MINISTERSTWO EDUKACJI NARODOWEJ
INSTYTUT INFORMATYKI UNIWERSYTETU WROCŁAWSKIEGO
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

VIII OLIMPIADA INFORMATYCZNA
2000/2001

WARSZAWA, 2001

MINISTERSTWO EDUKACJI NARODOWEJ
INSTYTUT INFORMATYKI UNIWERSYTETU WROCŁAWSKIEGO
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

VIII OLIMPIADA INFORMATYCZNA
2000/2001

WARSZAWA, 2001

Autorzy tekstów:

prof. dr hab. Zbigniew Czech
dr hab. Krzysztof Diks
dr hab. Wojciech Guzicki
dr Marcin Jurdziński
dr Marcin Kubica
dr hab. Krzysztof Loryś
dr Adam Malinowski
mgr Marcin Mucha
Krzysztof Onak
prof. dr hab. Wojciech Rytter
mgr Marcin Sawicki
Tomasz Waleń

Autorzy programów na dyskietce:

Andrzej Gąsienica–Samek
mgr Marcin Mucha
Marek Pawlicki
Piotr Sankowski
mgr Marcin Sawicki
Marcin Stefaniak
Tomasz Waleń
Paweł Wolff

Opracowanie i redakcja:

dr hab. Krzysztof Diks
Tomasz Waleń

Skład: Tomasz Waleń

Pozycja dotowana przez Ministerstwo Edukacji Narodowej.

Druk książki został sfinansowany przez PROKOM SOFTWARE SA

ISBN 83–906301–7–6

Spis treści

Wstęp

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

Już po opublikowaniu sprawozdań z VII Olimpiady odbyła się 12–ta Międzynarodowa Olimpiada Informatyczna (Pekin, Chiny, 23–30 września, 2000). W tych zawodach Polskę reprezentowali laureaci VII Olimpiady Informatycznej — Tomasz Czajka, Tomasz Malesiński, Krzysztof Onak i Grzegorz Stelmaszek. W zawodach wzięły udział 4–osobowe ekipy z 71 krajów. Wśród najlepszych młodych informatyków z całego świata nasi reprezentanci spisali się znakomicie. Złote medale zdobyli Tomasz Czajka i Krzysztof Onak, natomiast Tomasz Malesiński zdobył srebrny medal. Więcej o olimpiadzie w Pekinie można znaleźć pod adresem internetowym `www.ioi2000.org.cn`.

W tym roku Polska była organizatorem 7–ej Bałtyckiej Olimpiady Informatycznej (Sopot, 16–17 czerwca, 2001). W zawodach wzięli udział uczniowie z Danii, Estonii, Finlandii, Niemiec, Łotwy, Litwy, Polski i Szwecji. Z każdego kraju, z wyjątkiem Danii i Polski, przejechało po sześciu zawodników. Danię reprezentował jeden zawodnik, natomiast Polskę 12 zawodników — czołówka z finałów VIII Olimpiady Informatycznej. Kraje bałtyckie od lat zaliczają się do czołówki młodzieżowej informatyki. W tak doborowej stawce Polacy wypadli bardzo dobrze zdobywając 3 złote medale (Michał Adamaszek, Paweł Parys, Krzysztof Kluczek), 4 srebrne medale (Tomasz Malesiński, Karol Cwalina, Arkadiusz Pawlik, Piotr Stańczyk) i 2 brązowe medale (Marek Żylak i Marcin Michalski). Impreza została przeprowadzona bardzo sprawnie dzięki sprawdzonej ekipie organizatorów z olimpiady krajowej, jak i współorganizatorów: firm Prokom Software S.A. i Combidata Poland Sp. z o.o., oraz miasta Sopot. Więcej o samej imprezie można przeczytać pod adresem `www.ii.uni.wroc.pl/boi`.

W dniach 14–21 czerwca 2001 roku, w Tampere w Finlandii, odbyła się 13–ta Międzynarodowa Olimpiada Informatyczna. W Olimpiadzie udział wzięło 272 zawodników z 74 krajów. Polska była reprezentowana przez złotych medalistów VIII Olimpiady Informatycznej: Pawła Parysa, Tomasza Malesińskiego, Mateusza Kwaśnickiego i Karola Cwalinę. Każdy z naszych reprezentantów zdobył medal: Tomek — złoty, Paweł i Mateusz — srebrne, Karol — brązowy. Należy pokreślić dobrą passę naszych reprezentantów, którzy corocznie z Olimpiady Międzynarodowej przywożą medale, a od siedmiu lat zawsze są wśród nich medale złote. Sukcesy polskich olimpijczyków i doskonała organizacja przez Polskę imprez międzynarodowych zostały dostrzeżone i Polska uzyskała w Tampere prawo organizacji Międzynarodowej Olimpiady Informatycznej w roku 2005. Więcej o olimpiadzie w Tampere można znaleźć na stronach `www.ioi2001.edu.fi`.

Olimpiada w Tampere była ważnym etapem w rozwoju olimpiad informatycznych. Po raz pierwszy zawodnicy pracowali w środowisku linuksowym i korzystali z kompilatorów gcc i fpc. Zawody przebiegły gładko i wydaje się, że to nowe środowisko programistyczne na długo zagości na olimpiadach informatycznych.

Ten rok był także rewolucyjny w krajowej olimpiadzie. Po raz pierwszy na wielką skalę wykorzystaliśmy Internet do komunikacji z zawodnikami. W I etapie zawodnicy mogli zgłaszać swoje rozwiązania również przez Internet. W etapach II i III każdy mógł konkurować z najlepszymi korespondencyjnie przez Internet. Przy okazji Bałtyckiej Olimpiady Informatycznej przeprowadziliśmy wraz z Gazetę Wyborczą i firmą Prokom Software internetowe zawody programistyczne pod nazwą „Pogromcy Algorytmów”. Zawody cieszyły się dużą popularnością i myślimy o ich powtórzeniu. Olimpiada krajowa zmierza stopniowo w kierunku środowiska linuksowego. Takie środowisko było już dostępne w trzecim etapie.

Tak wielkie zmiany nie byłyby możliwe bez zaangażowania wielu osób współpracujących z Olimpiadą, w szczególności pracowników i studentów Instytutów Informatyki Uniwersytetów Warszawskiego i Wrocławskiego. Wszystkim serdecznie dziękuję.

Prezentowana książeczka zawiera zadania wraz z rozwiązaniami z VIII Olimpiady Informatycznej. Na dyskietce załączono programy wzorcowe i testy, które posłużyły do sprawdzenia rozwiązań zawodników. Przedstawiamy też zadania z tegorocznych olimpiad, bałtyckiej i międzynarodowej oraz olimpiady w Pekinie. 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 2001

Sprawozdanie z przebiegu VIII Olimpiady Informatycznej 2000/2001

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 2000/2001 odbyły się zawody VIII Olimpiady Informatycznej. Olimpiada Informatyczna jest trójstopniowa. Integralną częścią rozwiązania każdego zadania zawodów I, II i III stopnia jest program napisany w języku programowania wysokiego poziomu (Pascal, C, C++). Zawody I stopnia miały charakter otwartego konkursu przeprowadzonego dla uczniów wszystkich typów szkół młodzieżowych.

6 października 2000 r. rozesłano plakaty zawierające zasady organizacji zawodów I stopnia oraz zestaw 4 zadań konkursowych do 3350 szkół i zespołów szkół młodzieżowych ponadpodstawowych oraz do wszystkich kuratorów i koordynatorów edukacji informatycznej. Zawody I stopnia rozpoczęły się dnia 16 października 2000 roku. Ostatecznym terminem nadsyłania prac konkursowych był 13 listopada 2000 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, w dniach 6–8.02.2001r., natomiast zawody III stopnia odbyły się w ośrodku firmy Combidata Poland S.A. w Sopocie, w dniach 26–30.03.2001r.

Uroczystość zakończenia VIII Olimpiady Informatycznej odbyła się w dniu 30.03.2001r. w Zespole Szkół Handlowych w Sopocie.

SKŁAD OSOBOWY KOMITETÓW OLIMPIADY INFORMATYCZNEJ

Komitet Główny:

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

z–cy przewodniczącego:
- prof. dr hab. Maciej M. Sysło (Uniwersytet Wrocławski)
- dr Andrzej Walat (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 (Uniwersytet Wrocławski)
- dr hab. Jan Madey, prof. UW (Uniwersytet Warszawski)
- prof. dr hab. Wojciech Rytter (Uniwersytet Warszawski)
- mgr Krzysztof J. Święcicki (Ministerstwo Edukacji Narodowej)
- dr Maciej Ślusarek (Uniwersytet Jagielloński)
- dr 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, a Prezydium — 4 zebrania. 26 stycznia 2001r. przeprowadzono seminarium przygotowujące przeprowadzenie 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 (Uniwersytet Wrocławski)
z–ca przewodniczącego:
dr Helena Krupicka (Uniwersytet Wrocławski)
sekretarz:
inż. Maria Woźniak (Uniwersytet Wrocławski)
członkowie:
mgr Jacek Jagiełło (Uniwersytet Wrocławski)
dr Tomasz Jurdziński (Uniwersytet Wrocławski)
dr Przemysława Kanarek (Uniwersytet Wrocławski)
dr Witold Karczewski (Uniwersytet Wrocławski)

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

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

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

Górnośląski Komitet Okręgowy
przewodniczący:
prof. dr hab. Zbigniew Czech (Politechnika Śląska w Gliwicach)
z–ca przewodniczącego:
mgr inż. Stanisław 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)
z–ca 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 w Krakowie jest Instytut Informatyki Uniwersytetu Jagiellońskiego, ul. Nawojki 11 w Krakowie.

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:
Tomasz Czajka
Wojciech Dudek
mgr Marcin Mucha
Krzysztof Onak
Arkadiusz Paterek
Marek Pawlicki
mgr Marcin Sawicki
Piotr Sankowski
Marcin Stefaniak
Tomasz Waleń
Paweł Wolff

ZAWODY I STOPNIA

W VIII Olimpiadzie Informatycznej wzięło udział 1534 zawodników. Po dokładnym sprawdzeniu prac przez Jury wykryto 10 prac wykonanych przypuszczalnie niesamodzielnie. Wystosowano pismo do zawodników z prośbą o wyjaśnienia. Dwóch zawodników przysłało swoje wyjaśnienia. Komitet uznał je za wystarczające i zakwalifikował obu zawodników do kolejnych etapów. Pozostałych ośmiu zawodników zdyskwalifikowano.

Dwóch zawodników przysłało prace na uszkodzonych dyskietkach i nie zostali oni sklasyfikowani w zawodach I stopnia.

W zawodach I stopnia VIII Olimpiady Informatycznej sklasyfikowano 1524 zawodników.

Decyzją Komitetu Głównego Olimpiady do zawodów zostało dopuszczonych 13 uczniów z gimnazjów i 2 uczniów ze szkół podstawowych:

- Gimnazjum w Brwinowie: Łukasz Kidziński
- Gimnazjum nr 1 w Bydgoszczy: Marcin Możejko
- Gimnazjum nr 1 im. M. Konopnickiej w Gdyni: Piotr Kowalczyk, Remigiusz Modrzejewski
- Gimnazju nr 24 w Gdyni: Michał Duczmal, Filip Wolski, Bartosz Michałowski
- Gimnazjum w Gliwicach: Piotr Kupisiewicz
- Gimnazjum w Lublinie: Jakub Klimkiewicz
- Gimnazjum nr 34 w Łodzi: Bartosz Janiak
- Gimnazjum nr 3 w Poznaniu: Marcin Mikołajczak
- Gimnazjum nr 2 w Rzeszowie: Piotr Kaleta
- Gimnazjum nr 13 we Wrocławiu: Miłosz Kordecki
- S. P. w Krakowie: Robert Obryk
- S. P. w Warszawie: Paweł Marczewski

Z rozwiązaniami:

czterech zadań nadeszło	659 prac
trzech zadań nadeszło	631 prac
dwóch zadań nadeszło	175 prac
jednego zadania nadeszło	59 prac

Kolejność województw pod względem liczby uczestników była następująca:

mazowieckie	249
małopolskie	170
śląskie	158
pomorskie	126
dolnośląskie	112
kujawsko–pomorskie	102
wielkopolskie	100
łódzkie	90
podkarpackie	90
lubelskie	62
zachodniopomorskie	61
warmińsko–mazurskie	54
lubuskie	45
świętokrzyskie	39
podlaskie	34
opolskie	28

W zawodach I stopnia najliczniej reprezentowane były szkoły:

V L. O. im. A Witkowskiego w Krakowie	50 uczniów
III L. O. im. Marynarki Wojennej RP w Gdyni	48
VIII L. O. im. A. Mickiewicza w Poznaniu	32
XIV L. O. im. St. Staszica w Warszawie	32
I L. O. im. St. Staszica w Lublinie	19
IV L. O. im. T. Kościuszki w Toruniu	15
XIV L. O. im. Polonii Belgijskiej we Wrocławiu	15
VIII L. O. im. M. Skłodowskiej–Curie w Katowicach	14
V L. O. im. Ks. J. Poniatowskiego w Warszawie	13
VI L. O. im. J. i J. Śniadeckich w Bydgoszczy	12
II L. O. im. C. K. Norwida w Tychach	11
I L. O. im. M. Kopernika w Łodzi	10
L. O. im. Króla Władysława Jagiełły w Dębicy	10
V L. O. w Bielsku–Białej	10
XXVII L. O. im. T. Czackiego w Warszawie	10
Z. S. O. nr 1 im. St. Dubois w Koszalinie	10
I L. O. im. J. Słowackiego w Elblągu	9
I L. O. im. Ziemi Kujawskiej we Włocławku	9
III L. O. im. A. Mickiewicza we Wrocławiu	9
IV L. O. im. J. Korczaka w Olkuszu	9
V L. O. im. A. Struga w Gliwicach	9
VI L. O. im. T. Reytana w Warszawie	9
VI L. O. im. W. Sierpińskiego w Gdyni	9
I L. O. im. B. Krzywoustego w Głogowie	8
I L. O. im. T. Kościuszki w Legnicy	8
I. L. O. im. S. Żeromskiego w Ełku	8
II L. O. im. R. Traugutta w Częstochowie	8
VII L. O. im. K. K. Baczyńskiego we Wrocławiu	8
X L. O. im. Królowej Jadwigi w Warszawie	7
I L. O. im. A. Mickiewicza w Białymstoku	7
I L. O. w Bydgoszczy	7
II L. O. im. M. Skłodowskiej–Curie w Gorzowie Wlkp.	7
II L. O. w Słupsku	7
IV L. O. w Bydgoszczy	7
L. O. im. St. Małachowskiego w Płocku	7
V L. O. im. A. Asnyka w Szczecinie	7
VI L. O. im. J. Kochanowskiego w Radomiu	7
Zespół Szkół Technicznych w Ostrowie Wielkopolskim	7
I L. O. im. 1–go Maja w Bełchatowie	6

I L. O. im. B. Nowodworskiego w Krakowie	6
I L. O. im. Jana III Sobieskiego w Oławie	6
II L. O. im. J. Chreptowicza w Ostrowcu Świętokrzyskim	6
II L. O. im. Jana Hetmana Zamojskiego w Lublinie	6
II L. O. w Wałbrzychu	6
III L. O. im. M. Skłodowskiej–Curie w Opolu	6
IX L. O. im. C. K. Norwida w Częstochowie	6
L. O. w Żurominie	6
V L. O. im. S. Żeromskiego w Gdańsku	6
XIII L. O. im. L. Lisa–Kuli w Warszawie	6
XIII L. O. w Szczecinie	6
I L. O. im. B. Prusa w Siedlcach	5
I L. O. im. H. Sienkiewicza w Łańcucie	5
I L. O. im. M. Konopnickiej w Suwałkach	5
I L. O. im. M. Kopernika w Gdańsku	5
I L. O. im. Ruy Barbosa w Warszawie	5
I l. O. im. Z. Nałkowskiej w Wołominie	5
I L. O. w Jaśle	5
II L. O. im. J. Śniadeckiego w Kielcach	5
II L. O. im. M. Kopernika w Mielcu	5
IX L. O. im. K. Libelta w Poznaniu	5
Katolickie L. O. w Krakowie	5
Szkoła Przymierza Rodzin w Warszawie	5
V L. O. w Elblągu	5
VII L. O. w Zielonej Górze	5
XX L. O. im. J. Słowackiego w Łodzi	5
XXXV L. O. im. B. Prusa w Warszawie	5
Zespół Szkół Elektronicznych i Technicznych w Olsztynie	5
Zespół Szkół Elektrycznych w Gorzowie Wielkopolskim	5

Ogólnie najliczniej reprezentowane były miasta:

Warszawa	157	Olkusz	9
Kraków	100	Ostrowiec Św.	9
Gdynia	74	Legnica	9
Poznań	56	Siedlce	9
Wrocław	47	Stalowa Wola	9
Łódź	42	Ełk	8
Lublin	38	Łańcut	8
Bydgoszcz	36	Mielec	8
Toruń	27	Ostrów Wlkp.	8
Gdańsk	26	Słupsk	8
Szczecin	26	Suwałki	8
Dąbrowa Górnicza	25	Tarnów	8
Katowice	23	Wałbrzych	8
Częstochowa	21	Inowrocław	7
Rzeszów	19	Piotrków Tryb.	7
Gorzów Wlkp.	17	Sosnowiec	7
Kielce	17	Milanówek	6
Bielsko Biała	16	Nowy Sącz	6
Elbląg	16	Oława	6
Koszalin	16	Września	6
Gliwice	15	Żuromin	6
Olsztyn	15	Ciechanów	5
Włocławek	14	Jasło	5
Opole	14	Ostrołęka	5
Płock	14	Ostrzeszów	5
Zielona Góra	14	Pabianice	5
Tychy	12	Piła	5

Białystok	11	Przemyśl	5
Głogów	11	Racibórz	5
Dębica	10	Wołomin	5
Radom	10	Żory	5
Bełchatów	9	Żywiec	5

Zawodnicy uczęszczali do następujących klas

do klasy IV	szkoły podstawowej	1 zawodnik
do klasy VI	szkoły podstawowej	1 zawodnik
do klasy I	gimnazjum	4 zawodników
do klasy II	gimnazjum	9
do klasy I	szkoły średniej	132
do klasy II		339
do klasy III		498
do klasy IV		479
do klasy V		43

18 zawodników nie podało informacji do której klasy uczęszczają.
Zawodnicy najczęściej używali następujących języków programowania:

Pascal firmy Borland	822
C/C++ firmy Borland	611

Ponadto pojawiły się:

Borland Delphi	32
DJGPP	23
FPC	11
GNU C/C++	10
Visual C	9
Watcom C/C++	3
TMT Pascal	3

Komputerowe wspomaganie umożliwiło sprawdzenie prac zawodników kompletem 53 testów.
Poniższa tabela przedstawia liczby zawodników, którzy uzyskali określone liczby punktów za poszczególne zadania, w zestawieniu ilościowym i procentowym:

	Mapa		*Przedziały*		*Monocyfrowe reprezentacje*		*Gra w zielone*	
	liczba zawodn.	czyli	liczba zawodn.	czyli	liczba zawodn.	czyli	liczba zawodn.	czyli
100 pkt.	397	26,1%	202	27%	491	32,2%	20	1,3%
99–75 pkt.	121	7,9%	411	27%	169	11,1%	7	0,5%
74–50 pkt.	100	6,6%	442	29%	117	7,7%	13	0,9%
49–1 pkt.	793	52%	208	13,6%	623	40,9%	360	23,6%
0 pkt.	113	7,4%	261	17,1%	124	8,1%	1124	73,7%

W sumie za wszystkie 4 zadania:

SUMA	liczba zawodników	czyli
400 pkt.	12	0,9%
399–300 pkt.	210	13,8%
299–200 pkt.	398	26,1%
199–1 pkt.	876	57,4%
0 pkt.	28	1,8%

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

ZAWODY II STOPNIA

Do zawodów II stopnia zakwalifikowano 277 zawodników, którzy osiągnęli w zawodach I stopnia wynik nie mniejszy niż 279 pkt.

Zawody II stopnia odbyły się w dniach 6–8 lutego 2001 r. w pięciu stałych okręgach oraz w Sopocie:

- w Toruniu — 34 zawodników z następujących województw:
  - kujawsko–pomorskie (23)
  - warmińsko–mazurskie (9)
  - podlaskie (1)
  - wielkopolskie (1)
- we Wrocławiu — 59 zawodników z następujących województw:
  - dolnośląskie (16)
  - łódzkie (5)
  - opolskie (5)
  - śląskie (8)
  - wielkopolskie (25)
- w Warszawie — 69 zawodników z następujących województw:
  - lubelskie (5)
  - łódzkie (8)
  - mazowieckie (44)
  - podkarpackie (3)
  - świętokrzyskie (7)
  - warmińsko–mazurskie (1)
- w Krakowie — 50 zawodników z następujących województw:
  - małopolskie (44)
  - podkarpackie (6)
- w Katowicach — 21 zawodników z następujących województw:
  - śląskie (21)
- w Sopocie — 44 zawodników z następujących województw:
  - zachodniopomorskie (1)
  - pomorskie (32)
  - warmińsko–mazurskie (1)

W zawodach II stopnia najliczniej reprezentowane były szkoły:

V L.O. im. A. Witkowskiego w Krakowie	30 uczniów
III L.O. im. Marynarki Wojennej RP w Gdyni	25
XIV L.O. im. St. Staszica w Warszawie	19
VIII L.O. im. A. Mickiewicza w Poznaniu	15
VI L.O. im. J. i J. Śniadeckich w Bydgoszczy	10
XIV L. O. im. Polonii Belgijskiej we Wrocławiu	6
I L. O. im. St. Staszica w Lublinie	5
VI L. O. im. W. Sierpnińskiego w Gdyni	4
V L. O. w Bielsku Białej	4
IV L. O. im. T. Kościuszki w Toruniu	4
II L. O. im. M. Kopernika w Łodzi	4
IX L. O. im. C. K. Norwida w Częstochowie	3
I L. O. im. A. Osuchowskiego w Cieszynie	3

VIII L. O. im. M. Skłodowskiej–Curie w Katowicach	3
Z. S. Technicznych w Ostrowie Wielkopolskim	3
XIII L. O. w Szczecinie	3
V L. O. im. A. Asnyka w Szczecinie	3
I L. O. im. Ziemi Kujawskiej we Włocławku	3
III L. O. im. A. Mickiewicza we Wrocławiu	3
Z. S. O. Nr 2 w Wałbrzychu	3
VI L. O. im. T. Reytana w Warszawie	3

Ogólnie najliczniej reprezentowane były miasta:

Kraków	36	zawodników	Bielsko–Biała	4
Warszawa	36		Kielce	4
Gdynia	29		Olsztyn	4
Poznań	16		Opole	4
Bydgoszcz	13		Cieszyn	3
Wrocław	10		Gorzów Wlkp.	3
Szczecin	8		Koszalin	3
Łódź	7		Ostrów Wlkp.	3
Częstochowa	6		Wałbrzych	3
Katowice	5		Włocławek	3
Lublin	5		Żywiec	3
Toruń	5			

6 lutego odbyła się sesja próbna, na której zawodnicy rozwiązywali nie liczące się do ogólnej klasyfikacji zadanie "Gorszy Goldbach". W dniach konkursowych zawodnicy rozwiązywali zadania: "Spokojna komisja", "Wyspa", "Mrówki i biedronka" oraz "Podróż", każde oceniane maksymalnie po 100 punktów.

Czterech zawodników nie stawiło się na zawody.

Podczas zawodów okręgowych Jury wykryło dwie prace niesamodzielne. Po wyjaśnieniu wszystkich okoliczności Komitet podjął decyzję o zdyskwalifikowaniu jednego zawodnika, drugiemu udzielono upomnienia.

Do automatycznego sprawdzania 4 zadań konkursowych zastosowano łącznie 62 testy.

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

- Gorszy Goldbach

	liczba zawodników	czyli
100 pkt	19	6,9%
99–75 pkt	20	7,2%
74–50 pkt	18	6,5%
49–1 pkt	46	16,6%
0 pkt	174	62,8%

- Spokojna komisja

	liczba zawodników	czyli
100 pkt	0	0%
99–75 pkt	4	1,5%
74–50 pkt	8	2,9%
49–1 pkt	99	35,7%
0 pkt	166	59,9%

- Podróż

	liczba zawodników	czyli
100 pkt	16	5,8%
99–75 pkt	7	2,5%
74–50 pkt	12	4,3%
49–1 pkt	52	18,8%
0 pkt	190	68,6%

- Mrówki i biedronka

	liczba zawodników	czyli
100 pkt	0	0%
99–75 pkt	2	0,7%
74–50 pkt	15	5,4%
49–1 pkt	55	19,9%
0 pkt	205	74%

- Wyspa

	liczba zawodników	czyli
100 pkt	12	4,4%
99–75 pkt	1	0,4%
74–50 pkt	4	1,4%
49–1 pkt	4	1,4%
0 pkt	256	92,4%

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

SUMA	liczba zawodników	czyli
400 pkt.	0	0%
399–300 pkt.	1	0,4%
299–200 pkt.	9	3,3%
199–1 pkt.	163	58,8%
0 pkt.	104	37,5%

Zawodnikom przesłano listy z wynikami zawodów i dyskietkami zawierającymi ich rozwiązania i testy, na podstawie których oceniano prace.

Równocześnie z zawodami okręgowymi odbywał się Internetowy Konkurs Programistyczny, podczas którego uczestnicy rozwiązywali zadania olimpijskie, a następnie przesyłali swoje rozwiązania przez Internet.

Po sprawdzeniu tych rozwiązań Komitet Główny wyróżnił następujących zawodników, nagradzając ich książkami ufundowanymi przez WNT:

(1) Bartosz Nowierski (Politechnika Poznańska) z wynikiem 354 pkt.,

(2) Marcin Meinardi (Akademia Górniczo–Hutnicza w Krakowie) z wynikiem 184 pkt.

(3) Andrzej Szombierski (V L. O. im. A. Witkowskiego w Krakowie) z wynikiem 178 pkt.

(4) Grzegorz Swat (VI LO im. J. Kochanowskiego) z wynikiem 121 pkt

(5) Adam Dzedzej (Uniwersytet Gdański) z wynikiem 120 pkt

(6) Piotr Kowalski (Uniwersytet Warszawski) z wynikiem 92 pkt.

ZAWODY III STOPNIA

Zawody III stopnia odbyły się w ośrodku firmy Combidata Poland S.A. w Sopocie w dniach od 26 do 30 marca 2001 r.

W zawodach III stopnia wzięło udział 44 najlepszych uczestników zawodów II stopnia, którzy uzyskali wynik nie mniejszy niż 114 pkt. Zawodnicy pochodzili z następujących województw:

śląskie	10
małopolskie	9
mazowieckie	7
dolnośląskie	5
pomorskie	3
zachodniopomorskie	3
kujawsko–pomorskie	2
wielkopolskie	2
lubelskie	1
warmińsko–mazurskie	1
podlaskie	1

Niżej wymienione szkoły miały w finale więcej niż jednego zawodnika:

V L. O. im. A. Witkowskiego w Krakowie	8 zawodników
XIV L.O. im. St. Staszica w Warszawie	5
III L. O. im. Marynarki Wojennej RP w Gdyni	3
XIV L. O. im. Polonii Belgijskiej we Wrocławiu	4

26 marca odbyła się sesja próbna, na której zawodnicy rozwiązywali nie liczące się do ogólnej klasyfikacji zadanie: "Wędrowni treserzy pcheł". W dniach konkursowych zawodnicy rozwiązywali zadania: "Naszyjnik", "Zwiedzanie miasta" oceniane maksymalnie po 60 punktów, oraz "Bank", "Kopalnia złota" i "Łańcuch", każde oceniane maksymalnie po 40 punktów.

Zastosowano zestaw łącznie 92 testów.

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

- Naszyjnik

	liczba zawodników	czyli
60 pkt.	0	0%
59–40 pkt.	6	13,7%
39–20 pkt.	13	29,5%
19–1 pkt.	13	29,5%
0 pkt.	12	27,3%

- Zwiedzanie miasta

	liczba zawodników	czyli
60 pkt.	3	6,8%
59–40 pkt.	1	2,3%
39–20 pkt.	4	9,1%
19–1 pkt.	6	13,6%
0 pkt.	30	68,2%

- Bank

	liczba zawodników	czyli
40 pkt.	0	0%
39–30 pkt.	0	0%
29–20 pkt.	1	2,3%
19–1 pkt.	22	50%
0 pkt.	21	47,7%

- Kopalnia złota

	liczba zawodników	czyli
40 pkt.	0	0%
39–30 pkt.	0	0%
29–20 pkt.	2	4,5%
19–1 pkt.	40	90,9%
0 pkt.	2	4,5%

- Łańcuch

	liczba zawodników	czyli
40 pkt.	20	45,5%
39–30 pkt.	3	6,8%
29–20 pkt.	3	6,8%
19–1 pkt.	15	34,1%
0 pkt.	3	6,8%

W sumie za wszystkie 5 zadań:

SUMA	liczba zawodników	czyli
240 pkt.	0	0%
239–180 pkt.	1	2,3%
179–120 pkt.	3	6,8%
119–1 pkt.	40	90,9%
0 pkt.	0	0%

W dniu 30 marca 2001 roku, w sali gimnastycznej Zespołu Szkół Handlowych w Sopocie, ogłoszono wyniki finału VIII Olimpiady Informatycznej 2000/2001 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:

(1) Paweł Parys, L. O. im. St. Staszica w Tarnowskich Górach, laureat I miejsca, 180 pkt. (komputer — PROKOM; roczny abonament na książki — WNT)

(2) Tomasz Malesiński, Zespół Szkół Elektrycznych w Białymstoku, laureat I miejsca, 154 pkt. (komputer — PROKOM)

(3) Mateusz Kwaśnicki, III L. O. we Wrocławiu, laureat I miejsca, 131 pkt. (komputer — PROKOM)

(4) Karol Cwalina, XIV L. O. im. St. Staszica w Warszawie, laureat I miejsca, 130 pkt. (komputer — PROKOM)

(5) Michał Adamaszek, V L. O. w Bielsku–Białej, laureat II miejsca, 112 pkt. (drukarka laserowa — PROKOM)

(6) Bartosz Walczak, V L. O. im. A. Witkowskiego w Krakowie, laureat II miejsca, 102 pkt. (drukarka laserowa — PROKOM)

(7) Krzysztof Kluczek, L. O. im. S. Żeromskiego w Bartoszycach, laureat II miejsca, 98 pkt. (drukarka laserowa — PROKOM)

(8) Adam Fuksa, V L. O. im. A. Witkowskiego w Krakowie, laureat II miejsca, 88 pkt. (drukarka laserowa — PROKOM)

(9) Pawel Walter, V L. O. im. A. Witkowskiego w Krakowie, laureat II miejsca, 86 pkt. (drukarka laserowa — PROKOM)

(10) Arkadiusz Pawlik, V L. O. im. A. Witkowskiego w Krakowie, laureat III miejsca, 84 pkt. (drukarka atramentowa — PROKOM)

(11) Tomasz Kmiecik, VIII L. O. im. M. Skłodowskiej–Curie w Katowicach, laureat III miejsca, 83 pkt. (drukarka atramentowa — PROKOM)

(11) Jakub Piędel, XIV L. O. im. St. Staszica w Warszawie, laureat III miejsca, 83 pkt. (drukarka atramentowa — PROKOM)

(12) Grzegorz Herman, V L. O. im. A. Witkowskiego w Krakowie, laureat III miejsca, 81 pkt. (drukarka atramentowa — PROKOM)

(13) Grzegorz Gutowski, V L. O. im. A. Witkowskiego w Krakowie, laureat III miejsca, 76 pkt. (drukarka atramentowa — PROKOM)

(14) Grzegorz Stelmaszek, XIV L. O. im. Polonii Belgijskiej we Wrocławiu, laureat III miejsca, 71 pkt. (drukarka atramentowa — PROKOM)

(15) Piotr Stańczyk, XIV L. O. im. St. Staszica w Warszawie, laureat III miejsca, 68 pkt. (drukarka atramentowa — PROKOM)

(16) Marcin Michalski, III L. O. im. Marynarki Wojennej RP w Gdyni, laureat III miejsca, 65 pkt. (drukarka atramentowa — PROKOM)

(17) Jakub Żytka, I L. O. im. E. Dembowskiego w Gliwicach, laureat III miejsca, 63 pkt. (drukarka atramentowa — PROKOM)

Wszyscy finaliści otrzymali książki ufundowane przez WNT, a ci którzy nie byli laureatami otrzymali upominki ufundowane przez Ogólnopolską Fundację Edukacji Komputerowej. Wszystkim laureatom i finalistom wysłano przesyłki zawierające dyskietki z ich rozwiązaniami oraz testami, na podstawie których oceniono ich prace.

Ogłoszono komunikat o powołaniu reprezentacji Polski na:

- Olimpiadę Informatyczną Centralnej Europy w składzie:

(1) Paweł Parys
(2) Karol Cwalina
(3) Bartosz Walczak
(4) Adam Fuksa

zawodnikami rezerwowoymi zostali:

(5) Arkadiusz Pawlik
(6) Tomasz Kmiecik

- Międzynarodową Olimpiadę Informatyczną w składzie:

(1) Paweł Parys
(2) Tomasz Malesiński
(3) Mateusz Kwaśnicki
(4) Karol Cwalina

zawodnikami rezerwowymi zostali:

(5) Michał Adamaszek
(6) Bartosz Walczak

- Bałtycką Olimpiadę Informatyczną w składzie:
zespół I

(1) Paweł Parys
(2) Tomasz Malesiński
(3) Mateusz Kwaśnicki
(4) Karol Cwalina
(5) Michał Adamaszek
(6) Bartosz Walczak

zawodnikami rezerwowymi zostali:

(7) Krzysztof Kluczek
(8) Paweł Walter

zespół II

(1) Adam Fuksa
(2) Arkadiusz Pawlik
(3) Tomasz Kmiecik
(4) Piotr Stańczyk
(5) Marcin Michalski
(6) Marek Żylak

zawodnikami rezerwowymi zostali:

(7) Marcin Pilipczuk
(8) Roman Łomowski

- obóz czesko–polsko–słowacki: reprezentacja (wraz z rezerwowymi) na Międzynarodową Olimpiadę Informatyczną,

- obóz rozwojowo–treningowy im. A. Kreczmara dla finalistów Olimpiady Informatycznej: laureaci i finaliści Olimpiady, z pominięciem zawodników z ostatnich klas szkół średnich.

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

Sekretariat wystawił łącznie 18 zaświadczeń o uzyskaniu tytułu laureata i 26 zaświadczeń o uzyskaniu tytułu finalisty VIII 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:

- Ryszard Parys (BUTIH "EKOKAL", Kalety)
  - Paweł Parys (laureat I miejsca)
- Joanna Ewa Łuszcz (Zespół Szkół Elektrycznych w Białymstoku)
  - Tomasz Malesiński (laureat I miejsca)
- Halina Kwaśnicka (Politechnika Wrocławska)
  - Mateusz Kwaśnicki (laureat I miejsca)
- Andrzej Gąsienica–Samek (student Uniwersytetu Warszawskiego)
  - Karol Cwalina (laureat I miejsca)
  - Piotr Stańczyk (laureat III miejsca)
  - Jakub Piędel (laureat III miejsca)
  - Piotr Cerobski (finalista)
  - Marcin Pilipczuk (finalista)
- Anna Kowalska (V L. O. Bielsko–Biała)
  - Michał Adamaszek (laureat II miejsca)
- Andrzej Dyrek (Uniwersytet Jagielloński w Krakowie)
  - Bartosz Walczak (laureat II miejsca)
  - Adam Fuksa (laureat II miejsca)
  - Paweł Walter (laureat II miejsca)
  - Grzegorz Gutowski (laureat III miejsca)
  - Grzegorz Herman (laureat III miejsca)
  - Arkadiusz Pawlik (laureat III miejsca)
  - Andrzej Pezarski (finalista)
  - Michał Zmarz (finalista)
- Wojciech Kmiecik (Kopalnia Węgla Kamiennego "Wujek" w Katowicach)
  - Tomasz Kmiecik (laureat III miejsca)
- Ewa Stelmaszek (DC Edukacja we Wrocławiu)
  - Grzegorz Stelmaszek (laureat III miejsca)
- Ryszard Szubartowski (III L. O. im. Marynarki Wojennej RP w Gdyni)
  - Marcin Michalski (laureat III miejsca)
  - Piotr Stefaniak (finalista)
  - Dominik Wojtczak (finalista)
- Walenty Żytka (Politechnika Śląska w Gliwicach)
  - Jakub Żytka (laureat III miejsca)
- Michał Bartoszkiewicz (Akademia Medyczna we Wrocławiu)

  - Michał Bartoszkiewicz (finalista)
- Jolanta Bąk (Żywiec)
  - Michał Bąk (finalista)
- Mariusz Blank (Lucent Technologies w Bydgoszczy)
  - Kamil Blank (finalista)
- Elżbieta Burlaga (Zespół Szkół Mechaniczno–Elektrycznych w Żywcu)
  - Dariusz Karcz (finalista)
- Jan Chróścicki (I L. O. im. B. Prusa w Siedlcach)
  - Marek Żylak (finalista)
- Jolanta Dębińska–Banak (I L. O. im. J. Kasprowicza w Raciborzu)
  - Dawid Huczek (finalista)
- Anna Fąfera (I L. O. w Szczecinie)
  - Sławomir Kolasiński (finalista)
- Mirosław Kulik (I L. O. im. St. Staszica w Lublinie)
  - Michał Mirosław (finalista)
- Ewa Kutyłowska (Zespół Szkół nr 3 we Wrocławiu)
  - Jarosław Kutyłowski (finalista)
- Bronisław Machura (Katowice)

  Marcin Machura (finalista)
- Narmi Michejda (Warszawa)
  - Noe Michejda (finalista)
- Marek Noworyta (Centrozap S.A. w Katowicach)
  - Filip Noworyta (finalista)
- Leon Plebanek (III L. O. im. A. Mickiewicza w Tarnowie)
  - Wojciech Matyjewicz (finalista)
- Jan Przewoźnik ("Integracja" Gorzów Wlkp.)
  - Maciej Przewoźnik (finalista)
- Włodzimierz Raczek (V L. O. w Bielsku–Białej)
  - Bartosz Sułkowski (finalista)
- Aleksy Schubert (XIV L. O. im. St. Staszica w Warszawie)
  - Marcin Pilipczuk (finalista)
- Mateusz Smul (XIV L. O. im. Polonii Belgijskiej we Wrocławiu)
  - Paweł Gawrychowski (finalista)
- Krzysztof Stefański (VIII L. O. im. A. Mickiewicza w Poznaniu)
  - Grzegorz Sobański (finalista)
- Michał Szuman (XIII L. O. w Szczecinie)

  - Mateusz Greszta (finalista)
  - Michał Jaszczyk (finalista)

- Iwona Waszkiewicz (VI L. O. im. J. i J. Śniadeckich w Bydgoszczy)
  - Kamil Blank (finalista)
  - Roman Łomowski (finalista)

Zgodnie z rozporządzeniem MEN w sprawie olimpiad tylko wyróżnieni nauczyciele otrzymają nagrody pieniężne. Decyzją Komitetu Głównego Olimpiady Informatycznej wyróżniono następujące szkoły, z których w zawodach finałowych brało udział więcej niż dwóch zawodników:

- III L. O. im. Marynarki Wojennej RP w Gdyni (1 laureat i 2 finalistów),
- V L. O. im. A. Witkowskiego w Krakowie (6 laureatów i 2 finalistów),
- XIV L. O. im. St. Staszica w Warszawie (3 laureatów i 2 finalistów),
- XIV L. O. im. Polonii Belgijskiej we Wrocławiu (1 laureat i 3 finalistów).

Podobnie jak w II etapie, tak i w III został zorganizowany Internetowy Konkurs Programistyczny. Poniższa lista przedstawia uczestników wyróżnionych w tym konkursie, którzy otrzymali nagrody książkowe. Dodatkowo zawodnik z najlepszym wynikiem, uczeń szkoły średniej, został zaproszony do uczestnictwa w obozie rozwojowo–treningowym im. A. Kreczmara dla finalistów Olimpiady Informatycznej.

Lista wyróżnionych zawodników:

Piotr Balwierz	V Liceum Ogólnokształcące w Krakowie	79 pkt.
Bartłomiej Romański	XIV L.O. im. St. Staszica w Warszawie	66 pkt.
Przemysław Dzierżak	VI LO im. W. Sierpińskiego w Gdyni	62 pkt.
Piotr Jakóbczyk	Techniczne Zakłady Naukowe	61 pkt.
Bartosz Nowierski	Politechnika Poznańska	60 pkt.
Michał Czardybon	Politechnika Śląska	59 pkt.
Piotr Jańczyk	III LO im. Marynarki Wojennej w Gdyni	58 pkt.
Grzegorz Swat	VI LO im. J. Kochanowskiego w Radomiu	52 pkt.

Warszawa, 18 maja 2001 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 2000 roku

Zasady organizacji zawodów w roku szkolnym 2000/2001

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 2000/2001.

§1 WSTĘP

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

§2 ORGANIZACJA OLIMPIADY

(1) Olimpiadę przeprowadza Komitet Główny Olimpiady Informatycznej.

(2) Olimpiada Informatyczna jest trójstopniowa.

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

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

(5) Zawody I stopnia mają charakter otwarty i polegają na samodzielnym 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 250 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 — 16.10–13.11.2000 r.
  ogłoszenie wyników:
    w witrynie Olimpiady — 9.12.2000 r.,
    pocztą – 20.12.2000 r.
zawody II stopnia — 6–8.02.2001 r.
  ogłoszenie wyników:
    w witrynie Olimpiady — 24.02.2001 r.,
    pocztą – 3.03.2001 r.
zawody III stopnia — 26–30.03.2001 r.

§3 WYMAGANIA DOTYCZĄCE ROZWIĄZAŃ ZADAŃ ZAWODÓW I STOPNIA

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

- 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 13 listopada 2000 r. (decyduje data stempla pocztowego). Prosimy o zachowanie dowodu nadania przesyłki.

- poprzez witrynę Olimpiady o adresie www.oi.pjwstk.waw.pl, do godziny 12.00 (w południe) dnia 13 listopada 2000 r. 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.

Rozwiązania dostarczane w inny sposób nie będą przyjmowane.

(2) Ocena za rozwiązanie 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:

  (a) programu (tylko jednego) w postaci źródłowej i skompilowanej,
  (b) opisu algorytmu rozwiązania zadania z uzasadnieniem jego poprawności.

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 w pliku nazwanym SPIS.TRC
  - do każdego rozwiązanego zadania — programy w postaci źródłowej i skompilowanej oraz opis algorytmu zapisany w postaci pliku txt

  dyskietka powinna być oznaczona imieniem i nazwiskiem

- kartkę z następującymi danymi (tylko w przypadku korzystania ze zwykłej poczty):
  - imię i nazwisko
  - datę i miejsce urodzenia
  - dokładny adres zamieszkania i ewentualnie numer telefonu
  - nazwę, adres, województwo i numer telefonu szkoły oraz klasę, do której uczęszcza
  - nazwę i numer wersji użytego języka programowania
  - opis konfiguracji komputera, na którym rozwiązano zadania

(11) Poprzez witrynę o adresie `www.oi.pjwstk.waw.pl` można uzyskać odpowiedzi na pytania dotyczące Olimpiady. Pytania należy przysyłać na adres: `olimpiada@jack.oeiizk.waw.pl`. Komitet Główny może nie udzielić odpowiedzi na pytanie jedynie 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 XIII Międzynarodową Olimpiadę Informatyczną w 2001 roku na podstawie wyników zawodów III stopnia i regulaminu tej olimpiady. Szczegółowe zasady zostaną podane po otrzymaniu formalnego zaproszenia na XIII 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 II Obozie Naukowo Treningowym im. Antoniego Kreczmara, który odbędzie się w okresie wakacji 2001 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. Każdy uczestnik, który przeszedł do zawodów wyższego stopnia oraz dyrektor jego szkoły otrzymują informację o miejscu i terminie następnych zawodów.

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

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

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

(1) Aby Twoje rozwiązanie mogło zostać właściwie ocenione, zastosuj się do ustaleń zawartych w „Zasadach organizacji zawodów" i treściach zadań.

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

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

(4) Twój program powinien odczytywać plik wejściowy z bieżącego katalogu i zapisywać plik wyjściowy również do bieżącego katalogu.

(5) Twój program nie powinien oczekiwać na jakąkolwiek czynność, np. naciśnięcie klawisza, ruch myszą, wpisanie liczby lub litery.

(6) Dane testowe są bezbłędne, zgodne z warunkami zadania i podaną specyfikacją wejścia. Twój program nie musi tego sprawdzać.

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

(8) Staraj się dobrać taką metodę rozwiązania zadania, która jest nie tylko poprawna, ale daje wyniki w jak najkrótszym czasie i w możliwe małej pamięci.

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

Zawody I stopnia

Zawody I stopnia — opracowania zadań

Tomasz Śmigielski
Treść zadania

Marcin Mucha
Opracowanie, Program

Mapa gęstości

Dane są:

- *liczby naturalne* $n > r \geq 0$,
- F *— tabelka* $n \times n$ *wypełniona liczbami ze zbioru* $\{0, 1\}$*; kolumny i wiersze tabelki są ponumerowane od 1 do* n*; liczbę znajdującą się w* i*-tej kolumnie i* j*-tym wierszu tabelki oznaczamy przez* $F[i, j]$.

Jeśli $[i, j]$ *i* $[i', j']$ *są dwoma miejscami w tabelce* F*, to odległością między nimi nazywamy liczbę* $max\,(|i - i'|, |j - j'|)$.

Należy obliczyć tabelkę W, $n \times n$ *(do elementów tej tabelki odwołujemy się tak samo, jak do elementów tabelki* F*) taką, że* $W[i, j]$ *jest sumą wszystkich liczb z tabelki* F *leżących w odległości co najwyżej* r *od* $[i, j]$.

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `map.in` *liczby* n, r *oraz tabelkę* F,
- *obliczy tabelkę* W,
- *zapisze ją do pliku tekstowego* `map.out`.

Wejście

W pierwszym wierszu pliku tekstowego `map.in` *znajdują się dwie dodatnie liczby całkowite oddzielone pojedynczą spacją:* n *i* r*, gdzie* $0 \leq r < n \leq 250$.
W kolejnych n *wierszach znajduje się opis tabelki* F*. Każdy z tych wierszy zawiera* n *liczb ze zbioru* $\{0, 1\}$*, pooddzielanych pojedynczymi odstępami,* i*-ta liczba zapisana w* $j + 1$*-szym wierszu to* $F[i, j]$.

