000\,000$.

Wyjście

Twój program powinien zapisać w pierwszym wierszu pliku tekstowego `lic.out` *jedną liczbę całkowitą — wyznaczoną liczbę liczb B-gładkich.*

Przykład

Dla pliku wejściowego `lic.in`*:*
```
30 10 5
```
poprawną odpowiedzią jest plik wyjściowy `lic.out`*:*
```
4
```

Rozwiązanie (Wojciech Guzicki)

Narzucającą się metodą rozwiązania jest przeglądanie kolejno wszystkich liczb z rozważanego przedziału i dzielenie ich przez wszystkie liczby nie większe od B. Jeśli po tych wszystkich dzieleniach otrzymamy 1, to liczba, którą dzieliliśmy, jest B-gładka. A oto odpowiednia procedura:

```
licznik := 0;
for k := n to n + m do begin
  l := k;
  for d := 2 to B do
    while l mod d = 0 do l := l div d;
  if l = 1 then licznik := licznik + 1;
end;
```

Po wykonaniu tego prostego algorytmu zmienna *licznik* podaje szukaną liczbę liczb B-gładkich należących do przedziału $[n, n+m]$.

Algorytm ten można znacznie przyspieszyć, jeśli będziemy dzielić nie przez wszystkie liczby z przedziału $[2, B]$, ale tylko przez liczby pierwsze z tego przedziału. W tym celu należy na początku programu stworzyć tablicę wszystkich liczb pierwszych mniejszych bądź równych B (np. za pomocą sita Eratostenesa). Raz utworzoną tablicę liczb pierwszych mniejszych od 1000000 można zapamiętać i wykorzystać następnie w każdym wywołaniu programu. Warto wspomnieć tu, że liczb pierwszych mniejszych od 1000000 jest 78498.

Mimo tego przyspieszenia, algorytm jest nadal bardzo nieefektywny — próbujemy w nim wykonać mnóstwo niepotrzebnych dzieleń. Lepsza jest metoda *sita* – nazwę usprawiedliwia podobieństwo do sita Eratostenesa.

Zapisujemy liczby z przedziału $[n, n+m]$ w tablicy. Wiemy, że przez 2 może się dzielić tylko co druga liczba, przez 3 co trzecia, przez 5 co piąta itd. Będziemy dzielić tylko te liczby.

Tworzymy tablicę długości $m+1$ i zapisujemy w niej wszystkie liczby z przedziału $[n, n+m]$. W poniższym programie jest to tablica *Przedział* indeksowana liczbami od 0 do m. Następnie dla każdej liczby pierwszej p (w programie jest to liczba *Pierwsze*$[k]$) nie większej od B znajdujemy pierwszą liczbę w tej tablicy podzielną przez p. W tym celu najpierw wyznaczamy resztę r z dzielenia n przez p (w poniższym programie jest to zmienna *reszta*). Jeśli $r = 0$, to tą liczbą jest n (ma ona w naszej tablicy indeks 0). Jeśli zaś $r \neq 0$, to tą liczbą jest $n + (p - r)$; ma ona w naszej tablicy indeks $p - r$. Znalezioną liczbę dzielimy przez p tak długo, jak tylko jest to możliwe. Następnie zwiększamy indeks o p: tylko co p-tą liczbę warto dzielić przez p. A oto program realizujący przesiewanie przez sito:

```
for i := 0 to m do Przedział[i] := n + i;
k := 1;
p := Pierwsze[k];
while p ⩽ B do begin
  reszta := n mod p;
  if reszta = 0 then indeks := 0 else indeks := p − reszta;
  while indeks ⩽ m do begin
    repeat
      Przedział[indeks] := Przedział[indeks] div p;
    until Przedział[indeks] mod p ≠ 0;
    indeks := indeks + p;
  end;
  k := k + 1;
  p := Pierwsze[k];
end;
```

Teraz wystarczy policzyć, na ilu miejscach w naszej tablicy znajduje się liczba 1. Ten program działa znacznie szybciej od poprzedniego, głównie dzięki temu, że nie wykonujemy wielu niepotrzebnych prób dzielenia przez kolejne liczby pierwsze. Jest to szczególnie widoczne dla bardzo dużych liczb n, gdy sama procedura dzielenia wymaga znacznie dłuższego czasu niż w przypadku liczb małych.

Algorytm oparty na metodzie sita można jeszcze nieco przyspieszyć. Po pierwsze, zauważmy, że w czasie dzielenia przez daną liczbę pierwszą p za każdym razem dokonywaliśmy sprawdzenia, czy można wykonać następne dzielenie. Tego sprawdzenia można uniknąć. Popatrzmy na przykład. Co drugą liczbę dzielimy przez 2 (tylko raz). Następnie co czwartą dzielimy jeszcze raz przez 2: w ten sposób co czwarta będzie podzielona przez 4. Teraz co ósmą jeszcze raz dzielimy przez 2: łącznie te liczby będą podzielone przez 8. I tak dalej. Oczywiście za każdym razem trzeba znaleźć najmniejszą liczbę, od której zaczniemy kolejne dzielenia. W podobny sposób można postąpić dla innych liczb pierwszych.

Jeśli mamy do czynienia z bardzo dużymi liczbami, np. mającymi ponad 100 cyfr, to możemy spróbować jeszcze jednego sposobu przyspieszenia algorytmu. Mianowicie bardzo wiele dzieleń i tak wykonamy niepotrzebnie: po wszystkich wykonanych dzieleniach przekonamy się, że dana liczba nie jest B-gładka. A wiemy już, że dzielnie bardzo dużych liczb trwa długo. Możemy obliczyć przybliżoną wartość logarytmu każdej z tych liczb (wystarczy dokładność kilkunastu cyfr po przecinku, dostępna w standardowych typach rzeczywistych) oraz logarytmy liczb pierwszych nie większych od B. Następnie, zamiast dzielić przez daną liczbę pierwszą p, odejmujemy jej logarytm. Odejmowanie liczb rzeczywistych jest znacznie szybsze od dzielenia dużych liczb całkowitych. Po wykonaniu wszystkich odejmowań patrzymy na liczby, których logarytm niewiele różni się od zera (przypominamy: $\log 1 = 0$). Nie możemy oczekiwać, że otrzymamy dokładnie 0, ze względu na popełniane błędy zaokrągleń. Jednak tylko stosunkowo niewiele obliczonych logarytmów będzie bliskich 0 i wtedy te liczby naprawdę dzielimy przez liczby pierwsze mniejsze od B. W ten sposób wiele niepotrzebnych dzieleń zastąpimy znacznie szybszymi odejmowaniami i wykonamy tylko niezbędne dzielenia. Oczywiście ta metoda jest nieprzydatna w przypadku liczb tak małych jak w zadaniu.

Algorytm pierwszy oraz algorytm oparty na sicie mają tę zaletę, że dają w wyniku nie tylko liczbę liczb B-gładkich, ale także pozwalają znaleźć te liczby. Wyznaczanie liczb B-gładkich pewnych szczególnych postaci jest najbardziej czasochłonnym krokiem w nowoczesnych algorytmach rozkładu dużych liczb na czynniki pierwsze. Metoda sita znajduje w tych algorytmach ważne zastosowanie.

Dla małych liczb n możemy podać jeszcze jeden algorytm rozwiązania zadania, tym razem oparty na rekurencji. Oznaczmy symbolem $C(n, B)$ liczbę liczb B-gładkich w przedziale $[1, n]$. Liczbę l liczb B-gładkich w przedziale $[n, n+m]$ wyznaczymy wtedy ze wzoru

$$l = C(n+m, B) - C(n-1, B).$$

W jaki sposób można obliczyć $C(n,B)$? Najprościej przez równanie rekurencyjne:

$$C(n,p_k) = C(\lfloor n/p_k \rfloor, p_k) + C(n, p_{k-1}),$$

gdzie p_k oznacza k-tą liczbę pierwszą. Mianowicie zbiór liczb p_k-gładkich rozbijamy na dwa podzbiory:

- liczby niepodzielne przez p_k — jest ich $C(n, p_{k-1})$;
- liczby podzielne przez p_k — jest ich tyle samo, co liczb p_k-gładkich w przedziale $[1, \lfloor n/p_k \rfloor]$.

Na podstawie tych równań rekurencyjnych można już napisać program. Nie będzie on jednak działał szybko i trzeba wprowadzić szereg usprawnień. Po pierwsze, można zapamiętać w tablicy wartości $C(n,B)$ dla małych n i B. Zamiast obliczać wielokrotnie te same wartości, program weźmie je z tablicy. Po drugie, można częściowo rozwikłać równanie rekurencyjne i te prostsze przypadki wpisać do programu. Na przykład mamy:

$$C(n,p) = n \quad \text{dla } n \leqslant p$$

oraz

$$C(n,2) = \lfloor \log_2 n + 1 \rfloor.$$

Podobnych usprawnień można znaleźć więcej, niektóre z nich zostały wpisane do programu wzorcowego.

Mamy zatem dwa algorytmy; każdy z nich działa szybko dla różnych wartości n i m. Dla małych n i dużych m szybciej działa algorytm rekurencyjny, dla dużych n i małych m szybsze jest sito. Na podstawie testów przeprowadzonych dla różnych zestawów danych (w zakresie zgodnym z warunkami zadania), w programie wzorcowym przyjęto następujące ograniczenie: jeśli $n > 800m$, to stosujemy metodę sita, w przeciwnym przypadku stosujemy algorytm rekurencyjny.

Testy

Rozwiązania zawodników były testowane za pomocą 27 testów. Można je podzielić na następujące grupy:

- testy poprawnościowe (testy 1–18);
- testy wydajnościowe (19–20);
- testy dla odróżnienia metody sita (21–27).

W poniższej tabeli znajdują się opisy testów:

nr	n	m	B	wynik
1	1000	100	10	6
2	1234	567	89	261
3	9876	5432	1	0
4	10987	6543	21	315
5	13579	23456	789	12522
6	1	1	1	1
7	10000	10000	9000	8856
8	19	1	10	1
9	20	1	10	2
10	235421	23415	10000	15364
11	2345678	23471	10000	11667
12	23423456	23434	10000	8371
13	234322342	23434	10000	5584
14	1342342342	34232	10000	5904
15	23423434	100000	10000	35614
16	1345344567	100000	10000	17183
17	234234342	23423	100000	10667
18	1342342343	23423	100000	8680
19	234324233	100000	100000	45544
20	1342342323	100000	100000	37121
21	2000000000	1000000	100000	352858
22	20000100	10000000	100100	5753651
23	200000000	10000100	100000	4621219
24	2000001000	10000000	100000	3527205
25	1000000000	10000000	1000000	5666589
26	200000000	10001000	1000000	6442366
27	200000000	30000000	100100	13794053

Dodatek: Rozwiązanie oparte na rekurencji (Przemysława Kanarek)

Oznaczmy przez $C(n,B)$ liczbę liczb B-gładkich w przedziale $[1,n]$. Choć właściwe zadanie polega na wyliczeniu, ile jest liczb B-gładkich w przedziale $[n,n+m]$, dla m zdecydowanie mniejszego niż n, rozwiążemy nieco ogólniejszy problem. Pokażemy, jak obliczyć $C(n,B)$, czyli odpowiemy na pytanie, ile jest liczb B-gładkich w przedziale $[1,n]$. Potem, by rozwiązać początkowe zadanie, obliczymy $C(n+m,B)-C(n-1,B)$.

Nasze rozwiązanie będzie wykorzystywać kilka spostrzeżeń dotyczących liczb B-gładkich, które pozwalają niewielkim kosztem redukować problem obliczania $C(n,B)$ do problemów obliczania $C(n',B')$ dla mniejszych argumentów, tzn. $n'<n$ i/lub $B'<B$ (spostrzeżenia 1–4). Redukcję będziemy przeprowadzać tak długo, aż dojdziemy do przypadku (n,B), w którym n i/lub B są na tyle małe, że będziemy mogli wartość $C(n,B)$ wyliczyć bezpośrednio ze wzoru (spostrzeżenia 5 i 5' dla $B\leqslant 3$) lub skorzystać ze stablicowanych wartości $C(n,B)$, które wyliczymy wcześniej i umieścimy w tablicy D (spostrzeżenie 6).

Na początku obliczmy i umieśćmy w tablicy $P[1,r]$ w porządku rosnącym wszystkie liczby pierwsze nie większe niż maksymalna możliwa wartość B, tak więc $P[i]$ jest i-tą liczbą pierwszą. Dla liczby naturalnej x będziemy mówić, że liczba $P[i]$ jest ***bliska z lewej*** x (zapiszemy to $P[i]\hookrightarrow x$), jeżeli $P[i]$ jest największą liczbą pierwszą nie większą niż x.

Spostrzeżenie 1 Dla $B\geqslant n$ zachodzi równość $C(n,B)=n$.
Uzasadnienie Uwaga ta jest oczywista i podajemy ją jedynie po to, by mieć „komplet” stosowanych spostrzeżeń i redukcji. $\square$

Spostrzeżenie 2 Dla $B>\lfloor\frac{n}{2}\rfloor$ zachodzi równość $C(n,B)=C(n,\lfloor\frac{n}{2}\rfloor)+|\{p : p \text{ jest liczbą pierwszą i } \frac{n}{2}<p\leqslant B\}|$.
Uzasadnienie Aby wykazać powyższą równość rozważymy oddzielnie liczby B-gładkie mające duży czynnik pierwszy — większy niż $\frac{n}{2}$ — i oddzielnie liczby B-gładkie, których wszystkie czynniki pierwsze są mniejsze lub równe $\frac{n}{2}$ (są to w rzeczywistości liczby $\frac{n}{2}$-gładkie). Łatwo zauważyć, że liczby z pierwszej grupy są dość szczególne — muszą to być liczby pierwsze. Wynika to z faktu, że liczba z przedziału $[1,n]$ może mieć w rozkładzie na czynniki pierwsze liczbę większą niż $n/2$ tylko wówczas, gdy jest sama liczbą pierwszą (i jednocześnie swoim jedynym dzielnikiem pierwszym). Gdyby bowiem była liczbą złożoną, to jej drugi podzielnik musiałby być równy co najmniej 2 i cała liczba byłaby tym samym większa niż n.