Wyjście

Plik tekstowy `map.out` *powinien zawierać dokładnie* n *wierszy, w* i*-tym wierszu powinny być zapisane kolejno wartości* $W[1, i] \ldots W[n, i]$ *pooddzielane pojedynczymi odstępami.*

Przykład

Dla pliku wejściowego `map.in`*:*

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

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

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

Rozwiązanie

W całym opracowaniu (które jest jednak nieco bardziej teoretyczne od programu) będziemy zakładali, że funkcja F jest określona na całym zbiorze $Z \times Z$, przy czym poza rozważanym kwadratem się zeruje. To pozwoli na nieco swobodniejsze posługiwanie się sumami, a rozważenie przypadków brzegowych w programie jest dość proste (jeśli przypadki brzegowe będą w którymś momencie istotne, to zostanie to odnotowane).

Rozwiązanie optymalne polega na dynamicznym policzeniu sum:

$$S[i,j] = \sum_{1 \leq x \leq i} \sum_{1 \leq y \leq j} F[x,y]$$

Poszukiwane wartości wyrażają się przez $S[i,j]$ następująco:

$$W[i,j] = S[i,j] - S[i-r-1,j] - S[i,j-r-1] + S[i-r-1,j-r-1].$$

Obliczanie sum $S[i,j]$ można wykonać w naturalny sposób w miejscu (tzn. w tablicy F) w czasie $O(n^2)$. Wyliczanie każdego $W[i,j]$ odbywa się już potem w czasie stałym. Tak więc cały algorytm ma złożoność $O(n^2)$. Program implementujący ten algorytm znajduje się w pliku `map.pas`.

Inne rozwiązania

Zawsze można $W[i,j]$ obliczyć przy pomocy czterech zagnieżdżonych pętli, czyli po prostu z definicji. Taki algorytm ma złożoność $O(n^2r^2)$ i nie należy do najszybszych. Jego implementacja znajduje się w pliku `map1.pas`.

Chwila zastanowienia wystarcza na znalezienie następującej sprytnej formuły:

$$W[i,j] = W[i,j-1] + \sum_{i-r \leq k \leq i+r} F[k,j+r] - \sum_{i-r \leq k \leq i+r} F[k,j-r-1]$$

Innymi słowy: jeśli przesuwamy kwadracik o jedno pole w prawo, to musimy jedną kolumnę odjąć, a drugą dodać. Wystarczy więc "faktycznie" policzyć $W[i,j]$ dla $j = 1$, a dalej możemy już w czasie $2r$ korzystać z powyższej formuły. Jaki jest łączny czas działania takiego algorytmu? Na "faktyczne obliczenia" — $O(nr^2)$. Na poprawki $O(n^2r)$. Łącznie $O(n^2r)$. Program implementujący ten algorytm znajduje się w pliku `map2.pas`.

Oczywiście analogiczną formułę można zastosować, aby policzyć $W[i,1]$ przy pomocy $W[i-1,1]$. Wtedy musielibyśmy "faktycznie" policzyć tylko $W[1,1]$. Wydaje się to jednak bezcelową komplikacją, skoro i tak program będzie miał złożoność $O(n^2r)$. Nie jest to jednak takie proste. Poprawka bowiem nie zawsze wymaga czasu $2r$. Czasami dodawana lub odejmowana kolumna jest poza zakresem i wtedy nic nie trzeba robić. Im większe r, tym częściej się to zdarza. W szczególności, jeśli r jest bardzo duże (np. $r = n-1$), to program "poprawiający" w pionie i w poziomie będzie działał w czasie $O(n^2)$, a "poprawiający" tylko w poziomie w czasie $O(n^3)$, z tego prostego powodu, że będzie musiał policzyć wszystkie $W[i,1]$. W związku z powyższym program "poprawiający" i w pionie i w poziomie można uznać za oddzielne rozwiązanie, a jego implementacja znajduje się w pliku `map3.pas` (choć pesymistycznie nie jest on szybszy od programu "poprawiającego" tylko w poziomie).

Testy

Wszystkie testy są losowe (nie bardzo widać, co mogłoby być lepsze). Różnią się rozmiarami, wartością r (z treści opracowania wynika, że generalnie najtrudniejsze testy to takie, dla których $r \approx n/2$) i czasem gęstością jedynek. Testy $1-4$ to testy poprawnościowe. W szczególności test 1 sprawdza odporność na $r = 0$, a test 3 na r bardzo duże. Dalej idą coraz trudniejsze testy wydajnościowe. Testy $11-13$ to najtrudniejsze testy jakie udało się wymyśleć. Po drodze jest test 8, który może przejść nawet słaby algorytm, o ile przypadek małej liczby jedynek rozważa oddzielnie. Test 10 jest z kolei celowo dobrany pod algorytm poprawiający w obu kierunkach (tak naprawdę na testach $11-13$ ten algorytm też radzi sobie nieco lepiej).

Wydaje się, że świetnie napisany i zoptymalizowany algorytm, działający w czasie $O(n^2r)$, przejdzie większość testów (być może nawet wszystkie). Oto charakterystyki poszczególnych testów:

- `map0.in` — test z treści zadania;
- `map1.in` — $n = 5$,$r = 0$;
- `map2.in` — $n = 10$, $r = 5$;
- `map3.in` $n = 20$, $r = 19$;

- `map4.in` $n = 50$, $r = 3$;
- `map5.in` $n = 100$, $r = 50$;
- `map6.in` $n = 150$, $r = 50$;
- `map7.in` $n = 200$, $r = 100$;
- `map8.in` $n = 250$, $r = 10$, rzadka macierz;
- `map9.in` $n = 250$, $r = 50$;
- `map10.in` $n = 250$, $r = 249$;
- `map11.in` $n = 250$, $r = 125$;
- `map12.in` $n = 250$, $r = 125$;
- `map13.in` $n = 250$, $r = 125$.

Wojciech Guzicki
Treść zadania, Opracowanie

Marek Pawlicki
Program

Przedziały

Dany jest ciąg n przedziałów domkniętych $[a_i; b_i]$, *gdzie* $i = 1, 2, \ldots, n$. *Suma tych przedziałów może być przedstawiona w postaci sumy parami rozłącznych przedziałów domkniętych. Zadanie polega na znalezieniu przedstawienia tej sumy w postaci sumy minimalnej liczby parami rozłącznych przedziałów domkniętych. Przedziały tworzące to przedstawienie należy zapisać w pliku wyjściowym w rosnącej kolejności. Mówimy, że dwa przedziały rozłączne* $[a; b]$ *i* $[c; d]$ *są ustawione w rosnącej kolejności wtedy i tylko wtedy, gdy* $a \leq b < c \leq d$.

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `prz.in` *opis ciągu przedziałów,*
- *wyznaczy parami rozłączne przedziały spełniające warunki zadania,*
- *zapisze wyznaczone przedziały w rosnącej kolejności do pliku tekstowego* `prz.out` .

Wejście

W pierwszym wierszu pliku tekstowego `prz.in` *znajduje się jedna liczba całkowita n, spełniająca nierówności* $3 \leq n \leq 50\,000$. *Jest to liczba przedziałów. W* $(i+1)$*-szym wierszu pliku,* $1 \leq i \leq n$, *znajduje się opis przedziału* $[a_i; b_i]$ *w postaci dwóch liczb całkowitych* a_i *i* b_i *oddzielonych pojedynczym odstępem, będących odpowiednio jego początkiem i końcem,* $1 \leq a_i \leq b_i \leq 1\,000\,000$.

Wyjście

W kolejnych wierszach pliku tekstowego `prz.out` *należy zapisać opisy znalezionych parami rozłącznych przedziałów. W każdym wierszu ma być zapisany opis jednego przedziału w postaci dwóch liczb całkowitych oddzielonych pojedynczym odstępem, będących odpowiednio początkiem i końcem tego przedziału. Przedziały w pliku wyjściowym powinny być zapisane w rosnącej kolejności.*

Przykład

Dla pliku wejściowego `prz.in`:

```
5
5 6
1 4
10 10
6 9
8 10
```

poprawną odpowiedzią jest plik wyjściowy `prz.out`:

```
1 4
5 10
```

Rozwiązanie

Przedziały $[a_i; b_i]$ sortujemy w kolejności wzrastania początków. Jako bieżący bierzemy pierwszy przedział. Niech będzie to przedział $[a; b]$. Niech kolejnym przedziałem będzie $[c; d]$. Jeśli $c \leq b$ i $d > b$, to modyfikujemy przedział bieżący: będzie nim odtąd $[a; d]$. Jeśli natomiast $c > b$, to zapisujemy przedział bieżący, a nowym przedziałem bieżącym będzie odtąd $[c; d]$. Po wyczerpaniu wszystkich przedziałów zapisujemy przedział bieżący.

Załóżmy, że dane o przedziałach zostały zapisane w tablicy `P` następującej postaci:

```
type
   Przedzial=record
     Poczatek, Koniec : longint;
  end
   Przedzialy=array[1.. n] of Przedzial;
var
   P: Przedzialy;
```

Załóżmy następnie, że tablica `P` jest już posortowana w kolejności wzrastania początków. Opisany wyżej algorytm można zapisać w postaci następującego fragmentu programu:

```
Biezacy:= P[1];
for i:= 2 to n do
  if ( P[ i]. Poczatek ≤ Biezacy.Koniec)
     and ( P[ i]. Koniec > Biezacy.Koniec) then
        Biezacy.Koniec := P[ i]. Koniec
  else if P[ i]. Poczatek > Biezacy.Koniec then begin
        Zapisz( Biezacy);
        Biezacy:= P[ i]
    end;
Zapisz( Biezacy);
```

Procedura `Zapisz` zapisuje dane o przedziale `Biezacy` do pliku wyjściowego.

Jak widać, zadanie to jest dość łatwe. Jednak przy implementacji tego prostego algorytmu mogą wystąpić pewne trudności. Spróbujemy je teraz omówić.

1. Czas działania programu zależy od wyboru procedury sortującej. Nie będziemy tu omawiać różnych sposobów sortowania; algorytmy sortujące były wielokrotnie opisywane w sprawozdaniach z wcześniejszych Olimpiad Informatycznych, można też je znaleźć w wielu popularnych podręcznikach. Wspomnimy tu tylko o tym, że wybór wolno działającej procedury sortującej (np. sortowanie przez wybieranie, sortowanie przez wstawianie, sortowanie bąbelkowe – działających w czasie $O(n^2)$) spowoduje przekroczenie dopuszczalnego limitu czasu dla danych zapisanych w plikach `PRZ7.IN` i `PRZ8.IN`. Zadanie można też rozwiązać z pominięciem sortowania, dołączając kolejny przedział w odpowiednim miejscu do aktualnej listy przedziałów parami rozłącznych. Taki program też będzie działał na ogół w czasie $O(n^2)$. Testy sprawdzające zostały dobrane w taki sposób, by odróżnić rozwiązania wykorzystujące sortowanie w czasie kwadratowym od rozwiązań wykorzystujących sortowanie w czasie $O(n \log n)$ (sortowanie szybkie lub sortowanie stogowe) lub sortowanie w czasie $O(n)$ (sortowanie pozycyjne). Program wzorcowy `PRZ.PAS` używa właśnie sortowania pozycyjnego.

2. Druga trudność polega na braku pamięci. Jeśli dysponujemy kompilatorem dopuszczającym używanie dowolnie długich tablic, to z tą trudnością się nie zetkniemy. Jeśli jednak użyjemy np. kompilatora *Turbo Pascala*, to nie będziemy mogli zadeklarować jednej tablicy zawierającej dane o wszystkich przedziałach. Mianowicie początek i koniec każdego przedziału jest liczbą typu *longint*; zajmuje więc 4 bajty. Dane o każdym przedziale zajmują więc 8 bajtów. Maksymalna liczba przedziałów (50 tysięcy) będzie wymagać 400 000 bajtów. Te dane musimy zapisać w kilku tablicach, używając do tego zmiennych dynamicznych. Można teraz wybrać jedno z kilku rozwiązań. Pierwsze polega na posortowaniu każdej tablicy oddzielnie i wybieraniu jako kolejnego przedziału najmniejszego z przedziałów najmniejszych w tych tablicach (podobnie jak czyni się to w procedurze sortowania przez łączenie). Drugie rozwiązanie polega na traktowaniu tych kilku tablic jak jednej długiej tablicy. Zamiast odwołania `P[i]` do i-tego miejsca w tablicy `P` musimy teraz za każdym razem **obliczyć**, do którego miejsca w której tablicy się odwołujemy. Zaprogramowanie jednoczesnego sortowania takich tablic wymaga trochę staranności, ale nie jest bardzo trudne.

Do tego zadania ułożono 8 testów:

- `prz1.in`: trzy przedziały $[1; 1]$;
- `prz2.in`: 10 przedziałów zachodzących na siebie;

- `prz3.in`: 10 przedziałów rozłącznych, zapisanych w odwrotnej kolejności;
- `prz4.in`: 20 przedziałów takich, że każdy następny zawiera poprzedni;
- `prz5.in`: 6 przedziałów stykających się brzegami;
- `prz6.in`: 1000 losowo wybranych przedziałów;
- `prz7.in`: 20000 losowo wybranych przedziałów;
- `prz8.in`: 50000 losowo wybranych przedziałów.

Wszystkie rozwiązania poprawne (również działające w czasie kwadratowym) przechodziły przez testy od 1 do 6; przez ostatnie dwa testy przechodziły tylko programy wykorzystujące szybsze algorytmy sortowania.

Wojciech Rytter
Treść zadania, Opracowanie

Marcin Stefaniak
Program

Liczby antypierwsze

Oznaczmy przez LiczbaDzielników*(i) liczbę dzielników liczby i (włącznie z 1 i i). Dodatnią liczbę całkowitą nazywamy* **antypierwszą***, gdy ma ona więcej dzielników niż wszystkie dodatnie liczby całkowite mniejsze od niej. Przykładowymi liczbami antypierwszymi są liczby: 1, 2, 4, 6, 12 i 24.*

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `ant.in` *dodatnią liczbę całkowitą n,*
- *wyznaczy największą liczbę antypierwszą nie przekraczającą n,*
- *zapisze wyznaczoną liczbę w pliku tekstowym* `ant.out`*.*

Wejście

W jedynym wierszu pliku tekstowego `ant.in` *znajduje się jedna liczba całkowita n,* $1 \leq n \leq 2\,000\,000\,000$*.*

Wyjście

W jedynym wierszu pliku `ant.out` *Twój program powinien zapisać dokładnie jedną liczbę całkowitą — największą liczbę antypierwszą nie przekraczającą n.*

Przykład

Dla pliku wejściowego `ant.in`*:*
`1000`
poprawną odpowiedzią jest plik wyjściowy `ant.out`*:*
`840`

Rozwiązanie

Efektywne rozwiązanie zadania opiera się na prostych własnościach liczb antypierwszych:

- liczb antypierwszych mniejszych od zadanego n jest bardzo mało;
- liczby antypierwsze mają bardzo proste rozkłady względem liczb pierwszych;
- liczby pierwsze występujące w tych rozkładach są małe.

Aby obliczyć liczbę dzielników liczby $n > 1$, można rozłożyć ją na czynniki pierwsze. Jeśli

$$n = p_1^{a_1} * p_2^{a_2} * \ldots * p_k^{a_k}$$

to

$$\mathit{LiczbaDzielników}(n) = (a_1 + 1) * (a_2 + 1) * \ldots * (a_k + 1).$$

Widać, że liczba dzielników danej liczby nie zależy od wielkości liczb pierwszych występujących w jej rozkładzie na czynniki pierwsze, a jedynie od krotności, z jaką do tego rozkładu wchodzą.

Przykład
Przykładowymi liczbami antypierwszymi są liczby 1, 2, 4, 6, 12, 24 oraz *duże* liczby:

$$27935107200 = 2^7 3^3 5^2 7^1 11^1 13^1 17^1 19^1$$

$$2248776129600 = 2^6 3^3 5^2 7^2 11^1 13^1 17^1 19^1 23^1$$

$$LiczbaDzielników(27935107200) = 8 \cdot 4 \cdot 3 \cdot 2 \cdot 2 \cdot 2 \cdot 2$$
$$LiczbaDzielników(2248776129600) = 7 \cdot 4 \cdot 3 \cdot 3 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2$$

Przyjmijmy, że przez p_i rozumiemy i-tą kolejną liczbę pierwszą, a przez a_i rozumiemy krotność p_i w rozkładzie n na czynniki pierwsze (możliwe, że $a_i = 0$).

Jeżeli ciąg liczb a_i nie jest nierosnący, to n nie może być liczbą antypierwszą, gdyż wtedy moglibyśmy tak zamienić miejscami pewne krotności, że otrzymalibyśmy mniejszą liczbę o tej samej liczbie dzielników. Dlatego możemy ograniczyć poszukiwania liczb antypierwszych do liczb o „nierosnących" rozkładach na czynniki pierwsze. Łatwo możemy wygenerować wszystkie takie rozkłady dla liczb nie przekraczających danego ograniczenia górnego (kolejne rozkłady wyznaczamy w porządku leksykograficznym).

Jak się okazuje, liczb antypierwszych $\leq 2\,000\,000\,000$ jest niedużo, dokładnie 68. Pierwsze 41 liczb antypierwszych przedstawiono w poniższej tabeli, razem z potęgami w rozkładach na kolejne liczby pierwsze.

2	1					
4	2					
6	1	1				
12	2	1				
24	3	1				
36	2	2				
48	4	1				
60	2	1	1			
120	3	1	1			
180	2	2	1			
240	4	1	1			
360	3	2	1			
720	4	2	1			
840	3	1	1	1		
1260	2	2	1	1		
1680	4	1	1	1		
2520	3	2	1	1		
5040	4	2	1	1		
7560	3	3	1	1		
10080	5	2	1	1		
15120	4	3	1	1		
20160	6	2	1	1		
25200	4	2	2	1		
27720	3	2	1	1	1	
45360	4	4	1	1		
50400	5	2	2	1		
55440	4	2	1	1	1	
83160	3	3	1	1	1	
110880	5	2	1	1	1	
166320	4	3	1	1	1	
221760	6	2	1	1	1	
277200	4	2	2	1	1	
332640	5	3	1	1	1	
498960	4	4	1	1	1	
554400	5	2	2	1	1	
665280	6	3	1	1	1	
720720	4	2	1	1	1	1
1081080	3	3	1	1	1	1
1441440	5	2	1	1	1	1
2162160	4	3	1	1	1	1
2882880	6	2	1	1	1	1

Niech $p_1, p_2, \ldots, p_t$ będzie najdłuższym ciągiem kolejnych liczb pierwszych takich, że ich iloczyn nie przekracza n. Liczymy najpierw t. Istotnym jest to, że t jest stosunkowo małe, zatem liczba liczb pierwszych i ciągów do rozważenia nie jest bardzo duża. Aby znaleźć największą liczbę antypierwszą mniejszą lub równą n, rozważmy (najlepiej w kolejności leksykograficznej) wszystkie ciągi $i_1, i_2, \ldots i_t$ spełniające dwa poniższe warunki.

$$p_1^{i_1} p_2^{i_2} \ldots p_t^{i_t} \leq n$$

$$i_1 \geq i_2 \geq i_3 \ldots \geq i_t \geq 0$$

Jak się okazuje, takich ciągów, potencjalnych kandydatów na liczby antypierwsze $\leq 2\,000\,000\,000$, jest niedużo – około 1500. Bez trudu możemy więc je przesortować i jeden raz przeglądając znaleźć największą liczbę antypierwszą nie przekraczającą danego ograniczenia górnego.

Obserwacja

Przyjrzyjmy się strukturze pierwszych 40 liczb antypierwszych. W tabeli pokazane są wszystkie liczby antypierwsze mniejsze od 3 milionów. Zauważmy, że ostatnie potęgi, poza dwoma przypadkami, są równe 1. Przedostatnie potęgi są też nieduże. Nie jest to przypadek, ponieważ zachodzi poniższy fakt (dosyć trudny do uzasadnienia, ale prawdziwy). Korzystając z tego można ograniczyć przestrzeń rozpatrywanych danych.

Fakt

Dla każdej liczby antypierwszej, oprócz liczb 2 i 36, ostatnia potęga w rozkładzie na liczby pierwsze jest równa 1, natomiast potęgi druga i trzecia od końca nie przekraczają nigdy 4.

Zauważmy olbrzymią różnicę w liczbie liczb antypierwszych mniejszych od zadanej liczby n (dostatecznie dużej) w porównaniu z liczbą liczb pierwszych. Właśnie dzięki temu testowanie liczb antypierwszych jest znacznie łatwiejsze od testowania liczb pierwszych.

Testy

Zadanie testowane było na zestawie 21 danych testowych.

- `ant0.in` — $n = 1000$, wynik: 840;
- `ant1.in` — $n = 1$, wynik: 1;
- `ant2.in` — $n = 3$, wynik: 2;
- `ant3.in` — $n = 5$, wynik: 4;
- `ant4.in` — $n = 100000$, wynik: 83160;
- `ant5.in` — $n = 8$, wynik: 6;
- `ant6.in` — $n = 5041$, wynik: 5040;
- `ant7.in` — $n = 20159$, wynik: 15120;
- `ant8.in` — $n = 15120$, wynik: 15120;
- `ant9.in` — $n = 75000$, wynik: 55440;
- `ant10.in` — $n = 150000$, wynik: 110880;
- `ant11.in` — $n = 1000000$, wynik: 720720;
- `ant12.in` — $n = 3000$, wynik: 2520;
- `ant13.in` — $n = 8000000$, wynik: 7207200;
- `ant14.in` — $n = 15000000$, wynik: 14414400;
- `ant15.in` — $n = 30000$, wynik: 27720;
- `ant16.in` — $n = 60000000$, wynik: 43243200;
- `ant17.in` — $n = 110000000$, wynik: 73513440;
- `ant18.in` — $n = 354218765$, wynik: 294053760;
- `ant19.in` — $n = 600000000$, wynik: 551350800;
- `ant20.in` — $n = 1000000000$, wynik: 735134400;
- `ant21.in` — $n = 2000000000$, wynik: 1396755360.

Marcin Jurdziński
Treść zadania, Opracowanie

Tomasz Waleń
Program

Gra w zielone

Gra **w zielone** *jest grą dla dwóch graczy — nazwijmy ich Ania i Bolek — polegającą na przesuwania pionka po planszy.*

Część pól planszy jest pokolorowana na zielono, a pozostałe są białe. Wszystkie pola są ponumerowane liczbami naturalnymi z zakresu $1 \ldots (a+b)$, pola o numerach z zakresu $1 \ldots a$ należą do Ani, natomiast pola o numerach $(a+1) \ldots (a+b)$ należą do Bolka.

Dla każdego pola dany jest zbiór **następników**, *zawierający te pola planszy, do których można z niego przejść w jednym ruchu. Zbiory te zostały tak dobrane, że z pola należącego do Ani można przejść w jednym ruchu tylko na pole należące do Bolka i odwrotnie.*

Na początku gry ustawiamy pionek na dowolnym polu, a następnie gracze na przemian przestawiają pionek ze swojego pola na dowolny następnik tego pola — należący do przeciwnika. Grę rozpoczyna właściciel pola, z którego zaczynamy rozgrywkę. Zakładamy, że wszystkie pola mają niepuste zbiory następników, a więc zawsze można wykonać ruch. Gra kończy się w momencie, gdy pionek stanie po raz drugi na tym samym polu. Jeśli w sekwencji ruchów, od pierwszego do powtórnego zajęcia tego pola, pionek stanął przynajmniej raz na polu zielonym, to wygrywa Ania, w przeciwnym przypadku wygrywa Bolek.

Powiemy, że Ania ma **strategię wygrywającą dla danego pola początkowego**, *jeśli istnieje metoda gwarantująca jej wygraną w rozgrywce zaczynającej się od tego pola, niezależnie od tego, jakie ruchy będzie wykonywał Bolek.*

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `gra.in` *opis planszy do gry w zielone,*
- *znajdzie zbiór pól planszy, dla których Ania ma strategię wygrywającą,*
- *zapisze wynik w pliku tekstowym* `gra.out`.

Wejście

W pierwszym wierszu pliku tekstowego `gra.in` *zapisane są dwie nieujemne liczby całkowite a, b oddzielone pojedynczym odstępem, oznaczające odpowiednio: liczbę pól należących do Ani i liczbę pól należących do Bolka. Liczby a, b spełniają warunek: $1 \le a + b \le 3\,000$. W kolejnych $a+b$ wierszach opisano pola planszy — najpierw pola należące do Ani, a następnie pola należące do Bolka. Wiersz o numerze $i + 1$, dla $1 \le i \le a + b$, zawiera na początku liczby całkowite z, k oddzielone pojedynczym odstępem, oznaczające odpowiednio kolor pola o numerze i (0 oznacza kolor biały, 1 — kolor zielony), oraz liczbę następników tego pola. Następnie w tym wierszu zapisane jest k liczb całkowitych ($1 \le k < a + b$), pooddzielanych pojedynczymi odstępami, będącymi numerami następników danego pola. Liczba pól zielonych na planszy nie przekracza 100. Suma liczb następników wszystkich pól na planszy nie przekracza $30\,000$.*

Wyjście

Pierwszy wiersz pliku tekstowego `gra.out` *powinien zawierać dokładnie jedną liczbę całkowitą p, oznaczającą liczbę pól, dla których Ania ma strategię wygrywającą. Następne p wierszy powinno zawierać numery tych pól zapisane w kolejności rosnącej — każda liczba powinna zostać zapisana w osobnym wierszu.*

Przykład

Dla pliku wejściowego `gra.in`:

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

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

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

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

Rozwiązanie

Ścisłe sformułowanie faktu, że w grze w zielone Ania ma strategię wygrywającą z danego pola początkowego nie jest banalne i jest zaledwie naszkicowane w treści zadania. Jego sprecyzowanie oraz algorytmiczną charakteryzację należy zatem do pewnego stopnia traktować jako część zadania postawionego przed uczestnikami olimpiady.

Przez P będziemy oznaczać zbiór wszystkich pól na planszy, a przez Z zbiór wszystkich pól zielonych.

Aby ułatwić opis strategii wygrywających w grze w zielone oraz ich poszukiwanie, wygodnie jest rozważyć następującą uproszczoną wersję gry. Niech $X \subseteq Z$ będzie pewnym zbiorem zielonych pól na planszy. *Uproszczona gra w zielone* toczy się w sposób podobny do zwyczajnej gry w zielone. Różnica polega na tym, że w uproszczonej grze w zielone rozgrywka może się zakończyć wcześniej: gdy pionek odwiedzi zielone pole ze zbioru X po raz pierwszy, rozgrywka zostaje przerwana i Ania zwycięża.

Przez *Wymuś*(X) oznaczamy zbiór pól, z których Ania ma strategię wygrywającą w uproszczonej grze w zielone. Zauważ, że *Wymuś*(X) jest zbiorem pól z których Ania może wymusić przesunięcie pionka na pewne zielone pole ze zbioru X w taki sposób, że pionek nie staje na żadnym polu dwa razy.

W poniższym fakcie wykazujemy, że rozwiązanie uproszczonej gry w zielone dla $X = Z$, tj. obliczenie zbioru *Wymuś*(Z), jest pomocne w ustaleniu pewnego zbioru pól, z których Bolek ma strategię wygrywającą w normalnej grze w zielone.

Fakt 1 (Strategia wygrywająca dla Bolka) *Bolek ma strategię wygrywającą z każdego pola, które nie jest w zbiorze Wymuś*(Z)*.*

Dowód. Fakt, że pole p nie należy do zbioru *Wymuś*(Z) oznacza, że w uproszczonej grze w zielone rozpoczynającej się w polu p, Bolek potrafi zapobiec dojściu do jakiegokolwiek zielonego pola. Cała rozgrywka toczy się więc i kończy w białej części planszy — to gwarantuje zwycięstwo Bolka w zwyczajnej grze w zielone z pola p.
$\square$

Przykład 2 *W tym i w poniższych przykładach ilustrujemy pojęcia i fakty opisane w tym opracowaniu na przykładzie gry w zielone z przykładu z treści zadania.*
Zauważ, że $Z = \{4, 5, 6\}$ *oraz Wymuś*$(Z) = \{1, 2, 4, 5, 6, 7\}$*. W polach spoza zbioru Wymuś*(Z)*, tj. w polach ze zbioru* $\{3, 8\}$*, Bolek ma strategię wygrywającą polegającą na wybraniu pola 3 jako następnika w polu 8.*

Następnie spróbujemy ustalić jakie warunki gwarantują, że strategia wygrywająca dla Ani w uproszczonej grze w zielone daje Ani zwycięstwo także w normalnej grze w zielone. W tym celu przydatne okazuje się pojęcie pułapki dla Bolka.

Definicja 3 (Pułapka dla Bolka) *Powiemy, że zbiór pól Wymuś*(X) *jest* pułapką *dla Bolka, jeśli dla każdego pola* p *ze zbioru* X *mamy:*

1. *pole* p *należy do Ani i* istnieje *następnik pola* p *znajdujący się w zbiorze Wymuś*(X)*; lub*
2. *pole* p *należy do Bolka i* każdy *następnik pola* p *znajduje się w zbiorze Wymuś*(X)*.*

Przykład 4 *Niech* $X = \{4, 6\}$*. Zbiór Wymuś*$(X) = \{1, 2, 4, 6, 7\}$ *jest pułapką dla Bolka, ponieważ:*

- *istnieje następnik pola 4 (tj. pole 7) w zbiorze Wymuś*(X)*, oraz*
- *każdy następnik pola 6 (tj. zarówno pole 1 jak i pole 2) należy do zbioru Wymuś*(X)*.*

Fakt 5 (Strategia wygrywająca dla Ani) *Jeśli zbiór Wymuś*(X) *jest pułapką dla Bolka, to Ania ma strategię wygrywającą z każdego pola w zbiorze Wymuś*(X)*.*

Dowód. Rozważmy następującą strategię dla Ani:

- w każdym białym polu Ani ze zbioru $Wymuś(X)$, Ania gra zgodnie z jej strategią wygrywającą w uproszczonej grze w zielone;
- w każdym zielonym polu Ani ze zbioru $Wymuś(X)$, tj. w każdym polu Ani ze zbioru X, Ania wybiera następnika który należy do zbioru $Wymuś(X)$: taki następnik istnieje, bo zbiór $Wymuś(X)$ jest pułapką dla Bolka.

Zauważ, że w każdej rozgrywce zgodnej z tą strategią, pionek nigdy nie opuszcza zbioru $Wymuś(X)$. Co więcej, żaden cykl (tj. fragment rozgrywki rozpoczynający się i kończący w tym samym polu) w takiej rozgrywce nie składa się tylko z białych pól, ponieważ strategia Ani na białych polach jest wygrywająca w uproszczonej grze w zielone. Dlatego każda rozgrywka zgodna z powyższą strategią jest wygrywająca dla Ani w zwyczajnej grze w zielone. □

Pozostaje nam do rozwiązania sytuacja, w której zbiór $Wymuś(Z)$ nie jest pułapką dla Bolka. Niech $Z_1 \subseteq Z$ będzie zbiorem zielonych pól, które spełniają warunki 1 lub 2 definicji pułapki dla Bolka, dla $X = Z$.

Zauważ, że w zielonych polach nie należących do zbioru Z_1, Bolek ma następującą strategię wygrywającą: najpierw przesuń pionek w jednym kroku do pola nie należącego do zbioru $Wymuś(Z)$, a następnie użyj strategii wygrywającej zagwarantowanej przez Fakt 1. Możemy zatem odrzucić zielone pola spoza zbioru Z_1, jako "bezużyteczne" dla Ani.

Aby ponowić próbę użycia Faktu 5 do wyznaczenia strategii wygrywającej dla Ani, rozważmy zbiór $Wymuś(Z_1)$. Jeśli ten zbiór jest pułapką dla Bolka, to rozwiązanie gry w zielone jest gotowe (można wykazać — patrz dowód ogólniejszego Faktu 7 poniżej — że także we wszystkich białych polach spoza zbioru $Wymuś(Z_1)$ Bolek ma strategię wygrywającą.) W przeciwnym wypadku należy ponownie odrzucić zielone pola, które okazały się bezużyteczne dla Ani, itd.

Powyższe rozumowanie motywuje następującą metodę obliczania kolejnych, coraz mniejszych zbiorów zielonych pól, które kandydują do roli pól dających Ani strategię wygrywającą (poprzez Fakt 5).

```
 1: procedure NajwiększaPułapkaDlaBolka
 2: begin
 3:    Z_0 := Z
 4:    i := 0
 5:    while Wymuś(Z_i) nie jest pułapką dla Bolka do
 6:    begin
 7:       Z_{i+1} := zbiór pól z Z_i spełniających warunki 1 lub 2 definicji 3
 8:       i := i + 1
 9:    end
10: end
```

Przykład 6
Z przykładu 2 mamy $Z_0 = Z = \{4, 5, 6\}$ oraz $Wymuś(Z_0) = \{1, 2, 4, 5, 6, 7\}$. Zauważ, że Z_0 nie jest pułapką dla Bolka, bo pole 5 należy do Ani, ale jedyny następnik pola 5, tj. pole 8, nie należy *do zbioru $Wymuś(Z_0)$. Pole 5 jest jedynym polem z Z_0 nie spełniającym warunków 1 lub 2 definicji 3, więc $Z_1 = \{4, 6\}$. Z przykładu 4 wiemy, że zbiór $Wymuś(Z_1)$ jest pułapką dla Bolka.*

Zauważ, że jeśli warunek kontynuacji wykonywania pętli 5:–9: jest spełniony, to istnieje przynajmniej jedno zielone pole w zbiorze Z_i *nie* spełniające warunków 1 lub 2 definicji 3. Dlatego ciąg zbiorów Z_i jest malejący, a stąd wynika, że liczba iteracji pętli 5:–9: wynosi $O(k)$.

Niech ℓ będzie najmniejszą liczbą dla której zbiór $Wymuś(Z_\ell)$ jest pułapką dla Bolka. Oczywiście Ania ma strategię wygrywającą ze zbioru $Wymuś(Z_\ell)$ — wynika to z Faktu 5. Aby wykazać, że $Wymuś(Z_\ell)$ to zbiór *wszystkich* pól z których Ania ma strategię wygrywającą, wystarczy udowodnić, że z każdego pola spoza zbioru $Wymuś(Z_\ell)$, strategię wygrywającą ma Bolek.

Fakt 7 (Strategia wygrywająca dla Bolka: ogólny przypadek)
Dla każdego $i \leq \ell$, Bolek ma strategię wygrywającą z każdego pola spoza zbioru $Wymuś(Z_i)$.

Dowód. Dowód przebiega przez indukcję ze względu na i. Zauważ, że dla $i = 0$ teza wynika wprost z Faktu 1.

Załóżmy, że dla pewnego $i < \ell$, Bolek ma strategię wygrywającą z każdego pola spoza zbioru $Wymuś(Z_i)$. Wykażemy, że wtedy Bolek ma także strategię wygrywającą z każdego pola spoza zbioru $Wymuś(Z_{i+1})$.

Wystarczy udowodnić, że Bolek ma strategię wygrywającą z każdego pola ze zbioru $Wymuś(Z_i) \setminus Wymuś(Z_{i+1})$ — dla pól nie należących do zbioru $Wymuś(Z_i)$ teza wynika z hipotezy indukcyjnej.

Rozważmy dwa rodzaje pól ze zbioru $Wymuś(Z_i) \setminus Wymuś(Z_{i+1})$:

1. Jeśli pole jest zielone, to nie spełnia warunków 1 lub 2 definicji 3, więc Bolek może z tego pola w jednym kroku wymusić przesunięcie pionka poza zbiór $Wymuś(Z_i)$, skąd ma strategię wygrywającą (z hipotezy indukcyjnej).

2. Jeśli pole jest białe, to Bolek korzysta ze strategii w uproszczonej grze w zielone, która zapobiega dojściu pionka do zbioru $Wymuś(Z_{i+1})$. Jeśli w wyniku rozgrywki pionek stanie na jednym z zielonych pól w zbiorze $Wymuś(Z_i) \setminus Wymuś(Z_{i+1})$, wtedy Bolek postępuje jak w przypadku 1 i wygrywa. W przeciwnym razie rozgrywka toczy się i kończy w białej części zbioru $Wymuś(Z_i) \setminus Wymuś(Z_{i+1})$, więc jest wygrywająca dla Bolka. □

Aby precyzyjnie oszacować koszt czasowy działania algorytmu *NajwiększaPułapkaDlaBolka* musimy ustalić, jaki jest koszt rozwiązywania uproszczonej gry w zielone, tj. obliczania zbioru $Wymuś(X)$. Powiemy, że warunek $AniaWymusza(T,p)$ zachodzi dla pewnego zbioru pól T, oraz pola p, jeśli mamy:

- pole p należy do Ani i *istnieje* następnik pola p należący do zbioru T, lub
- pole p należy do Bolka i *każdy* następnik pola p należy do zbioru T.

Ćwiczenie 8 (Poprawność algorytmu *ObliczZbiórWymuś*)
Wykaż, że poniższa procedura ObliczWymuś(X) wyznacza poprawnie zbiór pól z których Ania ma strategię wygrywającą w uproszczonej grze w zielone, tj. zbiór Wymuś(X).

```
procedure  ObliczWymuś(X)
begin
   W := X
   while istnieje pole p ∉ W takie, że AniaWymusza(W,p) do
      dodaj pole p do zbioru W
   return(W)
end
```

Procedurę *ObliczWymuś* można zaimplementować tak, aby działała w czasie $O(m)$, gdzie m jest sumą liczb następników dla wszystkich pól. Przykładowa implementacja znajduje się na załączonej dyskietce.

Powyższą algorytmiczną analizę struktury strategii wygrywających dla Ani i Bolka w grze w zielone można podsumować w następujący sposób.

Twierdzenie 9 (Poprawność algorytmu *NajwiększaPułapkaDlaBolka*)
Niech ℓ będzie najmniejszą liczbą, dla której w procedurze NajwiększaPułapkaDlaBolka zbiór Wymuś(Z_ℓ) jest pułapką dla Bolka. Wtedy Ania ma strategię wygrywającą z pól ze zbioru Wymuś(Z_ℓ), a Bolek ma strategię wygrywającą ze wszystkich pozostałych pól. Algorytm można zaimplementować tak, by działał w czasie $O(k \cdot m)$, gdzie k jest liczbą zielonych pól, a m jest sumą liczb następników dla wszystkich pól.

Inne rozwiązania

W tym podrozdziale omówimy alternatywny warunek wystarczający i konieczny dla istnienia strategii wygrywających dla Bolka lub Ani w Grze w Zielone. Interesującą cechą tego warunku jest jego lokalność: jeśli każde pole jest w odpowiedniej, dość łatwej do sprawdzenia relacji ze zbiorem jego następników, to mamy gwarancję, że istnieje "globalna" strategia wygrywająca dla odpowiedniego gracza.

Definicja 10 (Dobre etykietowanie dla Bolka)
Rozważmy funkcję $\beta : P \to \{0,1,2,\ldots,k,\infty\}$: z każdym polem $p \in P$, związana jest etykieta $\beta(p)$, która jest liczbą naturalną nie większą niż k, lub symbolem ∞. Przyjmujemy umownie, że ∞ jest większa od każdej liczby naturalnej, czyli $i < \infty$, dla każdej liczby naturalnej i. Powiemy, że etykietowanie β jest dobre dla Bolka w polu *p, jeśli $\beta(p) = \infty$ lub spełnione są następujące warunki:*

- *jeśli pole p należy do Ani, to dla* każdego *następnika r pola p zachodzi warunek PostępBolka$(\beta; p, r)$;*
- *jeśli pole p należy do Bolka, to* istnieje *następnik r pola p taki, że zachodzi warunek PostępBolka$(\beta; p, r)$;*

gdzie warunek PostępBolka$(\beta; p, r)$ jest spełniony wtedy i tylko wtedy gdy:

1. *$\beta(p) > \beta(r)$, jeśli pole p jest zielone, oraz*
2. *$\beta(p) \geq \beta(r)$, jeśli pole p jest białe.*

Mówimy, że etykietowanie β jest dobre *dla Bolka, jeśli β jest dobre dla Bolka we wszystkich polach planszy.*

Przykład 11 *Zauważ, że zbiór P pól Gry w Zielone z treści zadania jest równy $P = \{1, 2, 3, \ldots, 8\}$. Niech etykietowanie $\beta : P \to \{0, 1, 2, 3, \infty\}$ będzie zdefiniowane przez następującą tabelę.*

p	1	2	3	4	5	6	7	8
$\beta(p)$	∞	∞	1	∞	3	∞	∞	1

Etykietowanie β jest dobre dla Bolka ponieważ:

- *pole 3 należy do Ani, pole 8 jest jedynym następnikiem pola 3, oraz zachodzi PostępBolka$(\beta; 3, 8)$, tj. $\beta(3) \geq \beta(8)$; zauważ, że wystarczy tu nieostra nierówność, bo pole 3 jest białe;*
- *pole 5 należy do Ani, pole 8 jest jedynym następnikiem pola 5, oraz zachodzi PostępBolka$(\beta; 5, 8)$, tj. $\beta(5) > \beta(8)$; zauważ, że wymagana tu jest ostra nierówność, bo pole 5 jest zielone;*
- *pole 8 należy do Bolka i dla pola 3, które jest następnikiem pola 8, zachodzi PostępBolka$(\beta; 8, 3)$, tj. $\beta(8) \geq \beta(3)$.*

Za tą—na pierwszy rzut oka—zawiłą definicją kryje się następująca prosta intuicja. O etykiecie $\beta(p)$ pola p — o ile ta etykieta jest liczbą naturalną, a nie symbolem ∞ — można myśleć jako o górnym ograniczeniu na liczbę zielonych pól, które Bolek może być zmuszony (przez Anię) odwiedzić w rozgrywce, jeśli Bolek wybiera zawsze następnika o jak najmniejszej etykiecie. Wykażemy, że wybieranie następnika o najmniejszej etykiecie gwarantuje Bolkowi zwycięstwo w grze w zielone, jeśli etykietowanie jest dobre dla Bolka.

Fakt 12 (Strategia wygrywająca dla Bolka)
Jeśli etykietowanie $\beta : P \to \{0, 1, 2, \ldots, k, \infty\}$ jest dobre dla Bolka, to Bolek ma strategię wygrywającą z każdego pola p takiego, że $\beta(p) \neq \infty$.

Dowód: Wykażemy, że następująca strategia jest wygrywająca dla Bolka: w każdym polu p należącym dla Bolka, Bolek wybiera następnika r pola p, o jak najmniejszej etykiecie $\beta(r)$. Zauważmy, że z definicji dobrego etykietowania dla Bolka wynika, że wtedy dla każdej rozgrywki $p_1, p_2, p_3, \ldots, p_i$ mamy

$$\beta(p_1) \geq \beta(p_2) \geq \beta(p_3) \geq \cdots \geq \beta(p_i).$$

Niech $p_j = p_i$, dla pewnego $j < i$, tj. $p_j = p_i$ jest pierwszym polem na którym pionek stanął dwa razy. Z powyższego warunku wynika, że $\beta(p_j) \geq \beta(p_{j+1}) \geq \beta(p_{j+2}) \geq \cdots \geq \beta(p_i) = \beta(p_j)$, więc musi być $\beta(p_j) = \beta(p_{j+1}) = \beta(p_{j+2}) = \cdots = \beta(p_i)$. Z definicji dobrego etykietowania dla Bolka — a dokładniej z punktu 1 definicji warunku *PostępBolka*(β, p, r) — wynika, że pola $p_j, p_{j+1}, p_{j+2}, \ldots, p_i$ są białe. Inaczej mówiąc, żadne pole odwiedzone pomiędzy pierwszą i drugą wizytą w polu $p_j = p_i$ nie jest zielone, czyli rozgrywka jest wygrywająca dla Bolka. □

Powyższy fakt oznacza, że istnienie dobrego etykietowania β dla Bolka, dla którego zachodzi $\beta(p) \neq \infty$, jest warunkiem wystarczającym dla istnienia strategii wygrywającej dla Bolka z pola p. Poniżej naszkicujemy dowód faktu, że jest to również warunek konieczny, oraz przedstawimy wydajny sposób znajdowania dobrych etykietowań. Rozważmy następującą procedurę poszukiwania dobrego etykietowania dla Bolka:

```
procedure PodnośEtykietowanie
begin
    dla każdego pola p do β(p) := 0
    while β nie jest dobre dla Bolka do
    begin
        wybierz pole p w którym β nie jest dobre dla Bolka
        β(p) := Popraw(β, p)
    end
end
```

gdzie *Popraw*(β, p) to najmniejsza etykieta nie mniejsza od $\beta(p)$, która przypisana polu p gwarantuje, że etykietowanie β staje się dobre w polu p. Wartość *Popraw*(β, p) można obliczyć na przykład tak:

$$Popraw(\beta, p) = \begin{cases} \beta(p) & \text{jeśli } \beta \text{ jest dobre dla Bolka w polu } p, \\ MaxMin(\beta, p) & \text{jeśli } p \text{ jest białe,} \\ \infty & \text{jeśli } p \text{ jest zielone i } MaxMin(\beta, p) \geq k, \\ 1 + MaxMin(\beta, p) & \text{jeśli } p \text{ jest zielone i } MaxMin(\beta, p) < k; \end{cases}$$

gdzie $MaxMin(\beta, p)$ jest zdefiniowane w następujący sposób:

$$MaxMin(\beta,p) = \begin{cases} \max\{\beta(r)\ :\ r \text{ jest następnikiem } p\} & \text{jeśli } p \text{ należy do Ani,} \\ \min\{\beta(r)\ :\ r \text{ jest następnikiem } p\} & \text{jeśli } p \text{ należy do Bolka.} \end{cases}$$

Warto tu zwrócić uwagę na to, że $Popraw(\beta, p) \geq \beta(p)$ oraz, że $Popraw(\beta, p) > \beta(p)$, jeśli etykietowanie β nie jest dobre w polu p.

Przykład 13 *Dla skrócenia i uproszczenia ilustracji działania algorytmu PodnośEtykietowanie wprowadzamy następującą notację na oznaczenie ciągu kilku operacji Popraw:*

$$\begin{aligned} Popraw(\beta,[p_1,p_2]) &= Popraw\big(Popraw(\beta,p_1),p_2\big) \\ Popraw(\beta,[p_1,p_2,p_3]) &= Popraw\Big(Popraw\big(Popraw(\beta,p_1),p_2\big),p_3\Big) \\ &\ \vdots \end{aligned}$$

Rozważmy następujące wykonanie procedury PodnośEtykietowanie:

p	1	2	3	4	5	6	7	8
$\beta_0(p)$	0	0	0	0	0	0	0	0
$\beta_1(p) = Popraw\big(\beta_0,[4,5,6]\big)(p)$	0	0	0	1	1	1	0	0
$\beta_2(p) = Popraw\big(\beta_1,[1,2]\big)(p)$	1	1	0	1	1	1	0	0
$\beta_3(p) = Popraw(\beta_2,7)(p)$	1	1	0	1	1	1	1	0
$\beta_4(p) = Popraw\big(\beta_3,[4,6,1,2,7]\big)(p)$	2	2	0	2	1	2	2	0
$\beta_5(p) = Popraw\big(\beta_4,[4,6,1,2,7]\big)(p)$	3	3	0	3	1	3	3	0
$\beta_6(p) = Popraw\big(\beta_5,[4,6,1,2,7]\big)(p)$	∞	∞	0	∞	1	∞	∞	0

Etykietowanie β_6 jest dobre dla Bolka. Zauważ, że etykietowanie β_6 jest "mniejsze" niż etykietowanie β z przykładu 11, w tym sensie, że dla każdego pola $p \in P$, mamy $\beta_6(p) \leq \beta(p)$.