Obliczenie $C(n,B)$ możemy więc sprowadzić do obliczenia $C(n,\lfloor\frac{n}{2}\rfloor)$, jeżeli uda nam się dowiedzieć, ile jest liczb w zbiorze $\{p : p \text{ jest liczbą pierwszą i } \frac{n}{2}<p\leqslant B\}$. Ale to możemy łatwo sprawdzić znajdując poprzez wyszukiwanie binarne w tablicy P liczby pierwsze bliskie z lewej $\lfloor\frac{n}{2}\rfloor$ i B. Jeżeli będą to odpowiednio $P[i]$ i $P[j]$, to $C(n,B)=C(n,\lfloor\frac{n}{2}\rfloor)+j-i$. $\square$

Spostrzeżenie 3 Dla $B>\sqrt{n}$ zachodzi równość $C(n,B)=C(n,\lfloor\sqrt{n}\rfloor)+\sum_p\lfloor\frac{n}{p}\rfloor$, gdzie sumowanie przebiega po wszystkich liczbach pierwszych p należących do przedziału $[\lfloor\sqrt{n}\rfloor+1..B]$.
Uzasadnienie W tym spostrzeżeniu oddzielnie liczymy liczby B-gładkie mające czynnik pierwszy większy niż $\sqrt{n}$ i oddzielnie takie liczby, których wszystkie czynniki pierwsze są mniejsze lub równe $\sqrt{n}$ (czyli liczby $\sqrt{n}$-gładkie). Aby obliczyć, ile jest liczb w pierwszej grupie zauważmy, że w rozkładzie na czynniki pierwsze dowolnej liczby $x\in[1,n]$ może wystąpić tylko jedna liczba pierwsza p z przedziału $[\lfloor\sqrt{n}\rfloor+1,B]$ (dwa takie czynniki sprawiłyby, że $x>n$). Co więcej, liczb podzielnych przez p jest w przedziale $[1,n]$ dokładnie $\lfloor\frac{n}{p}\rfloor$ i wszystkie one są B-gładkie, bo ich największym dzielnikiem pierwszym jest p, a $p\leqslant B$. $\square$

Spostrzeżenie 4 Dla dowolnych n i B zachodzi równość $C(n,B)=C(n,p')+C(\lfloor\frac{n}{p}\rfloor,p)$, gdzie p' i p są *kolejnymi* liczbami pierwszymi ($p'<p$) i p jest bliska z lewej B, to znaczy $p'\hookrightarrow p-1$ i $p\hookrightarrow B$.
Uzasadnienie Podobnie jak w poprzednich spostrzeżeniach, tu również rozważymy oddzielnie dwie grupy liczb B-gładkich. Pierwsza grupa to liczby B-gładkie podzielne przez p. Druga grupa, to liczby B-gładkie nie mające dzielnika p, a więc mające wszystkie dzielniki pierwsze mniejsze od p — są to liczby p'-gładkie.

Aby policzyć, ile jest liczb w pierwszej grupie rozważmy wszystkie liczby z przedziału $[1,n]$ podzielne przez p, czyli $1\cdot p,\ 2\cdot p,\ \ldots,\ \lfloor\frac{n}{p}\rfloor\cdot p$. Niech $x=d\cdot p$ będzie jedną z tych liczb. Zauważmy, że x jest p-gładka tylko wówczas, gdy czynnik d jest p-gładki. Tak więc, by wybrać liczby p-gładkie podzielne przez p, wystarczy spośród wymienionych liczb wybrać te, w których czynniki d są p-gładkie, tzn. wybrać liczby p-gładkie ze zbioru $\{1,2,3,\ldots,\lfloor n/p\rfloor\}$. Takich liczb jest $C(\lfloor\frac{n}{p}\rfloor,p)$.

W drugiej grupie są liczby p-gładkie o czynnikach pierwszych mniejszych niż p, czyli liczby p'-gładkie. Tych liczb jest $C(n,p')$.

Przedstawiona w tym spostrzeżeniu zależność jest najbardziej skomplikowana obliczeniowo spośród dotychczas opisanych. Jej wyliczenie wymaga dwóch wywołań rekurencyjnych, a wiadomo (?) jaki to może mieć wpływ na czas obliczeń. Oczywiście stosujemy ją tylko wówczas, gdy B jest na tyle małe w stosunku do n, że nie możemy zastosować żadnej z wcześniej opisanych redukcji. Szczęśliwie, często okazuje się, że po zastosowaniu nawet jednego kroku tej redukcji otrzymujemy przypadki, które poddają się pozostałym redukcjom i pozwalają dalej zmniejszyć rozmiar problemu bez zbytniego nakładu pracy. $\square$

Kolejne spostrzeżenia dotyczą przypadków, gdy n lub B są wystarczająco małe, by szybko wyliczyć wartość $C(n,B)$.

Spostrzeżenie 5 $C(n,2) = \lfloor \log_2 n + 1 \rfloor$.
Uzasadnienie Wynika to z faktu, że liczby w przedziale $[1,n]$ dające się przedstawić jako iloczyn 2 (czyli 2-gładkie) to tylko to tylko 1, 2, $4 = 2^2$, $8 = 2^3$, ..., $2^{\log_2(n)}$. □

Można również wyprowadzić bezpośredni wzór na liczbę liczb 3-gładkich:

Spostrzeżenie 5' $C(n,3) = \sum_{i=0}^{\lfloor \log_3 n \rfloor} \lfloor \log_2 (n/3^i) + 1 \rfloor$.
Uzasadnienie Wzór ten otrzymujemy zauważając, że liczb 3-gładkich niepodzielnych przez 3 (czyli 2-gładkich) jest dokładnie $\lfloor \log_2 n + 1 \rfloor$ (jest to składnik sumy dla $i = 0$). Następnie widzimy, że liczby 3-gładkie podzielne przez 3 dokładnie raz (tzn. niepodzielne przez 9), to 3, $3 \cdot 2$, $3 \cdot 2^2$, $3 \cdot 2^3$, ... Ostatnia liczba w tym ciągu nie większa niż n to $3 \cdot 2^{\lfloor \log_2 (n/3) \rfloor}$, a więc liczb tych jest $\lfloor \log_2 (n/3) + 1 \rfloor$ (jest to składnik sumy dla $i = 1$). Dalej zauważamy, że kolejne składniki sumy odpowiadają liczbom 3-gładkim podzielnym dokładnie przez 3^2, 3^3, itd.
□

Podobne wzory można wyprowadzać dla kolejnych liczb pierwszych: 5, 7, 11, ... Jednak trzeba zdać sobie sprawę, że stopień ich skomplikowania rośnie bardzo szybko i tym samym coraz trudniej się takie wartości wylicza. W praktyce proponujemy więc poprzestać na bezpośrednim wyliczaniu $C(n,B)$ ze wzoru dla $B \leqslant 3$.

Spostrzeżenie 6 W naszym zadaniu warto wstępnie wyliczyć i zapamiętać w tablicy wartości $C(n,P[i])$ dla odpowiednio małych n i $P[i]$. Jednak zapisanie tych wartości w tablicy kwadratowej nie byłoby ekonomicznym rozwiązaniem, ponieważ dla małych n mamy niewiele liczb pierwszych $P[i]$ mniejszych od n (tylko dla takich n warto zapamiętywać $C(n,P[i])$), a dla większych n takich liczb jest znacznie więcej. Powinniśmy więc zastosować tablicę jednowymiarową, nazwijmy ją D, zawierającą zapisane jeden za drugim „wiersze" odpowiadające kolejnym wartościom n. Dodatkowa tablica J powinna pomagać nam w stwierdzeniu od jakiej pozycji w D występują wartości C dla ustalonego n (czyli, gdzie w D jest „wiersz" odpowiadający n), tzn. wartości $C(n,3)$, $C(n,5)$, ..., $C(n,P[i_n])$ występowałyby na pozycjach $D[J[n]..J[n+1]-1]$. □

Wykorzystując wszystkie powyższe spostrzeżenia możemy napisać program efektywnie obliczający wartość $C(n,B)$ dla wystarczająco dużych (tak dużych, jak jest to wymagane w zadaniu) wartości n i B. Schemat tego programu jest prosty. Dla zadanych n i B należy jedynie sprawdzić, którą z redukcji należy zastosować (oczywiście w pierwszej kolejności próbujemy zastosować spostrzeżenia kończące obliczenia, tzn. 5, 5' lub 6, a dopiero potem sprawdzamy, czy można zastosować redukcję 1, 2, 3 czy 4) i rekurencyjnie rozwiązujemy zadanie dla mniejszych n i B.

Oszacowanie złożoności przedstawionego rozwiązania jest trudne. Trudno przewidzieć, ile kroków redukcji wynikających ze spostrzeżenia 4 trzeba będzie wykonać dla zadanego n, a to one głównie decydują o czasie obliczeń. Przyjmijmy więc empiryczne oszacowanie czasu działania algorytmu — program wykorzystujący wszystkie powyższe spostrzeżenia działa w rozsądnym czasie kilku sekund dla maksymalnych wartości n i B.

Piotr Chrząstowski
Wojciech Guzicki
Treść zadania

Piotr Chrząstowski
Opracowanie

Remigiusz Różycki
Program

Nawiasy

Działanie odejmowania nie jest łączne, np. $(5-2)-1=2$*, natomiast* $5-(2-1)=4$*, a zatem* $(5-2)-1 \neq 5-(2-1)$*. Wynika stąd, że wartość wyrażenia postaci* $5-2-1$ *zależy od kolejności wykonywania odejmowań. Zwykle przy braku nawiasów przyjmuje się, że wykonujemy działania w kolejności od lewej do prawej, czyli wyrażenie* $5-2-1$ *oznacza* $((5-2)-1)$*.*

Mamy dane wyrażenie postaci

$$x_1 \pm x_2 \pm \cdots \pm x_n,$$

gdzie $\pm$ *oznacza* + *(plus) lub* − *(minus), a* $x_1, x_2, \ldots, x_n$ *oznaczają (parami) różne zmienne. Chcemy w wyrażeniu postaci*

$$x_1 - x_2 - \cdots - x_n$$

tak rozstawić $n-1$ *par nawiasów, aby jednoznacznie określić kolejność wykonywania odejmowań i jednocześnie uzyskać wyrażenie równoważne danemu. Na przykład chcąc uzyskać wyrażenie równoważne wyrażeniu:*

$$x_1 - x_2 - x_3 + x_4 + x_5 - x_6 + x_7$$

możemy w:

$$x_1 - x_2 - x_3 - x_4 - x_5 - x_6 - x_7$$

rozmieścić nawiasy w następujący sposób:

$$(((x_1 - x_2) - ((x_3 - x_4) - x_5)) - (x_6 - x_7))$$

Uwaga: *Interesują nas tylko wyrażenia w pełni i poprawnie ponawiasowane. Wyrażeniem w pełni i poprawnie ponawiasowanym jest*

- *albo pojedyncza zmienna,*
- *albo wyrażenie postaci* $(w_1 - w_2)$*, w którym* w_1 *i* w_2*, to w pełni i poprawnie ponawiasowane wyrażenia.*

Nieformalnie mówiąc, nie interesują nas wyrażenia, w których występują na przykład konstrukcje postaci: $()$*,* (x_i) *lub* $((\ldots))$*. Nie jest w pełni ponawiasowane wyrażenie* $x_1 - (x_2 - x_3)$*, ponieważ brakuje w nim zewnętrznych nawiasów.*

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `naw.in` *opis danego wyrażenia postaci* $x_1 \pm x_2 \pm \cdots \pm x_n$*,*
- *obliczy na ile różnych sposobów (modulo 1 000 000 000) można rozstawić* $n-1$ *par nawiasów w wyrażeniu* $x_1 - x_2 - \cdots - x_n$*, tak aby jednoznacznie określić kolejność wykonywania odejmowań i jednocześnie uzyskać wyrażenie równoważne danemu,*
- *zapisze wynik w pliku tekstowym* `naw.out`*.*

Wejście

W pierwszym wierszu pliku wejściowego `naw.in` *zapisana jest jedna liczba całkowita* n*,* $2 \leqslant n \leqslant 5000$*. Jest to liczba zmiennych w danym wyrażeniu. W każdym z kolejnych* $n-1$ *wierszy jest zapisany jeden znak,* `+` *lub* `-`*. W* i*-tym wierszu zapisany jest znak występujący w danym wyrażeniu między* x_{i-1} *a* x_i*.*

Wyjście

Twój program powinien zapisać w pierwszym wierszu pliku wyjściowego `naw.out` *jedną liczbę całkowitą równą liczbie różnych sposobów (modulo 1 000 000 000), na jakie można powstawiać* $n-1$ *par nawiasów do wyrażenia* $x_1 - x_2 - \cdots - x_n$ *tak, aby jednoznacznie określić kolejność wykonywania odejmowań i jednocześnie uzyskać wyrażenie równoważne danemu.*

Przykład

Dla pliku wejściowego `naw.in`*:*
```
7
-
-
+
+
-
+
```
poprawną odpowiedzią jest plik wyjściowy `naw.out`*:*
```
3
```

Omówienie zadania

Zadanie „Nawiasy" było pozornie bardzo podobne do zadania „Minusy" z sesji próbnej trzeciego etapu. Tyle tylko, że zadanie „Minusy" zostało zrobione na 100 punktów przez prawie wszystkich uczestników, a „Nawiasów" nie udało się „na maksa" rozwiązać żadnemu. Różnica polegała na tym, że o ile w „Minusach" wystarczyło podać *przykładowe* ustawienie nawiasów, o tyle w przypadku „Nawiasów" należało zliczyć liczbę takich ustawień. Ponieważ liczba ta dla dużych danych wychodziła bardzo duża, więc i obliczenie jej sprawiało uczestnikom sporo trudności.

Jak mogliśmy się przekonać w rozdziale „Minusy", liczba wszystkich możliwych ustawień nawiasów dla n zmiennych jest równa $(n-1)$-szej liczbie Catalana. Z kolei liczby Catalana są duże. Ponieważ $C_k = \frac{1}{k+1}\binom{2k}{k}$, co jest w przybliżeniu[1] równe 4^k, więc widać że nawet gdyby ustawienia nawiasów rozłożyły się równomiernie, to i tak na jedną daną wejściową przypadałoby średnio ponad 2^n ustawień nawiasów. Wykładniczy charakter tej zależności powodował, że zliczanie wszystkich rozwiązań pojedynczo skazane było na niepowodzenie. W takich przypadkach warto jest zastosować zliczanie skumulowane, charakterystyczne dla techniki programowania zwanej *programowaniem dynamicznym*.

Chodzi w niej o to, aby w przypadku zadań, w których spodziewamy się sporych wyników, przy rozwiązywaniu zadania dla dużych danych wykorzystać obliczenia dla danych mniejszych. Używamy w tym celu tzw. metody *wstępującej*. W naszym przypadku odpowiada to mniej więcej podzieleniu naszego wyrażenia na mniejsze (niekoniecznie rozłączne) kawałki i po obliczeniu wyniku dla każdego z nich dokonanie syntezy uzyskanych rezultatów.

Przedstawmy zatem dwa rozwiązania: pierwsze dość standardowe w klasie zadań, gdzie występują liczby Catalana i drugie — specyficzne dla naszego zadania.

Rozwiązanie sześcienne

Zaczynamy od charakterystycznego dla programowania dynamicznego założenia. Przypuśćmy, że jakaś dobra wróżka potrafi nam dać odpowiedź na pytanie, jak dużo jest różnych ustawień nawiasów dla dowolnego ciągu krótszego od n. Czy mając taką wyrocznię pod ręką bylibyśmy w stanie skonstruować rozwiązanie? Oczywiście zakładamy, że wróżka jest w stanie poradzić sobie z każdymi danymi, byleby tylko ich długość nie przekraczała $n-1$.

Zauważmy, że zawsze wykonując działania o zadanym układzie plusów i minusów gdzieś musimy wykonać odejmowanie po raz ostatni. Czyli koniec końców nasze wyrażenie jest obliczane według schematu $(W_1)-(W_2)$, gdzie wyrażenia W_1 i W_2 są już dobrze obnawiasowanymi wyrażeniami o długości krótszej od n. Pozycja minusa oddzielającego W_1 od W_2 musi odpowiadać prawdziwemu ustawieniu minusa w zadanym ciągu. Jeżeli wyrażenie W_1 można uzyskać na w_1 sposobów nawiasowania wyrażenia z samymi minusami, a W_2 na w_2 sposobów, to przy tak ustalonej pozycji ostatniego minusa otrzymamy łącznie w_1w_2 sposobów obnawiasowania początkowego wyrażenia prowadzących do właściwego wyniku. W związku z tym wystarczy wysumować wszystkie odpowiedzi postaci w_1w_2 dla wszystkich możliwych ostatnich minusów. Nie przejmujmy się przy okazji tym, że być może jakiś konkretny minus nie będzie mógł być ostatni, bo np. to co za nim stoi nie da się obnawiasować. Wróżka jest na tyle miła, że w takim przypadku da nam odpowiedź $w_2 = 0$, powodując, że dany czynnik zaniknie.