Można myśleć o procedurze *PodnośEtykietowanie* jako o metodzie przybliżania etykietowania β „z dołu", do „najmniejszego" dobrego etykietowania. Istotnie, początkowe etykietowanie $\beta = \beta_0$, gdzie $\beta_0(p) = 0$, dla każdego pola p, jest „mniejsze" niż jakiekolwiek inne etykietowanie. (Dla ścisłości, mówimy, że etykietowanie β jest „mniejsze" lub równe niż etykietowanie β', jeśli $\beta(p) \leq \beta'(p)$, dla każdego pola p.) Kolejne „poprawki" polegające na podnoszeniu wartości etykietowania β w polu, w którym β nie jest dobre dla Bolka, zachowują własność, że etykietowanie β jest mniejsze lub równe niż każde dobre etykietowanie. Jest tak dlatego, ponieważ w procedurze *PodnośEtykietowanie* wartość etykiety w pewnym polu jest zawsze podnoszona tylko o tyle, o ile to jest konieczne, aby etykietowanie stało się dobre dla Bolka w tym polu.

Przykład 14 *Etykietowanie $\beta_6 : P \to \{1,2,3,\infty\}$:*

p	1	2	3	4	5	6	7	8
$\beta_6(p)$	∞	∞	0	∞	1	∞	∞	0

z przykładu 13 jest najmniejszym *dobrym etykietowaniem dla Bolka w grze z treści zadania. Inaczej mówiąc, etykietowanie β_6 jest mniejsze nie tylko od dobrego etykietowania beta dla Bolka z przykładu 11, ale jest nie większe niż* każde *dobre etykietowanie dla Bolka w tej grze.*

Zauważ, że każde pole może być poprawione przez procedurę *PodnośEtykietowanie* co najwyżej $k+1$ razy, zatem liczba powtórzeń pętli 4:–8: wynosi $O(k \cdot n)$.

Co więcej, etykietowanie β po zakończeniu procedury *PodnośEtykietowanie* jest dobrym etykietowaniem dla Bolka; w „najgorszym" przypadku mamy $\beta(p) = \infty$, dla każdego pola p — „największe" dobre etykietowanie. Z faktu 12 wynika, że Bolek ma strategię wygrywającą z każdego pola p takiego, że $\beta(p) \neq \infty$. *Najmniejsze* dobre etykietowanie dla Bolka, tj. etykietowanie β obliczane przez procedurę *PodnośEtykietowanie*, jest szczególnie interesujące ponieważ można z niego łatwo odczytać rozwiązanie gry w zielone, tj. zbiór pól z których istnieje strategia wygrywająca dla Ani.

Twierdzenie 15 (Strategia wygrywająca dla Ani)
Etykietowanie $\beta : P \to \{0,1,2,\ldots,k,\infty\}$ obliczone przez procedurę PodnośEtykietowanie ma taką własność, że Ania ma strategię wygrywającą z pola p wtedy i tylko wtedy gdy $\beta(p) = \infty$.

Dowód tego twierdzenia nie jest natychmiastowy i wymaga dość subtelnej argumentacji. Poniżej szkicujemy przydatne w tym celu pojęcia oraz ogólny tok rozumowania. Wypełnienie szczegółów dowodu pozostawiamy jako ćwiczenie dla dociekliwego czytelnika.

Definicja 16 (Dobre etykietowanie dla Ani)
Rozważmy etykietowanie $\alpha : P \to \{0, 1, 2, \ldots, n-k, \infty\}$. Powiemy, że etykietowanie α jest dobre dla Ani w polu p, *jeśli $\alpha(p) = \infty$ lub spełnione są następujące warunki:*

- *jeśli pole p należy do Ani, to* istnieje *następnik r pola p taki, że zachodzi warunek PostępAni$(\alpha; p, r)$,*
- *jeśli pole p należy do Bolka, to dla* każdego *następnika r pola p, zachodzi warunek PostępAni$(\alpha; p, r)$;*

gdzie warunek PostępAni$(\alpha; p, r)$ oznacza, że $\alpha(p) > \alpha(r)$, jeśli pole p jest białe. Mówimy, że etykietowanie α jest dobre *dla Ani, jeśli α jest dobre dla Ani we wszystkich polach planszy.*

Przykład 17 *Etykietowanie $\alpha : P \to \{0, 1, 2, 3, 4, 5, \infty\}$ zdefiniowane przez następującą tabelę jest dobre dla Ani.*

p	1	2	3	4	5	6	7	8
$\alpha(p)$	1	1	∞	0	∞	0	2	∞

Podobnie jak w przypadku dobrych etykietowań dla Bolka, można skonstruować strategię wygrywającą dla Ani, jeśli dane jest dobre etykietowanie dla Ani. Dowód następującego łatwego faktu pozostawiamy czytelnikowi.

Ćwiczenie 18 (Strategia wygrywająca dla Ani) *Wykaż, że jeśli etykietowanie $\alpha : P \to \{0, 1, 2, \ldots, n-k, \infty\}$ jest dobre dla Ani, to Ania ma strategię wygrywającą z każdego pola p takiego, że $\alpha(p) \neq \infty$.*

W celu wykazania Twierdzenia 15 do procedury *PodnośEtykietowanie* dodamy instrukcje obliczające etykietowanie $\alpha : P \to \{0, 1, 2, \ldots, n-k, \infty\}$. Zauważ, że te dodatkowe instrukcje nie mają wpływu na obliczenie etykietowania β i służą nam tylko jako pomoc w dowodzie Twierdzenia 15. Rozważmy następującą modyfikację procedury *PodnośEtykietowanie*:

```
procedure PodnośEtykietowanie'
begin
    dla każdego pola p do begin β(p) := 0; α(p) := ∞ end
    while β nie jest dobre dla Bolka do
    begin
        wybierz pole p w którym β nie jest dobre dla Bolka;
        β(p) := Popraw(β, p);
        if β(p) = ∞ then α(p) := Ustal(α, p)
    end
end
```

gdzie

$$\mathit{Ustal}(\beta, p) = \begin{cases} 0 & \text{jeśli pole } p \text{ jest zielone,} \\ 1 + \mathit{MinMax}(\beta, p) & \text{jeśli pole } p \text{ jest białe;} \end{cases}$$

oraz

$$\mathit{MinMax}(\alpha, p) = \begin{cases} \min\{\alpha(r) \ : \ r \text{ jest następnikiem } p\} & \text{jeśli } p \text{ należy do Ani,} \\ \max\{\alpha(r) \ : \ r \text{ jest następnikiem } p\} & \text{jeśli } p \text{ należy do Bolka.} \end{cases}$$

Przykład 19
Przekonaj się, że etykietowanie α obliczane przez procedurę PodnośEtykietowanie' jest równe etykietowaniu α z Przykładu 17.

Ćwiczenie 20 (Dobre etykietowanie dla Ani)
Wykaż, że etykietowanie: $\alpha : P \to \{0, 1, 2, \ldots, n-k, \infty\}$ obliczane przez procedurę PodnośEtykietowanie' jest dobre dla Ani.

Wskazówka. *Wykaż, że gdyby istniało pole p takie, że $\alpha(p) \neq \infty$ i α nie jest dobrym etykietowaniem dla Ani w polu p, to $\beta(p) \neq \infty$, gdzie β jest* najmniejszym *dobrym etykietowaniem dla Bolka.*

Powyższą analizę dobrych etykietowań dla Ani i Bolka można podsumować w następujący sposób.

Twierdzenie 21 (Poprawność algorytmu *PodnośEtykietowanie*)
Niech β będzie etykietowaniem obliczonym przez procedurę PodnośEtykietowanie. Wtedy Ania ma strategię wygrywającą z każdego pola p dla którego zachodzi $\beta(p) = \infty$, a Bolek ma strategię wygrywającą ze wszystkich pozostałych pól.

Ćwiczenie 22 (Wydajna implementacja algorytmu)
Zaimplementuj algorytm PodnośEtykietowanie w taki sposób, aby czas działania twojego programu był $O(k \cdot m)$.

Wskazówka. *Zaprojektuj strukturę danych umożliwiającą ustalanie w czasie $O(1)$, czy istnieje pole p, w którym aktualne etykietowanie nie jest dobre dla Bolka, oraz poprawianie wartości etykietowania w polu p i aktualizację struktury danych w czasie $O(d)$, gdzie d jest sumą liczb, następników pola p oraz jego „poprzedników”, tj. pól dla których p jest następnikiem.*

Ciekawy problem 23 *Przyjmijmy, że liczba zielonych pól wynosi $\Omega(n)$. Czy istnieje algorytm rozwiązujący gry w zielone działający w czasie $o(n \cdot m)$? W szczególności, czy istnieje algorytm rozwiązujący gry w zielone działający w czasie $O(m)$?*

Testy

Ocena rozwiązań zadania “Gra w zielone”, przedstawionych przez uczestników olimpiady, została oparta na zachowaniu się programów na kolekcji jedenastu testów, tj. jedenastu plików wejściowych zawierających opisy plansz różnych gier w zielone:

- `gra0.in`: Plik z treści zadania.
- `gra1.in`, `gra2.in`: Niewielkie plansze, zaprojektowane głównie z myślą o sprawdzaniu, czy program poprawnie rozwiązuje zadanie, nie wymagające — ze względu na mały rozmiar — aby algorytm był bardzo wydajny.
- `gra3.in`: Niewielka plansza o kilkudziesięciu polach z losowo dobranymi następnikami.
- `gra4.in`*: Plansza o umiarkowanej wielkości, z losowo dobranymi następnikami; mała średnia liczba następników.
- `gra5.in`*: Plansza o umiarkowanej wielkości, z losowo dobranymi następnikami; duża średnia liczba następników.
- `gra6.in`*: Plansza o dużej wielkości z losowo dobranymi następnikami.
- `gra7.in`*: Plansza o umiarkowanej wielkości, w której wszystkie pola przeciwnika o większych numerach niż numer danego pola są następnikami tego pola.
- `gra8.in`*: Plansza o bardzo dużej wielkości, z losowo dobranymi następnikami; mała średnia liczba następników.
- `gra9.in`*: Plansza o bardzo dużej wielkości, z losowo dobranymi następnikami; duża średnia liczba następników.
- `gra10.in`*: Plansza o umiarkowanej wielkości, na której przedstawione rozwiązania działają w czasie równym pesymistycznemu oszacowaniu, tj. $O(k \cdot m)$.

Testy oznaczone gwiazdką zawierają dodatkowe małe fragmenty planszy zaprojektowane z myślą o weryfikacji poprawności rozwiązania.

Zawody II stopnia

Zawody II stopnia — opracowania zadań

Wojciech Guzicki, Jarosław Wróblewski Treść zadania | **Wojciech Guzicki** Opracowanie | **Marcin Mucha** Program

Gorszy Goldbach

W roku 1742 C. Goldbach w liście do L. Eulera napisał, że jego zdaniem każda liczba całkowita $n > 5$ jest sumą trzech liczb pierwszych[1].

Euler odpisał, że jest to równoważne temu, że każda liczba parzysta $n \geq 4$ jest sumą dwóch liczb pierwszych. To jednak nie przybliżyło ich do rozwiązania podstawowego problemu: czy tak jest naprawdę.

Dziś wiemy, że jest tak dla liczb aż do $4 \cdot 10^{11}$ (wiemy też dużo więcej, ale cała hipoteza jest nadal problemem otwartym). Nie będziemy tego sprawdzać, postawimy sobie mniej ambitne zadanie. Okazuje się, że każda liczba naturalna $n \geq 10$ jest sumą różnych nieparzystych liczb pierwszych.

Zadanie

Twoje zadanie polega na napisaniu programu, który:

- *wczyta z pliku tekstowego* `gol.in` *liczby całkowite,*
- *znajdzie ich rozkłady na sumy różnych nieparzystych liczb pierwszych,*
- *zapisze wyniki w pliku tekstowym* `gol.out`.

Takich rozkładów może być wiele. Twój program może znaleźć jakikolwiek z nich.

Wejście

W pierwszym wierszu pliku tekstowego `gol.in` *zapisano jedną dodatnią liczbę całkowitą n, $n \leq 40$. W każdym z kolejnych n wierszy znajduje się jedna liczba całkowita z przedziału $[10, \ldots, 2\,000\,000\,000]$.*

Wyjście

Rozkład liczby k musi być zapisany w dwóch wierszach. W pierwszym wierszu należy zapisać jedną liczbę całkowitą $m \geq 1$, będącą liczbą składników rozkładu.

W drugim wierszu należy zapisać, w rosnącej kolejności, m różnych nieparzystych liczb pierwszych, których suma jest równa k, pooddzielanych pojedynczymi odstępami. Rozkłady powinny występować w kolejności zgodnej z kolejnością liczb w pliku wejściowym.

Przykład

Dla pliku wejściowego `gol.in`*:*

```
2
59
15
```

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

```
5
5 7 11 17 19
3
3 5 7
```

Rozwiązanie

Istnieje bardzo szybki algorytm pozwalający rozwiązać to zadanie. Wskażemy mianowicie 32 nieparzyste liczby pierwsze o tej własności, że każda dopuszczalna liczba n (tzn. z przedziału $[10; 2\,000\,000\,000]$) będzie sumą niektórych z tych liczb. Pokażemy też dokładnie, jak taki rozkład można znaleźć.

[1] Liczba pierwsza to liczba naturalna $n > 1$, która ma tylko dwa dzielniki naturalne: 1 oraz n.

Algorytm, który mamy na myśli, można odczytać z dowodu następującego twierdzenia ogólnego, udowodnionego w 1949 roku przez H. E. Richerta.

Twierdzenie. Każda liczba naturalna $n \geq 10$ jest sumą różnych nieparzystych liczb pierwszych.

Dowód. Najpierw przedstawimy każdą liczbą naturalną n spełniającą nierówności

$$10 \leq n \leq 46$$

w postaci sumy różnych nieparzystych liczb pierwszych wybranych spośród liczb 3, 5, 7, 11, 13, 17. A oto te rozkłady:

10	=	3+7,			
11	=	11,	29	=	5+7+17,
12	=	5+7,	30	=	13+17,
13	=	13,	31	=	3+11+17,
14	=	3+11,	32	=	3+5+7+17,
15	=	3+5+7,	33	=	5+11+17,
16	=	5+11,	34	=	3+7+11+13,
17	=	17,	35	=	5+13+17,
18	=	5+13,	36	=	3+5+11+17,
19	=	3+5+11,	37	=	7+13+17,
20	=	3+17,	38	=	3+5+13+17,
21	=	3+5+13,	39	=	3+5+7+11+13,
22	=	5+17,	40	=	5+7+11+17,
23	=	5+7+11,	41	=	11+13+17,
24	=	7+17,	42	=	5+7+13+17,
25	=	3+5+17,	43	=	3+5+7+11+17,
26	=	3+5+7+11,	44	=	3+11+13+17,
27	=	3+7+17,	45	=	3+5+7+13+17,
28	=	11+17,	46	=	5+11+13+17

Tabela 1.

Tych liczb jest 37. Wybieramy teraz największą liczbę pierwszą nie większą od 37. Będzie to sama liczba 37. Teraz do każdego z otrzymanych 37 rozkładów dołączamy liczbę 37. W ten sposób otrzymamy rozkłady liczb od $10 + 37 = 47$ do $46 + 37 = 83$ na sumy różnych nieparzystych liczb pierwszych. Łącznie z otrzymanymi wcześniej rozkładami mamy teraz rozkłady 74 liczb: od 10 do 83.

Wybieramy następną liczbę pierwszą: będzie to największa liczba pierwsza nie większa od 74; w naszym przypadku jest to liczba 73. Dołączamy tę liczbę do każdego z dotychczasowych rozkładów, otrzymując w ten sposób rozkłady wszystkich liczb aż do $83 + 73 = 156$. Mamy więc rozkłady 147 liczb od 10 do 156.

Wybieramy tym razem największą liczbę pierwszą nie większą od 147, dołączamy ją do znalezionych rozkładów i tak dalej.

Jest to bardzo naturalna metoda postępowania. Powstaje jednak pytanie, czy jest ona zawsze skuteczna. Mianowicie mogłoby się okazać, że kolejna wybrana liczba pierwsza (pamiętamy: wybieramy największą liczbę nie większą od liczby tych liczb, dla których już znaleźliśmy rozkład) jest równa poprzednio wybranej liczbie pierwszej. Inaczej mówiąc, mogłoby się okazać, że za ostatnio wybraną liczbą pierwszą występuje tak wiele liczb złożonych, że nie możemy dobrać odpowiednio kolejnej liczby pierwszej. Okazuje się jednak, że tak nie jest i można tego dowieść. Dowód wymaga jednak skorzystania z bardzo znanego twierdzenia z teorii liczb, tzw. twierdzenia Czebyszewa, które zapewne nie jest znane większości uczniów liceum. W naszym zadaniu jednak dowód twierdzenia nie jest potrzebny. Wystarczy bowiem sprawdzić, że taką liczbę można dobrać zawsze do momentu, gdy uzyskamy rozkłady wszystkich liczb nie większych od 2 miliardów. Przykłady takich liczb pierwszych zawarte są w tabeli 2.

W kolumnie pierwszej podajemy największą dotychczas użytą liczbę pierwszą p. W kolumnie drugiej podajemy przedział liczb, w którym ta liczba p jest użyta jako największa liczba rozkładu (z wyjątkiem pierwszego wiersza). Wreszcie w kolumnie trzeciej podajemy, ile liczb ma już znany rozkład (od 10 do największej liczby, dla której mamy znaleziony rozkład). Ta liczba jest o 9 mniejsza od największej liczby, dla której znamy rozkład.

Zwróćmy jeszcze uwagę na to, że w pierwszym wierszu występuje liczba pierwsza 17 jako największa liczba występująca w rozkładach liczb od 10 do 46. To nie znaczy oczywiście, że występuje ona w każdym z tych rozkładów. Rozkłady liczb od 10 do 46 muszą być brane z tabeli pierwszej.

Ostatnia liczba pierwsza p	przedział	Ile liczb (razem)
17	$\langle 10; 46\rangle$	37
37	$\langle 47; 83\rangle$	74
73	$\langle 75; 156\rangle$	147
139	$\langle 157; 295\rangle$	286
283	$\langle 296; 578\rangle$	569
569	$\langle 579; 1147\rangle$	1138
1129	$\langle 1148; 2276\rangle$	2267
2267	$\langle 2277; 4543\rangle$	4534
4523	$\langle 4544; 9066\rangle$	9057
9049	$\langle 9067; 18115\rangle$	18106
18097	$\langle 18116; 36212\rangle$	36203
36191	$\langle 36213; 72403\rangle$	72394
72383	$\langle 72404; 144786\rangle$	144777
144773	$\langle 144787; 289559\rangle$	289550
289543	$\langle 289560; 579102\rangle$	579093
579083	$\langle 579103; 1158185\rangle$	1158176
1158161	$\langle 1158186; 2316346\rangle$	2316337
2316337	$\langle 2316347; 4632683\rangle$	4632674
4632673	$\langle 4632684; 9265356\rangle$	9265347
9265309	$\langle 9265357; 18530665\rangle$	18530656
18530639	$\langle 18530666; 37061304\rangle$	37061295
37061293	$\langle 37061305; 74122597\rangle$	74122588
74122571	$\langle 74122598; 148245168\rangle$	148245159
148245127	$\langle 148245169; 296490295\rangle$	296490286
296490277	$\langle 296490296; 592980572\rangle$	592980563
592980559	$\langle 592980573; 1185961131\rangle$	1185961122
1185961087	$\langle 1185961132; 2000000000\rangle$	1999999991

$Tabela 2.$

Teraz już można łatwo napisać algorytm:

Dana jest liczba $n \geq 10$.
Dopóki $n > 46$ **powtarzaj**
  Znajdź w drugiej kolumnie przedział, do którego należy n.
  Zapamiętaj odpowiadającą mu liczbę p *z pierwszej kolumny.*
  $n := n - p$.
Teraz już $n \leq 46$.
Zapamiętaj znajdujące się w tabeli pierwszej liczby pierwsze występujące w rozkładzie n.
Zapamiętane liczby wypisz w kolejności rosnącej.

Powróćmy teraz do dowodu twierdzenia. Oznaczmy przez p największą użytą liczbę pierwszą i przez M największą liczbę, dla której znaleźliśmy rozkład. Niech k będzie liczbą tych liczb, dla których taki rozkład znaleźliśmy. Oczywiście wtedy $k = M - 9$.

Na początku mamy $p = 17$, $M = 46$ i $k = 37$. Zauważmy, że

$$k \geq 2p.$$

Niech teraz q będzie największą liczbą pierwszą nie większą od k. Wtedy $q > p$. Wynika to ze znanego twierdzenia Czebyszewa:

Twierdzenie. Dla każdej liczby naturalnej $l \geq 2$ istnieje liczba pierwsza r taka, że $l < r < 2l$.

Dowód twierdzenia Czebyszewa można znaleźć np. w książkach W. Sierpińskiego *Arytmetyka teoretyczna* lub *Teoria liczb, t. II.* Z twierdzenia Czebyszewa wynika, że istnieje nieparzysta liczba pierwsza r taka, że:

$$p < r < 2p \leq k.$$

Liczba q, która jest największą liczbą pierwszą nie większą od k, jest więc większa od p. Teraz do q ostatnich otrzymanych liczb dodajemy q:

$$\begin{gathered}(M-q+1)+q = M+1,\\ (M-q+2)+q = M+2,\\ \ldots\;\ldots\;\ldots\\ (M-2)+q = M+q-2,\\ (M-1)+q = M+q-1,\\ M+q = M+q.\end{gathered}$$

Otrzymaliśmy rozkłady liczb od $M+1$ do $M+q$ na sumę różnych liczb pierwszych: liczba q jest bowiem większa od wszystkich dotychczas użytych liczb pierwszych. Oczywiście również $M+q > M$.

Połóżmy teraz

$$M_1 = M+q, \quad p_1 = q, \quad k_1 = M_1 - 9.$$

Ponieważ $q \leq k$, więc

$$2q \leq k+q = M-9+q = M_1 - 9 = k_1.$$

Podstawiamy

$$M := M_1, \quad p := p_1, \quad k := k_1$$

i mamy nadal spełniony warunek $2p \leq k$. Możemy więc powtórzyć postępowanie itd.

Ponieważ $M_1 > M$, więc za każdym razem dostajemy rozkład co najmniej jednej nowej liczby na sumę różnych nieparzystych liczb pierwszych, co dowodzi, że każda liczba $n \geq 10$ ma taki rozkład. To kończy dowód twierdzenia Richerta.

Można podać wiele innych algorytmów rozwiązujących to zadanie. Najprostszy polega na wybieraniu największej liczby pierwszej p nie większej od n i następnie rozwiązaniu zadania dla liczby $n-p$ zamiast n. Ten algorytm nie zawsze prowadzi do sukcesu. Jeśli np. $n = 27$, to wybierzemy liczbę $p = 23$ i wtedy zadanie nie ma rozwiązania dla $n-p = 4$. Możemy jednak próbować zadbać o to, by otrzymana różnica była co najmniej równa 10: wtedy zadanie dla $n-p$ będzie miało rozwiązanie. To daje bardzo prosty algorytm:

Dana jest liczba $n \geq 10$.
Dopóki $n \geq 10$ **powtarzaj**
  Znajdź liczbę pierwszą p *taką, że* $\frac{n}{2} < p \leq n-10$.
  Zapamiętaj liczbę p.
  $n := n - p$.
Zapamiętane liczby pierwsze wypisz w kolejności rosnącej.

Warunek $\frac{n}{2} < p$ gwarantuje, że kolejno znajdowane liczby pierwsze będą różne. Mamy bowiem wtedy

$$n - p < \frac{n}{2} < p,$$

a więc następna liczba pierwsza będzie mniejsza od p.

Ten algorytm jednak nie jest poprawny, gdyż nie dla każdej liczby n istnieje liczba pierwsza p taka, że

$$\frac{n}{2} < p \leq n - 10.$$

Jednak taka liczba pierwsza p istnieje dla odpowiednio dużych liczb n. Niewielka modyfikacja dowodu twierdzenia Czebyszewa daje następujące wzmocnienie tego twierdzenia:

Twierdzenie. Jeśli $n \geq 21$, to w przedziale $\langle n+1; 2n \rangle$ istnieje co najmniej 5 liczb pierwszych.

Z tego twierdzenia wynika łatwo następujący wniosek:

Wniosek. Jeśli $n > 26$, to istnieje liczba pierwsza p taka, że

$$\frac{n}{2} < p \leq n - 10.$$

Zauważmy, że z tego wniosku wynika natychmiast inny dowód twierdzenia Richerta. Przeprowadzenie tego dowodu pozostawimy jako ćwiczenie dla Czytelnika. Możemy też zmodyfikować algorytm, tak by działał poprawnie dla wszystkich n:

Dana jest liczba $n \geq 10$.
Dopóki $n > 26$ **powtarzaj**
  Znajdź liczbę pierwszą p *taką, że* $\frac{n}{2} < p \leq n - 10$.
  Zapamiętaj liczbę p.
  $n := n - p$
Teraz już $n \leq 26$.
Zapamiętaj znajdujące się w tabeli pierwszej liczby pierwsze występujące w rozkładzie n.
Zapamiętane liczby pierwsze wypisz w kolejności rosnącej.

Ten algorytm może jednak działać dość długo. Liczby p najlepiej szukać „od góry”: po odjęciu liczby p od n dostaniemy liczbę dość małą i teraz już szybko znajdziemy rozkład tej liczby na sumę różnych liczb pierwszych. Warto jednak odnotować fakt, że znane są długie ciągi kolejnych liczb złożonych, tzw. luki. Luka długości k oznacza, że po liczbie pierwszej p, liczby $p+1, \ldots, p+k-1$ są złożone i dopiero liczba $p+k$ jest pierwsza. Oczywiście długości luk muszą być liczbami parzystymi. Znane są luki do długości 778 (dla liczb pierwszych $p < 72635119999997$) i niektóre inne. To oznacza, że poszukiwanie największej liczby pierwszej nie większej od $n - 10$ może potrwać dość długo, tym bardziej, że sprawdzanie, czy dana liczba jest pierwsza, też jest czasochłonne.

Najniżej występującą luką długości 100 jest luka między liczbami pierwszymi 396733 i 396833. Ale istnieją większe luki. Najdłuższa znana ma długość 863:

Po liczbie 6 505 941 701 960 039 występują 863 liczby złożone. Ta liczba jest większa od 2 miliardów, ale na przykład po liczbie 166726367 występują 194 liczby złożone. J. Young i A. Potler opublikowali (First occurrence of prime gaps, *Math. Comp.*, 1989, **52**, 221–224) tablicę takich luk. Niektóre testy zostały dobrane w taki sposób, by poszukiwanie liczby pierwszej p trwało długo; do tego celu została wykorzystana wspomniana tabela luk (użyto luk długości do 300).

Można jednak zauważyć, że znaleziona liczba pierwsza p jest „dobra” dla wielu liczb n; dokładniej dla liczb n spełniających nierówności

$$p + 10 \leq n < 2p.$$

Można teraz znaleźć taki zbiór skończony liczb pierwszych p, by przedziały $[p+10; 2p-1]$ pokrywały cały przedział $[10; 2\,000\,000\,000]$. Jest wiele metod szukania takich zbiorów. Jeden przykład podaliśmy wcześniej. Inny został wykorzystany w programie wzorcowym. Proponujemy Czytelnikowi jako ćwiczenie zastanowienie się, jak ten przykład został znaleziony.

Testy

Do sprawdzenia rozwiązań użyto 11 testów. Test `GOLO.IN` to test z treści zadania. Następne trzy testy to testy poprawnościowe (odpowiednio dla małych, średniej wielkości i dużych liczb). Następne testy sprawdzały szybkość działania programu i wykorzystywały wspomniane wcześniej luki. Te testy różniły się między sobą wielkością liczb i usytuowaniem luki po to, by sprawdzić różne sposoby szukania liczby pierwszej p mniejszej od $n - 10$.

Wojciech Rytter
Treść zadania, Opracowanie

Piotr Sankowski
Program

Spokojna komisja

W parlamencie **Demokratycznej Republiki Bajtocji***, zgodnie z* **Bardzo Ważną Ustawą***, należy ukonstytuować* **Komisję Poselską do Spraw Spokoju Publicznego***. Niestety sprawę utrudnia fakt, iż niektórzy posłowie wzajemnie się nie lubią.*

Komisja musi spełniać następujące warunki:

- *każda partia ma dokładnie jednego reprezentanta w Komisji,*
- *jeśli dwaj posłowie się nie lubią, to nie mogą jednocześnie być w Komisji.*

Każda partia ma w parlamencie dokładnie dwóch posłów. Wszyscy posłowie są ponumerowani liczbami od 1 do $2n$. Posłowie o numerach $2i-1$ i $2i$ należą do partii o numerze i.

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `spo.in` *liczbę partii oraz pary posłów, którzy się wzajemnie nie lubią,*
- *wyznaczy skład Komisji, lub stwierdzi, że nie da się jej ukonstytuować,*
- *zapisze wynik w pliku tekstowym* `spo.out`.

Wejście

W pierwszym wierszu pliku tekstowego `spo.in` *znajdują się dwie nieujemne liczby całkowite n i m. Liczba n, spełniająca warunki $1 \leq n \leq 8000$, oznacza liczbę partii. Liczba m, spełniająca warunki $0 \leq m \leq 20000$, oznacza liczbę par nielubiących się posłów. W każdym z kolejnych m wierszy zapisana jest para liczb naturalnych a i b, $1 \leq a < b \leq 2n$, oddzielonych pojedynczym odstępem. Oznacza ona, że posłowie o numerach a i b wzajemnie się nie lubią.*

Wyjście

Plik tekstowy `spo.out` *powinien zawierać pojedyncze słowo* `NIE`*, jeśli utworzenie Komisji nie jest możliwe. W przypadku, gdy utworzenie Komisji jest możliwe, plik* `spo.out` *powinien zawierać n liczb całkowitych z przedziału od 1 do $2n$, zapisanych w kolejności rosnącej i oznaczających numery posłów zasiadających w Komisji. Każda z tych liczb powinna zostać zapisana w osobnym wierszu. Jeśli Komisję można utworzyć na wiele sposobów, Twój program może wypisać dowolny z nich.*

Przykład

Dla pliku wejściowego `spo.in`*:*

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

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

```
1
4
5
```

Rozwiązanie

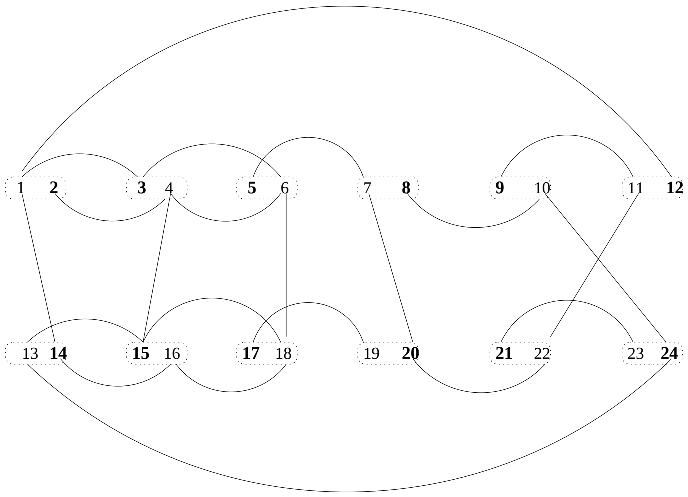
Niech X będzie zbiorem par posłów, którzy się wzajemnie nie lubią. Potraktujmy X jako krawędzie grafu nieskierowanego, a taki graf nazwijmy *grafem konfliktowym*. Przykład grafu konfliktowego ilustruje rysunek 1. Pogrubione numery oznaczają pewien zbiór posłów stanowiący "spokojną" Komisję.

Z grafem konfliktowym stowarzyszymy graf $\mathcal{G}$, który nazwiemy *grafem wymuszeń*. Jeśli x–$y \in X$ oraz $z \neq y$ należy do tej samej partii co y, to $x \to z$ jest skierowaną krawędzią w grafie wymuszeń. Oznaczmy

$$Wymuszone(x) \; = \; \{y \; : \; x \to^* y\},$$

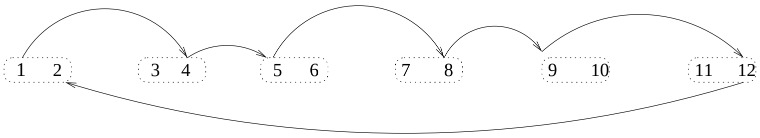
gdzie $\to^*$ oznacza, że w G można przejść z x do y po strzałkach. Zbiór $Wymuszone(x)$ składa się z posłów, których obecność w komisji jest wymuszona przez obecność w niej posła x. Zbiory tego typu mają następującą własność.

Jeśli K jest spokojną Komisją i $x \in K$, to $Wymuszone(x) \subseteq K$.

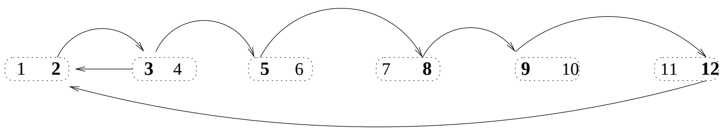


Rys. 1. Przykładowy graf konfliktowy dla 12 partii. Możliwym rozwiązaniem jest zbiór posłów, których numery na rysunku są pogrubione.

Posła x nazwiemy ***kłopotliwym***, jeśli zbiór $Wymuszone(x)$ zawiera dwóch posłów z tej samej partii (co jest niedozwolone). Na rysunku 2 (dla grafu konfliktowego z rysunku 1) poseł 1 jest kłopotliwy, natomiast poseł 2 z rysunku 3 jest niekłopotliwy.



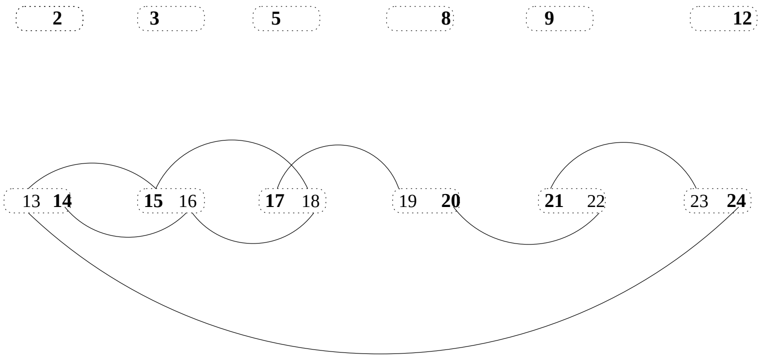
Rys. 2. Część grafu wymuszeń zaczynającego się od 1. Poseł 1 jest kłopotliwy, $\{1, \; 2\} \subseteq Wymuszone(1)$, a 1, 2 są posłami tej samej partii.



Rys. 3. Poseł 2 jest niekłopotliwy: $Wymuszone(2) \; = \; \{2, \; 3, \; 5, \; 8, \; 9, \; 12\}$.

W celu sformowania Komisji wykonujemy następujący algorytm:

Algorytm KOMISJA1;
początkowo zbiorem reprezentantów jest $K = \emptyset$;
while $|K| < n$ **do**
begin
niech π będzie pierwszą partią bez reprezentanta w K;
if obaj posłowie należący do π są kłopotliwi **then**
nie ma rozwiązania, STOP
else
begin
wybierz pierwszego niekłopotliwego posła $x \in \pi$;
$K := K \cup Wymuszone(x)$
end
end
return K;



Rys. 4. Po pierwszej iteracji otrzymujemy reprezentantów pierwszych sześciu partii. Jeśli odrzucimy pozostałych posłów z tych partii, to posłowie z sześciu partii pozostałych są niezależni (w sensie grafu konfliktowego) od wybranych posłów $2, 3, 5, 8, 9, 12$.

Zobaczmy jak działa ten algorytm na naszym przykładzie (Rys. 4). Na początku wybieramy pierwszą partię. Poseł 1 jest kłopotliwy, zatem wybieramy posła 2. $Wymuszone(2) = \{2,\ 3,\ 5,\ 8,\ 9,\ 12\}$. Zbiorem K staje się $\{2,\ 3,\ 5,\ 8,\ 9,\ 12\}$. Następną partią bez reprezentanta jest $\{13,\ 14\}$. Poseł 13 jest kłopotliwy, dodajemy do K zbiór $Wymuszone(14)$. Otrzymujemy końcowy rezultat.

Dowód poprawności.
Algorytm wybiera reprezentanta x z pierwszej partii, a następnie tworzy $K = Wymuszone(x)$. Partie bez reprezentantów w K są, w sensie grafu konfliktowego, całkowicie niezależne od K (zobacz rysunek 4).

Własność niezależności.
Niech $U = Wymuszone(x)$ i niech W będzie sumą teoriomnogościową partii, które są rozłączne z U. Jeśli x jest niekłopotliwy, to w grafie konfliktowym nie ma krawędzi między U i W.

Algorytm znajduje reprezentantów dla pozostałych partii, tak jak gdyby startował od początku. Formalnie poprawność można wykazać indukcyjnie ze względu na liczbę partii.

Zamiast umieszczać w komisji K grupy posłów możemy powiększać komisję po jednym pośle, stosując następującą wersję algorytmu KOMISJA1. Jest to wersja łatwiejsza do zaprogramowania, natomiast poprzednia wersja jest łatwiejsza z punktu widzenia poprawności. W tej wersji dostaniemy dokładnie tę samą komisję.

Oznaczmy i-tą partię przez π_i. Powiemy, że poseł x jest zgodny ze zbiorem posłów w K, gdy x nie jest w konflikcie z żadnym z posłów w K.

> **Algorytm KOMISJA**;
> początkowo zbiorem reprezentantów jest $K = \emptyset$;
> **for** $i := 1$ **to** n **do**
> **begin**
> wybierz pierwszego niekłopotliwego posła $x \in \pi_i$ zgodnego z K;
> **if** nie ma takiego posła **then** nie ma rozwiązania, STOP;
> $K := K \cup \{x\}$
> **end**
> **return** K;

Algorytm Komisja zaimplementowany bezpośrednio może działać zbyt wolno. Sprawdzenie, czy poseł jest niekłopotliwy może wymagać czasu proporcjonalnego do rozmiaru grafu wymuszeń. Zatem łączny czas działania algorytmu wynosi $O(n \cdot (n+m))$. W programie wzorcowym Komisja wybierana jest trochę sprytniej. Rozpoczynamy od znalezienia w grafie wymuszeń wszystkich silnie spójnych składowych, czyli podgrafów, w których od każdego wierzchołka można przejść do każdego innego wierzchołka idąc po strzałkach. Silnie spójne składowe można znaleźć w czasie proporcjonalnym do rozmiaru grafu (zobacz [11]). Jeśli istnieje choć jedna silnie spójna składowa zawierająca posłów z tej samej partii, to oczywiście "spokojnej" Komisji nie da się sformować. W przeciwnym przypadku patrzymy na graf silnie spójnych składowych — wierzchołkami są silnie silnie spójne składowe, a od składowej A prowadzi krawędź do B tylko wtedy, gdy w wyjściowym grafie istnieje krawędź od pewnego wierzchołka z A do pewnego wierzchołka z B. Następnie silnie spójne składowe sortujemy topologicznie i rozważamy je w otrzymanej kolejności. Jeśli aktualnie rozważana silnie spójna składowa nie została wcześniej odrzucona, to wybieramy wszystkich należących do niej posłów do Komisji. Dla każdego wybranego w ten sposób posła, odrzucamy silnie spójną składową, która zawiera jego partyjnego kolegę, oraz wszystkie składowe, z których ta składowa jest osiągalna. Jeśli w ten sposób wybierzemy n posłów odpowiadamy TAK, a przeciwnym razie NIE.

Testy

Zadanie testowane było na zestawie 13 danych testowych:

- `spo1.in` — mały test poprawnościowy;
- `spo2.in` — mały test poprawnościowy;
- `spo3.in` — średni test poprawnościowy;
- `spo4a.in` — średni test z odpowiedzią NIE, dla którego "backtracking" działa wykładniczo;
- `spo4b.in` — średni test wydajnościowy, dla którego komisja daje się sformować;
- `spo5a.in` — średni test z odpowiedzią NIE, dla którego "backtracking" działa wykładniczo;
- `spo5b.in` — średni test, dla którego komisja daje się sformować;
- `spo6a.in` — duży test z odpowiedzią NIE, dla którego "backtracking" działa wykładniczo;
- `spo6b.in` — duży test, dla którego komisja istnieje;
- `spo7.in` — test o maksymalnym rozmiarze;
- `spo8a.in` — duży test, dla którego rozwiązanie KOMISJA działa w czasie $O(n^2)$;
- `spo8b.in` — duży test, dla którego rozwiązanie KOMISJA działa w czasie $O(n^2)$ (trochę inny wariant);
- `spo8c.in` — duży test losowy z odpowiedzią TAK.

Testy $4a$ i $4b$, $5a$ i $5b$, $6a$ i $6b$ oraz $8a$, $8b$ i $8c$ były zgrupowane.

Wojciech Guzicki
Treść zadania, Opracowanie

Andrzej Gąsienica–Samek
Program

Wyspa

Na wyspie o kształcie wielokąta wypukłego o $2n$ bokach, znajduje się $2n-2$ państw — trójkątów, których wierzchołki są jednocześnie wierzchołkami wielokąta. Nie ma państw graniczących dokładnie z dwoma innymi państwami (zatem każde państwo graniczy albo tylko z jednym państwem, albo z trzema). Wynika stąd, że istnieje dokładnie n państw graniczących tylko z jednym państwem (są to państwa **nadmorskie***) oraz $n-2$ państw graniczących z trzema sąsiadami (są to państwa* **wewnątrzlądowe***). Państwa nadmorskie są ponumerowane liczbami od 1 do n, natomiast państwa wewnątrzlądowe mają numery od $n+1$ do $2n-2$. Gdy podróżujemy z jednego państwa do drugiego, to za przekroczenie każdej granicy musimy zapłacić ustaloną stawkę. Poszczególne stawki mogą być różne, ale przekroczenie granicy w obu kierunkach kosztuje tyle samo.*

Dla każdych dwóch państw, spośród n państw nadmorskich, znana jest suma opłat granicznych na drodze prowadzącej (lądem) od jednego państwa do drugiego przez najmniejszą liczbę granic. Zadanie polega na wyznaczeniu wszystkich opłat granicznych na całej wyspie. Dla każdego państwa nadmorskiego należy podać numer państwa, z którym ono graniczy, oraz wysokość odpowiedniej opłaty granicznej. Ponadto, dla każdego z $n-2$ państw wewnątrzlądowych należy podać numery trzech państw, z którymi ono graniczy, oraz wysokości opłat na granicach z tymi państwami.

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `wys.in` *sumy opłat granicznych na drogach pomiędzy poszczególnymi państwami nadmorskimi,*
- *obliczy opłaty graniczne na całej wyspie i wyznaczy, które państwa sąsiadują z którymi,*
- *zapisze wyniki w pliku tekstowym* `wys.out`*.*

Wejście

W pierwszym wierszu pliku tekstowego `wys.in` *znajduje się jedna dodatnia liczba całkowita n, $4 \leq n \leq 100$. Jest to liczba państw nadmorskich. W każdym z następnych n wierszy znajduje się n nieujemnych liczb całkowitych oddzielonych pojedynczymi odstępami. Liczba $d_{i,j}$, stojąca na j-tym miejscu w i-tym z tych wierszy, jest równa sumie opłat granicznych na drodze prowadzącej (lądem, przez najmniejszą liczbę granic) z państwa o numerze i do państwa o numerze j. Zakładamy przy tym, że na każdej granicy opłata graniczna jest liczbą całkowitą z przedziału $[1 \ldots 100]$. Oczywiście, $d_{i,j} = d_{j,i}$ oraz $d_{i,i} = 0$.*

Wyjście

W pierwszych n wierszach pliku tekstowego `wys.out` *należy zapisać po dwie liczby całkowite oddzielone pojedynczym ostępem. Pierwszą liczbą w i-tym wierszu powinien być numer państwa graniczącego z państwem o numerze i, a drugą wysokość opłaty granicznej na granicy między tymi dwoma państwami. W każdym z następnych $n-2$ wierszy należy zapisać po sześć liczb oddzielonych pojedynczymi odstępami. W wierszu o numerze i (licząc od początku pliku, a więc $i > n$) pierwszą liczbą ma być numer pierwszego państwa graniczącego z państwem o numerze i, drugą ma być wysokość opłaty granicznej na tej granicy, trzecią ma być numer drugiego państwa graniczącego z państwem o numerze i, czwartą wysokość opłaty granicznej na drugiej granicy, piątą ma być numer trzeciego państwa graniczącego z państwem o numerze i, szóstą ma być wysokość opłaty granicznej na trzeciej granicy. Państwa wewnątrzlądowe mogą być ponumerowane dowolnie liczbami od $n+1$ do $2n-2$.*

Przykład

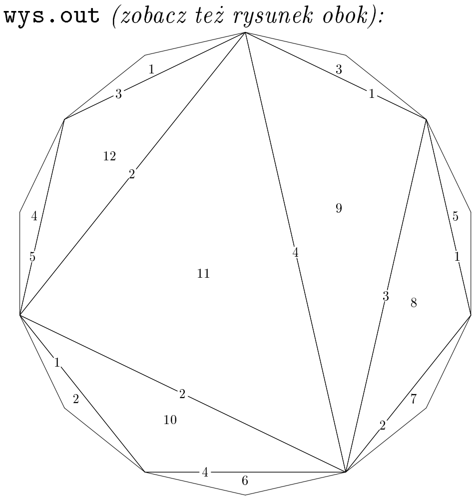
Dla pliku przykładowego `wys.in`*:*

```
7
0 8 10 8 13 11 14
8 0 8 10 11 5 12
```

```
10 8 0 12 5 11 6
8 10 12 0 15 13 16
13 11 5 15 0 14 3
11 5 11 13 14 0 15
14 12 6 16 3 15 0
```

poprawną odpowiedzią jest plik wyjściowy `wys.out` *(zobacz też rysunek obok):*

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



Rozwiązanie

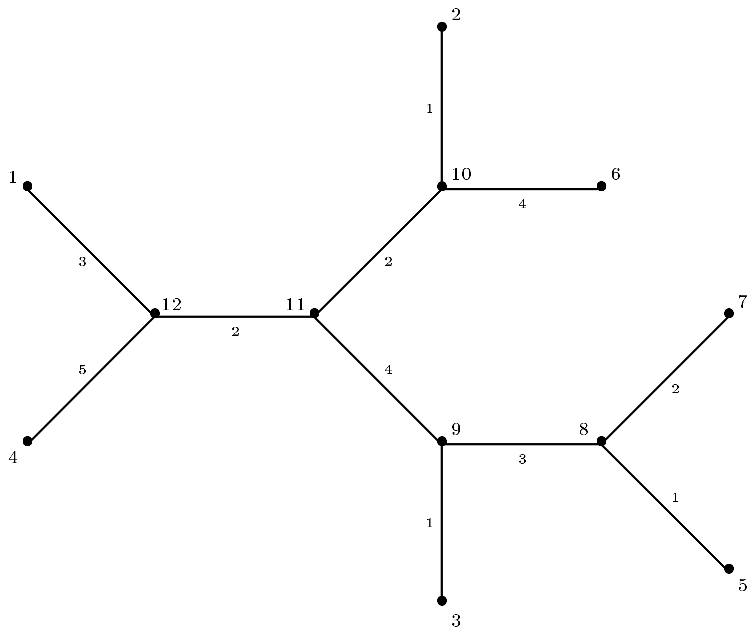
Będziemy używać terminologii teorii grafów — ułatwi to zaprezentowanie możliwych algorytmów prowadzących do rozwiązania zadania. Najprościej mówiąc, grafem nazywamy skończony zbiór punktów na płaszczyźnie (te punkty będziemy zaznaczać na rysunku „grubymi” kropkami) oraz pewien zbiór odcinków łączących te punkty. Wybrane punkty nazywamy **wierzchołkami** grafu, wybrane odcinki **krawędziami** grafu. Te krawędzie mogą się przecinać, będziemy tylko zakładać, że punkt przecięcia dwóch krawędzi nie może być wierzchołkiem grafu. Wierzchołki grafu będziemy numerować początkowymi liczbami naturalnymi, zaczynając od 1.