Zauważmy jeszcze, że wyrażenie W_2 będzie ze zmienionymi znakami w porównaniu z tym, co by się stało po otwarciu nawiasów.

Algorytm nasz będzie polegał na rozważeniu oryginalnego wyrażenia $x_1c_1x_2c_2x_3\cdots x_{n-1}c_{n-1}x_n$. Symbole c_i oznaczają tu znaki plus albo minus, dla $1 \leqslant i < n$. Jeżeli znak c_i jest plusem, to nie istnieją takie W_1 i W_2, aby granica między nimi przebiegała w miejscu i. Jeśli natomiast c_i jest minusem, to należy rozbić nasze

[1] W celu uzasadnienia tego wyniku można skorzystać ze wzoru Stirlinga pozwalającego przybliżyć wartość $n! \approx \sqrt{2\pi n}n^n e^{-n}$, gdzie e jest podstawą logarytmów naturalnych.

oryginalne wyrażenie na $(W_1)c_i(W_2)$ i spytać się wróżki o to, na ile sposobów można uzyskać wyrażenie W_1 oraz na ile sposobów można uzyskać wyrażenie W_2, w którym każdy minus zamienimy na plus i na odwrót. Otrzymane w ten sposób dla każdego minusa wyniki trzeba pomnożyć i dodać do globalnego licznika.

Skąd wziąć taką sympatyczną wróżkę? Otóż rolę wróżki może tu pełnić rekurencja. Wszak zawsze wyrażenia W_1 i W_2 są krótsze od całości wyrażenia, co jest warunkiem zastosowania schematu rekurencji. Musimy jeszcze wymyśleć, kiedy rekurencję przerwać. W oczywisty sposób dochodzimy do wniosku, że gdy ciąg jest jednoelementowy, to mamy dla niego dokładnie jedno obnawiasowanie.

Dochodzimy więc do następującej procedury

```
function ilenawiasów(i, j : integer; var C : nawiasy) : integer;
{ Zakładamy, że w tablicy C[1..n − 1] mamy umieszczone znaki c_i }
{ zakodowane na przykład za pomocą liczb 1 i −1. Zadaniem funkcji }
{ będzie wyznaczenie liczby obnawiasowań, która daje wartość }
{ wyrażenia x_i c_i x_{i+1} ··· x_{j−1} c_{j−1} x_j. }
var suma, k : integer;
begin
  if j − i < 2 then return 1;
  else begin
    suma := 0;
    for k := i to j − 1 do
      if C[k] = −1 then
        { tylko minus wart jest zachodu }
        begin
          odwróć(C, k + 1, j − 1);
            { procedura zamienia znaki w tablicy C }
            { od k + 1 do j − 1 na przeciwne }
          suma := suma
                  + ilenawiasów(i, k, C) * ilenawiasów(k + 1, j, C);
          odwróć(C, k + 1, j − 1);
            { ... i przywraca poprzednią postać tablicy C, }
            { aby dla kolejnych k zachować oryginalne wartości }
        end;
      return suma;
   end;
end;
```

Teraz wystarczy wywołanie

Write(*ilenawiasów*(1, *n*, *C*));

i powinniśmy uzyskać odpowiedź. Problem polega na tym, że co prawda opisana funkcja rzeczywiście oblicza szukaną wartość, ale jej koszt jest wykładniczy. Spowodowane jest to tym, że dla bardzo wielu odcinków $[i, j]$ będzie ona wielokrotnie wywoływana.

Wystarczy jednak zastosować technikę zwaną *spamiętywaniem*, aby pozbyć się tego kłopotu. Wymaga to zapamiętywania wszystkich pośrednich wyników za pierwszym razem, kiedy zostaną policzone. Przed jakimkolwiek wywołaniem rekurencyjnym będziemy sprawdzali, czy szukana wartość nie jest już przypadkiem obliczona. Jeśli tak jest, to pobierzemy ją po prostu z odpowiedniej tablicy, zamiast od początku wyliczać. Jak duża powinna być taka tablica na przechowywanie wyników? Wszystkich odcinków $[i, j]$ jest kwadratowo dużo (mniej więcej $\frac{n^2}{2}$). Dodatkowo, dla każdego odcinka należy pamiętać dwa wyniki: „pozytyw” i „negatyw”. Tablica C bowiem jest co chwila kawałkami odwracana i każdy odcinek występuje w algorytmie albo w wersji normalnej, albo odwróconej, z pozamienianymi znakami na przeciwne.

Zamiast jednak deklarować dwie osobne kwadratowe tablice W_1 i W_2, zadeklarujemy jedną o dwóch wierszach: w pierwszym będzie kwadratowa tablica odpowiadająca pozytywowi, a w drugim negatywowi.

```
type tablica = array[1..n,1..n] of integer;
{ Tablice będą wypełnione przydatnymi danymi tylko w połowie. }
{ Wartości o indeksach [i,j] takich, że i > j nie będziemy }
{ w ogóle wykorzystywali. W faktycznej realizacji sensownie byłoby }
{ zadeklarować jedną tablicę i górną jej część wykorzystywać }
{ do „pozytywu”, a dolną do „negatywu”. }
```

a następnie inicjalizujemy je umieszczając na przekątnej jedynki, a nad przekątną wartości -1, aby zaznaczyć, że są one jeszcze niepoliczone.

```
type nawiasy = array[1..n] of −1..1;
var
  t : array[0..1] of nawiasy;
  W : array[0..1] of tablica;
  { Tablica W zawiera dwa wiersze indeksowane 0..1. }
  { W pierwszym z nich, W[0], obliczamy liczbę rozstawień nawiasów }
  { dla zadanego ciągu, a w drugim, W[1], dla jego przeciwieństwa. }

function ilenawiasów(i, j : integer; k : integer) : integer;
{ Zakładamy, że w tablicy t, w wierszu t[0][1..n − 1], mamy umieszczone }
{ znaki c_i zakodowane na przykład za pomocą liczb 1 i −1, }
{ a w wierszu t[1][1..n − 1] ich przeciwieństwa. Zadaniem funkcji }
{ bedzie wyznaczenie liczby obnawiasowań, która daje wartość wyrazenia }
{ x_i c_i x_{i+1} ... x_{j−1} c_{j−1} x_j. Wyniki pośrednie będziemy }
{ umieszczali w tablicy W: dla ciagu danego w wierszu W[0], a dla }
{ przeciwnego w W[1]. Parametr k mówi nam o tym, czy obliczenia }
{ prowadzimy dla ciągu zadanego (k = 0), czy dla jego negatywu (k = 1). }
const minus = −1;
var
  suma, l : integer;
begin
  if W[k][i,j] < 0 then { Tej wartości jeszcze nie liczyliśmy. }
  begin suma := 0;
    for l := i to j − 1 do
      if t[k][l] = minus then { tylko minus wart jest zachodu }
        suma := suma + ilenawiasów(i, l, k)
                     * ilenawiasów(l + 1, j, 1 − k);
          { 1 − k przenosi nas do przeciwnego ciągu znaków }
          { wymuszonego przez minus, który ma przed nim stać }
    W[k][i,j] := suma;
  end;
  return W[k][i,j];
end;
```

Wartość funkcji *ilenawiasów*$(1, n, 0)$ jest poszukiwaną liczbą rozstawień nawiasów.

Przyjrzyjmy się złożoności naszego algorytmu. Aby obliczyć wartość dla całego ciągu trzeba zapytać w stosownym czasie o wartości dla każdej pary $i \leqslant j$. A dla każdej takiej pary należy wykonać pętlę przebiegającą wszystkie możliwe cięcia takiego ciągu na dwa podciągi. Takich par jest, jak wiemy, rzędu n^2 (dokładniej $\frac{1}{2}n^2$), a dodatkowa pętla wewnętrzna podnosi całkowity koszt do rzędu n^3 (dokładniej $\frac{1}{6}n^3$). Potrzeba nam tu jeszcze kwadratowej pamięci na spamiętanie tablicy W. Zatem koszt czasowy jest sześcienny, a pamięciowy kwadratowy. Łatwo zauważyć, że rozwiązanie sześcienne jest zbyt kosztowne jak na rozmiar danych, aby mogło być zaakceptowane dla $n = 5000$. Rzecz jasna pamięć rzędu n^2 dla tak dużych danych jest również nieakceptowalna.

Przedstawiony schemat algorytmu stosuje się w wielu zadaniach tego typu. Można o nich przeczytać w książkach [14] oraz [25].

Rozwiązanie kwadratowe

Rozwiązanie wzorcowe pochodzi od Remigiusza Różyckiego, jednego z jurorów olimpiady. Tekst tego rozdziału wykorzystuje w istotnej części opracowanie zadania jego autorstwa. Rozwiązanie to opiera się również na zasadzie programowania dynamicznego, ale dzięki pewnym obserwacjom zbija każdy z kosztów o czynnik n.

Podstawowy pomysł polega na zauważeniu, że każdemu poprawnie obnawiasowanemu wyrażeniu odpowiada dokładnie jedno wyrażenie nie do końca obnawiasowane, ale odpowiadające regule lewostronnej łączności, czyli takie, w którym, z założenia, w przypadku kilku następujących po sobie bez żadnych nawiasów odejmowań, działania wykonujemy od lewej do prawej. Zatem takie uproszczone nawiasowanie wymaga opuszczenia tych nawiasów, które da się zastąpić regułą lewostronnej łączności. Łatwo się przekonać, że odpowiedniość wyrażeń w pełni obnawiasowanych i wyrażeń obnawiasowanych minimalnie, ale lewostronnie łącznych, jest wzajemnie jednoznaczna. Nazwijmy pełne nawiasowanie wyrażenia jego *LR-nawiasowaniem*, a to, które powstaje z LR-nawiasowania przez usunięcie wszystkich par nawiasów niepotrzebnych przy założeniu lewostronnej łączności, *R-nawiasowaniem*.

Na przykład: LR-nawiasowaniu $x_1 - ((x_2 - x_3) - x_4)$ odpowiada R-nawiasowanie $x_1 - (x_2 - x_3 - x_4)$, gdyż to, że odejmowanie $x_2 - x_3$ należy wykonać przed odjęciem od czegokolwiek x_4 wynika z lewostronnej łączności. Zauważmy przy okazji, że w R-nawiasowanych wyrażeniach nigdy nie spotkamy więcej niż jednego nawiasu otwierającego przed daną zmienną.

Szukając odpowiedniego R-nawiasowania chcemy zatem do wyrażenia $x_1 - x_2 - \cdots - x_n$ wstawić tak nawiasy, aby otrzymać wyrażenie równoważne danemu przy założeniu lewostronnej łączności. Nasze decyzje można zapisać w postaci ciągu $(w_1, w_2, \ldots, w_{n-1})$, gdzie w_i są liczbami całkowitymi, przy czym $w_i \geqslant 0$ oznacza w_i nawiasów otwierających występujących bezpośrednio przed zmienną x_{i+1}, a $w_i < 0$ oznacza $-w_i$ nawiasów zamykających występujących bezpośrednio za zmienną x_{i+1}. Przykład: podane powyżej R-nawiasowanie zapiszemy jako ciąg $(1, 0, -1)$.

Naszym zadaniem jest obliczyć liczbę LR-nawiasowań wyrażenia $x_1 - x_2 - \cdots - x_n$ równoważnych wyrażeniu $x_1c_1x_2c_2\ldots x_{n-1}c_{n-1}x_n$. Ze względu na istniejącą wzajemną jednoznaczność R-nawiasowań i LR-nawiasowań, wystarczy wyznaczyć liczbę odpowiednich R-nawiasowań.

Twierdzenie *Ciąg $(w_1, w_2, \ldots, w_{n-1})$ jest R-nawiasowaniem wyrażenia $x_1 - x_2 - \cdots - x_n$ równoważnym z wyrażeniem $x_1c_1x_2c_2\ldots x_{n-1}c_{n-1}x_n$ wtedy i tylko wtedy gdy:*

1 *dla $i = 1, 2, \ldots, n-2$ zachodzi $\sum_{j=1}^{i} w_j \geqslant 0$ i $\sum_{j=1}^{n-1} w_j = 0$*

2 *dla $i = 1, 2, \ldots, n-2$:*

- **2a** *jeśli $c_i \neq c_{i+1}$, to $w_i = 1$ lub w_i jest ujemne i nieparzyste*
- **2b** *jeśli $c_i = c_{i+1}$, to w_i jest niedodatnie i parzyste*

3 *$w_{n-1} \leqslant 0$ i w_{n-1} jest:*

- **3a** *parzyste, jeśli c_{n-1} jest znakiem '-'*
- **3b** *nieparzyste, jeśli c_{n-1} jest znakiem '+'*

Dowód Warunek 1 to po prostu warunek tego, że mamy do czynienia z poprawnym nawiasowaniem: w każdym miejscu wyrażenia liczba nawiasów otwierających nie jest mniejsza, niż liczba nawiasów zamykających, a na końcu wyrażenia te liczby są sobie równe.

Warunek 2 dotyczy wszystkich znaków poza ostatnim i mówi, że wartości dodatnie, to same jedynki, które występują tylko wtedy gdy kolejne dwa znaki się różnią. Faktycznie nie ma powodu otwierać więcej niż jednego nawiasu, gdyż działa lewostronna łączność, która wymusi właściwą kolejność. Jeżeli dwa kolejne znaki c_i oraz c_{i+1} są takie same, to po pierwszym z nich nie może wystąpić (jedyny!) nawias otwierający, bo po otwarciu tego nawiasu znaki musiałyby się różnić. Jeżeli zaś po zmiennej x_{i+1} występuje kilka nawiasów zamykających, to w przypadku, gdyby miała być ich parzysta liczba (w tym 0), oznaczałoby to, że znak c_i jest identyczny z c_{i+1}. Zero jest oczywiste, gdyż brak nawiasów oznacza dwa kolejne minusy, a dodanie każdej pary nawiasów zamykających zmienia tylko jakość następujących po sobie znaków, zmniejszając wartość kodującej ich liczby o 2.

Warunek 3 jest praktycznie powtórzeniem warunku 2 przy założeniu, że na końcu wyrażenia sztucznie dopisujemy minus za zmienną x_n, jak gdyby spodziewając się jeszcze czegoś dalej. Gdyby bowiem nasze wyrażenie miało dalszy ciąg, to zgodnie z lewostronną łącznością nie wymagałoby żadnych nawiasów na początku i sam początek kodowany byłby dokładnie tak jak do tej pory.

Uzasadniliśmy więc, że jeśli wyrażenie jest R-nawiasowaniem, to jego kod spełnia warunki 1–3. Chwila zastanowienia wystarczy, żeby przekonać się, że analogiczne rozumowanie działa w przeciwną stronę. □

Powyższy fakt prowadzi do następującego algorytmu wykorzystującego programowanie dynamiczne. Niech $t(i,j)$, dla $i = 1, 2, \ldots, n-1$ i $j = 0, \ldots, i$, oznacza liczbę sposobów wstawienia nawiasów do wyrażenia $x_1 - x_2 - \cdots - x_{i+1}$ takich, że powstałe wyrażenie jest początkiem jakiegoś R-nawiasowania wyrażenia $x_1 - x_2 - \cdots - x_n$ równoważnego wyrażeniu $x_1c_1x_2c_2\ldots x_{n-1}c_{n-1}x_n$ i takich, że $\sum_{k=1}^{i} w_k = j$. Dla uproszczenia przyjmijmy, że dla i, j nie spełniających podanych ograniczeń, $t(i,j) = 0$. Zauważmy, że $t(i,j) = 0$ dla nieparzystych j, gdy c_{i+1}='-', i dla parzystych j, gdy c_{i+1}='+'. Chcemy obliczyć wartość $t(n-1, 0)$. Korzystając z rekurencyjnej zależności:

$$t(1,0) = 1 \text{ i } t(1,j) = 0, \text{ dla } j > 0, \text{ gdy } c_2 = c_1;$$

$$t(1,1) = 1 \text{ i } t(1,j) = 0, \text{ dla } j \neq 1, \text{ gdy } c_2 \neq c_1;$$

$$t(i,j) = \sum_{k \geqslant 0} \begin{cases} t(i-1, j+2k) \text{ gdy } c_i = c_{i+1} \\ t(i-1, j+2k-1) \text{ gdy } c_i \neq c_{i+1} \end{cases}$$

możemy to zrobić w czasie $O(n^2)$. Odpowiedni kod znajduje się w pliku `naw1.pas`. Użyto w nim kwadratowej tablicy. Łatwo jednak zauważyć, że w trakcie wykonywania algorytmu korzystamy jedynie z dwóch ostatnich kolumn tablicy. Wersja programu wykorzystująca ten fakt przedstawiona jest w pliku `naw2.pas`. Używamy w niej pamięci liniowej ($O(n)$). Koszt czasowy jest identyczny: też kwadratowy. Zauważmy, że ze względu na zupełnie inny charakter rekurencji nie można było analogicznych oszczędności pamięciowych zrobić w rozwiązaniu sześciennym omówionym w poprzednim rozdziale.

Jeszcze prostszy algorytm

Załóżmy, że nasz ciąg znaków jest uzupełniony o znak $c_n = -1$. Niech $alt(i)$ oznacza liczbę zmian znaku w ciągu $c_1 \ldots c_i$. Oczywiście $alt(n)$ jest parzyste, ponieważ $c_1 = c_n$ są minusami. Niech $w(i,j) = t(i, alt(i+1) - 2j)$. Zachodzi zależność:

$$w(i,j) = \begin{cases} w(i,j-1) + w(i-1,j), & \text{gdy } 2j \leqslant alt(i+1); \\ 0 & \text{w przeciwnym razie.} \end{cases}$$

Rozwiązaniem zadania jest $w(n-1, \frac{alt(n)}{2})$. Schemat algorytmu jest następujący:

```
for i := 1 to n − 1 do w[i] := 1;
for i := 2 to n − 2 do
  if (c[i] = '+') and (c[i + 1] = '-') then
    for k := i + 1 to n − 1 do
      w[k] := w[k] + w[k − 1];
Wypisz(w[n − 1]);
```

Algorytm ten działa w czasie $O(n^2)$ i pamięci $O(n)$. Program znajduje się w pliku `naw.pas` (29 wierszy).

Testy

Rozwiązania były testowane za pomocą 11 testów:

- `naw1.in...naw4.in` — małe testy poprawnościowe;
- `naw5.in...naw7.in` — testy losowe dla $n = 100, 200, 500$;
- `naw8.in...naw10.in` — testy wydajnościowe dla n równego odpowiednio 2500, 5000, 5000.

Wojciech Guzicki
Treść zadania, Opracowanie

Wojciech Dudek
Program

Szyfr

Dany jest ciąg dodatnich liczb całkowitych a_i (dla $i = 1, 2, \ldots, n$). Ciąg ten jest używany do szyfrowania n-bitowych wiadomości. Jeśli mamy wiadomość, której kolejne bity tworzą ciąg $(t_1, t_2, \ldots, t_n)$ ($t_i \in \{0, 1\}$), to po zaszyfrowaniu ma ona postać liczby:

$$S = t_1 a_1 + t_2 a_2 + \cdots + t_n a_n$$

Zadanie

Masz dane zaszyfrowane wiadomości oraz ciągi liczb (a_i), których użyto do ich zaszyfrowania. Twoje zadanie polega na odkodowaniu zaszyfrowanych wiadomości i zapamiętaniu ich w określonych plikach. Nie musisz dostarczać żadnego programu. Wystarczy, że zapiszesz odszyfrowane wiadomości.

Wejście

Dysponujesz kilkoma zestawami danych. Każdy zestaw jest zapisany w osobnym pliku: `szy`*k*`.in`*, gdzie k jest numerem zestawu. W pierwszym wierszu każdego z tych plików znajduje się jedna liczba całkowita n, $5 \leqslant n \leqslant 40$. W kolejnych n wierszach zapisany jest ciąg liczb (a_i): w $i+1$-ym wierszu zapisana jest jedna dodatnia liczba całkowita a_i. Suma liczb a_i nie przekracza $2\,000\,000\,000$. W $n+2$-im wierszu zapisana jest jedna liczba całkowita S — zaszyfrowana wiadomość, $0 \leqslant S \leqslant a_1 + a_2 + \cdots + a_n$.*

Wyjście

Dla każdego zestawu danych `szy*.in` *powinieneś utworzyć plik* `szy*.out` *zawierający odszyfrowaną wiadomość. W pierwszym wierszu tego pliku należy zapisać kolejne liczby t_i, bez żadnych odstępów między nimi. Dane testowe zostały dobrane tak, że zaszyfrowane wiadomości są określone jednoznacznie.*

Przykład

Dla pliku wejściowego `szy.in`*:*

```
24
19226985
123697
67356296
19721773
1113273
69335448
23680077
9029881
85168664
93676782
5253843
77616588
78572630
13375812
17199980
101508862
59248276
3505733
35790095
62028546
85726819
56462819
103373994
91757169
667509506
```

poprawną odpowiedzią jest plik wyjściowy `szy.out`*:*
`11000100010110110001010l`

Rozwiązanie

Problem, który mamy rozwiązać, jest szczególnym przypadkiem tzw. problemu pakowania plecaka. W całej ogólności został on bardzo dobrze opisany w książce M. Sysły ([26] str. 215–228), gdzie w szczególności można znaleźć dość skuteczne algorytmy oparte na metodzie programowania dynamicznego. Działają one dobrze wtedy, gdy suma liczb $W = a_1 + a_2 + \cdots + a_n$ jest dość mała (złożoność wynosi $O(nW)$). W naszym przypadku, gdy liczba W może wynosić około $2 \cdot 10^9$, te algorytmy są nieprzydatne.

Zadanie nie ma dobrego rozwiązania. Mianowicie jeśli dane są dowolne liczby całkowite a_1, a_2, ..., a_n i liczba całkowita S, to zadanie polegające na znalezieniu ciągu $(t_1, t_2, \ldots, t_n)$ o wyrazach ze zbioru $\{0, 1\}$ takiego, że

$$t_1a_1 + t_2a_2 + \cdots + t_na_n = S,$$

jest przykładem problemu NP-zupełnego. Nie możemy więc spodziewać się rozwiązania działającego w czasie wielomianowym. Jedyne znane rozwiązania tego problemu w całej ogólności działają w czasie wykładniczym. Przyjrzyjmy się zatem takim rozwiązaniom.

Najbardziej narzuca się metoda przeszukania wszystkich ciągów zerojedynkowych; też trzeba to zrobić zręcznie, by wielokrotnie nie dodawać do siebie tych samych liczb. Taki program działa w czasie $O(2^n)$.

Skuteczniejsza jest metoda przeszukiwania z nawrotami. Ten algorytm polega na przeglądaniu wszystkich przypadków, które mogą prowadzić do rozwiązania. Przypuśćmy, że w danym momencie zdecydowaliśmy się wybrać liczby t_1, t_2, ..., t_k, gdzie $k < n$. Niech

$$Suma = t_1a_1 + t_2a_2 + \cdots + t_ka_k.$$

Mamy teraz następujące możliwości:

1. Jeśli $Suma > S$, to dalsze poszukiwania nie mają sensu i musimy cofnąć się do ciągu liczb t_1, t_2, ..., t_{k-1}.
2. Jeśli $Suma + a_{k+1} + \cdots + a_n < S$, to też musimy zakończyć poszukiwania i cofnąć się do ciągu liczb t_1, t_2, ..., t_{k-1}.
3. Jeśli nie zachodzi żaden z powyższych przykładów, to rozważamy dwa przypadki: $t_{k+1} = 0$ i $t_{k+1} = 1$.

Aby lepiej wykorzystać możliwość przerywania poszukiwań (i cofania się do poprzedniego ciągu liczb t_i), sortujemy na początku liczby $a_1, a_2, \ldots, a_n$ od największej do najmniejszej. Wtedy przypadki, w których mamy się cofnąć, wystąpią wcześniej i będziemy musieli sprawdzić mniej przypadków. Ten algorytm działa zupełnie dobrze wtedy, gdy wśród liczb a_1, a_2, ..., a_n występują zarówno liczby małe, jak i duże. Na przykład dla ciągów „prawie geometrycznych", tzn. takich, że $a_i \approx c^i$ dla pewnej stałej c, działa on bardzo szybko. Natomiast jeśli wszystkie liczby a_i są podobnego rzędu wielkości, ten algorytm też będzie działał w czasie $O(2^n)$.

Można to zadanie rozwiązać szybciej. Zapisujemy wszystkie możliwe sumy $\sum_{i \leqslant n/2} t_ia_i$ oraz $\sum_{i > n/2} t_ia_i$ w dwóch tablicach P i Q. Niech tablica Q ma długość k. Zapisujemy też odpowiednie ciągi t_i (wystarczą do tego trzy bajty). Następnie sortujemy obie tablice. Daną sumę identyfikujemy za pomocą następującej procedury:

```
i := 1;
j := k;
OK := false;
while not OK do
  if P[i] + Q[j] < S then i := i + 1
  else if P[i] + Q[j] > S then j := j − 1
  else OK := true;
```

Szukany ciąg liczb t_i znajdujemy teraz łatwo z ciągów odpowiadających sumom $P[i]$ i $Q[j]$.

Ten program działa w czasie rzędu $2^{n/2}$, wymaga jednak bardzo dużej pamięci. To ograniczenie pamięci spowodowało ograniczenie górnej wartości n do 40. Interesujące jest jednak to, że jest to najszybszy algorytm rozwiązujący ten problem w całej ogólności. Program wzorcowy wykorzystuje właśnie ten algorytm.

Problem postawiony w tym zadaniu wziął się z zaproponowanej w 1978 roku przez Merklego i Hellmana metody szyfrowania. Opiszemy teraz krótko tę metodę.

Najpierw wybieramy ciąg liczb b_1, b_2, ..., b_n, dla których problem pakowania plecaka można rozwiązać bardzo łatwo. Na przykład wybieramy tzw. ciąg superrosnący, tzn. taki, że każda liczba b_k jest większa od sumy liczb poprzednich:

$$b_k > b_1 + b_2 + \cdots + b_{k-1}.$$

Znalezienie prostego algorytmu, za pomocą którego możemy w czasie liniowym wyznaczyć liczby $t_1, t_2, \ldots, t_n$ takie, że

$$t_1b_1 + t_2b_2 + \cdots + t_nb_n = S,$$

pozostawimy Czytelnikowi jako ćwiczenie.

Następnie wybieramy liczbę m większą od sumy wszystkich liczb b_i:

$$m > b_1 + b_2 + \cdots + b_n$$

oraz liczbę $u < m$ względnie pierwszą z m. Za pomocą algorytmu Euklidesa znajdujemy liczbę v taką, że

$$uv \equiv 1 \pmod{m}.$$

Następnie definiujemy liczby $a_1, a_2, \ldots, a_n$ wzorem

$$a_k = b_k u \bmod m.$$

Teraz liczb $a_1, a_2, \ldots, a_n$ możemy używać do szyfrowania. Ciąg bitów $t_1, t_2, \ldots, t_n$ szyfrujemy za pomocą sumy

$$S = t_1a_1 + t_2a_2 + \cdots + t_na_n.$$

Trudności związane z rozwiązaniem naszego zadania powodują, że jeśli liczba n jest dostatecznie duża (np. $n = 200$), to po zaszyfrowaniu danej wiadomości nikt nie będzie umiał jej rozszyfrować za pomocą znanych algorytmów. Jednakże osoba, która skonstruowała ciąg liczb $a_1, a_2, \ldots, a_n$ może wykorzystać ten sposób konstrukcji do rozszyfrowania wiadomości. Mianowicie oblicza ona liczbę

$$T = Sv \bmod m,$$

a następnie znajduje liczby $t_1, t_2, \ldots, t_n$ takie, że

$$t_1b_1 + t_2b_2 + \cdots + t_nb_n = T.$$

Pominiemy tu nietrudny dowód tego, że znalezione w ten sposób liczby $t_1, t_2, \ldots, t_n$ są tymi samymi liczbami, które zostały zaszyfrowane.

Autor szyfru zna metodę tworzenia ciągu liczb $a_1, a_2, \ldots, a_n$ i dlatego umie rozszyfrować każdą zaszyfrowaną wiadomość. Osoba postronna nie wie, w jaki sposób liczby $a_1, a_2, \ldots, a_n$ powstały. Wydają się one być liczbami zupełnie przypadkowymi i problem znalezienia liczb $t_1, t_2, \ldots, t_n$ wydaje się być problemem bardzo trudnym. Dlatego szyfr ten został zaproponowany jako tzw. szyfr z publicznym kluczem: każdy zna klucz szyfrowania, tzn. liczby $a_1, a_2, \ldots, a_n$ i umie zaszyfrować dowolną wiadomość. Tylko autor szyfru zna klucz rozszyfrowywania, tzn. liczby u, v, m oraz $b_1, b_2, \ldots, b_n$ i umie rozszyfrować wiadomość.

Szyfr ten jednak w ciągu kilku lat został złamany. W 1982 roku Shamir wskazał metodę, za pomocą której można znaleźć liczby v i m. Ciąg $a_1, a_2, \ldots, a_n$ nie jest przecież zupełnie dowolny. Wiemy, że powstał on z pewnego ciągu superrosnącego. Wystarczy znaleźć liczby v i m, by ten ciąg superrosnący odtworzyć. Dokładniej: Shamir wskazał metodę znajdowania pewnych liczb v i m. Okazuje się, że ten sam ciąg $a_1, a_2, \ldots, a_n$ może powstać z nieskończenie wielu różnych ciągów superrosnących $b_1, b_2, \ldots, b_n$ (a więc i nieskończenie wielu liczb u, v i m dobranych do tych ciągów superrosnących). Shamir wskazał, w jaki sposób można znaleźć jeden z tych nieskończenie wielu ciągów superrosnących.

Inną metodę zaproponowali nieco później Lagarias i Odlyzko. Zauważyli oni, że liczby $a_1, a_2, \ldots, a_n$ są bardzo duże i jest ich bardzo mało w stosunku do wielkości tych liczb. Ciąg superrosnący $b_1, b_2, \ldots, b_n$ rośnie bowiem co najmniej wykładniczo:

$$b_k \geqslant 2^{k-1}.$$

Zatem $m > 2^n$. Liczby $a_1, a_2, \ldots, a_n$ są na ogół podobnej wielkości: w zapisie dwójkowym mają około n cyfr. Jest ich tylko n, więc występują bardzo rzadko wśród liczb mniejszych od m. Lagarias i Odlyzko pokazali metodę rozwiązywania problemu pakowania plecaka dla bardzo rzadko rozmieszczonych liczb $a_1, a_2, \ldots, a_n$. Ich metoda okazała się niezwykle skuteczna w praktyce do łamania szyfru plecakowego.

Obie metody, Shamira oraz Lagariasa i Odlyzki, wykorzystują bardzo skomplikowane algorytmy, których nawet pobieżne opisanie znacznie wykraczałoby poza ramy tej książki.