Krawędziom grafu będziemy przyporządkowywać pewne liczby rzeczywiste. Liczbę rzeczywistą przyporządkowaną krawędzi łączącej wierzchołki k i l będziemy nazywać **długością** tej krawędzi i będziemy oznaczać symbolem $d_{k,l}$.

Liczbę krawędzi wychodzących z jednego wierzchołka nazywamy **stopniem** tego wierzchołka.

W naszym przypadku wierzchołki grafu będą odpowiadać państwom na wyspie. Dwa wierzchołki łączymy krawędzią, jeśli odpowiadające im państwa graniczą ze sobą. Wreszcie długością krawędzi będzie koszt przekraczania granicy. Zauważmy, że w naszym grafie stopień każdego wierzchołka jest równy 1 (dla państw nadmorskich) lub 3 (dla państw wewnątrzlądowych). Nasz graf ma jeszcze jedną ważną własność wynikającą z geometrii wyspy: nie ma w nim cykli. To znaczy, że nie można wyjść z jednego wierzchołka, poruszać się po krawędziach przez inne wierzchołki i powrócić do punktu wyjścia nie przechodząc przez żaden z wierzchołków więcej niż raz. Ponadto z każdego wierzchołka można dojść do każdego innego. Takie grafy nazywamy **drzewami**. Jedną z ważnych własności drzew, wynikającą wprost z tego, że w drzewie nie ma cykli, jest to, że każde dwa wierzchołki łączy tylko jedna droga przebiegająca wzdłuż krawędzi. Długość tej jedynej drogi łączącej wierzchołki k i l (równa sumie długości krawędzi, przez które ta droga przebiega) oznaczymy również symbolem $d_{k,l}$. Jeśli dwa wierzchołki są połączone krawędzią, to oczywiście drogą łączącą je jest ta krawędź, a więc długość tej drogi jest równa długości krawędzi. Zatem użycie tego samego oznaczenia na długość krawędzi i długość drogi nie będzie prowadzić do nieporozumień.

Popatrzmy teraz na przykład. Drzewo odpowiadające wyspie z treści zadania ma postać:



Wierzchołki stopnia 1 nazywamy **liśćmi** drzewa. Pozostałe wierzchołki będziemy nazywać **węzłami**.

Danymi dla zadania są długości dróg łączących liście drzewa. Zadanie polega na tym, by odtworzyć drzewo mając dane wyłącznie odległości między liśćmi. Jest to możliwe dzięki założeniu, że wszystkie węzły mają stopień równy 3. Najprostszy algorytm odtwarzania drzewa polega na znajdowaniu dwóch sąsiednich liści (dwa liście nazywamy *sąsiednimi*, gdy sąsiadują z tym samym węzłem) i zastępowaniu ich węzłem, z którym sąsiadują. Przypuśćmy, że liście o numerach i oraz j sąsiadują z węzłem o numerze k. Usuwamy z tablicy kosztów wiersze i kolumny o numerach i oraz j i dopisujemy nowy wiersz i nową kolumnę o numerze k. W jaki sposób obliczamy długość drogi z węzła k do dowolnego innego liścia m? Zauważmy, że

$$d_{i,m} + d_{j,m} = 2 \cdot d_{k,m} + d_{i,j}.$$

Stąd otrzymujemy wzór

$$d_{k,m} = \frac{1}{2} \cdot (d_{i,m} + d_{j,m} - d_{i,j}).$$

Oczywiście długości krawędzi (i,k) i (j,k) obliczamy natychmiast ze wzorów

$$d_{i,k} = d_{i,m} - d_{k,m}$$

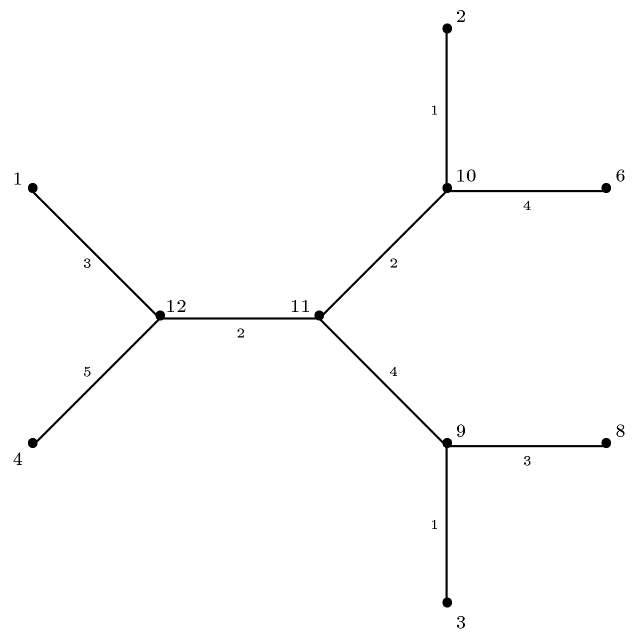
oraz

$$d_{j,k} = d_{j,m} - d_{k,m}.$$

Tak więc, gdy usuwamy dwa liście, tworzymy nowy liść o kolejnym numerze, znajdujemy odległości od tego nowego liścia do pozostałych liści i obliczamy długości usuniętych krawędzi. Wszystkie te dane zapisujemy w odpowiednich tabelach; wypiszemy je po zakończeniu algorytmu. Popatrzmy teraz na tabelę długości dróg.

	1	**2**	**3**	**4**	**5**	**6**	**7**
1	0	8	10	8	13	11	14
2	8	0	8	10	11	5	12
3	10	8	0	12	4	11	6
4	8	10	12	0	16	13	16
5	13	11	4	16	0	14	3
6	11	5	11	13	14	0	15
7	14	12	6	16	3	15	0

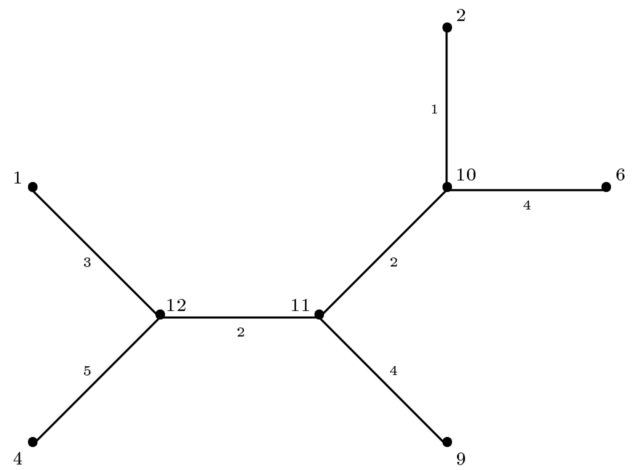
Weźmy sąsiednie liście o numerach 5 i 7. Usuwając je z drzewa otrzymujemy drzewo mniejsze, w którym mamy nowy liść o numerze 8:



Obliczając długości dróg z liścia 8 do pozostałych liści według powyższego wzoru, otrzymamy następującą tabelę długości dróg między liśćmi nowego drzewa:

	1	**2**	**3**	**4**	**6**	**8**
1	0	8	10	8	11	12
2	8	0	8	10	5	10
3	10	8	0	12	11	4
4	8	10	12	0	13	14
6	11	5	11	13	0	13
8	12	10	4	14	13	0

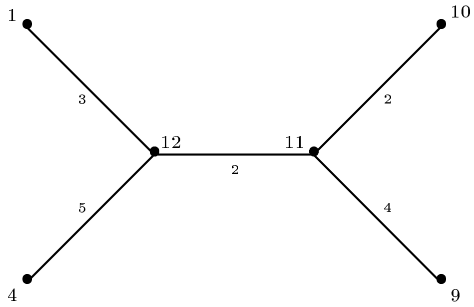
Teraz znajdujemy następne dwa sąsiednie liście: na przykład 3 i 8. Usuwamy je z drzewa i do tablicy długości dróg dopisujemy nowy liść o numerze 9. Nowe drzewo ma postać:



a odpowiednia tabela długości dróg między liśćmi tego drzewa ma postać:

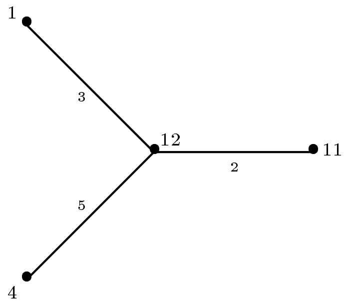
	1	**2**	**4**	**6**	**9**
1	0	8	8	11	9
2	8	0	10	5	7
4	8	10	0	13	11
6	11	5	13	0	10
9	9	7	11	10	0

Kontynuujemy to postępowanie. Znajdujemy kolejne sąsiednie liście: 2 i 6. Usuwamy je z drzewa i z tablicy długości dróg, dopisując nowy liść o numerze 10. Otrzymamy następne drzewo i odpowiadającą mu tablicę długości dróg:



	1	**4**	**9**	**10**
1	0	8	9	7
4	8	0	11	9
9	9	11	0	6
10	7	9	6	0

Jeszcze raz usuwamy dwa sąsiednie liście: tym razem o numerach 9 i 10. Dopisujemy nowy liść o numerze 11. Otrzymamy nowe drzewo i odpowiadającą mu tablicę długości dróg:

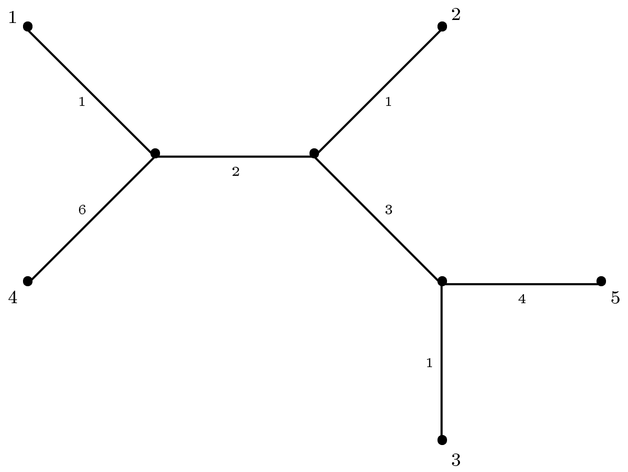


	1	**4**	**11**
1	0	8	5
4	8	0	7
11	5	7	0

Mamy teraz drzewo z trzema liśćmi. Ma ono tylko jeden wierzchołek stopnia 3: dajmy mu numer 12. Oczywiście teraz mamy

$$d_{1,12} = \frac{1}{2} \cdot (d_{1,11} + d_{1,4} - d_{4,11}),$$
$$d_{4,12} = \frac{1}{2} \cdot (d_{1,4} + d_{4,11} - d_{1,11}),$$
$$d_{11,12} = \frac{1}{2} \cdot (d_{4,11} + d_{1,11} - d_{1,4}).$$

W ten sposób całe drzewo zostało odtworzone. Jedynym istotnym problemem w tym algorytmie jest to, w jaki sposób znajdować dwa sąsiednie liście. W powyższym przykładzie za każdym razem wybieraliśmy dwa liście położone najbliżej siebie (tzn. liście i oraz j, dla których liczba $d_{i,j}$ jest najmniejsza). **To udało się tylko dlatego, że drzewo z przykładu zostało specjalnie dobrane!** Ten sposób wybierania sąsiednich liści na ogół jest niepoprawny. Popatrzmy na inny przykład drzewa i odpowiadającej mu tabeli odległości:



	1	**2**	**3**	**4**	**5**
1	0	4	7	7	10
2	4	0	5	9	8
3	7	5	0	12	5
4	7	9	12	0	15
5	10	8	5	15	0

Zauważmy, że najbliżej położone liście (o numerach 1 i 2) nie są sąsiednie! Gdybyśmy mimo to próbowali zastosować poprzedni algorytm do tej sytuacji, otrzymalibyśmy kolejno następujące tabele odległości:

	3	**4**	**5**	**6**
3	0	12	5	4
4	12	0	15	6
5	5	15	0	7
6	4	6	7	0

	4	**5**	**7**
4	0	15	7
5	15	0	4
7	7	4	0

Teraz poprzednie wzory dadzą nam następujące długości krawędzi łączących liście 4, 5 i 7 z wierzchołkiem 8:

$$d_{4,8} = 9, \quad d_{5,8} = 6 \quad \text{oraz} \quad d_{7,8} = -2.$$

Algorytm zakończył się niepowodzeniem, gdyż długość krawędzi nie może być liczbą ujemną.

Istnieją dwie metody znajdowania sąsiednich liści. Jedna z nich polega na obliczeniu dla każdego liścia i wielkości

$$r_i = \sum_j d_{i,j},$$

gdzie sumowanie jest rozciągnięte na wszystkie liście. Następnie dla każdej pary liści i oraz j obliczamy

$$D_{i,j} = d_{i,j} - \frac{r_i + r_j}{n-2}.$$

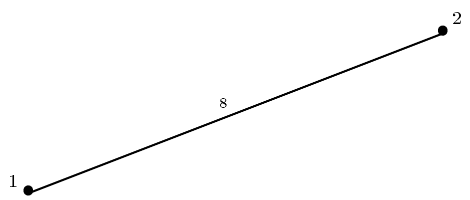
Można wtedy udowodnić twierdzenie mówiące, że liście i oraz j, dla których wartość $D_{i,j}$ jest najmniejsza, są sąsiednie.

Twierdzenie to jest dość sztuczne i nieoczekiwane. Potrzebny jest sposób prostszy i bardziej naturalny. Wybieramy dowolny liść, na przykład o najmniejszym numerze. Niech będzie to numer k. Szukamy liścia sąsiadującego z liściem o numerze l. W tym celu dla każdego liścia m wyznaczamy długość wspólnego odcinka dróg z k do l i z k do m. Jeśli te drogi rozchodzą się w węźle p, to

$$d_{k,p} = \frac{1}{2} \cdot (d_{k,l} + d_{k,m} - d_{l,m}).$$

Możliwe są teraz dwa przypadki. Jeśli liście k i l sąsiadują ze sobą, to węzeł p wypada zawsze w tym samym miejscu i to można łatwo stwierdzić. Jeśli liście k i l nie sąsiadują ze sobą, to znaleziona długość $d_{k,p}$ jest największa wtedy, gdy liść m sąsiaduje z liściem l. W obu przypadkach znajdujemy parę liści sąsiednich: najczęściej będzie to liść l i pewien liść m, może się też zdarzyć, że będą to liście k i l. Poszukiwanie takiej pary liści sąsiednich wymaga czasu $O(n)$, zatem cały algorytm działa w czasie $O(n^2)$.

Inny algorytm polega na sukcesywnym budowaniu drzewa. Zaczynamy od drzewa złożonego tylko z liści o numerach 1 i 2 i krawędzi łączącej te liście. Ma ona długość $d_{1,2}$. W naszym przykładzie to drzewo ma postać:



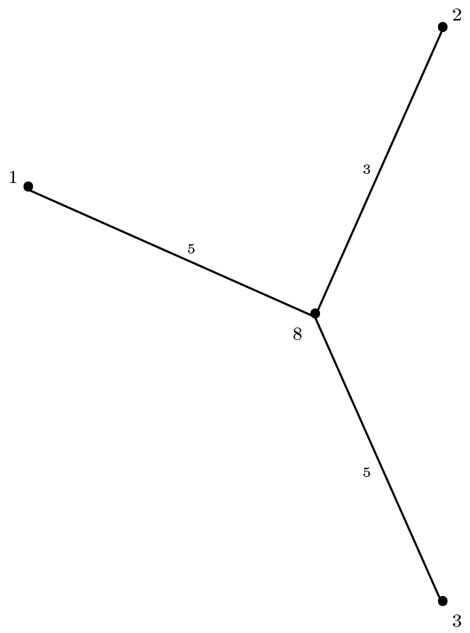
Teraz dołączamy następny liść. Na drodze z liścia 1 do liścia 2 znajduje się węzeł (nadamy mu numer 8), w którym odgałęzia się droga do liścia 3. Można łatwo wyznaczyć długość odcinka od liścia 3 do węzła 8:

$$d_{1,3} + d_{2,3} = 2 \cdot d_{8,3} + d_{1,2}.$$

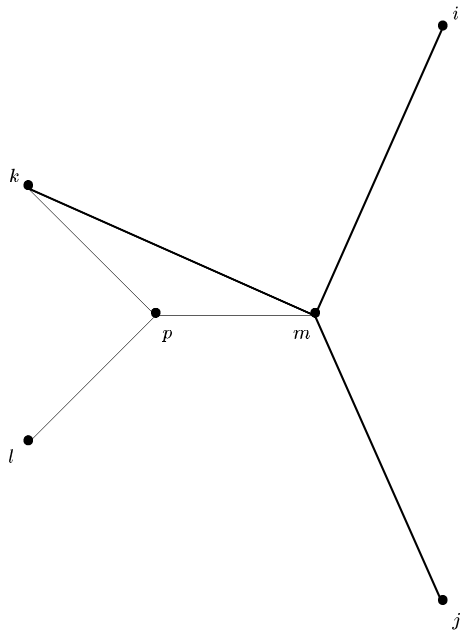
Stąd dostajemy $d_{8,3} = 5$ i następnie

$$d_{1,8} = d_{1,3} - d_{3,8} = 5, \quad d_{2,8} = d_{2,3} - d_{3,8} = 3.$$

Możemy teraz narysować drzewo:



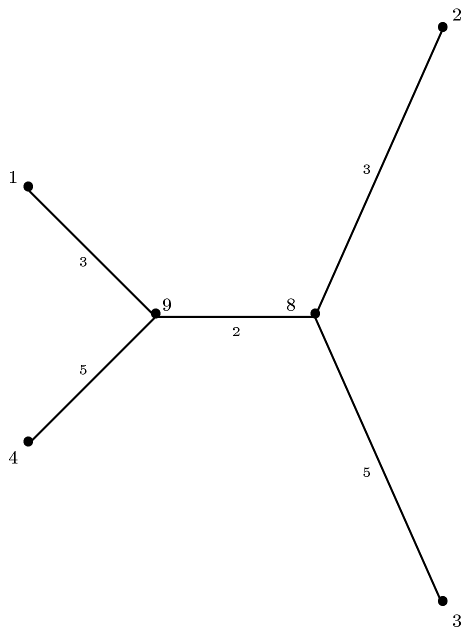
Teraz dołączamy kolejny liść, o numerze 4, wraz z węzłem sąsiadującym z nim. Mamy trzy możliwości. Droga prowadząca z liścia 4 do któregokolwiek z pozostałych liści musi „dołączyć" się do jednej z dróg: z liścia 1 do węzła 8, z liścia 2 do węzła 8 lub z liścia 3 do węzła 8. Oznaczmy liście i węzły literami: węzeł 8 oznaczmy literą m, liść 4 oznaczmy literą l, literą k oznaczmy ten z liści 1,2 lub 3, dla którego droga z k do l nie przechodzi przez m. Pozostałe dwa liście oznaczymy literami i oraz j. Wreszcie literą p oznaczymy węzeł na drodze z k do m, w którym odgałęzia się droga do l. Tę sytuację możemy przedstawić na rysunku:



Na rysunku grubszą linią zaznaczono krawędzie drzewa przed dołączeniem liścia l i węzła p, cieńszymi liniami zaznaczono krawędzie drzewa po dołączeniu tych nowych wierzchołków. Zauważmy, że musi być spełniona nierówność

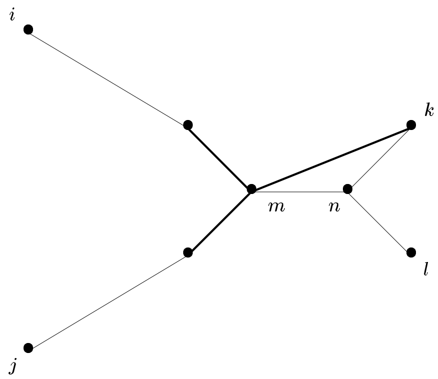
$$d_{i,k} + d_{j,l} > d_{i,j} + d_{k,l}.$$

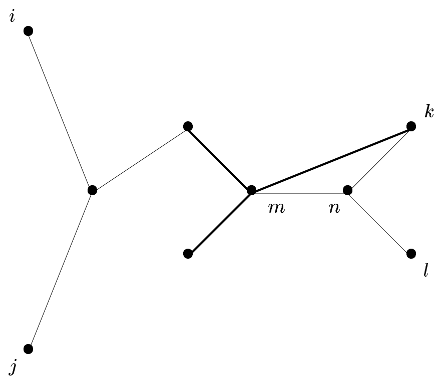
Sprawdzamy zatem tę nierówność dla trzech wyborów k: 1, 2 lub 3. Jedna z tych nierówności będzie spełniona; pokaże ona, w którym miejscu należy dołączyć do drzewa liść l i węzeł p. W naszym przykładzie okaże się, że liść k będzie liściem 1 i węzeł p dołączymy na drodze z liścia 1 do liścia 8. Węzłowi p nadamy teraz kolejny numer, tzn. 9. Tak samo jak poprzednio wyznaczamy odległości między węzłem 9 i wierzchołkami z nim sąsiadującymi. Otrzymamy nowe drzewo:



Teraz musimy dołączyć do drzewa następny liść, o numerze 5. Możliwe są dwa przypadki: droga od liścia 5 do zbudowanego dotychczas drzewa może „dołączyć się" do krawędzi łączącej liść z sąsiednim węzłem lub do krawędzi łączącej dwa węzły. Można łatwo stwierdzić, czy ta droga dołączyła się do krawędzi wychodzącej z liścia.

Przypuśćmy, że mamy dane drzewo i dołączamy nowy liść l. Załóżmy, że droga z liścia l do drzewa dołącza się do krawędzi łączącej liść k z sąsiednim węzłem m. Oznaczmy literą n węzeł, w którym ta droga dołącza się do krawędzi łączącej k z m. Niech wreszcie i i j będą dwoma dowolnymi liśćmi naszego drzewa, różnymi od liścia k. Na następnych dwóch rysunkach widzimy możliwe położenia tych liści i węzłów (grubą linią są zaznaczone krawędzie drzewa, cienką linią drogi w drzewie i nowe krawędzie po dołączeniu liścia l i węzła n).





Sytuacja ta jest możliwa tylko wtedy, gdy zachodzi nierówność

$$d_{i,k} + d_{j,l} > d_{i,j} + d_{k,l}.$$

Ten warunek można sprawdzić dla każdej krawędzi łączącej liść drzewa z sąsiednim węzłem. Może się jednak okazać, że żadna z tych krawędzi nie jest właściwa, czyli powyższa nierówność nie jest prawdziwa. Nowy liść musimy wtedy dołączyć do drzewa w krawędzi łączącej dwa węzły. Jak stwierdzić, która to krawędź?

Można postąpić dwojako. Przypuśćmy, że mamy dołączyć nowy liść l w węźle n znajdującym się na krawędzi łączącej dwa węzły m i k. Niech i i j będą dwoma pozostałymi sąsiadami węzła m. Możemy obliczyć odległości od liścia l do wierzchołków i, j i k (ćwiczeniem dla Czytelnika będzie, jak to zrobić). Dalej postępujemy tak samo: musi być spełniona nierówność

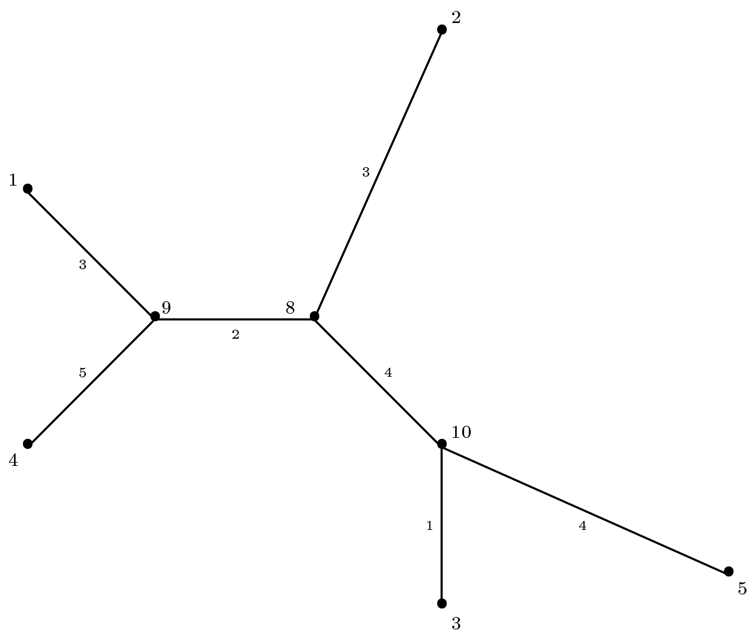
$$d_{i,k} + d_{j,l} > d_{i,j} + d_{k,l}.$$

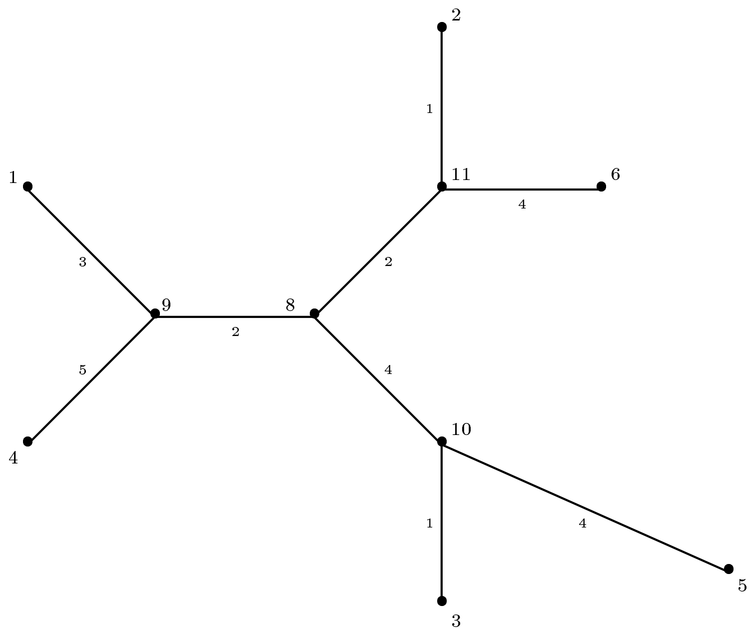
Można też postąpić inaczej. Na czas poszukiwania właściwej krawędzi usuwamy z drzewa krawędzie niewłaściwe. Dokładniej, przeszukujemy krawędzie łączące liście z sąsiednimi węzłami. Jeśli znajdziemy krawędź właściwą, to dołączamy nowy liść. Jeśli badana krawędź nie jest właściwa, to usuwamy ją wraz z liściem, którym ona się kończy. Jej drugi koniec może stać się teraz nowym liściem (zauważmy, że mogą ulec zmianie stopnie węzłów!). Wreszcie znajdziemy krawędź właściwą i wtedy dołączamy z powrotem krawędzie usunięte z drzewa. Ta metoda została użyta w programie wzorcowym.

Oczywiście długości krawędzi obliczamy korzystając z tych samych wzorów co poprzednio.

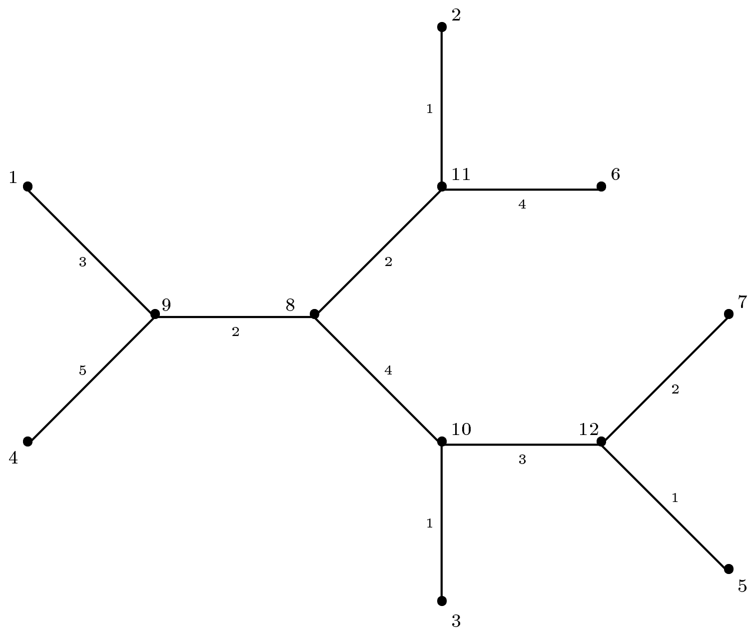
W ten sposób dołączamy kolejne liście i po dołączeniu ostatniego drzewo zostało zrekonstruowane. Czas działania tego algorytmu jest również rzędu $O(n^2)$.

W naszym przykładzie kolejne liście zostaną dołączone do krawędzi łączących inne liście z sąsiednimi węzłami. Na dalszych rysunkach widzimy kolejne drzewa powstające przez dołączenie następnych liści.





I wreszcie drzewo, o które chodziło:



Zadanie to ma zastosowanie w genetyce do wyznaczania tzw. drzew filogenetycznych. Są to drzewa, których liśćmi są gatunki zwierząt lub roślin, a odległości wskazują, jak bardzo dane gatunki są odległe od siebie ewolucyjnie. Rekonstrukcja drzewa sugeruje możliwe kierunki ewolucji: wskazuje, które gatunki mogły powstać bezpośrednio z innych (są to gatunki połączone krawędzią).

Testy

Testy zostały stworzone automatycznie, charakteryzuje je 5 wartości, przy tworzeniu drzewa binarnego:

- n,
- p_{min} — minimalny procent liści, jaki będzie miało lewe poddrzewo,
- p_{max} — maksymalny procent,
- o_{min} — minimalna opłata graniczna,

- o_{max} — maksymalna opłata.

Jeśli procent liści jest bliski 50, to otrzymamy drzewo pełne. Jeśli bliski 0, będzie to lista. Jeśli będzie dowolny od 0 do 100 to otrzymamy drzewo losowe, czyli zrównoważone.

test	n	p_{min}	p_{max}	o_{min}	o_{max}
wys1.in	6	0	100	1	7
wys2.in	10	0	100	1	7
wys3.in	15	0	100	1	1
wys4.in	15	0	5	1	10
wys5.in	15	45	55	1	10
wys6.in	20	0	100	1	20
wys7.in	40	0	100	1	50
wys8.in	40	0	5	1	50
wys9.in	70	45	45	1	1
wys10.in	100	50	50	1	100
wys11.in	100	0	5	100	100
wys12.in	100	0	100	1	100

Krzysztof Loryś
Treść zadania, Opracowanie

Tomasz Waleń
Program

Mrówki i biedronka

Jak wiadomo, mrówki potrafią "hodować" mszyce. Mszyce wydzielają słodką rosę miodową, którą spijają mrówki. Mrówki zaś bronią mszyc przed ich największymi wrogami — biedronkami.

Na drzewie obok mrowiska znajduje się właśnie taka hodowla mszyc. Mszyce żerują na liściach oraz w rozgałęzieniach drzewa. W niektórych z tych miejsc znajdują się również mrówki patrolujące drzewo. Dla ustalenia uwagi, mrówki są ponumerowane od jeden w górę. Hodowli zagraża biedronka, która zawsze siada na drzewie tam, gdzie są mszyce, czyli na liściach lub w rozgałęzieniach. W chwili, gdy gdzieś na drzewie usiądzie biedronka, mrówki patrolujące drzewo ruszają w jej stronę, aby ją przegonić. Kierują się przy tym następującymi zasadami:

- *z każdego miejsca na drzewie (liścia lub rozgałęzienia) można dojść do każdego innego miejsca (bez zawracania) tylko na jeden sposób; każda mrówka wybiera właśnie taką drogę do miejsca lądowania biedronki,*
- *jeżeli w miejscu lądowania biedronki znajduje się mrówka, biedronka natychmiast odlatuje,*
- *jeżeli na drodze, od aktualnego położenia mrówki do miejsca lądowania biedronki, znajdzie się inna mrówka, to ta położona dalej od biedronki kończy wędrówkę i zostaje w miejscu swojego aktualnego położenia,*
- *jeżeli dwie lub więcej mrówek próbuje wejść na to samo rozgałęzienie drzewa, to robi to tylko jedna mrówka — ta z najmniejszym numerem, a reszta mrówek pozostaje na swoich miejscach (liściach lub rozgałęzieniach),*
- *mrówka, która dociera do miejsca lądowania biedronki, przegania ją i pozostaje w tym miejscu.*

Biedronka jest uparta i znowu ląduje na drzewie. Wówczas mrówki ponownie ruszają, aby przegonić intruza.

Dla uproszczenia przyjmujemy, że przejście gałązki łączącej liść z rozgałęzieniem lub łączącej dwa rozgałęzienia, zajmuje wszystkim mrówkom jednostkę czasu.

Zadanie

Napisz program, który:

- *wczyta z pliku wejściowego* `mro.in` *opis drzewa, początkowe położenia mrówek oraz miejsca, w których kolejno siada biedronka,*
- *dla każdej mrówki znajdzie jej końcowe położenie i wyznaczy liczbę mówiącą, ile razy przegoniła ona biedronkę.*

Wejście

W pierwszym wierszu pliku tekstowego `mro.in` *znajduje się jedna liczba całkowita n, równa łącznej liczbie liści i rozgałęzień w drzewie, $1 \le n \le 5000$. Przyjmujemy, że liście i rozgałęzienia są ponumerowane od 1 do n. W kolejnych $n-1$ wierszach są opisane gałązki — w każdym z tych wierszy są zapisane dwie liczby całkowite a i b oznaczające, że dana gałązka łączy miejsca a i b. Gałązki pozwalają na przejście z każdego miejsca na drzewie, do każdego innego miejsca. W $n+1$-szym wierszu jest zapisana jedna liczba całkowita k, $1 \le k \le 1000$ i $k \le n$, równa liczbie mrówek patrolujących drzewo. W każdym z kolejnych k wierszy zapisana jest jedna liczba całkowita z przedziału od 1 do n. Liczba zapisana w wierszu $n+1+i$ oznacza początkowe położenie mrówki nr i. W każdym miejscu (liściu lub rozgałęzieniu) może znajdować się co najwyżej jedna mrówka. W wierszu $n+k+2$ zapisana jest jedna liczba całkowita l, $1 \le l \le 500$, mówiąca ile razy biedronka siada na drzewie. W każdym z kolejnych l wierszy zapisana jest jedna liczba całkowita z zakresu od 1 do n. Liczby te opisują kolejne miejsca, w których siada biedronka.*

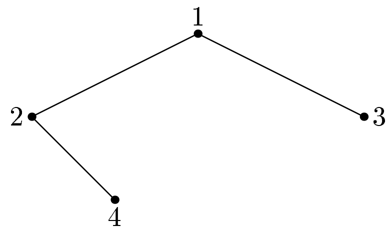
Wyjście

Twój program powinien zapisać k wierszy w pliku wyjściowym `mro.out`*. W i-tym wierszu powinny zostać zapisane dwie liczby całkowite oddzielone pojedynczym odstępem — końcowa pozycja i-tej mrówki (numer rozgałęzienia lub liścia) i liczba mówiąca, ile razy przegoniła ona biedronkę.*

Przykład

Dla pliku wejściowego `mro.in`*:*

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



Rysunek 1. Drzewo dla przykładowych danych

poprawną odpwiedzią jest plik wyjściowy `mro.out`*:*

```
1 0
4 2
```

Rozwiązanie

Narzucający się schemat rozwiązania polega na symulacji kolejnych zdarzeń (lądowań biedronki i pogoni mrówek). Po wylądowaniu biedronki w wierzchołku drzewa (tj. liściu lub rozgałęzieniu) symulujemy ruchy mrówek. Oznaczmy przez B wierzchołek, w którym wylądowała biedronka. Kolejno dla każdej mrówki (wg rosnących numerów) sprawdzamy czy droga między nią a wierzchołkiem B jest wolna, tj. czy w żadnym wierzchołku na tej drodze nie znajduje się jakaś inna mrówka. Jeśli droga jest wolna, przesuwamy mrówkę do następnego wierzchołka.

Pierwszy problem z jakim musimy sobie poradzić, to sposób wyznaczania drogi między wierzchołkami zajmowanymi przez mrówki a wierzchołkiem B. W tym celu musimy przyjąć odpowiednią reprezentację drzewa w pamięci. Wygodnie jest przyjąć, że dla każdego wierzchołka drzewa pamiętamy listę jego sąsiadów oraz wskaźnik na ojca. Początkowo jako korzeń możemy obrać dowolny wierzchołek drzewa a następnie, po każdorazowym lądowaniu biedronki, modyfikować to drzewo tak, by korzeniem zostawał wierzchołek B. Zauważmy, że modyfikacja ta ograniczać się będzie do odwrócenia wskaźników pomiędzy sąsiednimi wierzchołkami, leżącymi na ścieżce od poprzedniego korzenia do wierzchołka B. Teraz dla każdej mrówki droga do wierzchołka B wyznaczona jest poprzez wskaźniki na ojca. Oczywistym jest, że złożoność czasowa tej fazy algorytmu jest ograniczona przez $O(n)$, gdzie n jest liczbą wierzchołków w drzewie.

Drugi problem polega na efektywnym sprawdzaniu czy droga między wierzchołkiem, w którym znajduje się mrówka a wierzchołkiem B jest wolna. Jedno z rozwiązań polega na tym, by dla każdego wierzchołka pamiętać znacznik, mówiący czy dany wierzchołek leży na drodze wolnej. Początkowego ustawienia znaczników można dokonać prostą procedurą przechodzenia drzewa, np. procedurą przechodzenia w głąb. Każdy ruch mrówki może powodować konieczność "zablokowania" znaczników niektórych wierzchołków. Jeśli mrówka przesunęła się do wierzchołka v, to należy zablokować znaczniki wierzchołków leżących w poddrzewie o korzeniu v. Oczywiście każdy znacznik będzie zablokowany co najwyżej jeden raz, a więc i ta faza algorytmu ma złożoność czasową ograniczoną przez $O(n)$.

W rozwiązaniu wzorcowym przyjęto inną metodę rozwiązania problemu wolnych dróg. Wygląda ona na nieco bardziej skomplikowaną, jednak pozwala na bardziej efektywną implementację algorytmu, zwłaszcza w sytuacji gdy liczba mrówek m jest istotnie mniejsza od n. Symulując ruchy mrówek zapamiętujemy w wierzchołkach ich "ślady" (numer mrówki i numer kroku). Każda mrówka przesuwana jest do przodu dopóty, dopóki nie napotka wierzchołka z zapamiętanym śladem innej mrówki, bądź też któraś z mrówek nie dojdzie do wierzchołka B. Informacje zgromadzone w czasie symulacji pozwalają na wyliczenie właściwych odległości, na jakie powinny się przemieścić poszczególne mrówki. Zauważmy, że odległości te można by w prosty sposób obliczyć przechodząc drzewo w głąb: dla każdej mrówki jest to najmniejszy numer kroku zapamiętany w śladzie w wierzchołku znajdującym się na drodze tej mrówki do B. Taka metoda nie byłaby jednak specjalnie oszczędna. Jak łatwo zauważyć wszystkie istotne dla tych obliczeń informacje znajdują się w śladach zapamiętanych w tych wierzchołkach, w których doszło do "spotkań" mrówek ze śladami innych mrówek. Można więc w trakcie symulacji zbudować graf (a w istocie drzewo) spotkań i następnie zastosować do tego grafu procedurę przechodzenia w głąb. Ponieważ graf ten ma $O(m)$ wierzchołków, taką też złożoność ma procedura przechodzenia tego grafu.

Może się wydawać, że nie unikniemy jednak przechodzenia całego drzewa (a więc kosztu zależnego od n) w drugiej fazie algorytmu, ponieważ musimy pozacierać ślady przed symulacją kolejnego lądowania biedronki. To fakt. Możemy jednak pamiętać ślady nie w wierzchołkach drzewa, lecz w tablicy indeksowanej numerami wierzchołków. Zacieranie śladów możemy wówczas wykonać, stosując efektywne procedury zerowania tablicy.

Testy

Do testowania rozwiązania tego zadania użyto zestawu 15 testów:

- `mro1.in` — mały test poprawnościowy;
- `mro1a.in` — test poprawnościowy, mrówki we wszystkich węzłach;
- `mro1b.in` — test poprawnościowy, jedna mrówka;
- `mro2.in` — test poprawnościowy;
- `mro2a.in` — test poprawnościowy, drzewo z jednym węzłem;
- `mro2b.in` — test poprawnościowy;
- `mro3.in` — n=100, m=10, drzewo binarne;
- `mro4.in` — n=500, m=10, drzewo losowe;
- `mro5.in` — n=1000, m=10, drzewo binarne;
- `mro6.in` — n=3000, m=3, mrówki na gwieździe;
- `mro7.in` — n=5000, m=1000, mrówki na gwieździe;
- `mro8.in` — n=5000, drabinka, dwie biedronki;
- `mro9.in` — n=5000, dwie mrówki na liście z losowymi wypustkami, losowa biedronka;
- `mro10.in` — n=5000, dwie mrówki na liście;
- `mro11.in` — n=5000, m=3, mrówki na gwieździe.

Zbigniew J. Czech
Treść zadania, Opracowanie

Marcin Stefaniak
Program

Podróż

Rozważmy graf opisujący sieć komunikacji publicznej, np. sieć autobusową, tramwajową, metra, itp. Wierzchołki grafu, o numerach 1, 2, ..., n, reprezentują przystanki, a krawędzie $\langle p_i,\ p_j\rangle$, *(dla* $p_i \neq p_j$*) oznaczają możliwe, bezpośrednie przejazdy od przystanku* p_i *do* p_j $(1 \leq p_i, p_j \leq n)$.

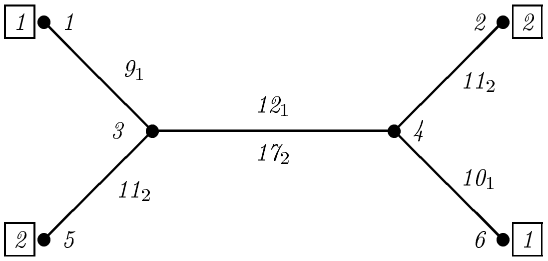
W sieci kursują pojazdy linii komunikacyjnych. Linie komunikacyjne oznaczone są numerami 1, 2, ..., k. Linia komunikacyjna l zdefiniowana jest przez ciąg przystanków $p_{l,1}, p_{l,2}, \ldots, p_{l,s_l}$, *przez które przejeżdżają pojazdy linii, oraz czasy przejazdów* $r_{l,1}, r_{l,2}, \ldots, r_{l,s_l-1}$ *pomiędzy przystankami —* $r_{l,1}$ *jest czasem przejazdu od przystanku* $p_{l,1}$ *do* $p_{l,2}$, *lub z powrotem, tj. od przystanku* $p_{l,2}$ *do* $p_{l,1}$; $r_{l,2}$ *jest czasem przejazdu od przystanku* $p_{l,2}$ *do* $p_{l,3}$, *itd. Wszystkie przystanki linii są różne, tj. dla* $i \neq j$ *zachodzi* $p_{l,i} \neq p_{l,j}$.

Na danej linii l pojazdy kursują z określoną częstotliwością c_l, *gdzie* c_l *jest liczbą ze zbioru* $\{6, 10, 12, 15, 20, 30, 60\}$. *Pojazdy linii wyruszają z przystanku* $p_{l,1}$ *o każdej "okrągłej" godzinie doby, g:0,* $(0 \leq g \leq 23)$, *a następnie zgodnie z częstotliwością linii, a więc o godzinach g:*c_l, *g:*$2c_l$, *... itd. (g:*c_l *oznacza `*c_l *minut po godzinie g"). Ruch pojazdów linii odbywa się jednocześnie w obu kierunkach: z przystanku* $p_{l,1}$ *do* p_{l,s_l}, *a także z przystanku* p_{l,s_l} *do* $p_{l,1}$. *Godziny odjazdów pojazdów linii z przystanku* p_{l,s_l} *są takie same, jak z przystanku* $p_{l,1}$.

W tak zdefiniowanej sieci komunikacji publicznej chcemy odbyć podróż z przystanku początkowego x, do przystanku końcowego y. Zakładamy, że podróż jest możliwa i nie będzie trwała dłużej niż 24 godziny. W trakcie podróży możemy się przesiadać dowolną liczbę razy z jednej linii komunikacyjnej na inną. Przyjmujemy, że czas dokonania przesiadki jest równy 0, jednakowoż, zmieniając linię musimy liczyć się z koniecznością czekania na pojazd linii, do którego chcemy się przesiąść. Naszym celem jest odbycie podróży z przystanku początkowego x, do przystanku końcowego y, w jak najkrótszym czasie.

Przykład

Na poniższym rysunku przedstawiono schemat sieci komunikacyjnej o 6 przystankach i dwóch liniach: $\boxed{1}$ *i* $\boxed{2}$. *Pojazdy linii* $\boxed{1}$ *kursują pomiędzy przystankami 1, 3, 4 i 6, a linii* $\boxed{2}$ *pomiędzy przystankami 2, 4, 3 i 5. Częstotliwości kursowania pojazdów linii wynoszą, odpowiednio,* $c_1 = 15$ *oraz* $c_2 = 20$. *Czasy przejazdów pomiędzy przystankami umieszczono obok krawędzi sieci opatrując je indeksami 1 i 2 dla poszczególnych linii.*



Załóżmy, że o godzinie 23:30 znajdujemy się na przystanku początkowym 5 i chcemy odbyć podróż do przystanku końcowego 6. Wówczas musimy odczekać 10 minut i o godzinie 23:40 wyjeżdżamy linią $\boxed{2}$. *Nasza podróż może mieć dwa warianty. W pierwszym wariancie, po dojechaniu do przystanku 3 o godzinie 23:51 i odczekaniu 3 minut, przesiadamy się o godzinie 23:54 na linię* $\boxed{1}$ *i dojeżdżamy do przystanku 6 o godzinie 0:16 (następnego dnia). W drugim wariancie, dojeżdżamy linią* $\boxed{2}$ *do przystanku 4 o godzinie 0:8, czekamy 13 minut i o godzinie 0:21 wsiadamy do pojazdu linii* $\boxed{1}$, *osiągając przystanek końcowy 6 o godzinie 0:31. Tak więc najwcześniej możemy dotrzeć do przystanku 6 o godzinie 0:16.*

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `pod.in` *opis sieci oraz linii komunikacyjnych, numer przystanku początkowego x, numer przystanku końcowego y, godzinę początkową* g_x *oraz minuty początkowe* m_x,
- *wyznaczy minimalny czas podróży od przystanku początkowego x, do przystanku końcowego y,*
- *zapisze do pliku tekstowego* `pod.out` *najwcześniejszą możliwą godzinę z minutami dotarcia do przystanku y — odpowiednio,* g_y *oraz* m_y.

Wejście

W pierwszym wierszu pliku tekstowego `pod.in` *zapisanych jest sześć liczb całkowitych, pooddzielanych pojedynczymi odstępami:*

- *liczba przystanków n ($1 \leq n \leq 1000$),*
- *liczba linii komunikacyjnych k ($1 \leq k \leq 2000$),*
- *numer przystanku początkowego x ($1 \leq x \leq n$),*
- *numer przystanku końcowego y ($1 \leq y \leq n$),*
- *godzina rozpoczęcia podróży g_x ($0 \leq g_x \leq 23$),*
- *minuta rozpoczęcia podróży m_x ($0 \leq m_x \leq 59$).*

Przystanki są ponumerowane od 1 do n, a linie komunikacyjne od 1 do k.

W kolejnych $3k$ wierszach opisane są kolejne linie komunikacyjne — opis każdej linii zajmuje trzy kolejne wiersze.

- *W pierwszym wierszu opisującym linię komunikacyjną l są zapisane dwie liczby całkowite, oddzielone pojedynczym odstępem: s_l, równa liczbie przystanków ($2 \leq s_l \leq n$), oraz c_l, równa częstotliwości kursowania pojazdów ($c_l \in \{6, 10, 12, 15, 20, 30, 60\}$).*
- *W drugim wierszu opisującym linię komunikacyjną l jest zapisanych s_l różnych liczb całkowitych, pooddzielanych pojedynczymi odstępami: $p_{l,1}, p_{l,2}, \ldots, p_{l,s_l}$ — numery kolejnych przystanków na tej linii ($1 \leq p_{l,i} \leq n$, dla $1 \leq i \leq s_l$).*
- *W trzecim wierszu opisującym linię komunikacyjną l jest zapisanych $s_l - 1$ liczb całkowitych, pooddzielanych pojedynczymi odstępami: $r_{l,1}, r_{l,2}, \ldots, r_{l,s_l-1}$ — czasy przejazdów (w minutach) pomiędzy kolejnymi przystankami na tej linii ($1 \leq r_{l,i} \leq 240$, dla $1 \leq i \leq s_l - 1$).*

Suma liczb przystanków, na wszystkich liniach razem, nie przekracza 4000 (tzn. $s_1 + s_2 + \ldots + s_k \leq 4000$).

Wyjście

Twój program powinien zapisać w pierwszym i jedynym wierszu pliku tekstowego `pod.out` *dwie liczby całkowite oddzielone pojedynczym odstępem: godzinę dotarcia do przystanku końcowego g_y ($0 \leq g_y \leq 23$) oraz minutę dotarcia do przystanku końcowego m_y ($0 \leq m_y \leq 59$).*

Przykład

Dla pliku wejściowego `pod.in`*:*

```
6 2 5 6 23 30
4 15
1 3 4 6
9 12 10
4 20
5 3 4 2
11 17 11
```

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

```
0 16
```

Rozwiązanie

W pierwszej kolejności zaprojektujemy strukturę danych, w której będziemy przechowywać rozkład jazdy pojazdów. Z zadania wynika, że sieć komunikacyjna działa całą dobę. Pojazdy linii komunikacyjnych wyruszają z przystanków krańcowych z określoną częstotliwością, c, gdzie c jest liczbą ze zbioru {6, 10, 12, 15, 20, 30, 60}. Ponieważ każda z tych częstotliwości dzieli liczbę 60, to pojazdy danej linii będą odjeżdżać z przystanków zawsze w tych samych minutach dowolnej godziny. Przykładowo, niech dla pewnej linii $c = 15$. Wówczas pojazdy tej linii wyruszają z przystanku p_1 o godzinie g minut 0, 15, 30, 45, gdzie $g \in \{0, 1, \ldots, 23\}$. Podobnie, czasy odjazdu z przystanku p_2 są równe godzina g minut $(0 + r_1) \bmod 60$, $(15 + r_1) \bmod 60$, $(30 + r_1) \bmod 60$, $(45 + r_1) \bmod 60$, gdzie r_1 jest czasem przejazdu od przystanku p_1 do p_2. Tak więc, zapamiętanie rozkładu jazdy na dowolnym przystanku wymaga przechowania jednej z chwil odjazdów wyrażonej jako liczba minut po (dowolnej) godzinie g (pozostałe chwile odjazdów można łatwo obliczyć znając częstotliwość kursowania linii). Dane określające rozkład jazdy dla dowolnego przystanku pi można przechować w rekordach o postaci:

```
type dane_odj = { dane dotyczące odjazdów z przystanku pi }
  record
     pi:  integer; { pi – przystanek aktualny }
     pj:  integer; { pj – przystanek następny (sąsiedni) }
     r:   byte; { "czysty" czas przejazdu od przystanku }
                { pi do pj }
     go:  byte; { jedna z chwil odjazdów pamiętana jako liczba }
                { minut po (dowolnej) godzinie g }
     c:   byte; { częstotliwość kursowania pojazdów linii, }
                { której dotyczy niniejszy rekord }
  end;
```

zaś cały rozkład jazdy w tablicy:

rozkl_jazdy: **array**[1 .. q] **of** *dane_odj*;

gdzie q jest całkowitą liczbą rekordów. Zauważmy, że dla dowolnego przystanku pi, z wyjątkiem przystanków krańcowych, liczba rekordów w rozkładzie jazdy jest równa podwojonej liczbie linii komunikacyjnych (bo linie prowadzą w obie strony), których pojazdy dojeżdżają, a następnie odjeżdżają z tego przystanku. Dla przystanków krańcowych liczba rekordów jest równa liczbie linii komunikacyjnych wychodzących z tych przystanków. Dla przykładowej sieci z treści zadania, $q = 12$. Liczby rekordów w rozkładzie jazdy dla wierzchołków od 1 do 6 są równe, odpowiednio, 1, 1, 4, 4, 1 oraz 1.

Zadanie można rozwiązać adaptując do tego celu algorytm Dijkstry znajdowania najkrótszych dróg w grafie (zobacz [14]). W naszym przypadku czas przejazdu między przystankami nie jest stały i zależy od chwili h, w której przybywamy na przystanek. Znając chwilę h, przystanki pi, pj oraz linię komunikacyjną, którą podróżujemy (tj. odpowiedni rekord w tablicy *rozkl_jazdy*), czas przejazdu pomiędzy przystankami pi oraz pj można wyznaczyć za pomocą funkcji:

```
function czas_przejazdu( pi, pj: integer; h: integer): integer;
begin
   czas_przejazdu := ((60 + pi − h) mod c) + r;
end
```

Wartość $(60 + go - h) \bmod c$ jest czasem oczekiwania na przystanku pi, zaś r jest "czystym" czasem przejazdu wybraną linią od przystanku pi do pj.

Stosując algorytm Dijkstry do rozwiązania naszego zadania będziemy wyróżniać wśród wszystkich przystanków sieci te z nich, dla których najkrótszy czas dojazdu z przystanku początkowego x został już ustalony. Do tego celu wprowadzimy tablicę:

ustal: **array**[1 .. n] **of** *boolean*;

w której wartość $ustal[i] = true$, $i = 1, 2, \ldots, n$, oznacza, że znamy już najkrótszy czas dojazdu od przystanku x do i. Początkowo inicjujemy wszystkie elementy na *false*, za wyjątkiem elementu $ustal[x]$, któremu przypisujemy wartość *true*. Będziemy także korzystać z tablicy:

D: **array**[1 .. n] **of** *integer*;

w której zapamiętamy najkrótsze czasy dojazdu wyrażone w minutach od przystanku x do przystanku i, $i = 1, 2, \ldots, n$, $i \neq x$. Elementy tablicy inicjujemy wartością ∞, za wyjątkiem przystanku x, dla którego przyjmujemy $D[x] = 0$. Oto rozwiązanie naszego zadania:

```
D[x] := 0; { przyjęcie czasu dojazdu do przystanku x jako równego 0 }
for k := 1 to n do
  { W każdej iteracji ustalany jest najkrótszy czas przejazdu }
  { z przystanku x do jednego z przystanków sieci. }
  Znajdź indeks w taki, że D[w] jest minimalne wśród wszystkich
    D[i], i = 1, 2, ..., n, dla których ustal[i] = false;
  if D[w] = ∞ then
```

```
8:      Nie istnieje ścieżka do przystanku końcowego; break;
9:    end if;
10:   ustal[w] := true;
11:   if w = y then
12:      break; { Zadanie rozwiązano; wynik: D[w] = D[y] }
13:   end if;
14:   for wszystkich połączeń (w, i) między sąsiednimi przystankami
15:         w oraz i, dla których ustal[i] = false do
16:     a := czas_przejazdu(w, i, (m_x + D[w]) mod 60);
17:     if (a ≠ ∞) and (D[w] + a < D[i]) then
18:        D[i] := D[w] + a;
19:     end if;
20:   end for;
21: end for;
22: t := g_x * 60 + m_x + D[y];
23: writeln((t div 60) mod 24, ' ', t mod 60);
```

W każdej iteracji instrukcji **for** w wierszach 2–21 zostaje dołączony jeden przystanek, dla którego zostaje ustalony minimalny czas dojazdu z przystanku początkowego x. Przystanek ten, o indeksie w, ma najkrótszy czas dojazdu $D[w]$ przez wszystkie przystanki, dla których najkrótsze czasy zostały już ustalone. W wierszach 14–20 sprawdzamy czy po dołączeniu przystanku w, czasy dojazdu do pozostałych przystanków o nieustalonym czasie dojazdu przez przystanek w nie uległy skróceniu. Jeżeli tak, to czasy te modyfikujemy.

Złożoność czasowa poprawiania tablicy D (wiersze 14–20) jest $O(q)$, gdzie q jest liczbą rekordów w tablicy *rozkl_jazdy*. Złożoność czasowa n-krotnego wyszukiwania minimalnej wartości $D[w]$ jest $O(n^2)$

W efektywnej implementacji algorytmu warto jest pogrupować rekordy tablicy *rozkl_jazdy* według numerów przystanków odjazdu, ponieważ uaktualniając tablicę D (wiersze 14–20) przeglądamy połączenia wychodzące z przystanku w. Numery przystanków są liczbami całkowitymi z zakresu $1..n$, więc grupowanie takie można zrealizować za pomocą sortowania pozycyjnego z użyciem liczników częstości w czasie $O(n+q)$ (zobacz [11]). Podsumowując, złożoność czasowa przedstawionego rozwiązania jest $O(n^2+q)$.

Testy

Do generowania losowych testów użyto generatora **gen_pod.pas**, który losuje zestaw danych o zadanych parametrach (bez gwarancji, że rozwiązanie istnieje). Ręcznie zadano stacje końcowe tak, żeby rozwiązanie było znajdowane pod koniec przeszukiwania.

- `pod0.in` — przykład z treści zadania;
- `pod1.in` — $n = 10$, stacja początkowa jest też stacją końcową;
- `pod2.in` — $n = 10$, $k = 5$, $q = 20$, losowy (k jest liczbą linii komunikacyjnych);
- `pod3.in` — $n = 20$, $k = 3$, $q = 30$, losowy;
- `pod4.in` — $n = 100$, $k = 1$, $q = 90$, podróż trwa dokładnie 24h;
- `pod5.in` — $n = 6$, $k = 30$, $q = 90$, losowy;
- `pod6.in` — $n = 200$, $k = 300$, $q = 1000$, losowy;
- `pod7.in` — $n = 300$, $k = 2000$, $q = 4000$, losowy;
- `pod8.in` — $n = 500$, $k = 100$, $q = 4000$, losowy;
- `pod9.in` — $n = 1000$, $k = 200$, $q = 2000$, losowy;
- `pod10.in` — $n = 1000$, $k = 500$, $q = 4000$, losowy.

Zawody III stopnia

Zawody III stopnia — opracowania zadań

Marcin Sawicki
Treść zadania, Opracowanie

Andrzej Gąsienica–Samek
Program

Wędrowni treserzy pcheł

W Bajtocji można spotkać wędrownych treserów pcheł. Tresowane pchły uczone są tańca, polegającego na wykonywaniu precyzyjnych skoków w rytm muzyki. Dokładnie wygląda to tak: treser układa na stole w rządku ponumerowane żetony, przy czym żetony nie muszą być ułożone po kolei. Na każdym żetonie, oprócz jego numeru, jest również napisany numer żetonu, na który powinna z niego skoczyć pchła. Następnie treser ustawia po jednej pchle na każdym z żetonów i włącza muzykę. Na początku każdego taktu, każda z pcheł wykonuje skok wprost na żeton, którego numer jest napisany na żetonie, na którym w danej chwili stoi. W trakcie tańca może się zdarzyć, że kilka pcheł znajdzie się na tym samym żetonie i razem wykonują dalsze skoki.

Załóżmy, że mamy n tresowanych pcheł i n żetonów. Jeśli podamy, jakie liczby znajdują się kolejno na żetonach numer $1, 2, \ldots, n$, to jednoznacznie opiszemy układ choreograficzny, jaki zaprezentują pchły. Jednak może się okazać, że dwa różne zestawy żetonów dają ten sam układ, jeśli tylko odpowiednio je ułożymy.

Przykład

Powiedzmy, że mamy trzy żetony. Jeśli z żetonu nr 1 należy skoczyć na żeton nr 2, z żetonu nr 2 na żeton nr 3, a z żetonu nr 3 na żeton nr 1 (w skrócie: $1 \to 2,\ 2 \to 3,\ 3 \to 1$), to pchły będą tańczyć „w kółko” i żadne dwie nigdy się nie spotkają na tym samym żetonie. Jest to inny układ tańca, niż np. $1 \to 2,\ 2 \to 3,\ 3 \to 3$, gdzie już po dwóch taktach wszystkie trzy pchły spotkają się na żetonie nr 3 i dalej będą razem skakać w miejscu.

Natomiast układy $1 \to 2,\ 2 \to 3,\ 3 \to 2,\ 4 \to 4$ oraz $1 \to 1,\ 2 \to 3,\ 3 \to 2,\ 4 \to 3$ są takie same — wystarczy ułożyć żetony na stole w rzędzie, w pierwszym przypadku w kolejności od lewej do prawej, a w drugim od prawej do lewej, a pchły odtańczą ten sam taniec.

Zadanie

Gawiedź bardzo się niecierpliwi, gdy pchły tańczą według tego samego układu więcej niż raz. Dlatego potrzebny jest program, który:

- *wczyta z pliku tekstowego* `pch.in` *liczbę przypadków testowych,*
- *dla każdego z przypadków wczyta z pliku* `pch.in` *opis dwóch zestawów żetonów i rozstrzygnie, czy żetony z tych zestawów można ułożyć na stole tak, by pchły wykonały taki sam taniec,*
- *wypisze odpowiedzi do pliku tekstowego* `pch.out`*.*

Wejście

W pierwszym wierszu pliku tekstowego `pch.in` *znajduje się jedna liczba całkowita d równa liczbie przypadków testowych, $1 \le d \le 100$.*

Kolejne $3d$ wierszy pliku `pch.in` *opisują kolejne przypadki testowe — każdy przypadek zajmuje trzy kolejne wiersze pliku. Pierwszy z nich zawiera jedną liczbę całkowitą $1 \le n \le 2\,000$, równą liczbie żetonów. Każdy z dwóch następnych wierszy zawiera opis zestawu n żetonów w postaci ciągu n liczb całkowitych z przedziału $1 \ldots n$, pooddzielanych pojedynczymi odstępami; i-ty wyraz ciągu oznacza numer żetonu, na który mają skakać pchły z żetonu nr i.*

Wyjście

Dla każdego z przypadków testowych z pliku `pch.in` *należy wypisać do pliku tekstowego* `pch.out` *dokładnie jeden wiersz, zawierający dokładnie jedną literę:*

- *T — jeśli oba zestawy żetonów można ułożyć tak, aby pchły wykonały taki sam taniec,*
- *N — w przeciwnym wypadku.*

Przykład

Dla pliku wejściowego `pch.in`*:*

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

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

```
N
T
```

Rozwiązanie

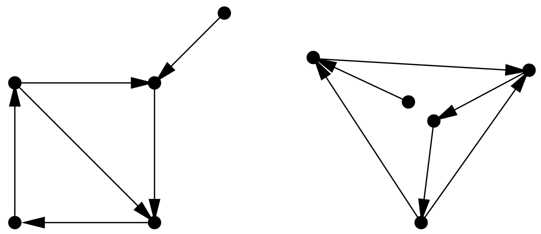
Wypada nam zacząć od przypomnienia, co to jest graf.

Graf są to kropki, z których niektóre połączone są strzałkami. Albo inaczej: graf G jest to para uporządkowana $G = \langle V, E \rangle$, gdzie V jest dowolnym zbiorem, który nazwiemy *zbiorem wierzchołków* grafu, zaś $E \subseteq V \times V$ jest *zbiorem krawędzi*: jeżeli $v, v' \in V$ są dwoma wierzchołkami grafu, to są one połączone strzałką od v do v' wtedy i tylko wtedy, gdy para $(v, v') \in E$.

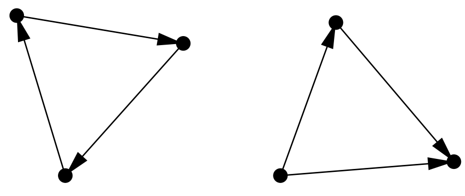
Zwróćmy uwagę, że w przyjętej przez nas definicji krawędź od v do v' to nie to samo, co krawędź od v' do v. Ponadto, dopuszczamy *pętle*, czyli krawędzie prowadzące od wierzchołka do niego samego.

Grafy są przydatne do opisu wielu praktycznych zagadnień, np. sieci komunikacyjnych, obwodów elektrycznych, zależności pomiędzy etapami przedsięwzięcia, a także, jak się zaraz okaże, pchlej choreografii.

Kiedy rysujemy graf, zazwyczaj nie ma dla nas znaczenia, jak na naszym rysunku rozmieszczone są wierzchołki, ważne jest tylko, które z którymi są połączone. Poza tym często wszystkie wierzchołki grafu zaznaczamy tak samo (np. kropką). Stąd pojawia się problem *izomorfizmu grafów*, polegający na rozstrzygnięciu, czy dane dwa grafy $G_1 = \langle V_1, E_1 \rangle$, $G_2 = \langle V_2, E_2 \rangle$ da się narysować tak samo? Innymi słowy, chodzi o sparowanie wierzchołków grafu G_1 z wierzchołkami grafu G_2 tak, by krawędzie w grafie G_1 dokładnie odpowiadały krawędziom pomiędzy wierzchołkami do pary w grafie G_2. Jeszcze inaczej, chodzi o znalezienie funkcji wzajemnie jednoznacznej $f : V_1 \longrightarrow V_2$ takiej, by $(v, v') \in E_1$ zachodziło wtedy i tylko wtedy, gdy $(f(v), f(v')) \in E_2$.



Rysunek 1: Para grafów izomorficznych



Rysunek 2: Para grafów nieizomorficznych

Wróćmy teraz do naszego zadania. Mamy opis pewnego tańca pcheł, czyli n żetonów ponumerowanych liczbami $1, 2, \ldots, n$, a na każdym z nich dodatkową liczbę mówiącą, na który żeton pchła ma skoczyć. Rozważmy graf, którego zbiorem wierzchołków będzie właśnie $V = \{1, 2, \ldots, n\}$, czyli każdemu żetonowi odpowiada dokładnie jeden wierzchołek grafu. Krawędzie poprowadzimy oczywiście od każdego żetonu do tego, którego numer jest na nim zapisany. Krawędzie będą zatem odpowiadać pchlim skokom.

Teraz naprawdę nietrudno zauważyć, że nasze zadanie to po prostu pytanie o izomorfizm tak utworzonych grafów! Zaglądamy zatem do indeksu dowolnego podręcznika algorytmiki, znajdujemy hasło „grafów izomorfizm” i ... co za rozczarowanie! Okazuje się, że nie jest znany żaden wielomianowy algorytm na rozstrzyganie o izomorfizmie grafów. Nikt też nie zdołał udowodnić, że taki algorytm nie może istnieć, co więcej nie wiadomo nawet, czy jest to tzw. problem NP–zupełny.[1]

Czyżby było aż tak źle? Pomyślmy... niektórzy spośród czytelników słyszeli może o problemie izomorfizmu drzew, dla którego istnieje rozwiązanie wielomianowe. Drzewa to grafy bez cykli. Być może grafy, które opisują pchle harce, też mają jakąś szczególną postać, która pozwala łatwo rozwiązać nasz problem? Okazuje się, że tak.

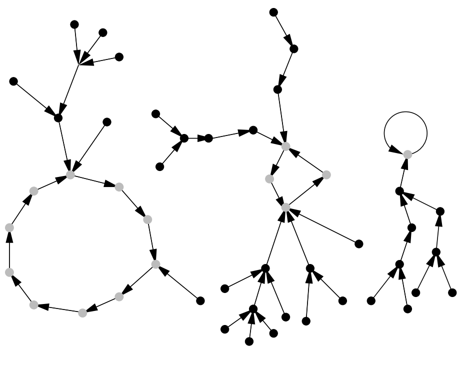
Przede wszystkim, w naszych grafach z każdego wierzchołka wychodzi *dokładnie* jedna krawędź. Oczywiście jest całe mnóstwo grafów, które nie mają tej właściwości, np. trzy spośród czterech narysowanych powyżej.

Zdefiniujmy *cykl* (skierowany) jako ciąg $(v_1, v_2, \ldots, v_n)$ parami różnych wierzchołków grafu $G = \langle V, E\rangle$ taki, że dla każdego $i = 1, 2, \ldots, n-1$ mamy w grafie krawędź $(v_i, v_{i+1}) \in E$ oraz dodatkowo $(v_n, v_1) \in E$. Zauważmy, że jeżeli ciąg $(v_1, v_2, \ldots, v_n)$ jest cyklem, to dla każdego $i = 2, 3, \ldots, n$ cyklem jest także ciąg $(v_i, v_{i+1}, \ldots, v_n, v_1, v_2, \ldots, v_{i-1})$, otrzymany przez cykliczne przesunięcie ciągu wyjściowego w lewo o $i-1$ pozycji. Umówmy się, że są to równoważne reprezentacje jednego i tego samego cyklu.

A co możemy powiedzieć o cyklach w naszych pchlich grafach? Wykażemy, że są one rozłączne, to znaczy że każdy wierzchołek należy do co najwyżej jednego cyklu. Dlaczego? Przypuśćmy, że wierzchołek $v = v_i = v'_j$ należy do dwóch cykli: $v_1, v_2, \ldots, v_n$ oraz $v'_1, v'_2, \ldots, v'_m$. Umówiliśmy się, że ciągi reprezentujące cykle możemy przesuwać cyklicznie, zatem nic nie stracimy zakładając, że $v = v_1 = v'_1$ (czyli że $i = j = 1$). Zamieniając w razie potrzeby cykle rolami możemy również przyjąć, że $n \leq m$. Wiemy, że z wierzchołka v (podobnie jak z każdego innego) wychodzi dokładnie jedna krawędź, wiemy też, że mamy w naszym grafie krawędź (v_1, v_2) oraz (v'_1, v'_2). Skoro zatem $v = v_1 = v'_1$, to także $v_2 = v'_2$. Rozumując tak dalej wnioskujemy, że $v_3 = v'_3, \ldots, v_n = v'_n$. Jeżeli byłoby $m > n$, to mielibyśmy następnie $v'_{n+1} = v_1 = v'_1$, lecz jest to niemożliwe, bo umówiliśmy się, że żaden wierzchołek w cyklu się nie powtarza. Czyli $n = m$ i nasze cykle okazały się być jednym i tym samym cyklem (a nawet tą samą jego reprezentacją)!

Każdy wierzchołek w naszym grafie albo należy do jakiegoś jednego cyklu, albo nie należy do żadnego. Jak wygląda ten drugi przypadek? Niech wierzchołek v_1 nie należy do żadnego cyklu. Prowadzi z niego dokładnie jedna krawędź, umówmy się, że do wierzchołka v_2, z którego z kolei prowadzi krawędź do v_3 itd. Otrzymujemy ciąg wierzchołków $v_1, v_2, v_3, v_4, \ldots$. Wiemy, że v_1 występuje tylko na początku ciągu (skoro nie leży na żadnym cyklu). Jednak w naszym grafie jest tylko skończenie wiele wierzchołków, więc od któregoś miejsca muszą się one zacząć powtarzać. Niech n będzie najmniejszą taką liczbą, że istnieje $i < n$ takie, że $v_i = v_n$. Oczywiście $i > 1$. Chwila uwagi wystarcza by stwierdzić, że wierzchołki $(v_i, v_{i+1}, v_{i+2}, \ldots, v_{n-1})$ tworzą cykl. Być może $n = i + 1$, mamy wówczas cykl jednoelementowy, czyli pętlę przy wierzchołku v_i.

Podsumujmy: startując od dowolnego wierzchołka i podążając po strzałkach, albo jesteśmy od razu na cyklu, albo po jakimś czasie wpadamy na cykl. Sytuacja przypomina morza i rzeki: jeżeli jesteśmy w wodzie, to albo jesteśmy w morzu, i wtedy wiadomo w jakim, albo jesteśmy w rzece i w końcu spłyniemy do morza. Do tego wiadomo, w zlewisku jakiego morza leży dana rzeka. Co więcej, analogia obejmuje i to, że rzeka ma dopływy. Istotnie, ścieżki prowadzące od różnych wierzchołków do tego samego cyklu mogą się łączyć jeszcze przed osiągnięciem cyklu, tworząc podczepione do cykli drzewa. Przykładowo, wygląda to tak, jak na rysunku 3. Wierzchołki, które leżą na cyklach, zaznaczono na szaro.



Rysunek 3: Przykład pchlego grafu

[1]Problemy NP–zupełne to m.in. problem cyklu Hamiltona, problem komiwojażera, problem 3-kolorowania grafu, sumy podzbioru, maksymalnej kliki, spełnialności formuł boolowskich i kilkaset innych. Wiadomo o tych problemach tyle, że albo wszystkie mają rozwiązania wielomianowe, albo żaden z nich takiego rozwiązania nie ma. Ponieważ przez wiele lat nie wymyślono dla żadnego z nich algorytmu wielomianowego (za to wymyślono wiele naprawdę pomysłowych algorytmów dla innych problemów), więc większość informatyków przypuszcza, że takie algorytmy nie istnieją. Wciąż jednak nikt nie potrafi tego udowodnić. Więcej o problemach NP i NP–zupełnych można poczytać w znakomitej książce [14].

Wcześniej wspominaliśmy o problemie izomorfizmu drzew, dla którego znamy rozwiązanie w czasie wielomianowym. Rozwiązanie to polega na wyznaczeniu dla drzewa D jego sygnatury $\sigma(D)$, to znaczy takiego drzewa, że jeżeli D i D' są izomorficzne, to $\sigma(D) = \sigma(D')$. Następnie po prostu porównujemy otrzymane sygnatury. Jak można zdefiniować taką sygnaturę?

Rozważmy najpierw prosty przykład porównywania ciągów liczb. Chcielibyśmy wiedzieć, czy dwa ciągi reprezentują ten sam zbiór wartości. Jedno z możliwych rozwiązań polega na posortowaniu ich i stwierdzeniu, czy są identyczne. Sortowanie to w tym wypadku właśnie wyznaczenie sygnatury.

Izomorfizm drzew (z wyróżnionym korzeniem) polega wyłącznie na permutowaniu synów każdego węzła. Sygnaturę wyznaczymy zatem, biorąc jakąś wyróżnioną permutację. Jaką? Np. najmniejszą, o ile nauczymy się porównywać drzewa. Przyjmijmy taką definicję:

- jeżeli korzeń drzewa D ma mniej synów, niż korzeń drzewa D', to $D < D'$
- jeżeli korzeń r drzewa D oraz korzeń r' drzewa D' mają po n synów, oraz $D_1, D_2, \ldots, D_n$ jest ciągiem poddrzew drzewa D, zakorzenionych w kolejnych synach r, analogicznie $D'_1, \ldots, D'_n$, oraz $D_1 = D'_1, D_2 = D'_2, \ldots, D_{k-1} = D'_{k-1}$ i przy tym $D_k < D'_k$ dla pewnego $k \leq n$, to także $D < D'$ (tzn. aby porównać drzewa D i D', potrzebujemy porównać leksykograficznie ciągi synów ich korzeni).

Chwili namysłu wymaga poprawność tej definicji. Niech każdy Czytelnik, który widzi ją po raz pierwszy, sam przeanalizuje, dlaczego z każdych dwóch nieidentycznych drzew jedno okazuje się być mniejsze od drugiego. Warto też sprawdzić, że jeżeli $D_1 < D_2$ oraz $D_2 < D_3$, to $D_1 < D_3$.

Za sygnaturę drzewa obieramy teraz po prostu najmniejsze możliwe w sensie określonego wyżej porządku drzewo, izomorficzne z danym. Wyznaczamy ją w ten sposób, że idąc od liści do korzenia, w każdym wierzchołku sortujemy synów w kolejności od tego, pod którym jest zaczepione najmniejsze poddrzewo (w zdefiniowanym wyżej sensie) do tego, pod którym zaczepione jest największe.

Sądzę, że nie jest już teraz trudno zdefiniować sygnatury dla zadania o pchłach. Nasze grafy rozpadają się na rozłączne cykle, z których do każdego podczepione są drzewa. Pierwszym krokiem jest zastąpienie każdego drzewa jego sygnaturą. Następnie można znaleźć dla każdego cyklu taką jego reprezentację (rotację) $(v_1, \ldots, v_n)$, by po zastąpieniu każdego wierzchołka w tym ciągu sygnaturą podczepionego do niego drzewa (być może zbudowanego tylko z korzenia), otrzymać leksykograficznie najmniejszy możliwy ciąg drzew. Z kolei można porównywać tak otrzymane sygnatury różnych cykli z podoczepianymi drzewami. W rozwiązaniu wzorcowym są one porównywane leksykograficznie. Inną możliwością byłoby najpierw porównywanie długości cykli, a następnie porównywanie leksykograficzne podczepionych drzew tylko dla cykli równej długości (analogicznie do definicji porządku na drzewach, podanej powyżej). Sygnaturą dla całego grafu jest uporządkowany ciąg wszystkich występujących w nim cykli. Sprawdzenie, że identyczne sygnatury otrzymujemy wtedy i tylko wtedy, gdy grafy są izomorficzne, nie powinno być problemem, jeśli ktoś potrafi udowodnić to dla drzew.

Jeżeli komuś powyższy opis nie wystarczył, to po szczegóły odsyłam do kodu programu wzorcowego.

Analiza złożoności rozwiązania

Mamy już jasność, że nasze zadanie da się rozwiązać w czasie wielomianowym. Spróbujmy dokładniej oszacować złożoność naszkicowanego algorytmu.

Wydzielenie w grafie cykli oraz wierzchołków nie leżących na cyklach i zbudowanie struktury ojców/synów w drzewach jest proste i może być wykonane w czasie liniowym.

Następnie dla każdego drzewa musimy obliczyć sygnaturę. Oznacza to konieczność posortowania synów każdego węzła. Sortowanie wymaga wykonania pewnej liczby porównań, a każde porównanie w pesymistycznym przypadku kosztuje tyle, co minimum z rozmiaru porównywanych poddrzew. W rozwiązaniu firmowym dla uproszczenia zastosowano sortowanie przez wstawianie. W sortowaniu tym każda para elementów jest porównywana co najwyżej raz. Pozwala to oszacować koszt wyznaczenia sygnatury całego drzewa poprzez następującą obserwację: każdy węzeł drzewa co najwyżej raz bierze udział w porównaniu z poddrzewem, zawieszonym w każdym innym węźle, leżącym na takiej jak on lub mniejszej głębokości (odległości od korzenia). Jeżeli drzewo ma n wierzchołków, pozwala to oszacować koszt obliczenia sygnatury tego drzewa przez $O(n^2)$ (gdyż w sumie we wszystkich operacjach porównania poddrzew każdy z wierzchołków będzie brał udział co najwyżej n razy). Oczywiście, jeżeli mamy las drzew, które w sumie mają n wierzchołków, to tym bardziej łączny koszt wyznaczenia sygnatury dla każdego z nich jest $O(n^2)$.

Z kolei trzeba wyznaczyć sygnatury dla poszczególnych cykli. Dla cyklu długości k wymaga to k porównań cykli (a dokładniej, różnych rotacji tego samego cyklu), by znaleźć minimalną rotację. Każde porównanie angażuje każdy z wierzchołków grafu co najwyżej raz, więc w sumie znalezienie minimalnej rotacji dla każdego z cykli wymaga $O(n^2)$ operacji.

Na koniec podobna argumentacja co przy wyznaczaniu sygnatur pozwala stwierdzić, że posortowanie przez wstawianie wszystkich cykli w grafie o n wierzchołkach również wymaga $O(n^2)$ operacji. Podsumowując, całe zadanie rozwiążemy w czasie $O(n^2)$.

Dla szczególnie dociekliwego Czytelnika mamy zadanie: czy można ten problem rozwiązać w czasie mniejszym od kwadratowego? Dla których etapów obliczenia (wyznaczanie sygnatur drzew, cykli, całego grafu) potrafiłbyś znaleźć szybszy algorytm?

Testy

Ponieważ zadanie wymaga podania jedynie odpowiedzi „tak" lub „nie", więc każdy z 10 właściwych testów obejmował od 20 do 100 przypadków, aby wyeliminować programy, próbujące zgadywać odpowiedź na chybił–trafił.

Testy zostały wygenerowane losowo, przy użyciu następujących parametrów:

- d — liczba przypadków
- n_{min}, n_{max} — ograniczenia na liczbę żetonów
- c_{min}, c_{max} — ograniczenia na liczbę cykli
- t_{min}, t_{max} — ograniczenia na liczbę drzew, doczepionych do każdego z cykli
- α_{min}, α_{max} — ograniczenia na to, jaki ułamek liczby żetonów w każdej ze składowych grafu znajduje się na cyklu
- p_{min}, p_{max} — ograniczenia na współczynnik, sterujący stopniem węzłów drzew (im wyższy współczynnik, tym większa statystycznie liczba synów w każdym węźle, a zatem drzewa szersze i niższe)

Wartości parametrów dla poszczególnych testów przedstawia tabela:

Nr	d	n_{min}	n_{max}	c_{min}	c_{max}	t_{min}	t_{max}	α_{min}	α_{max}	p_{min}	p_{max}
0	2	test z treści zadania									
1	100	4	5	1	5	1	5	0,01%	100%	0,0%	100%
2	100	9	10	1	10	1	10	0,01%	100%	0,0%	100%
3	100	19	20	1	20	1	20	0,01%	100%	0,0%	100%
4	100	90	100	1	100	1	100	0,01%	100%	0,0%	100%
5	100	90	100	1	10	1	5	0,01%	100%	0,0%	100%
6	100	90	100	1	10	1	5	0,01%	50%	0,0%	100%
7	100	90	100	1	10	1	5	0,01%	20%	0,1%	10%
8	40	400	500	1	500	1	500	0,01%	100%	0,0%	100%
9	5	1900	2000	500	2000	1	2000	0,01%	100%	0,0%	100%
	5	1900	2000	1	2	1	1	0,01%	5%	0,003%	0,03%
	5	1900	2000	1	2	1	1	0,01%	5%	96,7%	100%
	5	1900	2000	10	20	1	5	0,01%	20%	0,1%	1%
10	5	1900	2000	1	2	1	1	0,01%	5%	0,003%	0,03%
	5	1900	2000	1	2	1	1	0,01%	5%	96,7%	100%
	5	1900	2000	500	2000	1	2000	0,01%	100%	0,0%	100%
	5	1900	2000	10	20	1	5	0,01%	20%	0,1%	1%

Testy 1–3 były prostymi testami poprawnościowymi. Niewykluczone, że mogły być rozwiązane nawet przez algorytm wykładniczy, który np. szukałby izomorfizmu, badając wszystkie możliwe permutacje zbioru wierzchołków grafu. Na testach 4–7 pewne szanse miały rozwiązania, działające w czasie sześciennym. Przez ostatnie trzy testy przechodziły tylko algorytmy o złożoności czasowej $O(n^2)$.

A. Malinowski, W. Rytter Treść zadania | **A. Malinowski** Opracowanie | **Paweł Wolff** Program

Porównywanie naszyjników

W Bajtocji żyje bardzo znany jubiler Bajtazar. Zajmuje się on wyrobem naszyjników. Naszyjniki są zrobione z drogocennych kamieni nanizanych na nitkę. Do wyrobu naszyjników używa się 26-ciu rodzajów kamieni, będziemy je oznaczać małymi literami alfabetu (angielskiego): `a`, `b`, …, `z`. *Bajtazar postawił sobie za punkt honoru, aby nigdy nie wykonać dwóch takich samych naszyjników i przechowuje opisy wykonanych przez siebie naszyjników. Niektóre z tych naszyjników są bardzo długie. Dlatego też ich opisy mają skróconą postać. Każdy opis składa się z szeregu wielokrotnie powtórzonych sekwencji kamieni (wzorów). Opis naszyjnika to ciąg wzorów wraz z liczbami ich powtórzeń. Każdy wzór jest opisany za pomocą sekwencji liter reprezentujących kamienie tworzące wzór. Przykładowo, opis:* `abc` *2* `xyz` *1* `axc` *3 reprezentuje naszyjnik* `abcabcxyzaxcaxcaxc` *powstały przez dwukrotne powtórzenie wzoru* `abc`*, jednokrotne wystąpienie wzoru* `xyz` *i trzykrotne powtórzenie wzoru* `axc`*. Sprawę dodatkowo utrudnia fakt, iż naszyjniki nie mają widocznego początku, ani końca i można je dowolnie obracać w kółko. Powyższy opis reprezentuje również np. naszyjniki* `cabcxyzaxcaxcaxcab` *oraz* `xcaxcaxcabcabcxyza`.

Zadanie

Napisz program, który:

- *wczyta z pliku wejściowego* `nas.in` *dwa opisy naszyjników,*
- *sprawdzi, czy opisy te reprezentują takie same naszyjniki,*
- *zapisze wynik do pliku* `nas.out`.

Wejście

W pierwszym i drugim wierszu pliku tekstowego `nas.in` *znajdują się opisy naszyjników, po jednym w wierszu. Każdy z nich składa się z sekwencji liczb całkowitych i słów złożonych z małych liter alfabetu angielskiego, pooddzielanych pojedynczymi odstępami. Opis naszyjnika składa się z liczby całkowitej n równej liczbie wzorów występujących w opisie naszyjnika ($1 \leq n \leq 1\,000$), po której występuje n opisów powtórzeń wzorów. Opis powtórzeń i-tego wzoru składa się z: liczby całkowitej l_i równej długości wzoru ($1 \leq l_i \leq 10\,000$), słowa s_i złożonego z l_i małych liter alfabetu (angielskiego)* `a`, …, `z`, *reprezentującego wzór oraz liczby całkowitej k_i równej liczbie powtórzeń wzoru s_i ($1 \leq k_i \leq 100\,000$). Wiadomo, że suma liczb l_i (dla $i = 1, \ldots, n$) nie przekracza $10\,000$.*

Wyjście

Twój program powinien zapisać, w pierwszym i jedynym wierszu pliku wyjściowego `nas.out`*, słowo "TAK", jeśli obydwa opisy przedstawiają taki sam naszyjnik, lub słowo "NIE", w przeciwnym przypadku.*

Przykład

Dla pliku wejściowego `nas.in`*:*

```
3 3 abc 2 3 xyz 1 3 axc 3
4 4 cabc 1 4 xyza 1 3 xca 3 1 b 1
```

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

```
TAK
```

Zmiany w treści zadania

Podczas zawodów dokonano następującej zmiany w treści zadania:

Naszyjniki powstałe jeden z drugiego przez odwrócenie kolejności kamieni nie muszą być identyczne, a więc np. opisy 'abc' i 'cba' nie reprezentują tego samego naszyjnika.

Rozwiązanie

Zadanie sprowadza się do sprawdzenia, czy dwa dane słowa (ciągi znaków) są *cyklicznie równoważne*, to znaczy, czy jedno można otrzymać z drugiego przez jego cykliczne przesunięcie. Nietrudno zauważyć, że mające takie same długości słowa u i w są cyklicznie równoważne wtedy i tylko wtedy, gdy u występuje jako podsłowo słowa $w \cdot w$ (gdzie $\cdot$ oznacza konkatenację, czyli sklejenie słów). Nasz problem można by zatem rozwiązać w czasie proporcjonalnym do długości badanych słów, stosując efektywny algorytm wyszukiwania wzorca w tekście. Znane są dość wyrafinowane algorytmy wyszukiwania wzorca w czasie liniowym bez użycia pomocniczych tablic (zob. np. [10]), jednak cykliczną równoważność można sprawdzić znacznie prościej (zob. [11]):

Wprowadźmy oznaczenia:

- $|a|$ to długość słowa a;
- $u^{(k)} = u[k+1..n] \cdot u[1..k]$ (cykliczne przesunięcie słowa u o k pozycji w lewo);
- $D1 = \{1 \le k \le n : w^{(k-1)} >_L u^{(j)}$ dla pewnego $j\}$, gdzie $>_L$ oznacza porządek leksykograficzny na słowach;
- podobnie $D2 = \{1 \le k \le n : u^{(k-1)} >_L w^{(j)}$ dla pewnego $j\}$.

```
1:  { Algorytm sprawdzania, czy u powstaje przez cykliczne przesunięcie w }
2:  { Niech x = uu$, y = ww#, n=|| u|| }
3:  begin
4:      i := 0; j := 0; k := 1;
5:      while (i < n) and (j < n) and (k ≤ n) do
6:      begin
7:          k := 1;
8:          while x[i + k] = y[j + k] do
9:              k := k + 1;
10:         if k ≤ n then
11:             if x[i + k] > y[j + k] then
12:                 i := i + k
13:             else
14:                 j := j + k
15:     { Niezmiennik: [1..i] ⊆ D1, [1..j] ⊆ D2 }
16:     end
17: end;
18: { u jest cyklicznym przesunięciem w wtedy i tylko wtedy, gdy k > n }
```

Algorytm działa w czasie liniowym względem n, a jego poprawność wynika z podanego niezmiennika oraz z faktu, że jeśli $D1 = [1..n]$ lub $D2 = [1..n]$, to słowa u i w nie są cyklicznie równoważne.

Dodatkową trudność w zadaniu stanowi to, że opisy naszyjników są podane w formie skompresowanej. Żeby uzyskać program działający w czasie proporcjonalnym nie do faktycznego rozmiaru samych naszyjników, ale raczej do rozmiaru ich *opisów*, musimy odpowiednio zaimplementować pętlę w wierszach 8–9. W naszym przypadku wystarczy, żebyśmy potrafili efektywnie rozstrzygać, czy któreś ze słów a^l, b^r (gdzie a^l oznacza konkatenację l kopii słowa a) jest prefiksem (czyli fragmentem początkowym) drugiego słowa. Nietrudno pokazać, że jeśli $|a^l|, |b^r| \ge |a| + |b|$, to powyższy warunek jest równoważny temu, że $a \cdot b = b \cdot a$. Stąd wynika, że w celu stwierdzenia, czy któreś ze słów a^l, b^r jest prefiksem drugiego, wystarczy porównać tylko $|a| + |b|$ początkowych liter tych słów.

Usprawnienia wymagają jeszcze wiersze 12 i 14 w algorytmie. Z podanego niezmiennika wynika, że jeśli na pozycji $i + k$ (odpowiednio $j + k$) mamy do czynienia z co najmniej drugim powtórzeniem pewnego wzoru, to bez zaburzenia niezmiennika możemy od razu przeskoczyć do ostatniego powtórzenia tego wzoru.

Testy

Do sprawdzania rozwiązań zawodników użyto 11 grup testów (po 3 testy w każdej grupie). Oto ich krótka charakterystyka:

- małe testy poprawnościowe (grupy 1–3)

- średnie testy poprawnościowe (grupy 4–5): 5–10 wzorów długości około 100, powtarzających się około 500 razy
- duże testy wydajnościowe (grupy 6–8): wzory długości 1000–3000, powtarzające się około 10000 razy
- bardzo duże testy wydajnościowe (grupy 9–11): suma długości wzorów 5000–10000, 50000–100000 powtórzeń.

Zwiedzanie miasta

Bajtocka Agencja Turystyczna (w skrócie BAT) chce wejść na rynek oferując zwiedzanie Bajtogrodu autobusem–kabrioletem. Należy zbudować siedzibę firmy, w której będzie się zaczynało i kończyło zwiedzanie. Trasa zwiedzania musi przechodzić wszystkimi ulicami miasta, w przeciwnym przypadku turyści mogliby podejrzewać, że nie zobaczyli czegoś bardzo interesującego.

Ulice nie muszą być proste i mogą przebiegać tunelami lub wiaduktami. Wszystkie ulice są dwukierunkowe. Każda ulica łączy dwa skrzyżowania. Z każdego skrzyżowania w czterech kierunkach wychodzą ulice. Może się zdarzyć, że dwa skrzyżowania są połączone więcej niż jedną ulicą. Na ulicach nie wolno zawracać, ale można to robić na skrzyżowaniach. Ponadto wiadomo, że z każdego skrzyżowania da się dojechać do każdego innego.

Przy każdej ulicy, dokładnie w połowie drogi pomiędzy skrzyżowaniami, które łączy ulica, znajduje się szczególnie godna podziwu atrakcja turystyczna (np. piękny widok, pomnik lub inny zabytek), wywierająca na zwiedzających `wrażenie" określone nieujemną liczbą całkowitą. Siedziba BATu powinna znajdować się przy jednej z takich atrakcji.