To, że liczby $a_1, a_2, \ldots, a_n$ muszą być bardzo duże, wynika również z warunku jednoznaczności: dla każdej liczby S istnieje co najwyżej jeden ciąg liczb $t_1, t_2, \ldots, t_n$ taki, że

$$t_1a_1 + t_2a_2 + \cdots + t_na_n = S.$$

Mianowicie istnieje 2^n ciągów $t_1, t_2, \ldots, t_n$, a więc możemy otrzymać 2^n różnych sum S powyższej postaci. Zatem przynajmniej niektóre z liczb $a_1, a_2, \ldots, a_n$ muszą być duże; jeśli wszystkie mają być podobnej wielkości, to muszą mieć około n cyfr w dwójkowym układzie pozycyjnym.

Warunek jednoznaczności jest konieczny do tego, by ciągu $a_1, a_2, \ldots, a_n$ można było używać do szyfrowania: każdy kryptogram (czyli wiadomość zaszyfrowana) może być rozszyfrowany w jeden tylko sposób. Można łatwo pokazać, że ciąg $a_1, a_2, \ldots, a_n$ powstający z ciągu superrosnącego ma tę własność jednoznaczności.

Wielkość liczb $a_1, a_2, \ldots, a_n$ była przeszkodą w konstruowaniu testów. Liczby podane w przykładzie powstały w opisany sposób z pewnego ciągu superrosnącego. Jednak dla $n = 24$ nawet algorytm przeszukiwania z nawrotami działa wystarczająco szybko, by wykorzystać go do skutecznego rozwiązania zadania. Testy duże musiały być tworzone w inny sposób. Dla $n = 40$ liczby $a_1, a_2, \ldots, a_n$ byłyby wielkości rzędu 2^{40} i ich suma przekroczyłaby dopuszczalną wielkość $2 \cdot 10^9$, narzuconą przez ograniczenia pamięci. Wykorzystano kilka innych pomysłów, wymagających jednak sprawdzenia za każdym razem, czy rozwiązanie jest jednoznaczne.

VIII Bałtycka Olimpiada Informatyczna, Wilno 2002

VIII Bałtycka Olimpiada Informatyczna — treści zadań

Zbigniew J. Czech — Treść zadania
Marcin Kubica, Krzysztof Stencel — Tłumaczenie

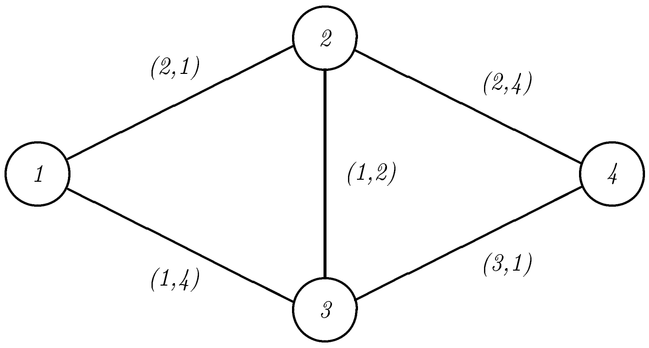
Dwukryterialny wybór drogi (Bicriterial routing)

W Bajtocji bardzo dynamicznie rozwija się sieć płatnych autostrad. Obecnie jest ona tak gęsta, że wybór najlepszej trasy przejazdu staje się problemem. Sieć autostrad składa się z dwukierunkowych odcinków łączących miasta. Każdy z takich odcinków charakteryzuje się czasem potrzebnym na jego przejechanie oraz opłatą, jaką należy za to uiścić.

Trasę przejazdu tworzą kolejne odcinki autostrad, którymi jedziemy. Czas potrzebny do przejechania trasy to suma czasów potrzebnych do przejechania odcinków tworzących trasę, a koszt przejechania trasy to suma opłat za przejechanie odcinków tworzących trasę. Im szybciej można przejechać trasę i im mniej to kosztuje, tym lepsza trasa. To znaczy, trasa jest lepsza od innej, gdy ma mniejszy czas i nie większy koszt lub odwrotnie: mniejszy koszt i nie gorszy czas. Trasę łączącą dwa miasta nazwiemy minimalną, jeśli nie istnieje lepsza od niej trasa łącząca te miasta. Nie zawsze jednak istnieje jedna minimalna trasa — może istnieć kilka nieporównywalnych, minimalnych tras lub nie istnieć żadna.

Przykład

Na poniższym rysunku przedstawiono przykładową sieć autostrad. Przy każdym odcinku autostrady podano w nawiasach koszt oraz czas przejazdu.



Rozważmy cztery różne trasy przejazdu z miasta 1 do miasta 4, wraz z ich kosztami i czasami przejazdu: 1–2–4 (koszt 4, czas przejazdu 5), 1–3–4 (koszt 4, czas przejazdu 5), 1–2–3–4 (koszt 6, czas przejazdu 4) i 1–3–2–4 (koszt 4, czas przejazdu 10). Trasy 1–3–4 i 1–2–4 są lepsze od 1–3–2–4. Mamy tutaj dwie minimalne pary koszt-czas: koszt 4, czas 5 (trasy 1–2–4 i 1–3–4), oraz koszt 6, czas 4 (trasa 1–2–3–4). Wybierając trasę musimy zdecydować, czy wolimy jechać krócej, ale drożej (trasa 1–2–3–4), czy dłużej, ale taniej (trasy 1–3–4 i 1–2–4).

Zadanie

Twoim zadaniem jest napisanie programu, który:

- *wczyta z pliku wejściowego* `bic.in` *opis sieci autostrad oraz początek i koniec trasy,*
- *obliczy, ile jest różnych minimalnych tras o zadanym początku i końcu, przy czym trasy o tym samym koszcie i czasie liczymy tylko raz — interesuje nas tylko, ile jest różnych minimalnych par koszt-czas,*
- *zapisze wynik w pliku wyjściowym* `bic.out`.

Wejście

W pierwszym wierszu pliku tekstowego `bic.in` *zapisane są cztery liczby całkowite pooddzielane pojedynczymi odstępami: liczba miast n (są one ponumerowane od 1 do n), $1 \leqslant n \leqslant 100$, liczba odcinków autostrad m, $0 \leqslant m \leqslant 300$, numery miast, w których trasa ma początek s i koniec e, $1 \leqslant s, e \leqslant n$, $s \neq e$. W kolejnych m wierszach opisane są odcinki autostrad — po jednym w wierszu. W każdym z tych wierszy zapisane są po cztery liczby całkowite, pooddzielane pojedynczymi odstępami, oznaczające kolejno: dwa końce odcinka autostrady p i r, $1 \leqslant p, r \leqslant n$, $p \neq r$, koszt przejazdu c, $0 \leqslant c \leqslant 100$, oraz czas przejazdu t, $0 \leqslant t \leqslant 100$. Dwa miasta mogą być połączone więcej niż jednym odcinkiem autostrady.*

Wyjście

Twój program powinien zapisać w jedynym wierszu pliku tekstowego `bic.out` *jedną liczbę całkowitą, liczbę minimalnych par koszt-czas dla tras prowadzących z s do e.*

Przykład

Dla pliku wejściowego `bic.in`*:*

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

poprawnym wynikiem jest plik wyjściowy `bic.out`*:*

```
2
```

Marcin Kubica, Krzysztof Stencel
Tłumaczenie

Klub tenisowy (Tennis club)

Klub tenisowy Matchball organizuje „tydzień otwartych drzwi”, który ma przyciągnąć nowych graczy do klubu. W ramach atrakcji dla odwiedzających poproszono kilka gwiazd tenisa o zagranie paru pokazowych meczy. Każda z gwiazd podała, ile meczy chce rozegrać. Organizatorzy chcą, aby gwiazdy też miały trochę zabawy, chcą więc tak zaplanować rozgrywki, aby żadni dwaj gracze nie grali ze sobą więcej niż raz.

Zadanie

Twoje zadanie polega na napisaniu programu, który pomoże dobrać graczy w pary w taki sposób, aby każdy z graczy grał tyle razy, ile chce i aby nie grał dwa lub więcej razy z tym samym przeciwnikiem. Oczywiście żaden z graczy nie może grać sam ze sobą.

Wejście

W pierwszym wierszu pliku wejściowego `TENNIS.IN` *zapisana jest liczba graczy* N *(*$2 \leqslant N \leqslant 1000$*). W kolejnych* N *wierszach zapisano liczbę meczy* G_i*, które chcą rozegrać poszczególni gracze (*$1 \leqslant G_i < N$*). Przyjmij, że gracze są ponumerowani od 1 do* N*, zgodnie z kolejnością ich życzeń w pliku wejściowym.*

Wyjście

Jeżeli nie można tak zaplanować meczy, aby spełnić życzenia wszystkich graczy, należy w pierwszym wierszu pliku wyjściowego `TENNIS.OUT` *zapisać* `NO SOLUTION`*. Jeżeli jest to możliwe, to należy w pierwszym wierszu tego pliku zapisać* `SOLUTION`*, a w kolejnych* N *wierszach należy zapisać rozwiązanie. W każdym z tych wierszy należy zapisać numery przeciwników gracza, dla którego liczbę meczy, które chce rozegrać, podano w odpowiadającym wierszu pliku wejściowego. W każdym z tych wierszy numery przeciwników muszą być podane w rosnącej kolejności i poodzielane odstępami. Jeżeli istnieje wiele rozwiązań, Twój program powinien zapisać jedno z nich.*

Przykłady

TENNIS.IN	TENNIS.OUT	TENNIS.IN	TENNIS.OUT
3	SOLUTION	3	NO SOLUTION
1	2	2	
2	1 3	2	
1	2	1	

Ocena

W tym zadaniu program otrzyma punkty zdobyte za testy, w których brak rozwiązania, tylko wtedy, gdy rozwiąże poprawnie przynajmniej połowę testów, w których istnieje rozwiązanie.

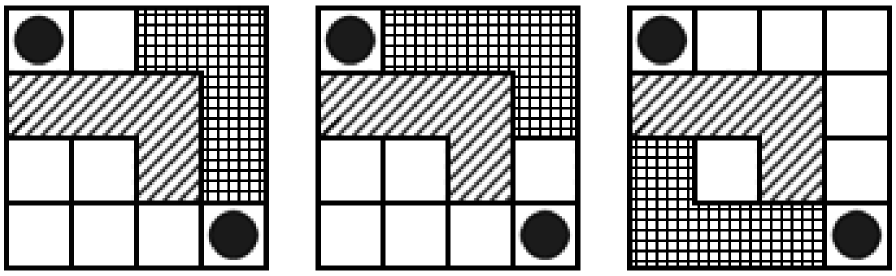
Marcin Kubica, Krzysztof Stencel
Tłumaczenie

L-gra (L-game)

L-grę opracował Edward de Bono, który uwielbiał gry, ale nienawidził koncentrować się na dużej liczbie pionów. Jego intencją było stworzenie najprostszej gry, jak tylko się da, ale wymagającej umiejętnej rozgrywki.

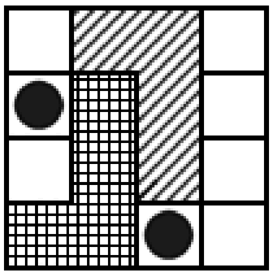
Każdy gracz ma tylko jeden pion, tzw. **L-pion**. *Są też dwa rodzaje neutralnych kwadratowych pionów. Plansza jest kwadratem 4 na 4 pola. Celem gry jest doprowadzenie na planszy do sytuacji, w której drugi gracz nie może poruszyć swego L-pionu.*

W początkowej sytuacji pierwszy gracz (i każdy gracz przy następnym ruchu) musi zacząć od przesunięcia L-pionu. Ruch może polegać na przesunięciu, obróceniu lub podniesieniu pionu, odwróceniu go i położeniu na dowolnym miejscu innym niż to zajmowane przed ruchem. Po poruszeniu L-pionu gracz może przenieść jeden (ale tylko jeden) z dwóch pionów neutralnych na dowolne wolne pole. Ruch pionem neutralnym nie jest obowiązkowy. Jego wykonanie zależy od gracza!



W każdej z tych trzech sytuacji gracz posługujący się pokratkowanym L-pionem może przesunąć swój L-pion do dwóch nowych położeń (dwóch pozostałych z tych trzech powyżej). Po przesunięciu L-pionu gracz może przenieść jeden z pionów neutralnych na dowolny z 6 pozostałych pól lub zdecydować się na nieporuszanie żadnego z nich. Jest tu więc $2 \cdot (6 + 6 + 1)$ *ruchów.*

Gracz wygrywa grę, gdy jego przeciwnik nie może poruszyć swego L-pionu. Jeśli w sytuacji przedstawionej poniżej gracz posługujący się pokratkowanym L-pionem ma wykonać ruch, przegrywa, ponieważ nie może przesunąć L-pionu do nowego, wolnego położenia na planszy.



Ta gra jest prosta, ale złożona ze względu na tak dużą liczbę ruchów. Jest w niej ponad 18000 położeń pionów na małej planszy, a w każdej chwili można wykonać nawet 195 różnych ruchów, z których tylko jeden prowadzi do wygranej.

Napisz program, który na podstawie bieżącej sytuacji na planszy i informacji o tym, który gracz ma wykonać ruch, stwierdzi, czy ten gracz ma ruch wygrywający, i wypisze ten ruch, jeśli on istnieje. Jeśli istnieje kilka ruchów wygrywających, należy wypisać jeden z nich. Jeśli nie ma ruchu wygrywającego, należy stwierdzić, czy gra skończy się remisem (przy założeniu perfekcyjnej gry obu graczy), czy też gracz wykonujący ruch przegra.

Ruch wygrywający to ruch, który powoduje, że niezależnie od ruchu przeciwnika, nie może on uniknąć przegranej, jeśli pierwszy gracz kontynuuje grę perfekcyjnie.

Gracz przegrywa grę, jeśli niezależnie od swoich ruchów nie może on uniknąć przegranej, jeśli przeciwnik kontynuuje grę perfekcyjnie.

Jeśli żaden z tych warunków nie jest spełniony (tzn. żaden gracz nie może zapewnić sobie wygranej), sytuacja gry jest remisowa.

Wejście

Dane wejściowe składają się z czterech wierszy, w których opisano trwającą grę. Kropka („.”) reprezentuje puste pole, litera „`x`*” oznacza pole z pionem neutralnym, znak „*`#`*” oznacza L-pion gracza, który ma wykonać ruch, a znak „*`*`*” oznacza L-pion drugiego gracza. Możesz założyć, że sytuacja jest dopuszczalna i że gracz, który ma wykonać ruch, ma co najmniej jeden dopuszczalny ruch w aktualnej sytuacji.*

Wyjście

Jeśli istnieje ruch wygrywający, należy wypisać sytuację po takim ruchu w takim samym formacie, jak w przypadku danych wejściowych. W przeciwnym wypadku należy wypisać zdanie „`No winnig move exists`*" w pierwszym wierszu, a w następnym albo „*`Draw`*", jeśli gra zakończy się remisem (przy założeniu perfekcyjnej gry) albo „*`Losing`*", gdy gracz wykonujący ruch przegra tę grę.*

Przykład

Przykładowe dane wejściowe	*Przykładowe dane wyjściowe*

`.***`
`#*.x`
`###.`
`x...`
	`.***`
	`x*#x`
	`###.`
	`....`

`...x`
`###.`
`#***`
`x..*`
	`No winning move exists`
	`Draw`

`.###`
`x#*x`
`***.`
`....`
	`No winning move exists`
	`Losing`

Marcin Kubica, Krzysztof Stencel
Tłumaczenie

Ograniczenia prędkości (Speed limits)

W naszym zabieganym świecie nie staramy się wybrać najkrótszej trasy, ale taką, której przejechanie zajmie jak najmniej czasu. Gdy jedziemy samochodem, najważniejsze są więc ograniczenia prędkości. Wyobraźmy sobie, że brakuje niektórych znaków ograniczenia prędkości. Nie można oczekiwać, że kierowcy znają na pamięć ograniczenia prędkości, więc jedynym sensownym wnioskiem jest to, że wcześniej przestrzegane ograniczenia prędkości nadal obowiązują po minięciu brakującego znaku drogowego. Napisz program, który obliczy najszybszą trasę przy założeniu wykorzystania braku znaków drogowych.