Przy doborze trasy zwiedzania należy brać pod uwagę zainteresowanie turystów, które może się zmieniać w trakcie zwiedzania. Przejechanie autobusem jednej bajtomili powoduje spadek zainteresowania o jeden. Przejechanie **po raz pierwszy** *obok danej atrakcji turystycznej zwiększa zainteresowanie turystów, o liczbę określająca wrażenie, jakie robi atrakcja. Początkowo poziom zainteresowania turystów jest równy wrażeniu, jakie robi atrakcja, przy której znajduje się siedziba BATu. Zainteresowanie turystów nie może w trakcie wycieczki nigdy spaść poniżej zera.*

Zadanie

Napisz program, który:

- *wczyta opis miasta z pliku tekstowego* `zwi.in`,
- *znajdzie trasę spełniającą podane wymagania, lub stwierdzi, że taka trasa nie istnieje,*
- *zapisze wynik do pliku tekstowego* `zwi.out`.

Wejście

W pierwszym wierszu pliku tekstowego `zwi.in` *znajduje się jedna liczba całkowita n określająca liczbę skrzyżowań, $1 < n \leq 10\,000$. Skrzyżowania są ponumerowane od 1 do n, a ulice są ponumerowane od 1 do $2n$. Kolejnych $2n$ wierszy opisuje ulice — $(i+1)$-szy wiersz w pliku opisuje ulicę o numerze i. W każdym wierszu znajdują się cztery liczby całkowite a, b, l, s oddzielone pojedynczymi odstępami. Liczby a i b to numery skrzyżowań, które łączy dana ulica, $1 \leq a, b \leq n$, $a \neq b$. Liczba l jest parzystą liczbą całkowitą będącą długością ulicy w bajtomilach, $2 \leq l \leq 1\,000$. Atrakcja turystyczna położona przy danej ulicy robi wrażenie określone liczbą s, $0 \leq s \leq 1\,000$.*

Wyjście

Pierwszy wiersz pliku tekstowego `zwi.out` *powinien zawierać jedno słowo* `TAK`, *jeżeli istnieje taka trasa, lub* `NIE`, *w przeciwnym przypadku. Jeśli odpowiedź jest pozytywna to kolejne wiersze powinny opisywać przykładową trasę. Drugi wiersz powinien zawierać dokładnie jedną liczbę całkowitą k równą liczbie skrzyżowań występujących na trasie zwiedzania. (Pamiętaj, że ulica, przy której ma znajdować się siedziba BATu łączy pierwsze i ostatnie skrzyżowanie). Oznaczmy przez s_i (dla $i = 1, 2, \ldots, k$) numer ulicy, którą podczas zwiedzania dojeżdża się do i-tego (w kolejności zwiedzania) skrzyżowania. Kolejny wiersz powinien zawierać dwie liczby całkowite s_1 i d równe odpowiednio numerowi ulicy, przy której należy zbudować siedzibę BATu oraz numerowi pierwszego skrzyżowania, przez które prowadzi trasa zwiedzania. Kolejne $k-1$ wierszy powinno zawierać po jednej liczbie całkowitej, odpowiednio s_2, s_3, …, s_k.*

Przykład

Dla pliku wejściowego `zwi.in`:

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

```
3 2 4 2
4 3 10 8
2 1 8 7
4 3 2 1
1 4 2 6
3 1 4 5
```

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

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

Rozwiązanie

W grafie nieskierowanym definiujemy *cykl Eulera* jako cykl przechodzący przez każdą krawędź grafu dokładnie raz. Rozpatrzmy graf, w którym wierzchołkami będą skrzyżowania, a krawędziami ulice. Nietrudno dostrzec, że istnieje w tym grafie cykl Eulera, ponieważ każdy wierzchołek ma parzysty stopień — z każdego skrzyżowania wychodzą cztery ulice. Znalezienie pewnej trasy przebiegającej po takim cyklu wydaje się być dobrym pomysłem, gdyż agencja turystyczna „traci" wówczas najmniej z zadowolenia turystów. Tymczasem mamy:

Fakt 1 *W grafie opisanym przez poprawne dane wejściowe istnieje cykl Eulera.*

Weźmy teraz pewien cykl o długości k, gdzie $k > 2$, z wierzchołkami ponumerowanymi kolejno od 1 do k. Przypiszmy każdemu wierzchołkowi pewną nieujemną liczbę — wierzchołkowi nr i przyporządkowujemy liczbę w_i. Niech l_i będzie długością krawędzi od wierzchołka nr i do $i+1$, jeśli $i < k$, albo do 1 w przeciwnym przypadku. Rozważamy teraz następującą sytuację: wybieramy pewien wierzchołek i obchodzimy cykl w kierunku zgodnym z numeracją $1 \to 2 \to \dots k \to 1$, wracając do punktu wyjściowego. Przypuśćmy, że przejście krawędzi kosztuje tyle, co jej długość, a w każdym wierzchołku otrzymujemy zwrot kosztu w_i dla tego wierzchołka. Niech $B = \sum_{i=1}^{k}(w_i - l_i)$ i przyjmijmy, że na początku dysponujemy kapitałem równym 0. Wówczas prawdziwe jest następujące stwierdzenie:

Fakt 2 *Cykl można obejść zachowując zawsze nieujemne konto wtedy i tylko wtedy, gdy* $B \geq 0$.

Dowód: Przypuśćmy najpierw, że cykl można obejść w ten sposób. Wtedy bilans podróży wyrażający się jako B musi być nieujemny. Teraz dowód w drugą stronę. Zakładamy, że $B \geq 0$. Rozpocznijmy symulację w wierzchołku o numerze 1. Przechodzimy cykl przy założeniu, że możemy mieć ujemny stan konta. Oznaczamy przez b_i stan konta już po dojściu do wierzchołka numer i, ale jeszcze przed pobraniem wyznaczonej rekompensaty. Na starcie mamy $b_1 = 0$. Dla pewnego j otrzymujemy minimalną wartość b_j. Wierzchołek o numerze j będzie naszym nowym wierzchołkiem startowym. Rozpoczynając drogę tym razem w j dostajemy nowe wartości b'_i takie, że:

$$b'_i = \begin{cases} b_i - b_j + B & \text{dla } i < j, \\ 0 & \text{dla } i = j, \\ b_i - b_j & \text{dla } i > j. \end{cases}$$

Widać już teraz, że b'_i nie może być ujemne dla żadnego i, gdyż dla każdego i zachodzi $b_i \geq b_j$, czyli $b_i - b_j \geq 0$.

Wracamy teraz do naszego zadania. Fakt 1 gwarantuje nam istnienie cyklu Eulera. W dowolnie znalezionym cyklu Eulera jako wierzchołki traktujemy teraz atrakcje turystyczne. Fakt 2 daje nam ostatecznie, że rozwiązanie zadania istnieje wtedy i tylko wtedy, gdy suma wrażeń dostarczanych przez atrakcje jest nie mniejsza od sumy długości ulic, a ponadto do znalezienia takiego rozwiązania można posłużyć się dowolnym cyklem Eulera. Możemy już zapisać główne kroki algorytmu rozwiązującego zadanie:

1. wczytaj dane;
2. sprawdź, czy wrażeń dostarczanych przez atrakcje turystyczne wystarczy na zwiedzenie miasta, a jeśli tak to:

   (a) znajdź cykl Eulera,
   (b) znajdź odpowiedni punkt startowy na skonstruowanym cyklu;

3. zapisz wynik.

Punkt 2(a) można zrealizować przechodząc graf w głąb po nieodwiedzonych jeszcze krawędziach. Odpowiednia procedura może mieć następujący pseudokod:

```
procedure euler( v);
begin
  for w in Sąsiedzi( v) do
    if not odwiedzona[ v–w] then
      begin
        odwiedzona[ v–w]:=true;
        euler( w);
        DopiszKrawędź( v–w);
       { dopisuje krawędź na koniec początkowo pustej listy }
      end
end
```

Podaną procedurę wywołujemy dla dowolnego wierzchołka, na przykład dla $v = 1$. Nie trudno dowieść, że znajduje ona cykl Eulera. Przy sprawnym zaimplementowaniu ten krok algorytmu wymaga czasu $O(n)$.

W implementacji punktu 2(b) algorytmu, można posłużyć się konstruktywnym dowodem faktu 2. Ta część algorytmu działa również w czasie $O(n)$. Podsumowując, cały algorytm działa w czasie $O(n)$.

Testy

Do oceny rozwiązań zawodników użyto kompletu 12 testów:

- `zwi1a.in` — prosty test poprawnościowy, $n = 4$;
- `zwi1b.in` — prosty test poprawnościowy na odpowiedź NIE, $n = 4$;
- `zwi2a.in` — graf pełny, $n = 5$;
- `zwi2b.in` — prosty test poprawnościowy, $n = 10$;
- `zwi3.in` — losowy graf, $n = 100$;
- `zwi4.in` — okrąg z losowymi wagami, $n = 1000$;
- `zwi5.in` — dwa nałożone na siebie losowe cykle, $n = 1000$;
- `zwi6.in` — drabinka, $n = 3000$;
- `zwi7.in` — dwa nałożone na siebie losowe cykle, $n = 5000$;
- `zwi8.in` — dwa równoległe okręgi połączone krawędziami, $n = 6000$;
- `zwi9.in` — graf losowy, $n = 10000$;
- `zwi10.in` — dwa nałożone na siebie losowe cykle, $n = 10000$.

Testy `zwi1a.in` i `zwi1b.in` oraz `zwi2a.in` i `zwi2b.in` były zgrupowane. Różnice pomiędzy sumą wrażeń i sumą długości ulic były w testach niewielkie, a krawędzie w plikach znajdowały się w losowej kolejności.

Marcin Kubica
Treść zadania, Opracowanie

Marcin Sawicki
Program

Bank

W Bajtocji funkcjonują cztery rodzaje waluty: denary, franki, grosze i talary, nie wymienialne między sobą. Przysparza to wiele kłopotów mieszkańcom Bajtocji.

Bajtocki Bank Biznesu (w skrócie BBB) na skutek pomyłki w rodzaju waluty stanął w obliczu utraty płynności gotówkowej. Zawarł on z klientami szereg umów na kredytowanie różnych przedsięwzięć. Wszystkie te umowy są zawarte według takiego samego wzoru:

- *umowa określa maksymalną wysokość kredytu w każdym rodzaju waluty,*
- *w ramach tak określonego limitu, gdy klient potrzebuje gotówki, to zgłasza się do BBB prosząc o określoną sumę w każdym rodzaju waluty; BBB może dowolnie długo zwlekać z wypłaceniem pieniędzy, ale dopóki klient nie przekracza maksymalnej wysokości kredytu, to prędzej czy później musi je klientowi wypłacić,*
- *po otrzymaniu pieniędzy klient może zgłaszać się po kolejne transze, aż do wyczerpania limitu,*
- *na koniec klient spłaca całość zaciągniętego kredytu, może z tym zwlekać dowolnie długo, ale prędzej czy później musi spłacić kredyt,*
- *klient nie ma obowiązku wykorzystania kredytu w maksymalnej wysokości,*
- *dla uproszczenia zakładamy, że klienci nie płacą żadnych odsetek ani prowizji.*

BBB nie dysponuje wystarczającą ilością gotówki, aby zaspokoić potrzeby swoich klientów, a bez ich wcześniejszego zaspokojenia kredyty nie będą spłacane. BBB poprosił Bajtocki Fundusz Walutowy (w skrócie BFW) o pomoc. BFW zgodził się pomóc BBB, ale zażądał, żeby BBB określił minimalne kwoty każdego rodzaju waluty, jakie BBB musi posiadać, aby móc doprowadzić do spłacenia przez klientów wszystkich kredytów (nawet jeżeli klienci będą chcieli wykorzystać swoje kredyty do maksymalnej ich wysokości).

Specjaliści BBB odkryli, że możliwych jest wiele odpowiedzi na tak postawione pytanie (por. przykład). BFW odpowiedziało, że interesują ich dowolne takie kwoty poszczególnych rodzajów walut, że gdyby zmniejszyć którąkolwiek z nich choćby o 1, to mogłyby nie wystarczyć do zakończenia realizacji wszystkich kredytów.

Zadanie

Napisz program, który:

- *wczyta z pliku* `ban.in` *maksymalne i aktualne wysokości kredytów klientów,*
- *wyznaczy minimalne kwoty poszczególnych rodzajów walut gwarantujące możliwość realizacji wszystkich kredytów,*
- *zapisze wynik w pliku* `ban.out`.

Jeśli jest możliwych wiele wyników, to Twój program powinien zapisać dowolny z nich.

Wejście

W pierwszym wierszu pliku tekstowego `ban.in` *jest zapisana jedna dodatnia liczba całkowita* n *równa liczbie klientów,* $1 \le n \le 8\,000$. *Klienci są ponumerowani od 1 do* n. *W kolejnych* n *wierszach jest zapisanych po osiem nieujemnych liczb całkowitych. W* $i+1$*-szym wierszu (dla* $i = 1, \ldots, n$*) zapisane są liczby* $m_{i,1}, m_{i,2}, m_{i,3}, m_{i,4}, w_{i,1}, w_{i,2}, w_{i,3}, w_{i,4}$, $(0 \le w_{i,j} \le m_{i,j} \le 50\,000$, *dla* $j = 1, \ldots, 4)$. *Liczby* $m_{i,j}$ *i* $w_{i,j}$ *określają odpowiednio maksymalną i aktualną wysokość kredytu klienta nr* i *w: denarach (*$j = 1$*), frankach (*$j = 2$*), groszach (*$j = 3$*) i talarach (*$j = 4$*).*

Wyjście

Twój program powinien zapisać w pierwszym (i jedynym) wierszu pliku tekstowego `ban.out` *cztery nieujemne liczby całkowite, pooddzielane pojedynczymi odstępami, określające minimalne kwoty gotówki, jakie musi posiadać BBB, odpowiednio w denarach, frankach, groszach i talarach.*

Przykład

Dla pliku wejściowego `ban.in`*:*

```
4
3 2 1 2 0 2 0 1
2 4 1 8 1 2 1 1
3 2 0 3 1 0 0 1
3 0 1 2 1 0 0 1
```

poprawną odpowiedzią może być plik tekstowy `ban.out`*:*

```
1 2 0 7
```

lub:

```
2 0 1 4
```

Zakleszczenie i algorytm bankiera

Zadanie to jest związane ze zjawiskiem zakleszczenia oraz algorytmem bankiera używanym do unikania zakleszczenia (zobacz [28], p. 7.5.3). Zjawisko zakleszczenia występuje w systemach operacyjnych, w których *współbieżnie* (tzn. równocześnie) może być wykonywanych wiele *procesów* (działających programów). Procesy te mogą używać rozmaitych zasobów systemowych, takich jak pamięć, czy urządzenia wejścia/wyjścia. Zasoby te są przydzielane procesom przez system operacyjny. Zakleszczenie występuje wówczas, gdy kilka procesów czeka nawzajem na siebie, prosząc system o przydzielenie zasobów, które są zajęte przez pozostałe procesy. Wyobraźmy sobie na przykład, że w systemie jest jeden napęd CD–ROM i jedna karta dźwiękowa, oraz że dwa procesy (P_1 i P_2) chcą uzyskać wyłączny dostęp do tych urządzeń. Załóżmy przy tym, że P_1 uzyskał już dostęp do CD–ROM'u i czeka na zwolnienie karty dźwiękowej, a P_2 ma już dostęp do karty dźwiękowej i czeka na zwolnienie CD-ROM'u. Jak łatwo zauważyć, te dwa procesy będą na siebie czekać w nieskończoność. Takie zjawisko nazywamy zakleszczeniem. Oczywiście zakleszczenie jest zjawiskiem niepożądanym.

Jeden ze sposobów radzenia sobie z zakleszczeniem polega na takim przydzielaniu przez system operacyjny zasobów procesom, aby unikać zakleszczenia. Pomysł polega na tym, aby utrzymywać system w *bezpiecznym* stanie, tzn. takim stanie, w którym mamy gwarancje, że wszystkie działające procesy mogą zostać wykonane aż do końca, bez zakleszczenia. System operacyjny przydziela zasoby procesom tylko wtedy, gdy prowadzi to do bezpiecznego stanu. W rezultacie, może się zdarzyć, że ze względów bezpieczeństwa proces musi czekać, mimo że potrzebne mu zasoby są dostępne.

W celu stwierdzenia czy stan systemu jest bezpieczny używany jest *algorytm bankiera*. Algorytm ten potrzebuje dodatkowej informacji: maksymalnych ilości zasobów jakie mogą być potrzebne poszczególnym procesom. Procesy deklarują maksymalne ilości potrzebnych zasobów w momencie uruchomienia. Algorytm bankiera opiera się na analogii między systemem operacyjnym, a opisanym w treści zadania systemem bankowym. Różne rodzaje zasobów to różne, wzajemnie nie wymienialne między sobą waluty. Klienci to procesy działające w systemie, a BBB to system operacyjny. Środki pieniężne jakimi dysponuje BBB to wolne zasoby, a pieniądze pożyczone klientom to zasoby przydzielone procesom. Natomiast maksymalne ilości potrzebnych zasobów deklarowane przez procesy, to wysokości limitów określone w umowach kredytowych.

Algorytm bankiera opiera się na następujących obserwacjach. Jeżeli w danym stanie system może doprowadzić do zakończenia wszystkich procesów, to może to również zrobić wykonując i kończąc te procesy w pewnej kolejności, po jednym na raz. Aby móc zakończyć jakiś proces, musimy mieć w systemie tyle wolnych zasobów, żeby zaspokoić jego potrzeby. W najgorszym przypadku ilość potrzebnych zasobów jest równa różnicy między maksymalną zadeklarowaną liczbą potrzebnych zasobów, a ilością aktualnie przydzielonych zasobów. Jednak po zakończeniu procesu w systemie może tylko przybyć wolnych zasobów, gdyż wszystkie zasoby przydzielone procesowi są zwalniane. Tak więc, aby odpowiedzieć na pytanie czy sytuacja jest bezpieczna, musimy stwierdzić, czy istnieje taka kolejność $P_1, P_2, \ldots, P_n$, w której możemy wykonywać i kończyć procesy. Ponadto, taką kolejność możemy konstruować w sposób zachłanny — jeżeli istnieje taka kolejność i wolne w danej chwili zasoby wystarczają do zakończenia procesu P, to istnieje również taka kolejność zaczynająca się od P.

Niech n będzie liczbą procesów, a m liczbą rodzajów zasobów (w naszym przypadku $m = 4$). Zakładamy, że są określone następujące cztery tablice:

- *wolne* — wektor długości m określający ilości wolnych zasobów poszczególnych rodzajów,
- *maks* — macierz $n \times m$ określająca zadeklarowane przez poszczególne procesy maksymalne ilości potrzebnych im zasobów,
- *przydzielone* — macierz $n \times m$ określająca ilości zasobów przydzielonych poszczególnym procesom,

- *potrzebne* — macierz $n \times m$ określająca maksymalne ilości zasobów potrzebnych do zakończenia poszczególnych procesów; tę macierz możemy zawsze wyznaczyć na podstawie wzoru $potrzebne[i,j] = maks[i,j] - przydzielone[i,j]$.

Dodatkowo zakładamy, że mamy do dyspozycji dwa pomocnicze wektory: *pom* i *zakończone* długości odpowiednio m i n. Wektor *pom* reprezentuje symulowaną ilość wolnych zasobów, a *zakończone* reprezentuje zbiór procesów, które udało się zakończyć. Algorytm bankiera ma następującą postać:

```
pom := wolne;
zakończone := (false, false, ..., false);
while istnieje takie i, że:
not zakończone[i] and ∀j potrzebne[i, j] ≤ pom[j]
do begin
  for j := 1 to m do
     pom[j] := pom[j] + przydzielone[i, j];
   zakończone[i] := true
end;
if zakończone = (true, true, ..., true) then
  system jest w stanie bezpiecznym
else
  system nie jest w stanie bezpiecznym.
```

Algorytm ten ma złożoność $O(n^2 m)$.

Algorytm bankiera jest zwykle używany do określenia, czy stan systemu po przydzieleniu zasobów jest bezpieczny. W niniejszym zadaniu problem jest postawiony trochę inaczej. System znalazł się w stanie niebezpiecznym i pytanie dotyczy minimalnej liczby wolnych zasobów potrzebnych do tego, aby stan był bezpieczny.

Rozwiązanie

Możemy zastosować do rozwiązania tego zadania algorytm bankiera. Dla określonych danych wejściowych może istnieć wiele poprawnych wyników. Nasze rozwiązanie wyznacza wynik, który jest najmniejszy w porządku leksykograficznym. Inaczej mówiąc, wyznaczamy najpierw najmniejszą liczbę potrzebnych denarów, następnie dla tak określonej liczby denarów najmniejszą liczbę potrzebnych franków, itd. Liczba potrzebnych denarów jest z jednej strony nieujemna, a z drugiej nie przekracza sumy limitów na denary we wszystkich umowach kredytowych. Konkretną wartość znajdujemy w tym przedziale za pomocą metody bisekcji, stosując za każdym razem algorytm bankiera i sprawdzając czy przy danej liczbie denarów i nieograniczonych zasobach pozostałych walut stan jest bezpieczny — jeżeli nie, to liczba denarów jest zbyt mała. Jak wspomnieliśmy powyżej, koszt algorytmu bankiera wynosi $O(n^2 m)$. Jeżeli oznaczymy przez s maksymalną wysokość limitu jednej waluty w umowie kredytowej, to koszt wyznaczenia minimalnej liczby potrzebnych denarów wynosi $O(n^2 m \log(ns))$. Po ustaleniu liczy denarów możemy w podobny sposób wyznaczyć minimalną liczbę potrzebnych franków, zakładając, że mamy nieograniczone zasoby groszy i talarów. Podobnie wyznaczamy minimalną liczbę potrzebnych groszy oraz talarów. W ten sposób otrzymujemy rozwiązanie o złożoności rzędu $O(n^2 m^2 \log(ns))$.

Okazuje się, że możemy to zadanie rozwiązać bardziej efektywnie. Tak jak w powyższym algorytmie ustalamy minimalne potrzebne ilości kolejnych walut. Powiedzmy, że w kolejnym kroku wyznaczamy minimalną ilość k-tej waluty. Wykonujemy wówczas zmodyfikowany algorytm bankiera, który na podstawie ustalonych ilości walut 1, $\ldots$, $k-1$ i przy założeniu, że mamy nieograniczone zasoby walut $k+1, \ldots, m$, wyznacza minimalną potrzebną kwotę k-tej waluty. Symulujemy zakończenie kredytów w pewnej kolejności. W tym celu symulujemy pulę dostępnych środków pieniężnych oraz utrzymujemy zbiór kredytów, do zakończenia których mamy wystarczającą ilość walut $1, \ldots, k-1$. Symulując kończenie kredytów wybieramy za każdym razem taki kredyt, do zakończenia którego mamy wystarczająca ilość walut $1, \ldots, k-1$ i który wymaga najmniej środków k-tej waluty. Jeśli mamy wystarczającą ilość środków, to po prostu symulujemy zakończenie i spłatę tego kredytu. Jeśli natomiast brakuje środków k-tej waluty, to odpowiednio zwiększamy ich ilość. W ten sposób, po zakończeniu symulacji znamy minimalną wymaganą ilość środków waluty k.

Aby uzyskać dobrą złożoność takiego algorytmu, musimy zastosować odpowiednią strukturę danych do przechowywania puli kredytów, do zakończenia których mamy wystarczającą ilość walut $1, \ldots, k-1$. Używamy do tego celu stogów, uporządkowanych według kwoty określonej waluty potrzebnej do zakończenia kredytu. Zakładamy, że są zaimplementowane następujące operacje na stogach:

- *heap_init*(h) — inicjuje pusty stóg h,
- *heap_empty*(h) — określa czy stóg h jest pusty,

- *heap_put*(h, e, k) — wkłada na stóg h element e skojarzony z kluczem k,
- *heap_min*(h) — określa element o najmniejszym kluczu znajdujący się na stogu h,
- *heap_get_min*(h) — zdejmuje ze stogu element o najmniejszym kluczu i przekazuje go jako wynik.

Algorytm ten jest zaimplementowany przez poniższą procedurę. Zakładamy przy tym, że zadeklarowane są następujące zmienne:

```
const
   N_MAX = 8000;
   K_MAX = 4;
var
   wys_max, wys: array[1.. N_MAX, 1.. K_MAX] of word;
   n: word;
   akt: array [1.. K_MAX] of longint;
```

Zmienna n to liczba kredytów, a tablice *wys_max* i *wys* zawierają odpowiednio limity kredytów i aktualne zadłużenie klientów.

```
procedure oblicz;
var
   heaps: array[1.. K_MAX] of heap;
   rez: array[1.. K_MAX] of longint;
   i, j, k, e: word;
   zmiana: word;
begin
  for i := 1 to K_MAX do akt[ i] := 0;
  for k := 1 to K_MAX do { Ustalanie limitu na k-tą walutę. }
  begin
    for i := 1 to k do rez[ i] := akt[ i];
    for i := 1 to k do heap_init( heaps[ i]);
    for i := 1 to n do
       heap_put( heaps[1], i, wys_max[ i,1]- wys[ i,1]);
    for i := 1 to n do
    begin
      { Wybieramy kredyty mieszczące się w limitach na waluty 1..k-1. }
      for j := 1 to k-1 do
        while
          (not heap_empty( heaps[ j])) and
          ( heap_min( heaps[ j]) ≤ rez[ j])
        do begin
           e := heap_get_min( heaps[ j]);
           heap_put( heaps[ j+1], e, wys_max[ e, j+1]− wys[ e, j+1])
        end;
      { Element wymagający najmniej środków waluty k. }
       e := heap_get_min( heaps[ k]);
      { Czy trzeba zwiększyć akt[ k]? }
      if rez[ k] < wys_max[ e, k]− wys[ e, k] then
      begin
         zmiana := wys_max[ e, k]− wys[ e, k]− rez[ k];
        inc( rez[ k], zmiana);
        inc( akt[ k], zmiana)
      end;
      { Realizujemy kredyt. }
      for j := 1 to k do
        inc( rez[ j], wys[ e, j]);
    end;
  end;
end;
```

Testy

Testy zostały wygenerowane losowo, przy czym wysokość aktualnie wykorzystanego kredytu dla żadnej waluty nie przekracza 9. Poniższa tabelka ilustruje wielkości testów. Wszystkie testy można znaleźć na załączonej dyskietce.

nr	n	maksymalna kwota kredytu
1	6	13
2	8	15
3	10	20
4	20	100
5	100	1000
6	1000	4000
7	2000	10000
8	6000	50000
9	7000	50000
10	8000	50000

Tomasz Waleń
Treść zadania, Opracowanie

Marcin Sawicki
Program

Kopalnia złota

Bajtazar, zasłużony pracownik Bajtockiej Kopalni Złota, przechodzi w tym roku na emeryturę. W związku z tym, zarząd kopalni postanowił go uhonorować. W nagrodę za wieloletnią pracę, Bajtazar może otrzymać działkę — wycinek kopalni mający postać prostokąta o bokach równoległych do osi współrzędnych oraz szerokości s i wysokości w — położoną w dowolnie przez siebie wybranym miejscu. Oczywiście nie wszystkie lokalizacje działki są równie cenne. Wartość działki mierzy się liczbą samorodków złota znajdujących się na jej terenie (jeśli samorodek leży na granicy działki, to również znajduje się na jej terenie).

Twoim zadaniem jest napisanie programu umożliwiającego wyznaczenie jaką wartość ma najcenniejsza spośród wszystkich możliwych lokalizacji działek.

Dla uproszczenia przyjmujemy, że teren kopalni jest nieograniczony, jakkolwiek samorodki występują jedynie na ograniczonym obszarze.

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `kop.in` *opis rozmieszczenia samorodków oraz wymiary działki,*
- *znajdzie wartość najcenniejszej spośród wszystkich lokalizacji działki, mierzoną liczbą znajdujących się na jej terenie samorodków,*
- *zapisze wynik w pliku tekstowym* `kop.out`*.*

Wejście

W pierwszym wierszu pliku tekstowego `kop.in` *zapisano dwie dodatnie liczby całkowite s i w oddzielone pojedynczym odstępem ($1 \leq s, w \leq 10\,000$), oznaczające odpowiednio szerokość i wysokość działki. W drugim wierszu zapisano jedną dodatnią liczbę całkowitą n ($1 \leq n \leq 15\,000$), oznaczającą liczbę samorodków znajdujących się na terenie kopalni. W kolejnych n wierszach zapisane są współrzędne poszczególnych samorodków. Każdy z tych wierszy zawiera dwie liczby całkowite x i y ($-30\,000 \leq x, y \leq 30\,000$), oddzielone pojedynczym odstępem, oznaczające odpowiednio współrzędną x i y samorodka.*

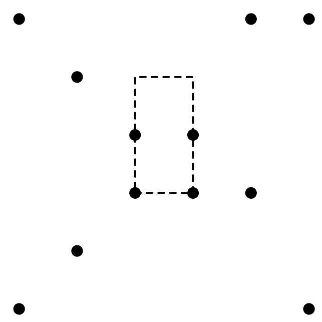
Wyjście

Plik tekstowy `kop.out` *powinien zawierać jedną liczbę całkowitą równą wartości najcenniejszej spośród wszystkich lokalizacji działek.*

Przykład

Dla pliku wejściowego `kop.in`*:*

```
1 2
12
0 0
1 1
2 2
3 3
4 5
5 5
4 2
1 4
0 5
5 0
2 3
3 2
```



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

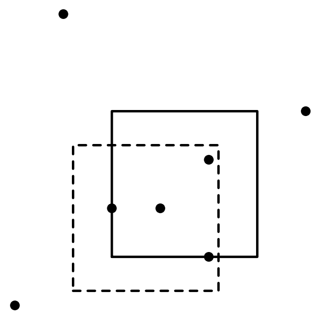
4

Rozwiązanie

Najprostszym rozwiązaniem tego zadania jest sprawdzenie wartości wszystkich możliwych działek, jednak z oczywistych powodów jest to rozwiązanie bardzo nieefektywne. Co prawda możemy ograniczyć się do takich działek, których wierzchołki pokrywają się z punktami kratowymi, ale nadal daje to $60\,000 \times 60\,000$ możliwości do sprawdzenia i co najgorsze, nawet w przypadku gdy ilość samorodków jest niewielka.

Do trochę lepszego rozwiązania prowadzi spostrzeżenie, iż zawsze możemy pokazać taką działkę o optymalnej wartości, na której lewym i dolnym brzegu leżą jakieś samorodki (każdą optymalną działkę można tak przesunąć by spełniała to kryterium, nie tracąc przy tym żadnych samorodków).

Rys. 1. Optymalna działka została zaznaczona przerywaną linią, a analogiczna działka po przesunięciu — linią ciągłą.



Zatem przestrzeń poszukiwań można już ograniczyć do $O(n^2)$ (na n różnych sposobów można wybrać samorodek na lewym brzegu i na n sposobów na dolnym brzegu), niestety nadal takie rozwiązanie będzie za mało efektywne w przypadku danych o dużych rozmiarach, ale dla danych o małych rozmiarach (np. prostych testów poprawnościowych) powinno być wystarczające.

Kod takiego rozwiązania może wyglądać następująco:

```
rozw := 0;
for i := 1 to n do
  for j := i to n do
  begin
     x := min(x_i,x_j);
     y := min(y_i,y_j);
     ile:= 0;
    { policzenie ile samorodków znajduje się w obrębie prostokąta }
    { (x,y)–(x+s,y+w) }
    for k := 1 to n do
      if (x <= x_i) and (y <= y_i) and
           (x_i <= x+s) and (y_i <= y+w) then inc( ile);
    if ile> rozw then rozw := ile
  end
```

Takie rozwiązanie ma złożoność $O(n^3)$, co przy maksymalnych danych jakie mogą wystąpić w zadaniu oznacza wykonanie około $3 \cdot 10^{12}$ operacji. Nawet przy założeniu, że dysponujemy szybkim komputerem, który potrafi wykonać 10^7 operacji w ciągu sekundy (jednak ta wartość jest bardzo uzależniona od wybranego języka programowania, kompilatora czy nawet metody kompilacji), a sam program przyspieszyć 3–krotnie (np. przez optymalizację), i tak oznaczałoby to, że program potrzebowałby na znalezienie rozwiązania 100 000s, czyli 27 godzin.

Jak więc ulepszyć poprzednie rozwiązanie? Można bardziej efektywnie zliczać liczbę samorodków w obrębie danego prostokąta, korzystając z bardziej zaawansowanych struktur danych, albo też można wykorzystać fakt, że wszystkie samorodki mamy dane na samym początku i analizować je wg. jakiegoś porządku.

W rozwiązaniu wzorcowym użyto popularnej techniki zwanej *zamiataniem*. Rozpatrzmy poziomą miotłę zamiatającą płaszczyznę od dołu do góry. Dla danego położenia miotły będziemy chcieli szybko obliczać wartości działek, których górny brzeg pokrywa się z aktualnym położeniem miotły. Podczas zamiatania płaszczyzny, dla każdego samorodka na miotle zaznaczamy przedział, w którym umieszczenie prawego górnego rogu działki obejmuje ten samorodek. Konieczne jest również dodanie sztucznych "ujemnych" samorodków, które będą służyły do usuwania samorodka (gdy miotła oddali się na odległość w).

Szkic takiego rozwiązania wygląda następująco:

```
S := { (x_i, y_i, +1) oraz (x_i, y_i + w + 1, -1): dla 1 ≤ i ≤ n };
rozw := 0;
przeglądaj S w kolejności rosnących współrzędnych y
    (x,y,z) := { kolejny element z S };
    if z= +1 then
        dodajPrzedział( x, x+s)
    else
        usunPrzedział( x, x+s);
    rozw := max( rozw, MaksymalnyPrzedział);
```

Funkcja *MaksymalnyPrzedzial* szuka punktu na osi X, który należy do największej liczby przedziałów i zwraca ich liczbę.
Dla danych z przykładu podanego w treści zadania zbiór S wygląda następująco:

$$(0,0,+1),\ (0,3,-1),\ (2,2,+1),\ (2,5,-1),\ (3,3,+1),\ (3,6,-1),\ (4,5,+1),\ (4,8,-1),\ (5,5,+1),\ (5,8,-1),\ (4,2,+1),\ (4,5,-1),\ (1,4,+1),\ (1,7,-1),\ (0,5,+1),\ (0,8,-1),\ (5,0,+1),\ (5,3,-1),\ (2,3,+1),\ (2,6,-1),\ (3,2,+1),\ (3,5,-1).$$

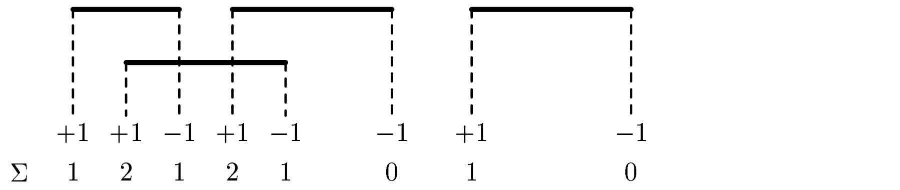
Zbiór S przeglądamy zgodnie z rosnącą współrzędną y, jednak wszystkie dla tej samej współrzędnej y trójki ze znakiem -1 powinny znaleźć się przed tymi ze znakiem $+1$.
W naszym wypadku mogłaby to być kolejność:

$$(0,0,+1),\ (5,0,+1),\ (2,2,+1),\ (4,2,+1),\ (3,2,+1),\ (5,3,-1),\ (0,3,-1),\ (3,3,+1),\ (2,3,+1),\ (1,4,+1),\ (3,5,-1).\ (4,5,-1),\ (2,5,-1),\ (4,5,+1),\ (5,5,+1),\ (0,5,+1),\ (3,6,-1),\ (2,6,-1),\ (1,7,-1),\ (4,8,-1),\ (5,8,-1),\ (0,8,-1).$$

Teraz pozostaje ustalić szczegóły: jakiej struktury danych użyć do reprezentowania przedziałów na miotle i jak szybko zrealizować funkcję *MaksymalnyPrzedział*?

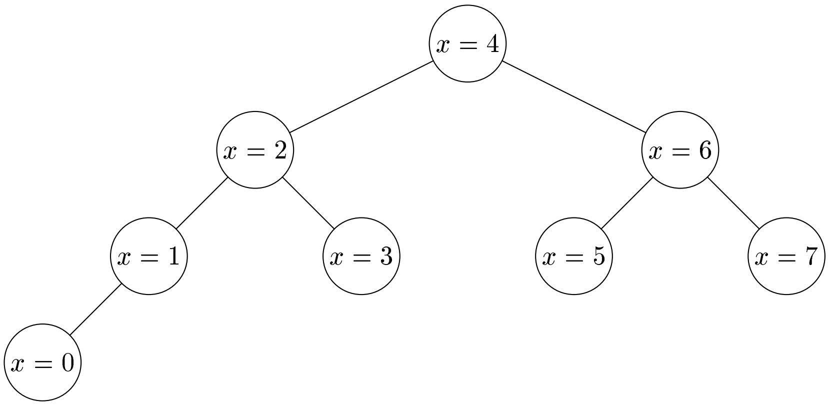
Odpowiedź na drugie pytanie można sprowadzić do problemu liczenia sum prefiksowych. Otóż gdy będziemy dodawać przedział $[a, b]$, to należy na osi X na pozycji a dodać $+1$, a na pozycji b dodać -1. Gdy będziemy chcieli usunąć jakiś przedział należy postąpić odwrotnie: na pozycji a dodać -1, a na b dodać $+1$. Teraz aby sprawdzić do ilu przedziałów należy dany punkt p, wystarczy zsumować wartości od minimalnej wartości na osi X do p, natomiast obliczenie funkcji *MaksymalnyPrzedzial* odpowiada znalezieniu maksymalnej sumy prefiksowej. Metoda ta została zilustrowana na rysunku 2.

Rys. 2. Przykład zastosowania sum prefiksowych do wyznaczania "maksymalnego" przedziału.



Bardzo często do reprezentowania miotły używa się drzew zrównoważonych takich jak AVL lub drzewa czerwono–czarne. Tak można postąpić i w tym wypadku, ale istnieje prostsze rozwiązanie. Ponieważ wszystkie punkty znamy zawczasu, możemy zbudować zrównoważone drzewo BST zawierające wszystkie wartości. Takie postępowanie może wydawać się trochę rozrzutne — w drzewach AVL przetrzymujemy tylko te wartości, które aktualnie są przetwarzane, a tu wszystkie. Jednak w tym wypadku punktów wcale nie jest aż tak dużo. Dodatkowo możemy takie drzewo ukryć w tablicy, dokładnie tak jak w przypadku kopców, a co za tym idzie, uniknąć dodatkowego kosztu związanego z przechowywaniem wskaźników, które definiują strukturę drzewa. Rysunek 3 pokazuje drzewo utworzone dla danych podanych w przykładzie zamieszczonym w treści zadania.

Rys. 3. Struktura miotły dla danych z przykładu podanego w treści zadania.



W każdym węźle drzewa oprócz wartości współrzędnej x (klucza) będziemy utrzymywać wartości:

- `MiotlaSuma` — oznaczającą sumę wartości dla kluczy z całego poddrzewa danego węzła (łącznie z nim samym),
- `MiotlaMaxSuma` — oznaczającą maksimum z wszystkich sum prefiksowych ciągu wartości zapisanych w poddrzewie danego węzła (przy czym bieżemy pod uwagę również pusty podciąg). Kolejność elementów w ciągu odpowiada porządkowi inorder.

Jak łatwo zauważyć, funkcja `MaksymalnyPrzedzial` sprowadza się do odczytania wartości pola `MiotlaMaxSuma` dla korzenia. Wstawianie wartości do miotły jest trochę bardziej kłopotliwe, należy bowiem aktualizować wszystkie te pola.
Dla danego klucza x i wartości w którą chcemy dodać, należy:

- znaleźć węzeł v zawierający klucz x,
- dodać do pola `MiotlaSuma` wartość w, dla wszystkich węzłów na ścieżce z v do korzenia,
- poprawić wartości `MiotlaMaxSuma` na tej samej ścieżce.

Przy aktualizowaniu pola `MiotlaMaxSuma` w węźle v mogą zajść dwa przypadki:

- podciąg o maksymalnej sumie kończy się w pewnym węźle v', o mniejszym kluczu niż ten z v (a więc znajdującym się w lewym poddrzewie) — w takim wypadku suma elementów takiego podciągu zapisana jest w polu `MiotlaMaxSuma` lewego syna węzła v;
- podciąg o maksymalnej sumie kończy się w pewnym węźle v', o większym lub równym kluczu niż ten z v — w takim wypadku, aby wyznaczyć wartość sumy elementów takiego podciągu należy dodać sumę elementów lewego poddrzewa do wartości przypisanych do węzła v (a więc wartość `MiotlaSuma` z węzła v po odjęciu sum z lewego i prawego poddrzewa) i dodać maksymalną sumę prefiksową z prawego poddrzewa.

Aktualizacja polega na wyliczeniu wartości obu podciągów i zapisaniu w węźle v wartości większego z nich.
Kod procedury realizującej dodawanie wartości do węzła drzewa wygląda następująco:

```
procedure WstawDoMiotly( x, wartosc: Integer);
begin
  { szukanie węzła z kluczem x }
   Wezel := 1;
  repeat
    if x < MiotlaX[ Wezel] then
        Wezel := Wezel * 2 { idziemy do lewego poddrzewa }
    else if x > MiotlaX[ Wezel] then
        Wezel := Wezel * 2 + 1 { idziemy do prawego poddrzewa }
    else
      break;
  until false;
  { Uaktualnianie sum na scieżce od Wezel do korzenia }
  repeat
    if Wezel * 2+1 ≤ IlePktowMiotly then begin
```

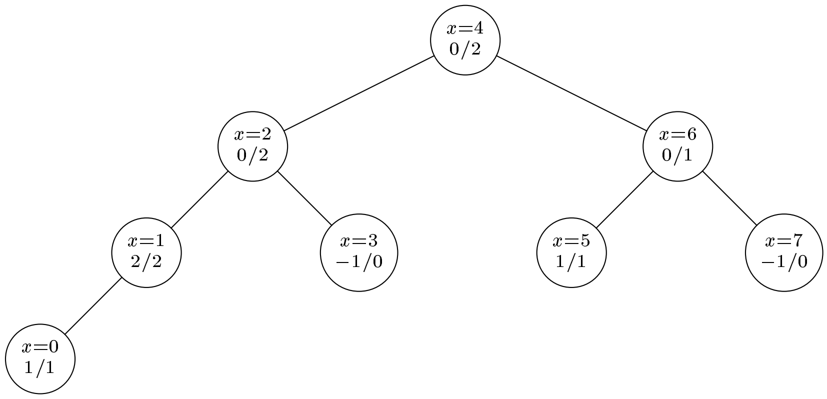
```
            MaxSumaPrawa := MiotlaMaxSuma[ Wezel * 2+1];
            SumaPrawa := MiotlaSuma[ Wezel * 2+1];
        end else begin { jeśli brak prawego syna }
            MaxSumaPrawa := 0; SumaPrawa := 0;
        end;
        if Wezel * 2 ≤ IlePktowMiotly then
            MaxSumaLewa := MiotlaMaxSuma[ Wezel * 2];
        else { jeśli brak lewego syna }
            MaxSumaLewa := 0;
        Inc( MiotlaSuma[ Wezel], k);
        MiotlaMaxSuma[ Wezel] := max(
            MaxSumaLewa, MiotlaSuma[ Wezel]− SumaPrawa+ MaxSumaPrawa);
        Wezel := Wezel div 2 { przejście do ojca w drzewie }
    until Wezel = 0;
end
```

Teraz można wprowadzić zmodyfikowaną procedurę zamiatania:

```
procedure Zamiataj;
begin
    rozw := 0;
    Q := { posortowane trójki wg. wsp. Y i znaku }
    { przetwarzanie samorodków zgodnie z rosnącą wsp. Y }
    while not empty Q do begin
        ( x, y, znak) := Q. extract;
        WstawDoMiotly( x, y, znak);
        WstawDoMiotly( x+szerokosc+1, y, −znak);
        rozw := max( rozw, MiotlaMaxSuma[1]);
    end
end
```

Rysunek 4 przedstawia stan miotły po przetworzeniu 3 pierwszych samorodków.

Rys. 4. Stan drzewa po dodaniu samorodków $(0,0)$, $(5,0)$ i $(1,1)$.



Podsumowując, aby otrzymać pełne rozwiązanie należy połączyć wszystkie wspomniane elementy:

```
    PrzygotujMiotle;
    PosortujPunkty;
    Zamiataj;
```

Do wykonania każdego z tych kroków potrzeba czasu $O(n \log n)$, stąd i całe rozwiązanie ma taką złożoność.

Testy

Rozwiązanie testowane było na 14 zestawach danych testowych, nie stosowano grupowania.

- `kop1.in` — $n = 12$, prosty test poprawnościowy;
- `kop2.in` — $n = 47$, prosty test poprawnościowy;
- `kop3.in` — $n = 48$, prosty test poprawnościowy;
- `kop4.in` — $n = 176$, prosty test poprawnościowy;
- `kop5.in` — $n = 9005$, samorodki ułożone w punktach kratowych (z dodanymi małymi zaburzeniami);
- `kop6.in` — $n = 5000$, samorodki zgrupowane wokół kształtu litery X (z dodanymi małymi zaburzeniami);
- `kop7.in` — $n = 10\,000$, samorodki zgrupowane wokół kształtu litery X (z dodanymi małymi zaburzeniami);
- `kop8.in` — $n = 14\,000$, samorodki zgrupowane wokół kształtu litery X (z dodanymi małymi zaburzeniami);
- `kop9.in` — $n = 15\,000$, samorodki zgrupowane wokół kształtu litery X (z dodanymi małymi zaburzeniami);
- `kop10.in` — $n = 15\,000$, dane losowe;
- `kop11.in` — $n = 15\,000$, dane losowe;
- `kop12.in` — $n = 15\,000$, dane losowe;
- `kop13.in` — $n = 15\,000$, dane losowe;
- `kop14.in` — $n = 15\,000$, dane losowe.

Marcin Kubica
Treść zadania, Opracowanie

Paweł Wolff
Program

Łańcuch

Bajtocja nie zawsze była państwem demokratycznym. W jej historii były również czarne karty. Pewnego razu, generał Bajtelski — przywódca junty żelazną ręką rządzącej Bajtocją — postanowił zakończyć stan wojenny, trwający od momentu przejęcia władzy, i zwolnić więzionych działaczy opozycji. Nie chciał jednak uwolnić przywódcy opozycji Bajtazara. Postanowił przykuć go do murów więzienia za pomocą **bajtockiego łańcucha**. *Bajtocki łańcuch składa się z połączonych ze sobą ogniw oraz przymocowanego do muru pręta. Choć ogniwa nie są połączone z prętem, to bardzo trudno jest je z niego zdjąć.*

– Generale, czemuś mnie przykuł do murów więzienia, miast uwolnić, jako to uczyniłeś z moimi kamratami! — wołał Bajtazar.
– Ależ Bajtazarze, wszak nie jesteś przykuty i z pewnością potrafisz sam zdjąć trzymające Cię ogniwa z pręta wystającego z murów więzienia. — przewrotnie stwierdził generał Bajtelski, po czym dodał — Uporaj się z tym jednak przed godziną cyfracyjną i nie dzwoń łańcuchami po nocy, gdyż w przeciwnym przypadku będę zmuszony wezwać funkcjonariuszy Cyfronicji Obywatelskiej.