Podano opis sieci ulic w bardzo zmotoryzowanym obszarze. Sieć składa się ze **skrzyżowań** *i* **ulic**. *Każda ulica jest jednokierunkowa, łączy dokładnie dwa skrzyżowania i ma co najwyżej jeden znak ograniczenia prędkości ustawiony na początku ulicy. Możesz założyć nieskończone przyspieszenie i to, że pozostałe jadące samochody nie mają wpływu na Twoją jazdę. Oczywiście nigdy nie wolno jechać szybciej niż obecne ograniczenia prędkości.*

Dane wejściowe

Pierwszy wiersz pliku wejściowego zawiera trzy liczby całkowite N, M i D, przy czym N ($2 \leqslant N \leqslant 150$) to liczba skrzyżowań ponumerowanych od 0 do $N-1$, M to liczba ulic, a D to numer skrzyżowania, do którego masz dotrzeć. Każdy z następnych M wierszy pliku wejściowego jest opisem jednej ulicy. Każdy zawiera cztery liczby całkowite A ($0 \leqslant A \leqslant N$), B ($0 \leqslant B \leqslant N$), V ($0 \leqslant V \leqslant 500$) i L ($1 \leqslant L \leqslant 500$), z których wynika, że ulica biegnąca od skrzyżowania A do skrzyżowania B ma ograniczenie prędkości V i długości L. Jeśli V jest równe zero, na tej ulicy nie ma znaku ograniczenia prędkości. Czas potrzebny do przejechania tej drogi to $T = L/V$, gdy $V \neq 0$, a w przeciwnym wypadku $T = L/V_{\mathrm{old}}$, przy czym V_{old} jest ograniczeniem prędkości przestrzeganym przez Ciebie zanim dojechałeś do tego skrzyżowania. Zauważ, że dzielenie należy wykonać na liczbach zmiennoprzecinkowych, aby uniknąć niepotrzebnych zaokrągleń. Na początku znajdujesz się na skrzyżowaniu 0, a Twoja prędkość wynosi 70.

Dane wyjściowe

Plik wyjściowy powinien zawierać pojedynczy wiersz zawierający liczby całkowite, będące opisami skrzyżowań leżących na najszybszej drodze od 0 do D. Skrzyżowania muszą być wypisane dokładnie w takim porządku, w jakim należy przez nie przejeżdżać, począwszy od 0 aż do D. Nigdy nie zdarzą się dwie najszybsze ścieżki o tym samym czasie przejazdu.

Przykład

Dane wejściowe:

```
6 15 1
0 1 25  68
0 2 30  50
0 5  0 101
1 2 70  77
1 3 35  42
2 0  0  22
2 1 40  86
2 3  0  23
2 4 45  40
3 1 64  14
3 5  0  23
4 1 95   8
5 1  0  84
5 2 90  64
5 3 36  40
```

Dane wyjściowe:

```
0 5 2 3 1
```

Wskazówka: Czas potrzebny na przejechanie tej trasy to 2,628 jednostek.

Marcin Kubica, Krzysztof Stencel
Tłumaczenie

Roboty (Robots)

Kilka robotów porusza się po dwuwymiarowej pełnej kracie. Stan każdego robota to jego aktualne położenie i kierunek. Każdy robot porusza się zgodnie z przypisanym mu skończonym ciągiem poleceń. Położenie jest opisane za pomocą pary liczb całkowitych (x, y)*. Istnieją cztery możliwe kierunki ruchu robota określone za pomocą liczb stopni: 0, 90, 180 i 270. Polecenia są dwojakiego rodzaju: obrót i krok. Polecenie obrotu ma jeden parametr* D*, którego wartością może być 90, 180 lub 270. To polecenie zmienia aktualny kierunek robota o* D *stopni. Kierunek* C *jest zmieniany na kierunek* $(C + D) \bmod 360$*. Polecenie kroku nie ma parametrów i powoduje krok robota w aktualnie obranym kierunku. Jeden krok w kierunku 0 zmienia położenie robota o* $(1, 0)$*, w kierunku 90 o* $(0, 1)$*, w kierunku 180 o* $(-1, 0)$*, w kierunku 270 o* $(0, -1)$*.*

Robot wykonuje polecenia ciągu jedno po drugim. Gdy wszystkie polecenia są zrealizowane robot zatrzymuje się w końcowym położeniu.

Inne roboty nie mają jakiegokolwiek wpływu na ruchy robota. W szczególności wiele robotów może znajdować się w tym samym położeniu.

Zanim roboty zaczną się poruszać, centrum sterowania może nakazać niektórym robotom usunięcie pewnych poleceń ze swoich ciągów. Centrum sterowania może więc wpływać na trasy robotów i ich końcowe położenia. Centrum sterowania pragnie zgromadzić wszystkie roboty w jednym końcowym położeniu, aby przeprowadzić inspekcję. Centrum chce do tego doprowadzić za pomocą minimalnej możliwej całkowitej liczby usuniętych poleceń.

Zadanie

Łącznie jest R $(2 \leqslant R \leqslant 10)$ *robotów. Każdy robot znajduje się w jakimś początkowym położeniu i ma początkowy kierunek oraz ciąg poleceń zawierający nie więcej niż 50 poleceń. Napisz program, który wyznaczy (o ile to możliwe) minimalną łączną liczbę poleceń, które należy usunąć z niektórych ciągów tak, aby wszystkie roboty zatrzymały się w jednym położeniu końcowym. Program ma również wyznaczyć to położenie. Jeśli jest kilka takich położeń, program ma znaleźć jedno z nich.*

Dane wejściowe

Dane wejściowe należy odczytać z pliku `ROBOTS.IN`*. Pierwszy wiersz tego pliku zawiera liczbę całkowitą* R $(2 \leqslant R \leqslant 10)$ *— liczbę robotów. Po niej następuje* R *bloków wierszy — w każdym bloku opisano jednego robota. Pierwszy wiersz bloku zawiera cztery liczby całkowite oddzielone pojedynczymi odstępami:* x*,* y *— początkowe położenie robota* (x, y)*,* d *— początkowy kierunek robota (*$d = 0$*, 90, 180 lub 270) i* n *— długość ciągu poleceń robota* $(1 \leqslant n \leqslant 50)$*. Dalej następuje* n *wierszy, w których zapisano ciąg poleceń, jedno polecenie w jednym wierszu. Wiersz z poleceniem kroku zawiera pojedynczą literę* `S` *jako pierwszy znak. Wiersz z poleceniem obrotu zawiera pojedynczą literę* `T` *jako pierwszy znak, po nim odstęp i parameter obrotu — liczba całkowita* D *(*$D = 90$*, 180 lub 270).*

Dane wyjściowe

Dane wyjściowe należy zapisać do pliku `ROBOTS.OUT`*. Jeśli nie da się sprawić, żeby wszystkie roboty zatrzymały się w tym samym końcowym położeniu poprzez usunięcie pewnych poleceń, program powienie wypisać* -1 *w pierwszym (i jedynym) wierszu pliku. W przeciwnym wypadku w pierwszym wierszu pliku należy wypisać całkowitą liczbę poleceń do usunięcia. W drugim wierszu należy wypisać wspólne końcowe położenie robotów — dwie liczby całkowite oddzielone pojedynczym odstępem.*

Przykład

Dla pliku wejściowego `ROBOTS.IN`*:*

```
2
2 0 270 5
S
T 180
S
S
S
1 -1 0 8
```

```
S
S
T 90
S
T 270
S
T 90
S
```

poprawnym plikiem wyjściowym `ROBOTS.OUT` *jest:*

```
2
2 1
```

Komentarz: Są tu dwa poruszające się roboty. Pierwszy z nich ma początkowe położenie $(2,0)$, początkowy kierunek 270 i ciąg poleceń złożony z 5 poleceń. Drugi z nich ma początkowe położenie $(1,-1)$, początkowy kierunek 0 i ciąg poleceń złożony z 8 poleceń. Minimalna całkowita liczba poleceń do usunięcia, których skasowanie sprawi, że roboty zatrzymają się w tym samym końcowym położeniu to 2. Można na przykład usunąć trzecie polecenie pierwszego robota i piąte polecenie drugiego robota. Wspólne końcowe położenie w tym wypadku to $(2,1)$.

Marcin Kubica, Krzysztof Stencel
Tłumaczenie

Słupki monet (Stacks of coins)

Kiedy siedziałeś ze swoim przyjacielem w waszym ulubionym barze mlecznym, przedstawił Ci on zasady prostej gry. Z identycznych monet ułożył kilka słupków. Ze słupka A można przenieść na słupek B tyle samo monet, ile zawiera słupek B przed wykonaniem ruchu; w wyniku takiego ruchu liczba monet w słupku B podwaja się. Zadanie polega na tym, aby za pomocą minimalnej liczby ruchów wyrównać słupki (tzn. doprowadzić do tego, aby były tej samej wysokości).

Ponieważ zasady gry nie były dla Ciebie do końca jasne, Twój przyjaciel przedstawił Ci przykład.

Mamy trzy słupki zawierające odpowiednio 4, 6 i 14 monet. Można je wyrównać przenosząc najpierw 6 monet z trzeciego słupka na drugi (słupki zawierają wówczas 4, 12 i 8 monet), a następnie przenosząc 4 monety ze słupka drugiego na pierwszy. Wówczas wszystkie słupki mają tę samą wysokość (zawierają po 8 monet) — zrobione! Ponieważ nie da się wyrównać słupków w mniejszej liczbie ruchów, podany ciąg ruchów ma minimalną długość równą 2.

Napisz program, który dla zadanego zestawu słupków monet wyznaczy ciąg ruchów minimalnej długości, który wyrównuje słupki. Ponieważ Twój przyjaciel nie ma zbyt dużo pieniędzy, nie użyje więcej niż 200 monet i nie ułoży za ich pomocą więcej niż 8 słupków.

Wejście

Plik wejściowy `stacks.in` *zawiera dwa wiersze. Pierwszy wiersz zawiera liczbę słupków S ($1 \leqslant S \leqslant 8$); słupki są ponumerowane od 1 do S. Drugi wiersz zawiera S liczb H_i ($1 \leqslant H_i \leqslant 70$, $1 \leqslant i \leqslant S$). H_i oznacza początkową wysokość słupka i (tzn. liczbę monet w tym słupku). Ponadto $\sum_{i=1}^{S} H_i \leqslant 200$.*

Wyjście

W pierwszym wierszu pliku wyjściowego `stacks.out` *należy zapisać minimalną liczbę M ruchów potrzebnych do wyrównania słupków. Możesz założyć, że dla danych wejściowych zawsze istnieje rozwiązanie (czyli łączna liczba monet zawsze dzieli się przez liczbę słupków) wymagające nie więcej niż 8 ruchów ($M \leqslant 8$).*

Kolejnych M wierszy opisuje ruchy, w kolejności ich wykonania. Każdy z tych wierszy zawiera dwie liczby F i T opisujące dany ruch, F oznacza słupek, z którego bierzemy monety, a T oznacza słupek, na który przekładamy monety.

Przykład

stacks.in	stacks.out
3 4 6 14	2 3 2 2 1

Plik źródłowy: `stacks.pas`, `stacks.c` *lub* `stacks.cpp`

Limit czasu: *2 sekundy na test*

Marcin Kubica, Krzysztof Stencel
Tłumaczenie

Trójkąty (Triangles)

Na płaszczyźnie zadano n równoramiennych trójkątów prostokątnych. Każdy taki trójkąt można opisać za pomocą trzech liczb całkowitych x, y, m ($m > 0$). Wierzchołki takiego trójkąta to punkty $(x; y)$, $(x + m; y)$ i $(x; y + m)$.

Napisz program, który wyznaczy całkowite pole powierzchni pokrytej tymi trójkątami.

Dane wejściowe

W pierwszym wierszu pliku tekstowego `tr.in` *podano jedną dodatnią liczbę całkowitą n ($n \leqslant 2000$). W następnych n wierszach pliku znajdują się opisy trójkątów (jednego trójkąta w jednym wierszu) — trzy liczby całkowite x_i, y_i i m_i, oddzielone odstępami ($1 \leqslant i \leqslant n$, $-10^7 \leqslant x_i \leqslant 10^7$, $-10^7 \leqslant y_i \leqslant 10^7$, $0 \leqslant m_i \leqslant 1000$).*

Dane wyjściowe

W pierwszym wierszu pliku tekstowego `tr.out` *należy zapisać jedną liczbę z dokładnie jedną cyfrą po kropce — całkowite pole powierzchni pokrytej trójkątami.*

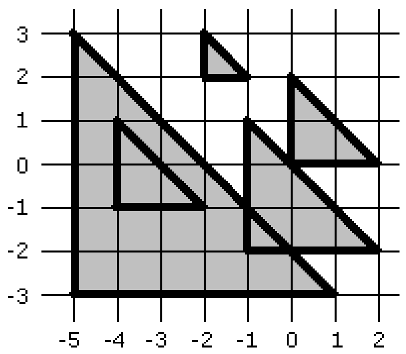
Przykład

Dane wejściowe (`tri.in`*):*

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

Dane wyjściowe (`tri.out`*)*

```
24.5
```



IX Olimpiada Informatyczna Europy Centralnej, 2002

IX Olimpiada Informatyczna Europy Centralnej — treści zadań

Krzysztof Diks, Krzysztof Stencel
Tłumaczenie

Autostrada i siedmiu krasnoludków (A highway and the seven dwarfs)

Dawno, dawno temu, w odległej galaktyce był sobie kraj, w którym żyło wiele rodzin krasnoludków. Ten kraj nosił nazwę Bajtocji. Każda rodzina żyła w jednym domu. Krasnoludki często odwiedzały swoich przyjaciół z innych rodzin. W Bajtocji nie było nieprawości, więc krasnoludki odwiedzały się wzajemnie.

Pewnego dnia ludzie żyjący w kraju sąsiadującym z Bajtocją, postanowili zbudować kilka prostoliniowych autostrad. Ludzie nie wiedzieli nic o istnieniu krasnoludków, więc kilka z planowanych autostrad przebiegało przez Bajtocję. Krasnoludki odkryły ten fakt i były z tego powodu bardzo nieszczęśliwe. Krasnoludki są malutkie i wolniutkie, więc nie są w stanie bezpiecznie przejść przez autostradę.

Krasnoludkom udało się za pomocą przekupstwa zdobyć plan sieci autostrad. Teraz potrzebują Twojej pomocy. Chciałyby w dalszym ciągu odwiedzać się nawzajem, więc nie podobają im się te autostrady, które przebiegają między ich domami, dzieląc zbiór domów na niepuste podzbiory. Po stwierdzeniu, które autostrady im się nie podobają, krasnoludki za pomocą magii uniemożliwią ludziom zbudowanie takich autostrad.

Krasnoludki to maluchy, więc nie są w stanie dosięgnąć klawiatury. Poprosiły Cię więc o pomoc.