Pomóż Bajtazarowi!

Ponumerujmy kolejne ogniwa łańcucha liczbami $1, 2, \ldots, n$. Ogniwa te możemy zakładać i zdejmować z pręta zgodnie z następującymi zasadami:

- *jednym ruchem możemy zdjąć lub założyć na pręt tylko jedno ogniwo,*
- *ogniwo nr 1 można zawsze zdjąć lub założyć na pręt,*
- *jeżeli ogniwa o numerach $1, \ldots, k-1$ (dla $1 \le k < n$) są zdjęte z pręta, a ogniwo nr k jest założone, to możemy zdjąć lub założyć ogniwo nr $k+1$.*

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `lan.in` *opis bajtockiego łańcucha,*
- *obliczy minimalną liczbę ruchów, które należy wykonać, aby zdjąć wszystkie ogniwa bajtockiego łańcucha z pręta,*
- *zapisze wynik w pliku tekstowym* `lan.out`.

Wejście

W pierwszym wierszu pliku tekstowego `lan.in` *zapisano jedną dodatnią liczbę całkowitą n, $1 \le n \le 1\,000$. W drugim wierszu zapisano n liczb całkowitych $o_1, o_2, \ldots, o_n \in \{0, 1\}$ pooddzielanych pojedynczymi odstępami. Jeśli $o_i = 1$, to ogniwo nr i jest założone na pręt, a jeśli $o_i = 0$, to jest z niego zdjęte.*

Wyjście

Plik tekstowy `lan.out` *powinien zawierać jedną liczbę całkowitą, równą minimalnej liczbie ruchów potrzebnych do zdjęcia wszystkich ogniw bajtockiego łańcucha z pręta.*

Przykład

Dla pliku wejściowego `lan.in`:

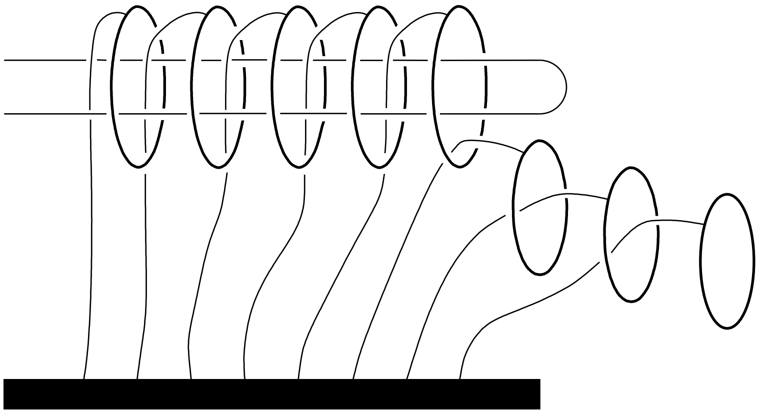
```
4
1 0 1 0
```

poprawną odpowiedzią jest plik wyjściowy `lan.out`:

```
6
```

Analiza problemu

Wielu spośród czytelników zastanawia się zapewne, czy opisany w zadaniu bajtocki łańcuch istnieje w rzeczywistości. Otóż tak, jest to chińska łamigłówka przedstawiona na poniższym rysunku.



Bajtocki łańcuch

Niech $o = (o_1, o_2, \ldots, o_n)$ i $o' = (o'_1, o'_2, \ldots, o'_n)$ będą dowolnymi ciągami (tej samej długości) opisującymi stany bajtockiego łańcucha. Oznaczmy przez $d(o, o')$ minimalną liczbę ruchów potrzebnych do przekształcenia łańcucha opisywanego przez o w łańcuch opisywany przez o'. Zadanie sprowadza się do wyznaczenia $d(o, (0, 0, \ldots, 0))$ dla o będącego ciągiem z pliku wejściowego. Zanim powiemy jak można obliczyć tę wartość, zastanówmy się nad ogólnymi własnościami funkcji d.

Zauważmy, że dla każdego ciągu o mamy $d(o, o) = 0$, gdyż nie trzeba wykonywać żadnych ruchów, aby łańcuch pozostał bez zmian. Oczywiście dla $o \neq o'$ mamy $d(o, o') > 0$. Zauważmy też, że ruchy jakie możemy wykonywać są symetryczne, a więc dla dowolnych o i o' mamy $d(o, o') = d(o', o)$. Ponadto, dla dowolnych o, o' i o'' zachodzi nierówność $d(o, o'') \leq d(o, o') + d(o', o'')$. Jest tak dlatego, że $d(o, o') + d(o', o'')$ jest minimalną liczbą ruchów potrzebnych do przekształcenia łańcucha opisywanego przez o w łańcuch opisywany przez o'', przy czym w trakcie tego przekształcenia, w pewnym momencie łańcuch jest opisywany przez o'. Wszystkie te własności sprawiają, że możemy traktować d jak *odległość* między różnymi *konfiguracjami* łańcucha, przy czym ostatnia z wymienionych własności odpowiada nierówności trójkąta.

Odległość d jest określona dla ciągów dowolnej długości. Zdjęcie lub założenie k-tego ogniwa może wymagać wkładania lub zdejmowania jedynie ogniw $1, 2, \ldots, k$. Położenie ogniw o numerach większych od k nie ma znaczenia dla liczby ruchów potrzebnych do zdjęcia lub założenia k-tego ogniwa. Tak więc, dla dowolnych ciągów o, o' oraz r zachodzi $d(o, o') = d(o \cdot r, o' \cdot r)$[1] Zastanówmy się ile wynosi $d(0^n, 0^{n-1} \cdot 1)$, czyli ile ruchów należy wykonać, aby zdjąć n-te ogniwo, zakładając, że ogniwa 1, 2, ..., $n-1$ są zdjęte i takie mają pozostać. Otóż, żeby zdjąć n-te ogniwo, należy najpierw założyć ogniwo $n-1$ i zdjąć ogniwa 1, 2, ..., $n-2$. Następnie po zdjęciu n-tego ogniwa należy zdjąć również ogniwo $n-1$. Tak więc:

$$\begin{gathered} d(0^n, 0^{n-1} \cdot 1) = \\ d(0^n, 0^{n-2} \cdot 1 \cdot 0) + d(0^{n-2} \cdot 1 \cdot 0, 0^{n-2} \cdot 1 \cdot 1) + d(0^{n-2} \cdot 1 \cdot 1, 0^{n-1} \cdot 1) = \\ d(0^n, 0^{n-2} \cdot 1 \cdot 0) + 1 + d(0^{n-2} \cdot 1 \cdot 1, 0^{n-1} \cdot 1) = \\ d(0^{n-1}, 0^{n-2} \cdot 1) + 1 + d(0^{n-2} \cdot 1, 0^{n-1}) = \\ 2\, d(0^{n-1}, 0^{n-2} \cdot 1) + 1 \end{gathered}$$

Uzyskujemy więc równanie rekurencyjne:

$$d(0^n, 0^{n-1} \cdot 1) = \begin{cases} 2\, d(0^{n-1}, 0^{n-2} \cdot 1) + 1 & \text{dla } n > 1 \\ 1 & \text{dla } n = 1 \end{cases}$$

Jak łatwo sprawdzić, rozwiązanie tego równania to:

$$d(0^n, 0^{n-1} \cdot 1) = 2^n - 1$$

Oznaczmy przez $R(o) = d(o, 0^n)$ oraz $S(o) = d(o, 0^{n-1} \cdot 1)$, gdzie n jest długością ciągu o. Naszym celem jest wyznaczenie wzoru na $R(o)$. Zauważmy, że zachodzą następujące zależności:

$$\begin{gathered} R(o \cdot 0) = R(o) \\ S(o \cdot 1) = R(o) \end{gathered}$$

[1] Symbol $\cdot$ oznacza operację sklejania ciągów. Dla uproszczenia, przez a^n będziemy oznaczać ciąg złożony z n elementów a, $a^n = \underbrace{a \cdot a \cdot \ldots \cdot a}_{n \text{ razy}}$.

Jeżeli ostatnie ogniwo jest założone, to żeby zdjąć wszystkie ogniwa musimy doprowadzić do konfiguracji, w której tylko ostatnie dwa ogniwa są założone, zdjąć ostatnie z nich, a następnie zdjąć przedostatnie:

$$\begin{aligned}R(o\cdot 1) = {} & d(o\cdot 1, 0^{n-1}\cdot 1\cdot 1) + 1 + d(0^{n-1}\cdot 1\cdot 0, 0^{n+1}) = \\ & d(o, 0^{n-1}\cdot 1) + 1 + d(0^{n-1}\cdot 1, 0^n) = \\ & S(o) + 2^n\end{aligned}$$

Podobnie, jeżeli ostatnie ogniwo jest zdjęte, a chcemy doprowadzić do konfiguracji, w której jedynie ostatnie ogniwo jest założone, to musimy doprowadzić do konfiguracji, w której jedynie przedostatnie ogniwo jest założone, założyć ostatnie ogniwo i zdjąć przedostatnie:

$$\begin{aligned}S(o\cdot 0) = {} & d(o\cdot 0, 0^{n-1}\cdot 1\cdot 0) + 1 + d(0^{n-1}\cdot 1\cdot 1, 0^n\cdot 1) = \\ & d(o, 0^{n-1}\cdot 1) + 1 + d(0^{n-1}\cdot 1, 0^n) = \\ & S(o) + 2^n\end{aligned}$$

Uzyskane zależności rekurencyjne mogą być już podstawą rozwiązania zadania. Możemy je jednak jeszcze uprościć. Zauważmy, że $R(o) = S((o_1, o_2, \ldots, 1 - o_n))$. Tak więc:

$$R(o) = \begin{cases} R((o_1, o_2, \ldots, o_{n-1})) & \text{dla } o_n = 0 \\ R((o_1, o_2, \ldots, 1 - o_{n-1})) + 2^{n-1} & \text{dla } o_n = 1 \end{cases}$$

Oznaczmy przez $p_i = (\sum_{j=i}^{n} o_j) \bmod 2$, czyli $p_i = 0$, gdy wśród $o_i, o_{i+1}, \ldots, o_n$ jest parzysta liczba jedynek, w przeciwnym przypadku $p_i = 1$. Wówczas mamy:

$$R(o) = \sum_{i=1}^{n} p_i 2^{i-1}$$

Wygodniej jednak jest zliczać jedynki od początku ciągu, a nie od jego końca. Oznaczmy więc przez $p'_i = \sum_{j=1}^{i} o_j$. Zauważmy, że $p_i = (p'_n - p'_{i-1}) \bmod 2$. Możemy więc wyrazić $R(o)$ następującym wzorem:

$$R(o) = \begin{cases} \sum_{i=1}^{n} p'_{i-1} 2^{i-1} & \texttt{dla } p'_n = 0 \\ \sum_{i=1}^{n} (1 - p'_{i-1}) 2^{i-1} = 2^n - 1 - \sum_{i=1}^{n} p'_{i-1} 2^{i-1} & \texttt{dla } p'_n = 1 \end{cases}$$

Ten wzór będzie podstawą naszego rozwiązania.

Rozwiązanie

Jak widać z analizy problemu, wynik może być bardzo dużą liczbą — może mieć nawet ponad 300 cyfr. Wymaga to zaimplementowania własnej arytmetyki. Przyjmujemy, że zostały zaimplementowane następujące operacje na „dużych liczbach”:

- *przypisz(d, l)* — nadaje dużej liczbie d wartość liczby całkowitej l,
- *dodajjeden(d)* — zwiększa dużą liczbę d o 1,
- *dodaj*(d_1, d_2) — do d_1 dodaje d_2,
- *odejmij*(d_1, d_2) — od d_1 odejmuje d_2,
- *zapiszwynik(p, d)* — zapisuje do pliku p liczbę d.

Wszystkie te operacje można zaimplementować w koszcie (czasowym i pamięciowym) $O(n)$, gdzie n jest liczbą cyfr.

W naszym rozwiązaniu będziemy używać następujących zmiennych. Pliki wejściowy i wyjściowy to odpowiednio *we* i *wy*. Zmienna n to liczba ogniw, o reprezentuje kolejne ogniwa. Zmienna logiczna *dodawac* reprezentuje kolejne wartości p'. Obliczany szereg wyliczamy na zmiennej *wynik*, równocześnie na zmiennej *potega2* obliczamy kolejne potęgi dwójki. Używamy też pomocniczej zmiennej całkowitej i. Oto rozwiązanie:

```
{ Inicjacja zmiennych. }
readln( we, n);
dodawac := false;
przypisz( wynik, 0);
przypisz( potega2, 1);
```

```
{ Sumowanie kolejnych elementów szeregu $\sum_{i=1}^{n} p'_{i-1}2^{i-1}$. }
for i := 1 to n do begin
  if dodawac then
     dodaj( wynik, potega2);
  dodaj( potega2, potega2);
  read( we, o);
  if o = 1 then
     dodawac := not dodawac
end;
{ Obliczenie i zapisanie wyniku. }
if dodawac then begin
   dodajjeden( wynik);
   odejmij( potega2, wynik);
   zapiszwynik( wy, potega2)
end else
   zapiszwynik( wy, wynik);
```

Zauważmy, że jeśli n jest długością danego ciągu, to liczba cyfr w wyniku jest rzędu $O(n)$, a liczba cyfr w największej obliczonej potędze 2 jest rzędu $\Theta(n)$. Tak więc złożoność pamięciowa programu jest rzędu $\Theta(n)$. Program ten wykonuje $\Theta(n)$ operacji, w tym operacje arytmetyczne na dużych liczbach. W rezultacie złożoność czasowa programu jest rzędu $\Theta(n^2)$.

Pełne rozwiązanie można znaleźć na załączonej dyskietce w pliku `lan.pas`.

Testy

Rozwiązania zawodników sprawdzano na 12 testach. Cztery z tych testów badały przypadki brzegowe: test nr 1 to łańcuch złożony z jednego założonego ogniwa, test nr 2 to trzy zdjęte ogniwa, test nr 8 to 100 ogniw, wszystkie założone, test nr 12 to 1 000 ogniw i tylko ostatnie z nich jest założone. Pozostałe testy to różne testy losowe. Wielkości testów przedstawiono w poniższej tabelce.

Nr testu	n	Wynik
1	1	1
2	3	0
3	9	410
4	25	7 568 677
5	28	167 445 844
6	30	390 843 256
7	100	1 111 099 096 870 146 813 528 342 860 237
8	100	845 100 400 152 152 934 331 135 470 250
9	300	liczba 89-cyfrowa
10	500	liczba 151-cyfrowa
11	1 000	liczba 301-cyfrowa
12	1 000	liczba 302-cyfrowa

XII Międzynarodowa Olimpiada Informatyczna, Pekin 2000

XII Międzynarodowa Olimpiada Informatyczna — treści zadań

Krzysztof Diks, Marcin Kubica
Tłumaczenie

Palindrome

Zadanie

Palindrom to sumetryczny napis, tzn. taki napis, który czytany z lewa na prawo i z prawa na lewo jest taki sam. Napisz program, który dla zadanego napisu wyznaczy minimalną liczbę znaków, które należy do niego wstawić, aby stał się palindromem.
Np. napis `Ab3bd` *można zamienić na palindrom (*`dAb3bAd` *lub* `Adb3bdA`*) wstawiając do niego 2 znaki. Nie można jednak tego zrobić wstawiając mniej niż 2 znaki.*

Wejście

Plik wejściowy nazywa się `PALIN.IN`*. Pierwszy wiersz tego pliku zawiera jedną liczbę całkowitą: długość zadanego napisu N, $3 \leq N \leq 5000$. Drugi wiersz zawiera napis długości N. Napis ten składa się z wielkich liter od* `A` *do* `Z`*, małych liter od* `a` *do* `z` *oraz cyfr od* `0` *do* `9`*. Małe i wielkie litery są różnymi znakami.*

Wyjście

Plik wyjściowy nazwa się `PALIN.OUT`*. Pierwszy wiersz tego pliku powinien zawierać jedną liczbę całkowitą równą szukanej minimalnej liczbie wstawianych znaków.*

Przykład

`PALIN.IN:`

```
5
Ab3bd
```

`PALIN.OUT:`

```
2
```

Car Parking

Zadanie

Na centralnym parkingu pod Wielkim Murem znajduje się długi rząd miejsc parkingowych. Jeden z końców tego rzędu uważa się za lewy, a drugi za prawy. Cały rząd jest wypełniony parkującymi samochodami. Każdy samochód jest ustalonej marki, ale może być wiele samochodów tej samej marki. Marki identyfikujemy liczbami całkowitymi. Pracownicy parkingu postanowili ustawić auta w niemalejącej kolejności względem ich marek, od lewego do prawego końca. Do tego celu wykorzystają następującą metodę postępowania. W tak zwanej rundzie, każdy z pracowników może wyjechać jednym samochodem z miejsca jego postoju (pracownicy pracują jednocześnie), a następnie zaparkować na zwolnionym w tej samej rundzie miejscu postojowym. Może się zdarzyć, że w danej rundzie nie wszyscy pracują. Oczywiście, pracownicy chcieliby wykonać swoją pracę w jak najmniejszej liczbie rund.

Niech N będzie liczbą samochodów, a W liczbą pracowników parkingu. Napisz program, który dla danej liczby marek samochodów na parkingu i danej liczby pracowników, znajduje sposób uporządkowania samochodów w co najwyżej $\lceil \frac{N}{W-1} \rceil$ rundach (zaokrąlenie w górę). Minimalna liczba rund nigdy nie jest większa niż $\lceil \frac{N}{W-1} \rceil$.

Rozważmy następujący przykład. Na parkingu znajduje się 10 samochodów o markach 1, 2, 3 i 4, oraz 4 pracowników. Na poczatku kolejność samochodów na parkingu, od lewej do prawej i względem ich marek, jest taka: 2 3 3 4 4 2 1 1 3 1. Minimalna liczba rund wynosi 3, a rundy można wykonać w taki sposób, że rozmieszczenie samochodów po każdej z nich jest następujące:

- *2 1 1 4 4 2 3 3 3 1 — po rundzie 1,*
- *2 1 1 2 4 3 3 3 4 1 — po rundzie 2,*
- *1 1 1 2 2 3 3 3 4 4 — po rundzie 3.*

Wejście

Nazwą pliku wejściowego jest `CAR.IN`*. Pierwszy wiersz pliku wejściowego zawiera trzy liczby całkowite. Pierwszą liczbą jest N — liczba samochodów na parkingu, $2 \leq N \leq 20000$. Drugą liczbą jest liczba marek M, $2 \leq M \leq 50$. Marki są ponumerowane od 1 do M. Na parkingu znajduje sie co najmniej jeden samochód każdej marki. Trzecia liczba jest liczbą pracowników parkingu W, $2 \leq W \leq M$. Drugi wiersz zawiera N liczb całkowitych — i-ta liczba jest marką i-tego samochodu w rzędzie, patrząc od lewego do prawgo końca.*

Wyjście

Nazwą pliku wyjsciowego jest `CAR.OUT`*. Pierwszy wiersz pliku wyjściowego powinien zawierać jedną liczbę całkowitą R — liczbę rund w znalezionym rozwiązaniu. Kolejne R wierszy zawiera opisy kolejnych rund od 1 do R. Każdy taki wiersz rozpoczyna się od liczby całkowitej C — liczby samochodów, które należy ruszyć w tej rundzie. Następnie pojawia się $2C$ liczb całkowitych — identyfikatorów pozycji samochodów. Pozycje samochodów są ponumerowane od lewego do prawego końca, kolejnymi liczbami $1, 2, \ldots N$. Pierwsze dwie liczby tworzą parę opisującą ruch jednego z samochodów: pierwsza liczba jest pozycją tego samochodu (od lewego końca) sprzed opisywanej rundy, a druga liczba jest pozycją tego samochodu (od lewego końca) po tej rundzie. Trzecia i czwarta liczba tworzą parę opisującą ruch innego samochodu, itd. Dla każdego z tych R wierszy może być wiele różnych rozwiązań, ale Twój program powinien wypisać tylko jedno z nich.*

Przykład

`CAR.IN`:

```
10 4 4
2 3 3 4 4 2 1 1 3 1
```

`CAR.OUT`:

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

Częściowa punktacja

Przypuśćmy, że twój program wypisał liczę rund R, a $\left\lceil \frac{N}{W-1} \right\rceil$ jest równe Q. Jeśli twój program wypisuje niepoprawne opisy rund lub w wyniku ich wykonania samochody nie zostaną uporządkowane, nie zdobywasz żadnych punktów. W przeciwnym przypadku otrzymasz liczbę punktów obliczoną w następujący sposób:

- $R \leq Q$ — *100% punktów*
- $R = Q + 1$ — *50% punktów*
- $R = Q + 2$ — *20% punktów*
- $R \geq Q + 3$ — *0% punktów*

Median strength

Zadanie

W nowym eksperymencie kosmicznym wykorzystanych jest N obiektów ponumerowanych od 1 do N. Wiadomo, że N jest liczbą nieparzystą. Każdy obiekt charakteryzuje się inną, ale nieznaną wagą, wyrażaną liczbą naturalną. Dla każdej wag Y mamy $1 \leq Y \leq N$. Obiekt o środkowej wadze to taki obiekt, dla którego mamy tyle samo obiektów od niego cięższych, co lżejszych. Napisz program, który wyznaczy obiekt o środkowej wadze. Niestety jedyny sposób na porównywanie wagi obiektów, to użycie narzędzia, które dla zadanych trzech różnych obiektów wyznacza, który spośród tych trzech obiektów ma środkową wagę.

Biblioteka

Dana jest biblioteka `device` *udostępniająca trzy operacje:*

- `GetN`*, bez argumentów, którą należy wywołać raz na początku, dająca w wyniku wartość N,*
- `Med3`*, mająca 3 argumenty będące numerami trzech różnych obiektów, dająca w wyniku numer obiektu o środkowej wadze,*
- `Answer`*, o jednym argumencie będącym numerem obiektu; wywołując te funkcje przekazuje się numer obiektu o środkowej wadze, funkcja ta kończy (w poprawny sposób) wykonanie programu.*

Biblioteka `device` *tworzy dwa pliki tekstowe:* `MEDIAN.OUT` *i* `MEDIAN.LOG`*. Pierwszy wiersz pliku* `MEDIAN.OUT` *zawiera jedną liczbę całkowitą — numer obiektu, który był argumentem* `Answer`*. Drugi wiersz jedną liczbę całkowitą — liczbę wywołań funkcji* `Med3` *wykonywanych przez Twój program. Komunikacja pomiędzy Twoim programem i biblioteką jest opisana w pliku* `MEDIAN.LOG`*.*

Instrukcja dla programujących w Pascalu: *Wstaw do kodu źródłowego programu następujące polecenie dołączające bibliotekę:*

```
uses device;
```

Instrukcja dla programujących w C/C++: *W kodzie źródłowym Twojego programu użyj polecenia:*

```
#include <device.h>
```

stwórz projekt `MEDIAN.PRJ` *i dodaj do niego pliki* `MEDIAN.C` *(*`MEDIAN.CPP`*) oraz* `DEVICE.OBJ`*.*

Testowanie

Możesz przetestować swój program tworząc plik tekstowy `DEVICE.IN`*. Plik ten musi zawierać dwa wiersze. W pierwszym wierszu musi być zapisana jedna liczba całkowita: liczba obiektów N. Drugi wiersz musi zawierać liczby całkowite od 1 do N podane w pewnej kolejności — i-ta liczba reprezentuje wagę obiektu nr i.*

Przykład

`DEVICE.IN`:

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

Powyższy plik `DEVICE.IN` *opisuje 5 danych obiektów o następujących wagach:*

Numer	*1*	*2*	*3*	*4*	*5*
Waga	*2*	*5*	*4*	*3*	*1*

Oto poprawna sekwencja 5–ciu wywołań biblioteki:

1. `GetN` *(w Pascalu) lub* `GetN()` *(W C/C++), daje w wwyniku 5,*

2. `Med3(1,2,3)`, *daje w wyniki 3,*

3. `Med3(3,4,1)`, *daje w wyniki 4,*

4. `Med3(4,2,5)`, *daje w wyniki 4,*

5. `Answer(4)`

Ograniczenia

- *Liczba obiektów N jest nieparzysta oraz $5 \leq N \leq 1499$.*
- *Numery obiektów i spełniają $1 \leq i \leq N$.*
- *Wagi obiektów Y spełniają $1 \leq Y \leq N$. Wszystkie wagi są różne,*
- *Biblioteka dla Pascala znajduje się w pliku* `device.tpu`*.*
- *Deklaracje funkcji i procedur bibliotecznych dla Pascala:*

```
function GetN : integer;
function Med3 (x, y, z : integer) : integer;
procedure Answer (m : integer);
```

- *Nazwy plików bibliotecznych dla C/C++:* `device.h`*,* `device.obj` *(używaj modelu pomięci large).*
- *Deklaracje funkcji bibliotecznych C/C++:*

```
int GetN (void);
int Med3 (int x, int y, int z);
void Answer (int m);
```

- *W trakcie każdego wykonania Twój program może wywoływać funkcję* `Med3` *co najwyżej 7777 razy.*
- *Twój program nie może odwoływać się (czytać/pisać) do żadnych plikach.*

Post Office

Wzdłuż prostoliniowej autostrady znajduje się wiele miast. Na autostradę można patrzeć jak na oś całkowitoliczbową. Miasta znajdują się w pewnych punktach na tej osi (o całkowitych współrzędnych) i różne miasta znajdują się w różnych punktach Odległóść między miastami definiuje się jako wartość bezwzględną z różnicy ich współrzędnych.

W miastach, niekoniecznie wszystkich, postanowiono wybudować urzędy pocztowe — w każdym mieście co najwyżej jeden. Należy znaleźć dla nich miejsce budowy w taki sposób, żeby zminimalizować sumę odległości miast od ich najbliższych urzędów pocztowych.

Napisz program, który mając dane położenie miast i liczbę planowanych urzędów pocztowych, obliczy najmniejszą możliwą sumę odległości miast od ich najbliższych urzędów pocztowych oraz odpowiadający tej sumie sposób usytuowania tych urzędów.

Wejście

Nazwą pliku wejściowego jest `POST.IN`*. Pierwszy wiersz zawiera dwie liczby całkowite: pierwszą z nich jest liczba miast* V, $1 \leq V \leq 300$, *a drugą liczba urzędów pocztowych* P, $1 \leq P \leq 30$, $P \leq V$. *Drugi wiersz zawiera* V *liczb całkowitych uporządkowanych rosnąco. Te liczby to współrzędne miast. Dla każdej współrzędnej* X *mamy* $1 \leq X \leq 10\,000$.

Wyjście

Nazwą pliku wyjściowego jest `POST.OUT`*. Pierwszy wiersz zawiera jedną liczbę całkowitą S, którą jest suma odległości miast od ich najbliższych urzędów pocztowych. Ich położenia są opisane w następnym wierszu. Drugi wiersz zawiera* P *liczb całkowitych w porządku rosnącym. Te liczby są współrzędnymi (pozycjami) miast, w których należy wybudować urzędy pocztowe. Może być wiele różnych sposobów lokalizacji urzędów pocztowych, ale Twój program musi podać tylko jeden z nich.*

Przykład

`POST.IN:`

```
10 5
1 2 3 6 7 9 11 22 44 50
```

`POST.OUT:`

```
9
2 7 22 44 50
```

Częściowa punktacja

Jeśli wynik działania Twojego programu nie jest zgodny z opisem wyjścia otrzymasz 0 punktów. W przeciwnym razie otrzymasz liczbę punktów obliczaną według poniższej tabelki. Jeśli Twój program wypisuje sumę S, *a najmniejszą możliwą sumą jest* S_{min}, *wówczas liczba punktów jest obliczana jak następuje* ($q = S/S_{min}$):

$q = S/S_{min}$	$q = 1.0$	$1.0 < q \leq 1.1$	$1.1 < q \leq 1.15$	$1.15 < q \leq 1.2$
c	10	5	4	3

$q = S/S_{min}$	$1.2 < q \leq 1.25$	$1.25 < q \leq 1.3$	$1.3 < q$
c	2	1	0

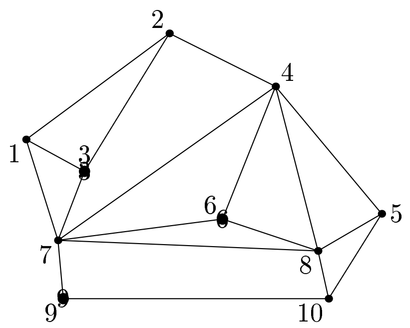
Krzysztof Diks, Marcin Kubica
Tłumaczenie

Walls

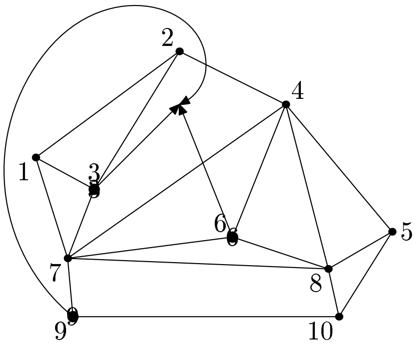
Zadanie

W pewnym kraju, wielkie mury pobudowano w taki sposób, że każdy z nich łączy dokładnie dwa miasta. Wielkie mury nie przecinają się wzajemnie. W ten sposób kraj został podzielony na obszary i żeby przejechać z jednego obszaru na inny, trzeba przedostać się przez mur łączący te obszary. Ponadto można przejść z miasta A do B idąc tylko przez miasta i po murach. **Pewne dodatkowe ograniczenia wynikają z formatu danych wejściowych**.

W opisywanym kraju działa klub podróżnika, którego członkowie żyją w miastach — w każdym mieście co najwyżej jeden członek. Od czasu do czasu członkowie klubu podróżują rowerami i unikają przejazdów przez miasta, ponieważ nie lubią spalin i korków ulicznych. Ponieważ pokonywanie murów z rowerem jest bardzo niewygodne, starają się wybrać tak obszar spotkania, żeby pokonać jak najmniej murów. Aby dostać się na spotkanie każdy z nich musi pokonać pewną liczbę (możliwe, że 0) murów. Chcą więc znaleźć taki obszar, który zminimalizuje sumaryczną liczbę pokonywanych przez nich murów (zwaną w skrócie liczbą pokonywanych murów). Taki obszar nazwiemy obszarem optymalnym.



Rys. 1.



Rys. 2.

Miasta są ponumerowane od 1 do N, gdzie N jest liczbą miast. Na rysunku 1 miasta są przedstawione jako ponumerowane węzły, a odcinki łączące węzły symbolizują mury. Przypuśćmy, że klub podróżnika ma 3 członków, którzy mieszkają w miastach 3,6 i 9. Na rysunku 2 zaznaczono optymalny obszar i drogę podróżników dla danych z rysunku 1. Liczba pokonywanych murów wynosi 2: podróżnik z miasta 9 pokonuje mur łączący miasta 2 i 4, a podróżnik z miasta 6 pokonuje mur łączący miasta 4 i 7.

Napisz program, który dla danych miast, obszarów i miejsca zamieszkania członków klubu, obliczy pewien optymalny obszar i minimalną liczbę murów pokonywanych przez członków klubu.

Wejście

Nazwą pliku wejściowego jest `WALLS.IN`*. Pierwszy wiersz zawiera jedną liczbę całkowitą: liczbę obszarów* M*,* $2 \leq M \leq 200$*. Drugi wiersz zawiera jedną liczbę całkowitą: liczbę miast* N*,* $3 \leq N \leq 250$*. Trzeci wiersz zawiera liczbę członków klubu* L*,* $1 \leq L \leq 30$*,* $L \leq N$*. Czwarty wiersz zawiera* L *różnych liczb całkowitych uporządkowanych rosnąco: numery miast, w których mieszkają członkowie klubu.*

Kolejne $2M$ *wierszy zawiera opisy obszarów — po dwa wiersze na obszar. Pierwsze dwa wiersze opisują pierwszy obszar, następne dwa opisują drugi obszar, itd. W każdej takiej parze wierszy, pierwszy z nich zawiera liczbę całkowitą* I*, równą liczbie miast na granicy opisywanego obszaru. Drugi wiersz zawiera numery* I *miast na granicy opisywanego obszaru, podane kolejno w porządku zgodnym z ruchem wskazówek zegara. Wyjątkiem jest obszar zewnętrzny (nieograniczony) opisywany jako ostatni. Wierzchołki na granicy tego obszaru są podane w kierunku przeciwnym do ruchu wskazówek zegara. Obszary są ponumerowane od 1 do* M*, w kolejności ich opisów w pliku wejściowym. Pamiętaj, że plik wejściowy zawiera opisy wszystkich obszarów, również obszaru zewnętrznego.*

Wyjście

Nazwą pliku wyjściowego jest `WALLS.OUT`*. Pierwszy wiersz zawiera jedną liczbę całkowitą: minimalną liczbę przekraczanych w sumie murów. Drugi wiersz zawiera jedną liczbę całkowitą: numer optymalnego obszaru spotkania. Może istnieć wiele takich obszarów i Twój program powinien wypisać numer tylko jednego z nich.*

Przykład

Podane poniżej pliki, wejściowy i wyjściowy, odpowiadają przykładowi podanemu w tekście:

`WALLS.IN`:

```
10
10
3
3 6 9
3
1 2 3
3
1 3 7
4
2 4 7 3
3
4 6 7
3
4 8 6
3
6 8 7
3
```

ciąg dalszy `WALLS.IN`:

```
4 5 8
4
7 8 10 9
3
5 10 8
7
7 9 10 5 4 2 1
```

`WALLS.OUT`:

```
2
3
```

Building with Blocks

Zadanie

Sześcian jednostkowy (lub krócej sześcian) to sześcian o wymiarach $1 \times 1 \times 1$, którego wierzchołki mają całkowite współrzędne. Dwa sześciany są połączone, jeśli mają wspólną ścianę. Trójwymiarowa bryła (lub krócej bryła), to niepusty zbiór sześcianów połączonych ze sobą (patrz rys. 1) Objętość bryły, to liczba tworzących ją sześcianów. Klocek, to bryła o objętości co najwyżej 4. Danych jest dokładnie 12 kształtów klocków (patrz rys. 2). Kolory klocków na rysunku nie mają żadnego znaczenia — mają jedynie polepszyć czytelność rysunku.

Zbiór klocków D nazywamy rozkładem bryły S, jeśli klocki należące do D tworzą w sumie bryłę S oraz żadne dwa klocki należące do D nie mają wspólnego sześcianu.

Napisz program który na podstawie opisów kształtów klocków oraz bryły S, wyznaczy najmniej liczny zbiór klocków będący rozkładem S. Twój program musi wypisać jedynie kształty klocków występujących w rozkładzie — każdy kształt tyle razy, ile klocków tego kształtu należy do rozkładu.

Wejście

W plikach wejściowych opisujemy sześciany za pomoca trójek liczb całkowitych x, y i z, będących współrzędnymi wierzchołka sześcianu o najmniejszej sumie współrzędnych $x + y + z$.

Kształty klocków są opisywane w pliku `TYPES.IN`*. Treść tego pliku jest taka sama dla wszystkich testów i zawiera ona opisy 12 klocków przedstawionych na rys. 2 (w kolejności ich numeracji). Każdy klocek jest opisany w kolejnych wierszach w pliku. Pierwszy wiersz zestawu zawiera jedną liczbę całkowitą I, reprezentującą numer kształtu klocka ($1 \leq I \leq 12$). Drugi wiersz zawiera objętość V klocka danego kształtu ($1 \leq V \leq 4$). Pozostałe V wierszy zawiera po trzy liczby całkowite x, y, z, reprezentujące sześciany tworzące klocek danego kształtu ($1 \leq x, y, z \leq 4$).*

Bryła S jest opisana w pliku `BLOCK.IN`*. Pierwszy wiersz tego pliku zawiera objętość V bryły S ($1 \leq V \leq 50$). Pozostałe V wierszy zawiera po trzy liczby całkowite x, y, z, reprezentujące sześciany tworzące bryłę ($1 \leq x, y, z \leq 7$).*

Wyjście

Plik wyjściowy nazywa się `BLOCK.OUT`*. Pierwszy wiersz tego pliku powinien zawierać jedną liczbę całkowitą M, będącą minimalną liczbą klocków, na które można rozłożyć zadaną bryłę. Drugi wiersz powinien zawierać M numerów kształtów klocków, na które można daną bryłę rozłożyć. Jeśli możliwych jest wiele odpowiedzi, Twój program powinien wypisać dowolną z nich.*

Przykład

TYPES.IN:

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

ciąg dalszy TYPES.IN

```
1 2 1
1 3 1
1 1 2
1 2 2
10
4
2 1 1
1 2 1
2 2 1
2 1 2
11
4
1 1 1
1 2 1
2 2 1
1 1 2
12
4
2 2 1
2 1 2
1 2 2
2 2 2
```

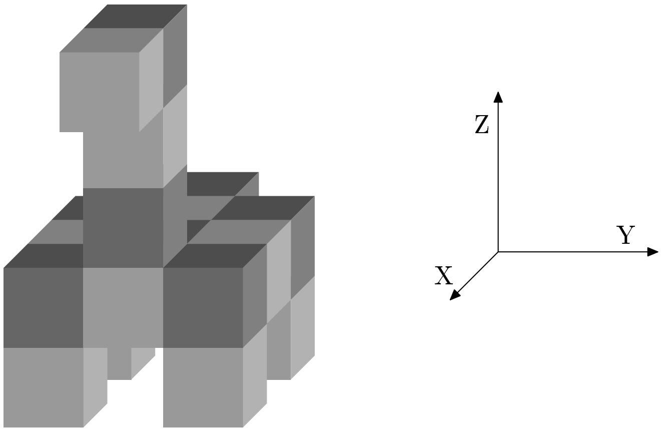
BLOCK.IN:

```
18
2 1 1
4 1 1
2 3 1
4 3 1
2 1 2
3 1 2
4 1 2
1 2 2
2 2 2
3 2 2
4 2 2
2 3 2
3 3 2
4 3 2
4 2 3
4 2 4
4 2 5
5 2 5
```

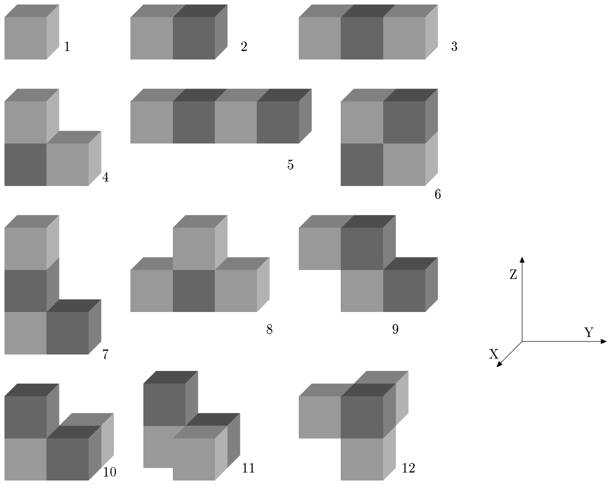
BLOCK.OUT:

```
5
7 10 2 10 12
```

Uwaga *Powyższe pliki opisują bryłę "konia" przedstawioną na rys. 1.*



Rys 1. Koń



Rys 2. 12 rodzajów klocków

XIII Międzynarodowa Olimpiada Informatyczna, Tampere 2001

XIII Międzynarodowa Olimpiada Informatyczna — treści zadań

Depot

Pewna fińska firma dysponuje wielkim prostokątnym magazynem. Boki magazynu są nazwane kolejno (idąc po obwodzie): lewy, górny, prawy i dolny. Wyodrębnienie w magazynie wierszy i kolumn podzieliło go na kwadraty o identycznych wymiarach. Wiersze są ponumerowane od góry liczbami 1, 2, ... Podobnie, kolumny są ponumerowane od lewej liczbami $1, 2, \ldots$.

W magazynie przechowywane są kontenery zawierające bezwartościowe rupiecie. Kontenery mają różne numery identyfikacyjne. Każdy kontener zajmuje jeden kwadrat. Magazyn jest tak duży, że liczba kontenerów, które kiedykolwiek mogą znaleźć się w magazynie, jest mniejsza zarówno od liczby wierszy, jak i liczby kolumn. Kontenery nie są nigdy usuwane z magazynu, ale czasami do magazynu są przysyłane nowe. Wejście do magazynu znajduje się w jego górnym lewym rogu.

Magazynier rozmieścił kontenery w okolicach górnego lewego rogu w taki sposób, że potrafi je odnajdywać na podstawie ich numerów identyfikacyjnych. W tym celu stosuje następującą metodę.

Przypuśćmy, że numerem identyfikacyjnym nowo wstawianego kontenera jest k (w skrócie kontenera k). Magazynier przegląda pierwszy wiersz, rozpoczynając od lewej strony, i szuka pierwszego kontenera o numerze identyfikacyjnym większym od k. Jeśli takiego kontenera nie ma, kontener k jest umieszczany bezpośrednio za ostatnim kontenerem w tym wierszu. Jeśli taki kontener l został znaleziony, to w miejsce kontenera l wstawiany jest kontener k, a kontener l jest umieszczany w następnym wierszu za pomocą tej samej metody. Jeśli magazynier dojdzie do wiersza bez kontenerów, to nowy kontener jest umieszczany w pierwszym (z lewej strony) kwadracie tego wiersza.

Załóżmy, że do magazynu trafiły kontenery 3,4,9,2,5,1 w tej właśnie kolejności. Wówczas rozmieszczenie tych kontenerów w magazynie jest następujące:

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

Do magazynu przyszedł Pan Kierownik i przeprowadził z magazynierem następującą rozmowę:

Kierownik: *Czy kontener 5 przyszedł przed kontenerem 4?*

Magazynier: *Ależ Panie Kierowniku, to jest niemożliwe!*

Kierownik: *Ach, jesteś w stanie określić kolejność napływania kontenerów, znając tylko ich rozmieszczenie w magazynie!*

Magazynier: *Eeee, no tak ogólnie, to nie. Na przykład, kontenery, które aktualnie znajdują się w magazynie mogły przyjść w kolejności 3,2,1,4,9,5, lub w kolejności 3,2,1,9,4,5, lub w jednej z 14 innych kolejności.*

Pan Kierownik nie chciał wyjść na głupka, więc poszedł sobie. Pomóż Panu Kierownikowi i napisz program, który mając dane rozmieszczenie kontenerów w magazynie obliczy wszystkie możliwe kolejności, w których mogły one przybyć do magazynu.

WEJŚCIE

Plik wejściowy nazywa się `depot.in`*. Pierwszy wiersz zawiera jedną liczbę całkowitą* R *— liczbę wierszy zawierających co najmniej po jednym kontenerze. Kolejne* R *wierszy w pliku wejściowym zawiera informacje o wierszach* $1, \ldots, R$*, poczynając od góry. Na początku każdego z tych wierszy jest zapisana liczba całkowita* M *— liczba kontenerów w tym wierszu. Następnie zapisanych jest* M *liczb całkowitych — numery identyfikacyjne kontenerów umieszczonych w tym wierszu, poczynając od lewej. Wszystkie numery identyfikacyjne* I *spełniają nierówności:* $1 \leq I \leq 50$*. Dla liczby kontenerów* N *w magazynie zachodzi* $1 \leq N \leq 13$*.*

WYJŚCIE

Plik wyjściowy nazywa się `depot.out`*. Liczba wierszy w pliku wyjściowym jest równa liczbie możliwych kolejności przybycia kontenerów do magazynu. Każdy z tych wierszy zawiera* N *liczb całkowitych — numery identyfikacyjne kontenerów w potencjalnej kolejności ich przybycia do magazynu. Różne wiersze powinny opisywać różne kolejności.*

PRZYKŁAD

Przykład 1:

depot.in	depot.out
3	3 2 1 4 9 5
3 1 4 5	3 2 1 9 4 5
2 2 9	
1 3	

Przykład 2:

depot.in	depot.out
2	3 1 2
2 1 2	1 3 2
1 3	

PUNKTACJA

Jeśli plik wyjściowy zawiera niepoprawną kolejność, lub nie zawiera żadnych kolejności, to Twój program dostanie 0 punktów za taki test. W przeciwnym razie punkty za test są przyznawane w następujący sposób. Jeśli plik wyjściowy zawiera wszystkie możliwe kolejności bez powtórzeń, to Twój program dostanie 4 punkty. Jeśli plik wyjściowy zawiera co najmniej połowę możliwych kolejności i każdą z nich dokładnie raz, to Twój program dostanie 2 punkty. Jeśli plik wyjściowy zawiera mniej niż połowę możliwych kolejności, lub niektóre z nich się powtarzają, Twój program dostanie 1 punkt.

Krzysztof Diks, Marcin Kubica
Tłumaczenie

Double Crypt

Rozwinięty Standard Szyfrowania (z angielskiego AES) korzysta z nowego algorytmu szyfrowania. Używa się w nim trzech bloków, każdy złożony ze 128 bitów. Dla danego bloku komunikatu p (tekst jawny) i bloku klucza k, funkcja szyfrująca E w standardzie AES oblicza zaszyfrowany blok c (tekst zaszyfrowany):

$$c = E(p, k).$$

Funkcją odwrotną dla funkcji szyfrującej E jest funkcja deszyfrująca D taka, że:

$$D(E(p, k), k) = p, \; E(D(c, k), k) = c.$$

W Podwójnym Szyfrowaniu AES używa się kolejno dwóch niezależnych bloków kluczy k1 i k2:

$$c_2 = E(E(p, k_1), k_2)$$

.

W tym zadaniu dana jest także liczba całkowita s. We wszystkich kluczach tylko $4 \cdot s$ skrajnie lewych bitów ma znaczenie, podczas gdy wszystkie pozostałe bity (skrajnie prawe 128 — $4 \cdot s$) są równe 0.

Twoim zadaniem jest odtworzenie pary kluczy szyfrujących dla pewnych komunikatów zaszyfrowanych za pomocą Podwójnego AES. Znasz zarówno tekst jawny p, odpowiadający mu podwójnie zaszyfrowany tekst c2, jak i strukturę kluczy szyfrujących zadaną przez liczbę całkowitą s.

Algorytmy szyfrowania i deszyfrowania w systemie AES są dostępne w bibliotece. Do oceny przedkładasz odtworzone klucze, a nie program do ich odtwarzania.