Zadanie

Na płaszczyźnie danych jest N punktów (domów) i pewna liczba linii prostych (autostrad). Dla każdej zadanej prostej Twoim zadaniem jest ustalenie, czy wszystkie N punktów leży po tej samej stronie prostej, czy nie. Twój program musi podać odpowiedź dla aktualnie przetwarzanej prostej, **zanim** *odczyta opis następnej prostej. Możesz założyć, że żadna autostrada nie przebiega przez żaden z domów.*

Opis danych wejściowych i wyjściowych

Twój program ma czytać dane ze standardowego wejścia (`stdin` *w C/C++,* `input` *we FreePascalu) i zapisywać wynik na standardowe wyjście (*`stdout` *w C/C++,* `output` *we FreePascalu). Pierwszy wiersz danych wejściowych zawiera jedną liczbę całkowitą N ($0 \leqslant N \leqslant 100\,000$). W następnych N wierszach podano współrzędne domów — i-ty z nich zawiera dwie liczby rzeczywiste x_i, y_i ($-10^9 \leqslant x_i, y_i \leqslant 10^9$) oddzielone pojedynczym znakiem odstępu — są to współrzędne i-tego domu.*

Każdy z następnych wierszy danych zawiera cztery liczby rzeczywiste X_1, Y_1, X_2, Y_2 ($-10^9 \leqslant X_1, Y_1, X_2, Y_2 \leqslant 10^9$) pooddzielane pojedynczymi odstępami. Są to współrzędne dwóch różnych punktów autostrady $[X_1, Y_1]$ i $[X_2, Y_2]$. Dla każdego wiersza danych wejściowych Twój program ma wypisać jeden wiersz zawierający słowo „`GOOD`*”, jeśli wszystkie zadane punkty leżą po tej samej stronie zadanej prostej, albo „*`BAD`*”, jeśli zadana prosta rozdziela te punkty. Po wypisaniu każdego wiersza danych wyjściowych Twój program ma wyczyścić bufor wyjściowy. W dalszej części zadania znajdziesz wyjaśnienie, jak to można zrobić.*

Zakończymy Twój program, gdy da on odpowiedź na pytanie o ostatnią autostradę. **Twój program nie powinien się sam zatrzymywać.** *Możesz założyć, że autostrad będzie nie więcej niż 100 000.*

Podprogramy wejścia i wyjścia w C/C++

Odczytanie jednego wiersza (zauważ, że nie ma odstępu po ostatnim `%lf`*):*
```
scanf("%lf %lf %lf %lf", &X_1, &Y_1, &X_2, &Y_2);
```
Zapisanie wyniku dla jednego wiersza:
```
printf("GOOD\n"); fflush(stdout);
```

Podprogramy wejścia i wyjścia w FreePascalu

Odczytanie jednego wiersza:
```
read(X_1, Y_1, X_2, Y_2);
```
Zapisanie wyniku dla jednego wiersza:
```
writeln('GOOD'); flush(output);
```

Ostrzeżenie

Radzimy wykorzystać typ `double` *do przechowywania liczb rzeczywistych (zarówno w C/C++, jak i we FreePascalu). Pamiętaj, że przy używaniu liczb rzeczywistych mogą pojawić się błędy zaokrągleń. Jeśli chcesz sprawdzić, czy dwie liczby rzeczywiste x i y są równe, nie badaj, czy $x = y$, ale czy $|x - y| < \varepsilon$, gdzie ε jest małą stałą (w wypadku tego zadania $\varepsilon = 10^{-4}$ w zupełności wystarczy).*

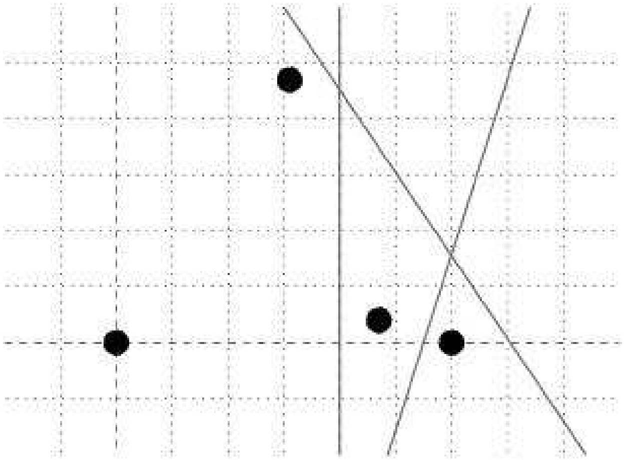
Przykład

Dane wejściowe:

```
4
0.0 0.0
6.00 -0.001
3.125 4.747
4.747 0.47
5 3 7 0
4 -4.7 7 4.7
4 47 4 94
```

Dane wyjściowe:

```
GOOD
BAD
BAD
```



Krzysztof Diks, Krzysztof Stencel
Tłumaczenie

Batalion Najeźdźcy (Conqueror's battalion)

W całej historii ludzkości zdarzyło się kilka dziwacznych bitew takich, jak ta we Francji w 1747...

W małej wiosce Bassignac-le-Haut leżącej na lewym brzegu rzeki Dordogne była sobie forteca tuż nad groblą Chastang. Znad grobli w kierunku fortecy prowadziły szerokie schody zbudowane z czerwonego marmuru. Pewnego dnia rano strażnik zauważył wielki batalion zbliżający się do fortecy pod dowództwem groźnego Najeźdźcy.

Gdy Najeźdźca stanął pod murami fortecy, czekał tam już na niego dowódca. Ponieważ dowódca fortecy dysponował jednynie niewielkim oddziałem, zaproponował Najeźdźcy: „Widzę za tobą wielu żołnierzy, którzy stoją na schodach. Możemy przeprowadzić małą grę. W każdej turze podzielisz swoich zołnierzy na dwie grupy w dowolny sposób. Następnie ja zdecyduję, która grupa zostaje, a która wraca do domów. Każdy pozostający żołnierz wejdzie w górę o jeden schodek. Jeśli co najmniej jeden żołnierz dotrze do najwyższego schodka, będziesz zwycięzcą. W przeciwnym wypadku przegrasz i zawrócisz w kierunku grobli Chastang."

Najeźdźcy spodobała się ta gra i natychmiast przystąpił do „oblężenia".

Zadanie

Jesteś Najeźdźcą. W kierunku fortecy wiedzie N schodów ($2 \leqslant N \leqslant 2\,000$). Masz co najwyżej $1\,000\,000\,000$ żołnierzy. Wiesz, ilu żołnierzy stoi na każdym ze schodków, przy czym najwyższy schodek ma numer 1, a najniższy N. Żaden z Twoich żołnierzy nie stoi na schodku nr 1.

Jeśli pozycja podana Twojemu programowi jest wygrywająca (tzn. istnieje strategia, która pozwala wygrać niezależnie od ruchów przeciwnika), Twój program powinien wygrać. W przeciwnym przypadku po prostu grać (i przegrać) poprawnie.

To jest program interakcyjny. Zagrasz przeciwko bibliotece zgodnej z poniższą specyfikacją. W każdej rundzie Twój program ma wskazać bibliotece grupę żołnierzy. Biblioteka zwróci 1 lub 2 określając, która grupa żołnierzy ma pozostać (1 oznacza grupę przez Ciebie wskazaną, a 2 resztę żołnierzy). Gdy gra się skończy (wygrałeś albo nie masz już żołnierzy), biblioteka poprawnie zakończy Twój program. Twój program nie może się kończyć w żaden inny sposób.

Interfejs biblioteki

Biblioteka `libconq` *udostępnia dwa podprogramy:*

- `start` — *zwraca liczbę N oraz wypełnia tablicę* `stairs` *liczbami żołnierzy stojących na schodach (tzn. na schodku* `i` *stoi* `stairs[i]` *żołnierzy).*
- `step` — *pobiera tablicę* `subset` *(z co najmniej N elementami[1]), w której zapisano wybraną przez Ciebie grupę żołnierzy.* `step` *zwraca 1 lub 2 zgodnie z powyższą specyfikacją. Grupę żołnierzy podaje się poprzez wskazanie liczby żołnierzy z każdego schodka, tak jak w wypadku funkcji* `start`.

Jeśli wskażesz niepoprawną grupę żołnierzy, gra zostanie przerwana, a Twój program otrzyma 0 punktów za ten konkretny test. **Pamiętaj, że schodki są ponumerowane od 1 także w przypadku C/C++.**

Oto deklaracje tych podprogramów we FreePascalu i w C/C++:

```
procedure start(var N : longint; var stairs : array of longint);
function step(subset : array of longint): longint;

void start(int *N, int *stairs);
int step(int *subset);
```

Poniżej znajdziesz przykład użycia biblioteki zarówno we FreePascalu jak i w C/C++. Oba fragmenty robią to samo: rozpoczynają grę i potem grają jedną turę. Wskazana grupa zawiera wszystkich żołnierzy z losowo wskazanych schodków. Twój prawdziwy program prawdopodobnie powinien rozgrywać tę grę w pętli nieskończonej.

Gorąco zachęcamy Cię do zdefiniowania tablic `stairs` *i* `subset` *dokładnie w taki sam sposób, jak zrobiliśmy to w poniższych przykładach. Zauważ, że biblioteka we FreePascalu zawsze zwraca wynik w pierwszych N elementach tablicy, niezależnie od tego, jak ją zdefiniowałeś. Biblioteka w C/C++ zwraca wynik w elementach o indeksach od 1 do N.*

Przykład dla FreePascala:

[1] Z co najmniej $N+1$ elementami w przypadku C/C++ — wyjaśniono to poniżej.

```
uses libconq;
var stairs: array[1..2000] of longint;
    subset: array[1..2000] of longint;
    i,N,result: longint;

...
start(N,stairs);
...
for i:=1 to N do
  if random(2)=0 then subset[i]:=0;
  else subset[i]:=stairs[i];
result:=step(subset);
...
```

Przykład dla C/C++:

```
#include "libconq.h"
int stairs[2001];
int subset[2001];
int i,N,result;

...
start(&N, stairs);
...
for (i=1;i<=N;i++)
  if (rand()%2==0) subset[i]=0;
  else subset[i]=stairs[i];
result=step(subset);
...
```

Musisz skonsolidować Twój program z biblioteką poprzez frazę `uses libconq;` *we FreePascalu, albo poprzez frazę* `#include "libconq.h"` *w C/C++ i dodanie* `libconq.c` *do argumentów kompilacji.*

Przykład gry

Ty:	*Biblioteka:*
start(N,stairs)	N=8, stairs=(0,1,1,0,3,3,4,0)
step(0,1,0,0,1,0,1,0)	zwraca 2
step(0,1,0,0,0,1,0,0)	zwraca 2
step(0,0,0,3,2,0,0,0)	zwraca 1
step(0,0,2,0,0,0,0,0)	zwraca 2
step(0,1,0,0,0,0,0,0)	zwraca 2
step(0,1,0,0,0,0,0,0)	bez powrotu: wygrałeś!

Zasoby

W witrynie WWW możesz znaleźć przykładowe biblioteki zarówno dla C/C++, jak i dla FreePascala. Te biblioteki są inne niż te, których użyjemy w czasie testowania. Możesz je wykorzystać, aby upewnić się, że Twoje wywołania funkcji bibliotecznych są poprawne. Przykładowa biblioteka czyta dane wejściowe z pliku `libconq.dat`*, który składa się z dwóch wierszy. W pierwszym wierszu zapisano liczbę schodków N. W drugim znajduje się N liczb całkowitych — są to liczby żołnierzy na schodkach od 1 do N.*

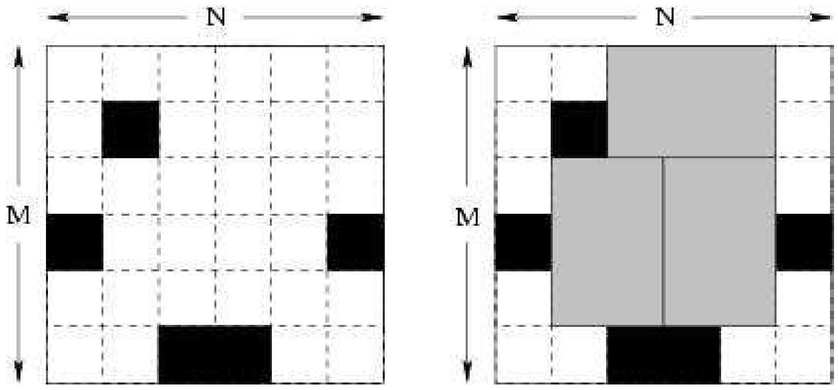
Plik `libconq.dat` *dla powyższego przykładu wyglądałby tak:*

```
8
0 1 1 0 3 3 4 0
```

Krzysztof Diks, Krzysztof Stencel
Tłumaczenie

Bugs Integrated, Inc.

Bugs Integrated, Inc. jest największym producentem nowoczesnych kości pamięci. Właśnie rozpoczynają produkcję nowej, sześcioterobajtowej kości Q-RAM. Każda kość składa się z sześciu jednostkowych kwadratów tworzących prostokąt o wymiarach 2×3. Produkcja kości Q-RAM odbywa się w następujący sposób: prostokątna płytka jest dzielona na $N \times M$ jednostkowych kwadratów. Następnie każdy z kwadratów jest bardzo uważnie testowany i wszystkie uszkodzone kwadraty są zaczerniane.



Na koniec płytka krzemowa zostaje pocięta na kości pamięci. Każda kość składa się z 2×3 (lub 3×2) jednostkowych kwadratów. Oczywiście żadna kość nie może zawierać kwadratu uszkodzonego (zaczernionego). Może się zdarzyć, że danej płytki nie można pociąć tak, żeby każdy nieuszkodzony kwadrat był fragmentem jakiejś kości. Kierownictwo przedsiębiorstwa chce, żeby jak najmniejsza liczba dobrych (nieuszkodzonych) kwadratów była marnowana, dlatego ważne jest, w jaki sposób pociąć płytkę, żeby dostać jak największą liczbę kości pamięci.

Zadanie

Dany jest zestaw płytek wraz z listami uszkodzonych kwadratów na każdej z nich. Napisz program, który policzy dla każdej płytki maksymalną liczbę kości, które można z niej wyciąć.

Wejście

Pierwszy wiersz pliku wejściowego zawiera dokładnie jedną liczbę D ($1 \leqslant D \leqslant 5$) oznaczającą liczbę płytek krzemowych. Następnie zapisano D bloków danych, każdy opisujący jedną płytkę. Pierwszy wiersz każdego bloku zawiera trzy liczby całkowite N ($1 \leqslant N \leqslant 150$), M ($1 \leqslant M \leqslant 10$), K ($0 \leqslant K \leqslant MN$) pooddzielane pojedynczymi odstępami: N jest długością płytki, M jej wysokością, a K jest liczbą uszkodzonych kwadratów na płytce. W kolejnych K wierszach opisano uszkodzone kwadraty. Każdy wiersz składa się z dwóch liczb całkowitych x i y ($1 \leqslant x \leqslant N$, $1 \leqslant y \leqslant M$) — współrzędnych jednego, uszkodzonego kwadratu (górny lewy kwadrat ma współrzędne $[1, 1]$, natomiast dolny prawy — $[N, M]$).

Wyjście

Dla każdej płytki z pliku wejściowego należy wypisać jeden wiersz zawierający największą liczbę kości pamięci, które można wyciąć z tej płytki.