WEJŚCIE

Zestawy danych dla tego zadania są zapisane w plikach tekstowych o nazwach double1.in — double10.in. Każdy plik wejściowy składa się z trzech wierszy. Pierwszy wiersz zawiera liczbę całkowita s, w drugim wierszu znajduje się blok jawnego tekstu p, natomiast trzeci wiersz zawiera blok tekstu zaszyfrowanego c2 otrzymany za pomocą Podwójnego Szyfrowania AES. Oba bloki są zapisane w postaci 32-znakowych słów szesnastkowych (o cyfrach `0,..., 9, A, ..., F`*). Biblioteka zawiera procedurę konwersji słów na bloki. Dla każdego zestawu danych wejściowych istnieje rozwiązanie.*

WYJŚCIE

Do oceny przedstawiasz 10 plików wyjściowych odpowiadających danym plikom wejściowym. Każdy plik wyjściowy składa się z trzech wierszy. Pierwszy wiersz zawiera tekst:

`#FILE double` *I*

gdzie I jest numerem odpowiedniego pliku wejściowego. Drugi wiersz zawiera blok klucza k1, a trzeci wiersz blok klucza k2, takie że: c2 = E(E(p, k1), k2). Oba bloki muszą być zapisane jako 32-znakowe słowa szesnastkowe (o cyfrach `0,..., 9, A, ..., F`*). Biblioteka zawiera procedurę do konwersji bloków na słowa. Jeśli istnieje wiele rozwiązań wystarczy, że przedstawisz tylko jedno z nich.*

PRZYKŁAD

Plik przedstawiony w tym przykładzie ma numer 0.

`double0.in`	*Możliwy plik wyjściowy*
`1`	`#FILE double 0`
`00112233445566778899AABBCCDDEEFF`	`A0000000000000000000000000000000`
`6323B4A5BC16C479ED6D94F5B58FF0C2`	`70000000000000000000000000000000`

BIBLIOTEKA

Biblioteka dla FreePascala (Linux: `aeslibp.p`*,* `aeslibp.ppu`*,* `aeslibp.o`*; Windows:* `aeslibp.p`*,* `aeslibp.ppw`*,* `aeslibp.ow*`*):*

```
type

  HexStr = String [ 32 ]; { only '0'..'9', 'A'..'F' }

  Block  = array [ 0..15 ] of Byte; { 128 bits }


procedure HexStrToBlock ( const hs: HexStr; var b: Block );

procedure BlockToHexStr ( const b: Block; var hs: HexStr );

procedure Encrypt ( const p, k: Block; var c: Block );

  { c = E(p,k) }

procedure Decrypt ( const c, k: Block; var p: Block );

  { p = D(c,k) }
```

Do dyspozycji masz program `aestoolp.pas`*, który pokazuje jak korzystać z takiej biblioteki.*
Biblioteka dla GNU C/C++ (Linux i Windows: `aeslibc.h`*,* `aeslibc.o*`*):*

```
typedef char HexStr[33]; /* '0'..'9', 'A'..'F', '\0'-terminated */

typedef unsigned char Block[16];  /* 128 bits */


void hexstr2block ( const HexStr hs, /* out-param */ Block b );

void block2hexstr ( const Block b, /* out-param */ HexStr hs );

void encrypt ( const Block p, const Block k, /* out-param */ Block c );

  /* c = E(p,k) */

void decrypt ( const Block c, const Block k, /* out-param */ Block p );

  /* p = D(c,k) */
```

Program `aestoolc.c` *pokazuje jak korzystać z tej biblioteki.*

OGRANICZENIA

Liczba określająca liczbę istotnych cyfr szesnastkowych w kluczu spełnia nierówność $1 \leq s \leq 5$*.*
Wskazówka: *Dobry program potrafi dla każdego dozwolonego pliku wejściowego odtworzyć klucze w czasie mniejszym niż 10s.*

Ioiwari Game

Rodzina gier Mankala jest znana ludzkości od dawna. W tym zadaniu rozważamy grę specjalnie wymyślona na potrzeby IOI. W grze bierze udział dwóch graczy. Do dyspozycji mają okrągłą planszę z siedmioma dołkami na obwodzie, a dodatkowo każdy z graczy ma swój dołek–bank. Gra rozpoczyna się od losowego rozłożenia 20 kamieni w dołkach na planszy w taki sposób, że w każdym z nich znajdują się co najmniej 2 i co najwyżej 4 kamienie. Gracze wykonują ruchy na przemian. Ruch gracza polega na wyborze niepustego dołka na planszy i wzięciu z niego wszystkich kamieni do ręki. Mając kamienie w ręku, gracz bierze pod uwagę kolejne dołki (w kierunku ruchu wskazówek zegara), zaczynając od następnego po opróżnionym, i wykonuje następujące czynności:

Więcej niż jeden kamień w ręce: Jeżeli w rozważanym dołku jest 5 kamieni, to bierze jeden kamień z dołka i przekłada go do swojego banku, w przeciwnym przypadku, odkłada jeden kamień z ręki do rozważanego dołka.

Jeden kamień w ręce: Jeżeli w rozważanym dołku jest co najmniej jeden, a co najwyżej 4 kamienie, to przekłada wszystkie kamienie z tego dołka oraz jeden z ręki do swojego banku, w przeciwnym przypadku (gdy w dołku jest 0 lub 5 kamieni) odkłada jedyny kamień z ręki do banku przeciwnika.

Gra się kończy, gdy wszystkie dołki na planszy są puste, a zwycięża ten gracz, który ma więcej kamieni w swoim banku.

Gracz, który rozpoczyna ma zawsze strategię wygrywającą. Twoim zadaniem jest napisanie programu, który gra w Ioiwari jako gracz rozpoczynający i wygrywa. Przeciwnik, program testujący, gra optymalnie, tzn. jeśli tylko dać mu szansę, to wygra, a Twój program przegra.

WEJŚCIE/WYJŚCIE

Twój program powinien czytać ze standardowego wejścia i pisać na standardowe wyjście. Twój program to gracz nr 1, a jego przeciwnik to gracz nr 2. Po rozpoczęciu, Twój program musi najpierw wczytać siedem liczb całkowitych $p1, \ldots p7$, *zapisanych w jednym wierszu: początkowe liczby kamieni odpowiednio w dołkach* $1, \ldots, 7$ *na planszy. Dołki na planszy są ponumerowane od 1 do 7, zgodnie z ruchem wskazówek zegara. Na samym początku gry banki graczy są puste. Twój program powinien grać jak następuje:*

Jeśli jest to ruch Twojego programu, to powinien on wypisać na standardowe wyjście numer dołka opisujący ruch.

Jeśli jest to ruch przeciwnika, to Twój program powinien wczytać ze standardowego wejścia numer (opróżnianego) dołka określający ten ruch.

NARZĘDZIA

Masz do dyspozycji program (ioiwari2 pod Linuxem, ioiwari2.exe pod Windowsami), który dla jednego początkowego rozmieszczenia kamieni gra optymalnie jako gracz nr 2. Program ten wypisuje na standardowe wyjście pierwszy wiersz, jaki powinien wczytać Twój program, opisujący początkowe rozmieszczenie kamieni w dołkach na planszy: `4 3 2 4 2 3 2`. *Następnie program ten gra, próbując czytać ze standardowego wejścia ruchy gracza nr 1 i wypisując swoje ruchy na standardowe wyjście. Możesz uruchomić swój program i ioiwari2 w oddzielnych okienkach i ręcznie wymieniać dane pomiędzy programami. Przebieg dialogu jest zapisywany w pliku ioiwari.out.*

WYTYCZNE PROGRAMISTYCZNE

W poniższych przykładach, ostatnia liczba całkowita z wejścia jest wczytywana na zmienną `last`, *a zmienna* `mymove` *zawiera Twój ruch.*

Jeśli programujesz w C++ i używasz iostreams, powinieneś stosować następującą konwencję wczytywania ze standardowego wejścia i wypisywania na standardowe wyjście:

```
cout<<mymove<<endl<<flush;
cin>>last;
```

Jeśli programujesz w C lub C++ i używasz scanf i printf, powinieneś stosować następującą konwencję wczytywania ze standardowego wejścia i wypisywania na standardowe wyjście:

```
printf("%d\n",mymove); fflush (stdout);
scanf ("%d", &last);
```

Jeśli programujesz w Pascalu, powinieneś stosować następującą konwencję wczytywania ze standardowego wejścia i wypisywania na standardowe wyjście:

```
Writeln(mymove);
Readln(last);
```

PRZYKŁAD

Oto poprawny ciąg 6 ruchów:

	Zawartości dołków i banków po wykonaniu ruchu								
Operacja/nr dołka	*1.*	*2.*	*3.*	*4.*	*5.*	*6.*	*7.*	*Bank1*	*Bank2*
Sytuacja początkowa	*4*	*3*	*2*	*4*	*2*	*3*	*2*	*0*	*0*
Ruch gracza nr 1: 2	*4*	*0*	*3*	*5*	*0*	*3*	*2*	*3*	*0*
Ruch gracza nr 2: 3	*4*	*0*	*0*	*4*	*1*	*4*	*0*	*3*	*4*
Ruch gracza nr 1: 5	*4*	*0*	*0*	*4*	*0*	*0*	*0*	*8*	*4*
Ruch gracza nr 2: 4	*0*	*0*	*0*	*0*	*1*	*1*	*1*	*8*	*9*
Ruch gracza nr 1: 5	*0*	*0*	*0*	*0*	*0*	*0*	*1*	*10*	*9*
Ruch gracza nr 2: 7	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*11*	*9*

PUNKTACJA

Za zwycięstwo w jednym teście Twój program otrzyma 4 punkty, za remis 2 punkty, a w pozostałych sytuacjach 0 punktów.

Mobile phones

Czwarta generacja stacji przekaźnikowych telefonii komórkowej w regionie Tampere działa w następujący sposób. Cały region jest podzielony na kwadraty. Kwadraty tworzą macierz rozmiaru $S \times S$, w której wiersze i kolumny są ponumerowane od 0 do $S-1$. W każdym kwadracie znajduje się jedna stacja. Liczba aktywnych telefonów wewnątrz każdego kwadratu może się zmieniać, ponieważ telefony mogą przemieszczać się pomiędzy kwadratami oraz być wyłączane lub włączane. Od czasu do czasu, każda stacja przesyła raport o zmianie liczby aktywnych telefonów w jej obszarze wraz z informacją o jej położeniu (wiersz–kolumna w macierzy).

Napisz program, który dostaje te raporty i odpowiada na pytania o aktualną liczbę wszystkich aktywnych telefonów w zadanym prostokątnym obszarze.

WEJŚCIE/WYJŚCIE

Dane wejściowe mają postać liczb całkowitych i są czytane ze standardowego wejścia. Odpowiedzi są wypisywane na standardowe wyjście, również jako liczby całkowite. Format danych wejściowych jest następujący. Każde pojedyncze dane zajmują jeden wiersz i składają się z numeru instrukcji, po którym następują jej parametry (liczby całkowite), zgodnie z następującą tabelką:

Instrukcja	*Parametry*	*Znaczenie*
0	S	*Ustalenie rozmiarów macierzy na $S \times S$ i wypełnienie jej zerami. Instrukcja ta pojawia się tylko raz i zawsze jako pierwsza.*
1	$X\ Y\ A$	*Dodanie A do liczby aktywnych telefonów w kwadracie (X, Y) macierzy. Wartość A może być dodatnia lub ujemna.*
2	$L\ B\ R\ T$	*Zapytanie o aktualną, łączną liczbę aktywnych telefonów w kwadratach (X, Y), dla $L \leq X \leq R, B \leq Y \leq T$.*
3		*Zakończenie programu. Instrukcja ta pojawia się tylko raz, jako ostatnia.*

Wszystkie dane wejściowe mieszczą się w podanych zakresach i nie trzeba tego sprawdzać. W szczególności, jeśli A jest ujemne, to możesz założyć, ze liczba telefonów w kwadracie nie spadnie poniżej 0. Numeracja wierszy i kolumn w macierzy zaczyna się od 0, np. dla macierzy 4×4, mamy $0 \leq X \leq 3$ i $0 \leq Y \leq 3$.

Twój program powinien odpowiadać wyłącznie na instrukcje nr 2. W tym przypadku, powinien on wypisać na standardowe wyjście jeden wiersz zawierający jedną liczbę całkowitą — odpowiedź na zapytanie.

WYTYCZNE PROGRAMISTYCZNE

W poniższych przykładach zmienna całkowita `last` *reprezentuje ostatnią liczbę całkowitą wczytaną z wiersza, a* `answer` *jest zmienną całkowitą zawierającą odpowiedź.*

Jeśli piszesz w C++ i używasz iostreams, powinieneś stosować następującą konwencję wczytywania ze standardowego wejścia i wypisywania na standardowe wyjście:

```
cin>>last;
cout<<answer<<endl<<flush;
```

Jeśli piszesz w C lub C++ i stosujesz `scanf` *i* `printf`*, powinieneś stosować następującą konwencję wczytywania ze standardowego wejścia i wypisywania na standardowe wyjście:*

```
scanf ("%d", &last);
printf("%d\n",answer); fflush (stdout);
```

Jeśli piszesz w Pascalu, powinieneś stosować następującą konwencję wczytywania ze standardowego wejścia i wypisywania na standardowe wyjście:

```
Read(last); ... Readln;
Writeln(answer);
```

PRZYKŁAD

stdin	*stdout*	*opis*
0 4		*Ustalenie rozmiaru macierzy na* 4×4.
1 1 2 3		*Do kwadratu (1,2) dodaj +3.*
2 0 0 2 2		*Zapytanie dotyczące prostokąta* $0 \leq X \leq 2$, $0 \leq Y \leq 2$.
	3	*Odpowiedź na zapytanie.*
1 1 1 2		*Do kwadratu (1, 1) dodaj +2.*
1 1 2 -1		*Do kwadratu (1, 2) dodaj -1.*
2 1 1 2 3		*Zapytanie dotyczące prostokąta* $1 \leq X \leq 2$, $1 \leq Y \leq 3$.
	4	*Odpowiedź na zapytanie.*
3		*Koniec.*

OGRANICZENIA

Wymiary macierzy	$S \times S$	$1 \times 1 \leq S \times S \leq 1024 \times 1024$
Wartość w kwadracie V w dowolnym momencie	V	$0 \leq V \leq 2^{15} - 1 (= 32767)$
Wielkość zmiany	A	$-2^{15} \leq A \leq 2^{15} - 1 (= 32767)$
Liczba instrukcji na wejściu	U	$3 \leq U \leq 60002$
Maksymalna łączna liczba telefonów w macierzy	M	$M = 2^3 0$

Spośród 20 zestawów danych, w 16 rozmiar macierzy nie przekracza 512×512.

UWAGA: *Program umożliwiający testowanie rozwiązań przez sieć podaje testowanemu programowi na standardowe wejście zawartość Twojego pliku testowego.*

Score

Score (wynik) jest grą dwuosobową, w której gracze przesuwają ten sam jeden pionek po planszy. Plansza do gry składa się z N pozycji ponumerowanych od 1 do N i strzałek. Każda strzałka prowadzi od jednej pozycji do innej. Każda pozycja należy do dokładnie jednego z graczy, którego nazywamy właścicielem tej pozycji. Dodatkowo każdej pozycji jest przypisana dodatnia wartość. Wszystkie wartości są różne. Pozycja 1 jest pozycją startową. Początkowo wynik (ang. score) każdego z graczy wynosi 0.

Reguły gry są następujące. Oznaczmy przez C pozycję pionka na planszy przed wykonaniem ruchu. Na początku gry C jest pozycją startową (nr 1). Ruch w grze polega na wykonaniu następujących czynności:

(1) *Jeśli wartość C jest większa niż aktualny wynik właściciela tej pozycji, to jego wynik wzrasta do wartości C. W przeciwnym przypadku wynik właściciela nie zmienia się. Wynik drugiego z graczy nie ulega zmianie.*

(2) *Następnie właściciel C wybiera jedną ze strzałek wychodzących z aktualnej pozycji pionka i przesuwa go na pozycje, do której prowadzi strzałka. Nowa pozycją staje się aktualną pozycją pionka. Zwróć uwagę, że ten sam gracz może wykonać kilka kolejnych ruchów z rzędu.*

Gra kończy się, gdy pionek powraca na pozycję startową. Zwycięzcą gry jest gracz z większym końcowym wynikiem. Strzałki są zawsze poprowadzone w następujący sposób:

- *Z każdej aktualnej pozycji pionka zawsze wychodzi co najmniej jedna strzałka.*
- *Każda pozycja P jest osiągalna z pozycji startowej, tzn. istnieje ciąg strzałek prowadzący z pozycji startowej do P.*
- *Masz gwarancję, że gra zawsze się kończy w skończonej liczbie ruchów.*

Napisz program, który gra w Score i wygrywa. W każdej grze, w której będzie uczestniczył Twój program podczas testowania, ma on możliwość zwyciężyć, niezależnie od tego czy akurat rozpoczął grę, czy nie. Przeciwnik zawsze gra optymalnie, tzn. jeśli dasz mu tylko szansę, to wygra, a Twój program przegra.

WEJŚCIE/WYJŚCIE

Twój program wczytuje dane ze standardowego wejścia i wypisuje wyniki na standardowe wyjście. Twój program to gracz nr 1, natomiast przeciwnik to gracz nr 2. Po uruchomieniu Twój program powinien najpierw przeczytać następujące dane ze standardowego wejścia.

Pierwszy wiersz zawiera jedną liczbę całkowita — liczbę pozycji N, $1 \leq N \leq 1\,000$. Pozycje są ponumerowane od 1 do N. Każdy z kolejnych N wierszy zawiera N liczb całkowitych opisujących strzałki. Jeśli istnieje strzałka prowadząca od pozycji i do pozycji j, to j–tą liczbą w i–tym z tych wierszy jest 1, w przeciwnym przypadku 0.

Kolejny wiersz zawiera N liczb całkowitych opisujących właścicieli poszczególnych pozycji. Jeśli właścicielem pozycji i jest gracz nr 1 (Twój program), to i–ta z tych liczb jest równa 1, w przeciwnym przypadku i–ta liczba jest równa 2.

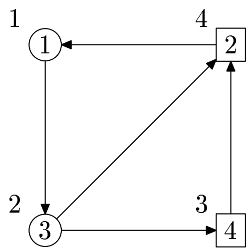
Kolejny wiersz zawiera N liczb całkowitych — wartości poszczególnych pozycji. Jeśli i–ta z tych liczb jest równa j, to wartością pozycji i jest j. Wszystkie wartości j są różne i spełniają nierówności $1 \leq j \leq N$.

Następnie rozpoczyna się gra.

Pozycją początkowa pionka jest pozycja nr 1. Twój program powinien grać następująco i zakończyć działanie po powrocie pionka na pozycję nr 1:

- *Jeśli jest to ruch Twojego programu, to Twój program powinien wypisać na standardowe wyjście numer następnej pozycji P, $1 \leq P \leq N$.*
- *Jeśli jest to ruch przeciwnika, to Twój program powinien wczytać ze standardowego wejścia numer następnej pozyji P, $1 \leq P \leq N$.*

Rozważmy następujący przykład. Plansza do gry jest przedstawiona na rysunku 1. Pozycje zaznacznone kółkami należą do gracza nr 1, a pozycje zaznacznone kwadratami należą do gracza nr 2. Wewnątrz każdej pozycji znajduje się jej wartość, a numer pozycji jest zapisany na zewnątrz. Poniżej przedstawiono rozgrywkę.



Rys. 1

stdin	*stdout*	*objaśnienie*
4		*N*
0 1 0 0		*Opis strzałek wychodzących z pozycji 1*
0 0 1 1		*Opis strzałek wychodzących z pozycji 2*
0 0 0 1		*Opis strzałek wychodzących z pozycji 3*
1 0 0 0		*Opis strzałek wychodzących z pozycji 4*
1 1 2 2		*Właściciele pozycji*
1 3 4 2		*Wartości pozycji*
	2	*Ruch gracza 1*
	4	*Ruch gracza 1*
1		*Gracz 2 wykonuje ruch na pozycję startową — gra się kończy.*

Po zakończeniu gry wynikiem Gracza 1 jest 3, natomiast Gracza 2 — 2. Gracz 1 wygrywa.

WYTYCZNE PROGRAMISTYCZNE

W przykładach poniżej `target` *jest całkowitoliczbową zmienną oznaczającą pozycję.*

Jeśli programujesz w C++ i korzystasz z iostreams, powinieneś stosować następującą konwencję czytania ze standardowego wejścia i pisania na standardowe wyjście:

```
cin>>target;
cout<<target<<endl<<flush;
```

Jeśli programujesz w C lub C++ i korzystasz ze `scanf` *i* `printf`*, powinieneś stosować następującą konwencję czytania ze standardowego wejścia i pisania na standardowe wyjście:*

```
scanf ("%d", &target);
printf("%d\n",target); fflush (stdout);
```

Jeśli programujesz w Pascalu, powinieneś stosować następującą konwencję czytania ze standardowego wejścia i pisania na standardowe wyjście:

```
Readln(target);
Writeln(target);
```

NARZĘDZIA

Do dyspozycji masz program (`score2` *w Linuxie,* `score2.exe` *w Windowsach), który wczytuje opis planszy do gry z pliku score.in zapisany w formacie przedstawionym na poprzedniej stronie. Program ten wypisze powyższą informację w tym samym formacie na standardowe wyjście. To wyjście może być wykorzystane jako wejście dla Twojego programu, do celów testowych. Następnie program* `score2` *gra używając strategii losowej — wczytuje ruchy Twojego programu ze standardowego wejścia i zapisuje swoje ruchy na standardowe wyjście.*

SPOSÓB OCENY

Dla jednego zestawu danych Twój program zdobędzie komplet punktów, jeśli wygra grę, a nie zdobędzie żadnych punktów, jeśli ją przegra. W trakcie oceny przez sprawdzaczkę Twój program jest uruchamiany dwukrotnie. Ograniczenie czasowe dla pierwszego uruchomienia jest o 1s większe od ograniczenia czasu podanego dla tego zadania. Ruchy (wejścia i wyjścia Twojego programu) wykonywane w trakcie gry są zapamiętywane. Następnie Twój program jest wykonywany dla danych wczytywanych z pliku i mierzony jest oficjalnie jego czas działania. Twój program MUSI wykonywać dokładnie TAKIE SAME RUCHY, jak podczas pierwszego wykonania (tzn. musi być DETERMINISTYCZNY).

Twofive

Tajne komunikaty pomiędzy Świętym Mikołajem i jego małymi pomocnikami szyfruje się zwykle w języku J-25. Alfabet tego języka jest taki sam jak alfabet angielski, z jednym wyjątkiem: nie ma w nim litery "Z", tzn. alfabet ten zawiera 25 liter od "A" do "Y", w takiej samej kolejności, co alfabet angielski. Każde słowo w języku J–25 składa się z dokładnie 25 różnych liter. Takie słowo można zapisać w tablicy 5 × 5, wpisując je w kolejne wiersze; np. słowo `ADJPTBEKQUCGLRVFINSWHMOXY` *zostanie zapisane następująco:*

```
A   D   J   P   T
B   E   K   Q   U
C   G   L   R   V
F   I   N   S   W
H   M   O   X   Y.
```

Po wpisaniu poprawnego słowa z języka J-25 do tablicy, litery we wszystkich wierszach i kolumnach będą uporządkowane rosnąco. Tak więc słowo `ADJPTBEKQUCGLRVFINSWHMOXY` *jest poprawne, a* `ADJPTBEGQUCKLRVFINSWHMOXY` *nie (porządek rosnący nie jest zachowany w drugiej i trzeciej kolumnie).*

Święty Mikołaj ma słownik, który jest listą wszystkich poprawnych słów języka J-25, ułożonych w porządku rosnącym (leksykograficznie) wraz z ich numerami porządkowymi zaczynającymi się od 1. Np. w tym słowniku `ABCDEFGHIJKLMNOPQRSTUVWXY` *jest słowem nr 1, zaś* `ABCDEFGHIJKLMNOPQRSUTVWXY` *jest słowem nr 2. W słowie nr 2, w porównaniu ze słowem nr 1, litery "U" i "T" są przestawione.*

Niestety słownik ten jest bardzo duży. Napisz program, który wyznacza numer porządkowy dowolnego zadanego słowa, a także słowo odpowiadające zadanemu numerowi. W słowniku nie ma więcej niż 2^{31} słów.

WEJŚCIE

Plik wejściowy ma nazwę `twofive.in` *i składa się z dwóch wierszy. Pierwszy wiersz zawiera jednoliterowy napis: "W" lub "N". Jeżeli jest to "W", to drugi wiersz zawiera poprawne słowo języka J-25, tzn. 25-znakowy napis. Jeżeli pierwszy wiersz zawiera "N", to drugi wiersz zawiera numer porządkowy poprawnego słowa z języka J-25.*

WYJŚCIE

Plik wyjściowy ma nazwę `twofive.out` *i składa się z jednego wiersza. Jeśli drugi wiersz pliku wejściowego zawiera słowo 25-literowe, to plik wyjściowy zawiera numer porządkowy tego słowa. Jeśli drugi wiersz pliku wejściowego zawiera liczbę, to plik wyjściowy zawiera 25-literowe słowo o tym numerze porządkowym.*

PRZYKŁAD

twofive.in	twofive.out
W ABCDEFGHIJKLMNOPQRSUTVWXY	2

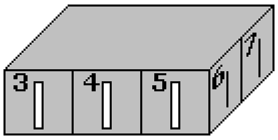
twofive.in	twofive.out
N 2	ABCDEFGHIJKLMNOPQRSUTVWXY

VII Bałtycka Olimpiada Informatyczna, Sopot 2001

VII Bałtycka Olimpiada Informatyczna — treści zadań

Box of Mirrors

Profesor Andrus uwielbia rozwiązywać przeróżne łamigłówki. Jedną z jego ulubionych jest "Lustrzana Skrzynka". Konstrukcję skrzynki najłatwiej opisać patrząc na nią z góry. Załóżmy więc, że widzimy poziomy przekrój skrzynki narysowany w prostokątnym układzie współrzędnych. Jest to prostokąt o bokach równoległych do osi układu podzielony na $n \times m$ kwadratowych pól (ułożonych w n wierszy i m kolumn). W każdym polu może być umieszczone lustro. Lustro jest ustawione pionowo po przekątnej pola biegnącej od lewego–dolnego do prawego–górnego narożnika (przekroju) pola. Obie strony lustra odbijają światło.



W zewnętrznych ścianach skrzynki, pośrodku każdego wiersza i każdej kolumny, znajdują się otwory, przez które może wpadać do wnętrza lub wychodzić na zewnątrz skrzynki wiązka światła. Przez każdy otwór można wpuścić do wnętrza skrzynki wiązkę światła jedynie w kierunku prostopadłym do ściany, w której znajduje się otwór. Taka wiązka odbijając się od lustra zmienia kierunek o 90 stopni. Gdy wiązka przechodzi przez puste pole (takie, na którym nie ma lustra), wówczas jej kierunek nie ulega zmianie. Otwory w ścianach skrzynki są ponumerowane od 1 do $2 \cdot (n + m)$. Numery są nadawane otworom zgodnie z kolejnością ich występowania na obwodzie skrzynki, począwszy od otworu w lewej ścianie górnego–lewego pola (na przekroju) i następnie w kierunku przeciwnym do ruchu wskazówek zegara (czyli idąc najpierw w dół lewej ściany). Ponieważ z zewnątrz nie widać układu luster, więc jedynym sposobem, by wywnioskować, jaki jest ten układ, jest wpuszczanie wiązek światła przez wybrane otwory i obserwowanie, przez które otwory takie wiązki wychodzą.

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `box.in` *rozmiar skrzynki i numery otworów, przez które wpadają i wychodzą wiązki światła,*
- *obliczy, na których polach znajdują się lustra, a które pola są puste,*
- *zapisze wynik w pliku tekstowym* `box.out`.

Jeżeli istnieje więcej niż jedno rozwiązanie, to program powinien podać dowolne z nich.

Wejście

W pierwszym wierszu pliku `box.in` *znajdują się dwie liczby naturalne: n (liczba wierszy pól, $1 \leq n \leq 100$) oraz m (liczba kolumn pól, $1 \leq m \leq 100$) oddzielone pojedynczym odstępem. Każdy z kolejnych $2 \cdot (n + m)$ wierszy zawiera po jednej liczbie naturalnej. Liczba w $(i + 1)$-szym wierszu oznacza numer otworu, przez który wyjdzie wiązka światła, która wpada do skrzynki przez otwór o numerze i.*

Wyjście

Twój program powinien zapisać w pliku wynikowym `box.out` *n wierszy, z których każdy powinien zawierać m liczb oddzielonych pojedynczymi odstępami. Liczba j-ta w i-ym wierszu powinna być równa* `1`*, jeżeli na polu w i-tym wierszu i j-tej kolumnie znajduje się lustro; w przeciwnym razie (gdy pole to jest puste) liczba ta powinna być równa* `0`.

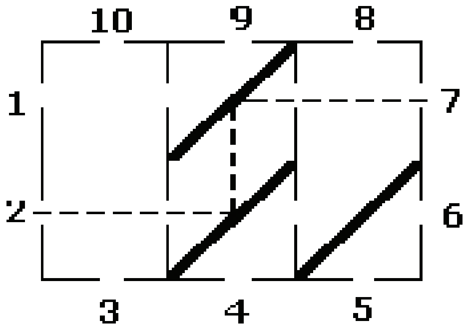
Przykład

Dla pliku wejściowego `box.in`*:*

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

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

```
0 1 0
0 1 1
```



Crack the Code

Kryptografia zajmuje się kodowaniem informacji w taki sposób, że tylko uprawniony odbiorca jest w stanie odczytać zakodowany tekst. Z kolei kryptoanaliza zajmuje się łamaniem kodów.

Załóż, że jesteś właśnie kryptoanalitykiem, a Twoim zadaniem jest odczytanie kilku zaszyfrowanych wiadomości przechwyconych przez policję w lokalu mafii.

Twoi koledzy już uzyskali program szyfrujący używany przez mafię. Jego tekst znajduje się w plikach `crack.pas` *i* `crack.c`*. To co zostało do zrobienia, to odwrócić algorytm szyfrujący i odgadnąć klucze użyte do zakodowania danych.*

Wraz z zaszyfrowanymi wiadomościami masz dostęp do kilku próbek tekstu niezaszyfrowanego, pochodzącego z tego samego źródła, co zakodowane wiadomości i — jak można przypuszczać — mającego podobną strukturę co do użytego języka, zasobu słów itp.

Zadanie

Twoje zadanie polega na odkodowaniu zaszyfrowanych wiadomości i zapamiętaniu ich w określonych plikach. Nie musisz dostarczać żadnego programu. Wystarczy, jeśli zapiszesz odkodowane teksty.

Wejście

Dysponujesz kilkoma zestawami danych. Jeden zestaw składa się z plików `cra`*n*`.*`*, gdzie n jest numerem zestawu. Każdy zestaw składa się z plików:*

- `cra*.in`*, zaszyfrowana wiadomość,*
- `cra*.txt`*, pliki tekstowe pochodzące z tego samego źródła, co zaszyfrowana wiadomość.*

Wyjście

Dla każdej zaszyfrowanej wiadomości `cra*.in`*, powienieneś utworzyć plik* `cra*.out` *z odszyfrowaną wiadomością.*

Excursion

Grupa podróżników zamierza odwiedzić wybrane miasta. Każdy podróżnik określa dwa życzenia dotyczące odwiedzenia bądź nieodwiedzenia danego miasta. Jedno życzenie to stwierdzenie, że się chce odwiedzić dane miasto, albo że się nie chce odwiedzać danego miasta. Jedno życzenie dotyczy tylko jednego miasta. Można w obu życzeniach odnieść się do tego samego miasta; zarówno zgodnie — wtedy oba życzenia są identyczne — jak i przeciwstawnie, czyli np. „Ja chcę odwiedzić miasto A" i „Ja nie chcę odwiedzać miasta A".

Zadanie

Twoje zadanie polega na napisaniu programu, który

- *wczyta życzenia podróżnika z pliku* `exc.in`*,*
- *określi, czy można wybrać taką listę (być może pustą) odwiedzanych miast, aby spełnić przynajmniej jedno życzenie każdego podróżnika,*
- *zapisze listę odwiedzanych miast w pliku wyjściowym* `exc.out`*.*

Jeżeli jest kilka możliwych rozwiązań, Twój program powinien zapisać dowolne z nich.

Wejście

Pierwszy wiersz pliku `exc.in` *zawiera dwie dodatnie liczby całkowite n oraz m ($1 \leq n \leq 20\ 000$, $1 \leq m \leq 8\ 000$); n jest liczbą podróżników, zaś m jest liczbą miast. Podróżnicy są ponumerowani od 1 do n, a miasta od 1 do m. Każdy z kolejnych wierszy pliku zawiera dwie różne od zera liczby całkowite oddzielone pojedynczym odstępem. Wiersz $i+1$ zawiera liczby w_i oraz w'_i oznaczające życzenia i-tego podróżnika, $-m \leq w_i, w'_i \leq m$, $w_i, w'_i \neq 0$. Liczba dodatnia oznacza, że podróżnik chce odwiedzić, a ujemna, że podróżnik nie chce odwiedzać miasta, którego numer jest równy wartości bezwzględnej tej liczby.*

Wyjście

W pierwszym wierszu pliku wyjściowego `exc.out`*, Twój program powinien zapisać jedną nieujemną liczbę l określającą proponowaną liczbę miast do odwiedzenia. W drugim wierszu tego pliku powinno się znaleźć dokładnie l dodatnich liczb całkowitych, określających miasta, które mają być odwiedzone. Liczby te muszą być podane w porządku rosnącym.*

Gdyby takiej listy miast nie dało się stworzyć, Twój program powinien umieścić w pierwszym i jedynym wierszu pliku wyjściowego jedno słowo `NO`*.*

Przykład

Dla pliku wejściowego `exc.in`*:*

```
3 4
1 -2
2 4
3 1
```

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

```
2
3 4
```

Marcin Kubica
Tłumaczenie

Knights

Dana jest szachownica o wymiarach $n \times n$, z której usunięto pewną liczbę pól. Należy wyznaczyć maksymalną liczbę skoczków (koników) szachowych, które można ustawić na pozostałych polach szachownicy tak, żeby żadne dwa skoczki nie atakowały się nawzajem.

	x		x	
x				x
		S		
x				x
	x		x	

Rysunek 1: Skoczek umieszczony w polu S atakuje pola oznaczone przez x.

Zadanie

Napisz program, który:

- *wczyta opis szachownicy z usuniętymi polami z pliku tekstowego* `kni.in`*,*
- *wyznaczy maksymalną liczbę wzajemnie nie atakujących się skoczków szachowych, które można ustawić na tej szachownicy,*
- *zapisze wynik w pliku tekstowym* `kni.out`*.*

Wejście

W pierwszym wierszu pliku tekstowego `kni.in` *znajdują się dwie liczby całkowite n i m, gdzie $1 \leq n \leq 200$, $0 \leq m < n^2$. Liczba n oznacza rozmiar szachownicy, a m oznacza liczbę usuniętych pól.*

W każdym z kolejnych m wierszy jest zapisana para liczb naturalnych x i y, gdzie $1 \leq x, y \leq n$, oddzielonych pojedynczym odstępem. Są to współrzędne usuniętych pól. Lewy górny róg szachownicy ma współrzędne (1,1), natomiast prawy dolny róg ma współrzędne (n,n). Pola nie powtarzają się.

Wyjście

Plik tekstowy `kni.out` *powinien zawierać dokładnie jeden wiersz, zawierający pojedynczą liczbę całkowitą równą maksymalnej liczbie wzajemnie nie atakujących się skoczków, które można ustawić na zadanej szachownicy.*

Przykład

Dla pliku wejściowego `kni.in`*:*

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

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

```
5
```

Marcin Kubica
Tłumaczenie

Mars maps

W roku 2051 kilka ekspedycji marsjańskich wybrało się w różne rejony Czerwonej Planety i wykonało mapy tych terenów. BAK (Bałtycka Agencja Kosmiczna) ma ambitne plany: zamierza stworzyć mapę całej planety. Aby przewidzieć skalę zadania pracownicy agencji muszą znać całkowitą powierzchnię skartowanego dotychchczas terenu. Twoje zadanie polega na napisaniu programu, który tę powierzchnię obliczy.

Zadanie

Napisz program, który:

- *wczyta z pliku wejściowego* `mar.in` *opis obszarów pokrytych przez mapy,*
- *obliczy całkowitą powierzchnię obszaru pokrytego przez mapy,*
- *zapisze wynik w pliku wyjściowym* `mar.out`*.*

Wejście

Pierwszy wiersz pliku wejściowego `mar.in` *zawiera jedną liczbę całkowitą* N *(*$1 \leq N \leq 10\,000$*). Jest to liczba dostępnych map. Każdy z następnych* N *wierszy zawiera cztery liczby całkowite* x_1, y_1, x_2 *oraz* y_2 *(*$0 \leq x_1 < x_2 \leq 30\,000$, $0 \leq y_1 < y_2 \leq 30\,000$*). Wartości* (x_1, y_1) *oraz* (x_2, y_2) *to współrzędne odpowiednio lewego–dolnego i prawego–górnego rogu opisywanej mapy. Każda z map ma kształt prostokąta o bokach równoległych do osi układu współrzędnych.*

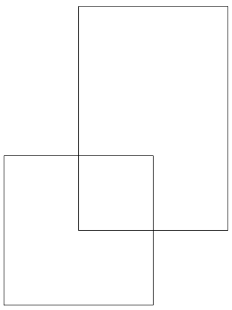
Wyjście

Plik wyjściowy `mar.out` *powinien zawierać jedną liczbę całkowitą* A *— całkowite pole powierzchni skartowanego obszaru (czyli pole powierzchni sumy wszystkich prostokątów).*

Przykład

Dla pliku wejściowego `mar.in`*:*

```
2
10 10 20 20
15 15 25 30
```



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

```
225
```

Postman

Wiejski listonosz musi dostarczać pocztę wszystkim mieszkańcom okolicy, którzy zamieszkują w wioskach i przy drogach łączących wioski.

Musisz pomóc listonoszowi wytyczyć trasę, która pozwoli mu przejechać wzdłuż każdej drogi i odwiedzić każdą wioskę w okolicy przynajmniej raz. Tak się szczęśliwie składa, że w rozważanych przykładach taka trasa zawsze istnieje. Jednak wytyczone trasy mogą się różnić jakością, tzn. listonosz może otrzymywać różną zapłatę za swą pracę w zależności od wybranej trasy (jak się za chwilę przekonamy, to nie zysk listonosza jest najważniejszy, a zysk jego firmy, czyli poczty). Mieszkańcy każdej wioski chcieliby, by listonosz docierał do nich jak najwcześniej. Każda wioska zawarła więc z pocztą następującą umowę: jeżeli i-ta wioska jest odwiedzana przez listonosza jako k-ta w kolejności (tzn. listonosz odwiedził $k-1$ **różnych** *wiosek, zanim po raz pierwszy dotarł do wioski i) oraz $k \leq w(i)$, to wioska płaci poczcie $w(i)-k$ euro. Jeśli jednak $k > w(i)$, to wówczas poczta płaci wiosce $k-w(i)$ euro. Ponadto poczta płaci listonoszowi jedno euro za każdy przejazd między dwiema kolejnymi wioskami na jego trasie.*

W rozważanej okolicy jest n wiosek, które oznaczamy liczbami naturalnymi od 1 do n. Poczta znajduje się w wiosce oznaczonej numerem 1, a więc trasa listonosza musi rozpoczynać się w tej wiosce. W każdej wiosce zbiega się 2, 4 lub 8 dróg. Pomiędzy dwiema wioskami może istnieć kilka różnych dróg; droga może także powracać do tej samej wioski, z której wyszła.

Zadanie

Twoim zadaniem jest napisanie programu, który:

- *wczyta opis wiosek i łączących je dróg z pliku tekstowego* `pos.in`,
- *znajdzie trasę, która prowadzi przez każdą wioskę i wzdłuż każdej drogi, i która pozwala osiągnąć poczcie maksymalny zysk (ewentualnie ponieść minimalną stratę),*
- *zapisze wynik w pliku tekstowym* `pos.out`.

Jeżeli istnieje więcej niż jedno rozwiązanie, to Twój program powinien obliczyć jedno z nich.

Wejście

W pierwszym wierszu pliku tekstowego `pos.in` *zapisane są dwie liczby naturalne n i m oddzielone pojedynczym odstępem; liczba n ($1 \leq n \leq 200$) oznacza liczbę wiosek, a m jest liczbą dróg. W każdym z kolejnych n wierszy znajduje się jedna liczba naturalna (dodatnia). Liczba w $(i+1)$-szym wierszu oznacza $w(i)$ ($1 < w(i) \leq 1\ 000$), czyli wstępną kwotę płaconą poczcie przez wioskę numer i (kwota ta jest oczywiście modyfikowana w opisany na początku zadania sposób). W każdym z kolejnych m wierszy znajdują się po dwie liczby naturalne oddzielone pojedynczym odstępem — oznaczają one numery wiosek, które łączy kolejna droga.*

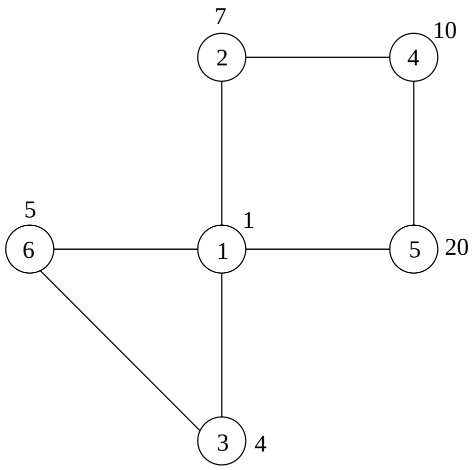
Wyjście

Twój program powinien zapisać jedną dodatnią liczbę naturalną k w pierwszym wierszu pliku tekstowego `pos.out`. *W kolejnym wierszu powinno znaleźć się $k+1$ liczb oznaczających numery wiosek odwiedzanych kolejno przez listonosza w ramach optymalnej trasy.*

Przykład

Dla pliku wejściowego `pos.in`:

```
6 7
1
7
4
10
20
5
2 4
1 5
2 1
4 5
3 6
1 6
1 3
```



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

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

Teleports

Wielki Czarodziej Bajtalf stworzył na Bałtyku dwie wyspy: Bornholm i Gotlandię. Na wyspach rozmieścił magiczne teleporty. Teleporty służą do szybkiego "podróżowania" – osoba umieszczona w jednym z teleportów w jednej chwili może się przenieść do innego teleportu. W każdym teleporcie, w trakcie produkcji, wpisuje się identyfikator jego teleportu docelowego, tzn. takiego, do którego może on przenosić "podróżników". Identyfikatora nie można już potem zmienić. Teleporty zostały rozmieszczone tak, by dla każdego teleportu, jego teleport docelowy znajdował się na drugiej wyspie.

Każdy teleport może być nastawiony na:

- *nadawanie — wówczas osoba, która się w nim znajduje zostaje przeniesiona do teleportu docelowego, o ile jest on (teleport docelowy) ustawiony na odbiór (patrz niżej),*
- *odbiór — wówczas może przyjąć podróżnika z innego teleportu.*

Pewnego dnia Wielki Czarodziej Bajtalf nakazał swoim uczniom, by nastawili teleporty tak, aby żaden z nich nie był bezużyteczny, tzn. tak, aby dla każdego teleportu nastawionego na odbiór istniał teleport przenoszący do niego podróżników nastawiony na nadawanie, i na odwrót, dla każdego teleportu nastawionego na nadawanie, jego docelowy teleport był nastawiony na odbiór.

Zadanie

Napisz program, który:

- *wczyta z pliku tekstowego* `tel.in` *opisy teleportów znajdujących się na obu wyspach,*
- *wyznaczy, jak należy nastawić teleporty, by żaden z nich nie był bezużyteczny,*
- *zapisze wynik do pliku tekstowego* `tel.out`*.*

Jeżeli istnieje wiele rozwiązań, to Twój program powinien wyznaczyć jedno z nich.

Wejście

W pierwszym wierszu pliku tekstowego `tel.in` *znajdują się dwie liczby całkowite m i n, $1 \leq m, n \leq 50\,000$, oddzielone pojedynczym odstępem; m oznacza liczbę teleportów znajdujących się na Bornholmie, a n – liczbę teleportów znajdujących się na Gotlandii. Teleporty na obu wyspach są ponumerowane odpowiednio od 1 do m i od 1 do n. Drugi wiersz pliku wejściowego zawiera m dodatnich liczb całkowitych (nie przekraczających n i oddzielonych pojedynczymi odstępami); k-ta z tych liczb jest numerem teleportu na Gotlandii, który jest teleportem docelowym k-tego teleportu z Bornholmu. Trzeci wiersz zawiera analogiczne dane dla teleportów z Gotlandii, tzn. n dodatnich liczb całkowitych (nie przekraczających m i oddzielonych pojedynczymi odstępami); k-ta z tych liczb jest numerem teleportu na Bornholmie, który jest teleportem docelowym k-tego teleportu z Gotlandii.*

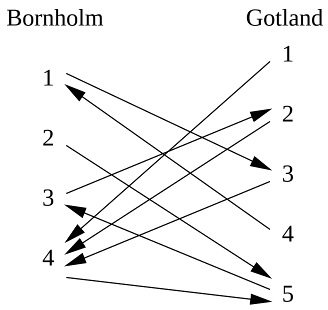
Wyjście

Twój program powinien zapisać w pliku wynikowym `tel.out` *dwa wiersze opisujące, jak należy nastawić teleporty, by żaden z nich nie był bezużyteczny. W pierwszym wierszu powinien znaleźć się opis ustawień teleportów na Bornholmie, a w drugim – opis ustawień teleportów na Gotlandii. Każdy opis, to napis długości równej odpowiednio m i n, złożony z zer lub jedynek. Jeżeli k-ty znak w wierszu jest równy* `1`*, to oznacza, że teleport o numerze k (na danej wyspie) jest ustawiony na nadawanie; jeśli odpowiedni znak jest równy* `0` *– to teleport jest nastawiony na odbiór.*

Przykład

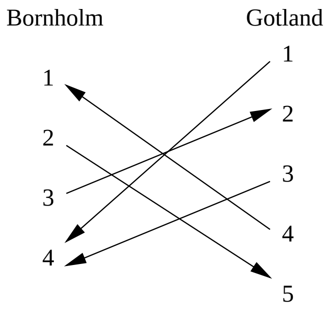
Dla pliku wejściowego `tel.in`*:*

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



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

```
0110
10110
```



Literatura

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

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

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

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

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

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

[14] T. H. Cormen, C. E. Leiserson, and 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, and O. Patashnik. *Matematyka konkretna.* PWN, Warszawa, 1996.

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

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

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

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

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

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

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

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

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

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

[28] A. Silberschatz and P. B. Galvin. *Podstawy systemów operacyjnych.* WNT, Warszawa, 2000.

Niniejsza publikacja stanowi wyczerpujące źródło informacji o zawodach VIII Olimpiady Informatycznej, przeprowadzonych w roku szkolnym 2000/2001. Książka zawiera zarówno informacje dotyczące organizacji, regulaminu oraz wyników zawodów. Zawarto także opis rozwiązań wszystkich zadań konkursowych. Do książki dołączona jest dyskietka zawierająca wzorcowe rozwiązania i testy do wszystkich zadań Olimpiady.

Książka zawiera też zadania z olimpiad międzynarodowych, XII i XIII, oraz 7–ej Bałtyckiej Olimpiady Informatycznej.

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

Olimpiada Informatyczna
jest organizowana przy współudziale

PROKOM
SOFTWARE SA

ISBN 83–906301–7–6