Przykład

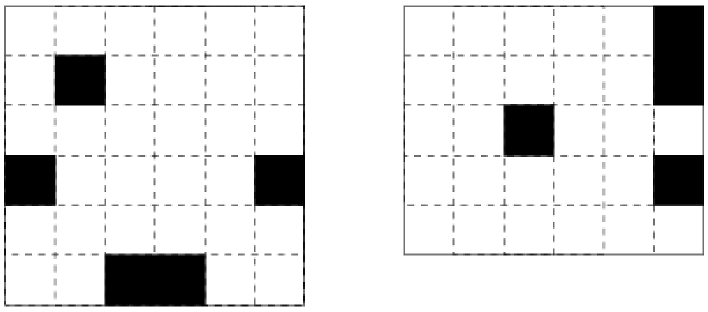
Wejście:

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

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

Wyjście:

```
3
4
```



Krzysztof Diks, Krzysztof Stencel
Tłumaczenie

Fikuśny płotek (A decorative fence)

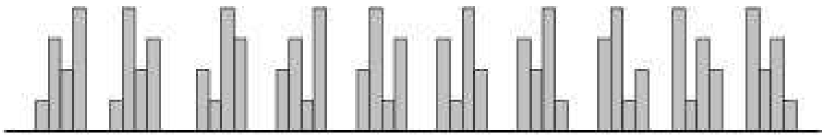
Rysiek właśnie zakończył budowę swojego nowego domu. Jedyne, co mu pozostało, to ogrodzić go fikuśnym drewnianym płotkiem. Niestety nie ma pojęcia, jak taki płotek ma wyglądać, więc postanowił kupić gotowy. Jeden z przyjaciół dał mu katalog GotowePłotki 2002 z przebogatą ofertą fikuśnych płotków. Po dokładnym przejrzeniu katalogu wiedział już, co chce.

Fikuśny płotek jest zbudowany z N drewnianych palików ustawionych pionowo w rzędzie.

- *Paliki mają różne długości — 1, 2, ... lub N jednostek.*
- *Każdy palik umieszczony między dwoma innym (sąsiednimi) jest albo większy, albo mniejszy od obu sąsiednich palików. (Zauważ, że wówczas wysokość płotka naprzemiennie rośnie i maleje.)*

Wynika stąd, że każdy fikuśny płotek składający się z N palików można jednoznacznie opisać jako permutację $a_1, \ldots, a_N$ liczb 1, ..., N taką, że $\forall_{1<i<N}(a_i - a_{i-1})(a_i - a_{i+1}) > 0$. Na odwrót każda taka permutacja opisuje pewien fikuśny płotek.

Oczywiście z N palików można zbudować wiele różnych fikuśnych płotków. Wydawca katalogu GotowePłotki 2002 postanowił uporządkować oferty płotków w następujący sposób: Płotek A (reprezentowany przez permutację $a_1, \ldots, a_N$) występuje w katalogu przed płotkiem B (reprezentowanym przez $b_1, \ldots, b_N$) wtedy i tylko wtedy, gdy istnieje takie i, że dla każdego $j < i$, $a_j = b_j$ oraz $a_i < b_i$. (Żeby stwierdzić, który z dwóch płotków występuje w katalogu wcześniej wystarczy wziąć odpowiadające im permutacje, znaleźć pierwsze miejsce, na którym się różnią i porównać wartości z tego miejsca.) Wszystkie fikuśne płotki zbudowane z N palików są ponumerowane (poczynając od 1) w kolejności ich wystąpień w katalogu. Taki numer nazywamy **numerem katalogowym**.



Oto wszystkie fikuśne płotki zbudowane z $N = 4$ palików uporządkowane według numerów katalogowych.

Po dokładnym obejrzeniu wszystkich fikuśnych płotków Rysiek postanowił zamówić niektóre z nich. Dla każdego zamówienia zapisał liczbę palików w płotku i numer katalogowy. Podczas wesołego wieczorku z przyjaciółmi chciał pokazać im zamówione płotki, ale zagubił gdzieś katalog. Jedyne, czym dysponował, to swoje notatki. Pomóż Ryśkowi i pokaż, jak wyglądają zamówione przez niego płotki.

Wejście

Pierwszy wiersz zawiera tylko jedną liczbę K ($1 \leqslant K \leqslant 100$) — liczbę zestawów danych wejściowych. Każdy z kolejnych K wierszy opisuje jeden zestaw danych wejściowych.

Każdy z tych wierszy zawiera dwie liczby całkowite N i C ($1 \leqslant N \leqslant 20$), oddzielone pojedynczym odstępem. Liczba N jest liczbą palików w płotku, a liczba C jest numerem katalogowym.

Możesz założyć, że liczba wszystkich fikuśnych płotków zbudowanych z 20 palików mieści się w zmiennej 64-bitowej ze znakiem (`long long` w C/C++, `int64` we FreePascalu). Można także przyjąć, że dane wejściowe są poprawne, w szczególności C wynosi co najmniej 1 i nie przekracza liczby fikuśnych płotków, które można zbudować z N palików

Wyjście

Dla każdego zestawu danych wejściowych należy wypisać jeden wiersz opisujący C-ty płotek w katalogu płotków, zbudowanych z N palików. Dokładniej, jeśli płotek opisuje permutacja $a_1, \ldots, a_N$, wówczas odpowiedni wiersz pliku wyjściowego powinien zawierać liczby a_i (we właściwej kolejności), pooddzielane pojedynczymi odstępami.

Przykład

Wejście:

```
2
2 1
3 3
```

Wyjście:

```
1 2
2 3 1
```

Królewscy strażnicy (Royal guards)

Dawno, dawno temu, w odległej galaktyce było sobie królestwo. Miało ono wszystko, co królestwu jest potrzebne, tzn. króla i jego zamek. Zamek miał plan prostokąta podzielonego na $M \times N$ jednostkowych kwadratów. Niektóre z tych kwadratów były ścianami, a niektóre były puste. Każdy z pustych kwadratów nazwiemy **komnatą**. *Król tego fikuśnego królestwa był wyjątkowym „świrem". Postanowił więc zbudować ukryte pułapki (z głodnymi aligatorami na dnie) w niektórych pokojach.*

To jednak nie wystarczyło szaleńcowi. Po tygodniu postanowił umieścić w zamku najwięcej strażników, jak to tylko możliwe. To nie było takie proste. Strażnicy byli wyszkoleni tak, aby natychmiast strzelać do zauważonych osób. Król musiał więc umieszczać strażników ostrożnie, ponieważ gdyby dwaj strażnicy mogli siebie zobaczyć, zastrzeliliby się nawzajem. Oczywiście król mimo swego szaleństwa nie chciał umieścić strażnika w komnacie z pułapką.

Dwaj strażnicy w komnatach widzą siebie nawzajem wtedy i tylko wtedy, gdy kwadraty odpowiadające ich komnatom są w tym samym wierszu lub tej samej kolumnie prostokąta oraz nie ma między nimi ścian. (Strażnicy patrzą jedynie w czterech kierunkach, dokładnie tak, jak porusza się szachowa wieża.)

Zadanie

Twoim zadaniem jest ustalić, ilu maksymalnie strażników król może umieścić w zamku (zgodnie z powyższymi regułami), oraz znaleźć jedno możliwe ustawienie tylu strażników w komnatach.

Dane wejściowe

Pierwszy wiersz pliku wejściowego zawiera dwie liczby całkowite N i M ($1 \leqslant N, M \leqslant 200$) — są to wymiary prostokątnego planu zamku — i-ty z następnych M wierszy zawiera N liczb $a_{i,1}, \ldots, a_{i,N}$ pooddzielanych pojedynczymi znakami odstępu, przy czym:

- $a_{i,j} = 0$ *oznacza, że kwadrat $[i, j]$ jest pusty (komnata bez pułapki),*
- $a_{i,j} = 1$ *oznacza, że kwadrat $[i, j]$ zawiera pułapkę,*
- $a_{i,j} = 2$ *oznacza, że kwadrat $[i, j]$ jest ścianą.*

Uwaga! Pierwsza współrzędna kwadratu jest numerem wiersza, natomiast druga współrzędna to numer kolumny.

Dane wyjściowe

Pierwszy wiersz pliku wejściowego powinien zawierać liczbę K — największą możliwą liczbę strażników, których król może umieścić w zamku. Następne K wierszy powinny opisywać jedno z możliwych ustawień K strażników w pustych komnatach zamku tak, żeby żaden z nich nie widział żadnego innego.

Bardziej precyzyjnie mówiąc, i-ty z tych wierszy powinien zawierać dwie liczby całkowite r_i, c_i oddzielone pojedynczym odstępem — są to współrzędne komnaty, w której ma znaleźć się i-ty strażnik (r_i jest numerem wiersza, a c_i numerem kolumny).

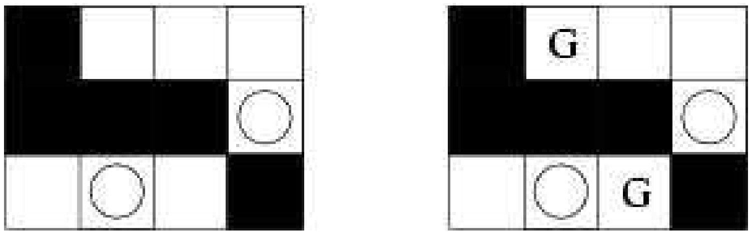
Przykład

Dane wejściowe:

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

Dane wyjściowe:

```
2
1 2
3 3
```



Zamek opisany przez przykładowe dane wejściowe i jedno z możliwych rozstawień strażników odpowiadające danym wyjściowym.

Krzysztof Diks, Krzysztof Stencel
Tłumaczenie

Przyjęcie urodzinowe (Birthday party)

Zbliża się dzień urodzin Jasia i jak co roku solenizant ma zamiar zorganizować przyjęcie urodzinowe. Chciałby, żeby na przyjęcie stawili się wszyscy jego przyjaciele, ale wie, że będzie to prawie niemożliwe. Na przykład w ostatnim tygodniu Zuzia rozstała się ze Stefkiem i prawie na pewno nie zgodzą się przyjść jednocześnie. Jasio spędził cały tydzień na rozmowach z przyjaciółmi zapraszając ich na przyjęcie. Część z osób odpowiedziała pozytywnie, ale większość wysunęła dodatkowe życzenie. (Jeśli zapraszasz mnie, to musisz także zaprosić mojego narzeczonego — zakomunikowała Wera. Jeśli zaprosisz bliźniaków od Witków, nie oczekuj, że przyjdziemy ja i Józek! — powiedział Pietrek.) Nagle Jasiek zorientował się, że spełnić wszystkie życzenia będzie prawie niemożliwe.

Zadanie

Dane są opisy życzeń, jakie otrzymał Jasiek od swoich przyjaciół. Twoje zadanie polega na znalezieniu takiej grupy ludzi, że jeśli Jasiek zaprosi na przyjęcie wszystkie osoby z tej grupy (i żadnych innych), wówczas wszystkie życzenia zostaną spełnione. Życzenia są opisane w następujący sposób:

- `nazwisko` *jest życzeniem. Takie życzenie jest spełnione wtedy i tylko wtedy, gdy Jasiek zaprosi* `nazwisko`.
- `-nazwisko` *jest życzeniem. Takie życzenie jest spełnione wtedy i tylko wtedy, gdy Jasiek nie zaprosi* `nazwisko`. *(W obu wypadkach* `nazwisko` *jest napisem złożonym z co najwyżej 20 małych liter bez odstępów.)*
- *Jeśli* `R1`, …, `Rk` *są życzeniami, wówczas* `(R1 & ... & Rk)` *jest życzeniem. Takie życzenie jest spełnione wtedy i tylko wtedy, gdy wszystkie życzenia* `R1`, …, `Rk` *są spełnione.*
- *Jeśli* `R1`, …, `Rk` *są życzeniami, wówczas* `(R1 | ... | Rk)` *jest życzeniem. Takie życzenie jest spełnione wtedy i tylko wtedy, gdy co najmniej jedno z życzeń* `R1`, …, `Rk` *jest spełnione.*
- `R1`, `R2` *są życzeniami, to* `(R1 => R2)` *jest życzeniem. Takie życzenie* **nie jest** *spełnione wtedy i tylko wtedy, gdy* `R1` *jest spełnione, natomiast* `R2` *nie jest spełnione.*

Dane wejściowe

W witrynie webowej znajdziesz 10 plików `party1.in`, …, `party10.in`. *Każdy plik jest wart 10 punktów.*

W pierwszym wierszu każdego pliku wejściowego znajduje się liczba przyjaciół Jasia — F. W każdym z kolejnych F wierszy znajduje się jedno nazwisko. Kolejny wiersz zawiera liczbę życzeń N. Każdy z następnych N wierszy zawiera jedno życzenie.

Dane wyjściowe

Dla każdego pliku wejściowego `partyX.in` *musisz utworzyć odpowiadający mu plik* `partyX.out` *zawierający jedno poprawne rozwiązanie. Pierwszy wiersz tego pliku powinien zawierać liczbę K osób, które Jasiek powinien zaprosić na przyjęcie. Kolejne K wierszy powinny zawierać nazwiska osób zaproszonych na przyjęcie — jedno nazwisko w jednym wierszu. Możesz przyjąć, że dla każdego pliku wejściowego istnieje (co najmniej jedno) rozwiązanie. Jeśli istnieje wiele rozwiązań, wystarczy, że wypiszesz tylko jedno z nich.*

Zgłaszanie rozwiązania

Rozwiązania zgłaszasz za pomocą serwisu webowego przekazując pliki `party*.out` *w taki sam sposób, w jaki zgłaszasz programy do oceny.*

Przykład

Dane wejściowe:

```
3
veronica
steve
dick
3
(veronica => dick)
(steve => -veronica)
(steve & dick)
```

Dane wyjściowe:

```
2
steve
dick
```

Bibliografia

[1] *I Olimpiada Informatyczna 1993/1994.* Warszawa–Wrocław, 1994.

[2] *II Olimpiada Informatyczna 1994/1995.* Warszawa–Wrocław, 1995.

[3] *III Olimpiada Informatyczna 1995/1996.* Warszawa–Wrocław, 1996.

[4] *IV Olimpiada Informatyczna 1996/1997.* Warszawa, 1997.

[5] *V Olimpiada Informatyczna 1997/1998.* Warszawa, 1998.

[6] *VI Olimpiada Informatyczna 1998/1999.* Warszawa, 1999.

[7] *VII Olimpiada Informatyczna 1999/2000.* Warszawa, 2000.

[8] *VIII Olimpiada Informatyczna 2000/2001.* Warszawa, 2001.

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

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

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

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

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

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

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

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

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

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

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

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

[21] W. Lipski, W. Marek. *Analiza kombinatoryczna.* PWN, Warszawa, 1986.

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

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

[24] B. Schieber, U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. *Siam Journal on Computing*, 17(6), December 1988.

[25] R. Sedgewick. *Algorithms.* Addison-Wesley, 1988.

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

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

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

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

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

Niniejsza publikacja stanowi wyczerpujące źródło informacji o zawodach IX Olimpiady Informatycznej, przeprowadzonych w roku szkolnym 2001/2002. Książka zawiera informacje dotyczące organizacji, regulaminu oraz wyników zawodów. Zawarto także opis rozwiązań wszystkich zadań konkursowych. Do książki dołączona jest dyskietka zawierająca wzorcowe rozwiązania i testy do wszystkich zadań Olimpiady.

Książka zawiera też zadania z VIII Bałtyckiej Olimpiady Informatycznej i IX Olimpiady Informatycznej Centralnej Europy.

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

Olimpiada Informatyczna
jest organizowana przy współudziale

PROKOM SOFTWARE SA

ISBN 83-917700-0-1