Polak Mistrzem Świata!

Filip Wolski zwycięzcą XVIII Międzynarodowej Olimpiady Informatycznej

MINISTERSTWO EDUKACJI NARODOWEJ
UNIWERSYTET WROCŁAWSKI
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

XIII OLIMPIADA INFORMATYCZNA

2005/2006

Olimpiada Informatyczna jest organizowana przy współudziale

PROKOM
SOFTWARE SA

WARSZAWA, 2006

MINISTERSTWO EDUKACJI NARODOWEJ
UNIWERSYTET WROCŁAWSKI
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

XIII OLIMPIADA INFORMATYCZNA
2005/2006

WARSZAWA, 2006

Autorzy tekstów:

Szymon Acedański
Marek Cygan
Karol Cwalina
prof. dr hab. Zbigniew Czech
dr hab. Krzysztof Diks
Adam Iwanicki
Maciej Jaśkowski
dr Przemysława Kanarek
dr Łukasz Kowalik
dr Marcin Kubica
mgr Jakub Pawlewicz
Jakub Radoszewski
mgr Krzysztof Sikora
Piotr Stańczyk
Bartosz Walczak
mgr Tomasz Waleń

Autorzy programów wzorcowych:

Szymon Acedański
Michał Adamaszek
Karol Cwalina
Marek Cygan
Krzysztof Dulęba
Adam Iwanicki
dr Łukasz Kowalik
Marcin Michalski
Paweł Parys
mgr Jakub Pawlewicz
Marcin Pilipczuk
Jakub Radoszewski
Piotr Stańczyk
Szymon Wąsik

Opracowanie i redakcja:

dr Przemysława Kanarek
Adam Iwanicki

Opracowanie i redakcja treści zadań:

dr Marcin Kubica

Skład:

Adam Iwanicki

Uwaga dla Czytelników egzemplarzy bezpłatnych książki w bibliotekach szkolnych. *Do tych egzemplarzy nie załączamy płyty z testami i rozwiązaniami wzorcowymi. Olimpiada nie ma środków na ten wydatek. Mamy nadzieję, że wobec powszechnej dostępności Internetu, umieszczenie programów wzorcowych i testów na stronie internetowej Olimpiady (www.oi.edu.pl) będzie wystarczające. Wszelkie odniesienia w tekście książki do „płyty" należy rozumieć, jako odniesienia do materiałów, które są dostępne na tej stronie.*

Pozycja dotowana przez Ministerstwo Edukacji Narodowej.

Druk książki został sfinansowany przez PROKOM SOFTWARE SA

ISBN 83–922946–1–0

Spis treści

Krzysztof Diks

Wstęp

Drogi Czytelniku!

Oddajemy do rąk czytelników sprawozdanie oraz zadania wraz z rozwiązaniami wzorcowymi z XIII Olimpiady Informatycznej. Miniony rok szkolny, rok XIII Olimpiady Informatycznej, był szczególny. W Olimpiadzie wzięła udział rekordowa liczba uczestników. Niewątpliwie duży wpływ na to miała akcja promująca konkursy programistyczne przeprowadzona z okazji XVII Międzynarodowej Olimpiady Informatycznej, która w ubiegłym roku odbyła się w Polsce, w Nowym Sączu.

Poziom naszej Olimpiady rośnie z roku na rok. Uczestnicy są coraz lepiej przygotowani, a organizatorzy starają się dostarczać ciekawych i wartościowych zadań. O tym, że tak jest w rzeczywistości świadczą wyniki laureatów XIII Olimpiady Informatycznej. Zacznijmy od wyniku najważniejszego. Po raz pierwszy w historii Polak, Filip Wolski z III LO im. Marynarki Wojennej w Gdyni, stanął na najwyższym podium Międzynarodowej Olimpiady Informatycznej. Nie gorzej spisali się na tych zawodach koledzy Filipa: Jakub Kallas, Marcin Andrychowicz i Michał Pilipczuk. Jakub i Michał zdobyli medale złote, a Michał medal srebrny. Równie dobrze wypadli nasi reprezentanci w dwóch ważnych, międzynarodowych konkursach programistycznych. W XII Olimpiadzie Krajów Bałtyckich — BOI 2006 zwyciężył Tomasz Kulczyński, złote medale zdobyli Marcin Andrychowicz i Wojciech Śmietanka, medale srebrne: Jakub Kallas i Maciej Klimek, natomiast Robert Obryk zdobył medal brązowy. W XIII Olimpiadzie Informatycznej Krajów Europy Środkowej zwyciężył Filip Wolski, Marcin Andrychowicz zdobył medal srebrny, a Michał Pilipczuk medal brązowy.

Sukcesy młodych polskich informatyków to przede wszystkim wynik ich wspaniałych umiejętności, dużej wiedzy i ciężkiej pracy. Bez wątpienia jednak wyrazy podziękowania należą się także ich rodzicom i opiekunom naukowym oraz szkołom z których pochodzą.

Słowa podziękowania kieruję też do moich koleżanek i kolegów z Olimpiady Informatycznej, członków Komitetu Głównego i Komitetów Okręgowych, jurorów i pracowników technicznych. Bez ich zaangażowania, ciężkiej i twórczej pracy nie byłoby tych sukcesów.

Jeszcze trzy ważne sprawy. Od tego roku będziemy przesyłali bezpłatny egzemplarz „Olimpiady Informatycznej ...” do wszystkich znanych nam szkół ponadgimnazjalnych dla szkolnej biblioteki. Mamy nadzieję, że w ten sposób nasze wydawnictwo będzie łatwiej i szerzej dostępne, lepiej służąc rozwojowi i popularyzacji informatyki.

Przy tej masowej wysyłce nie będziemy jednak załączali do książki płyty z testami i rozwiązaniami wzorcowymi. Prosimy o zrozumienie: przy przesyłaniu bezpłatnych egzemplarzy książki do szkolnych bibliotek, koszt płyt do całego nakładu byłby dla Olimpiady zbyt wysoki. Nie mamy środków na poniesienie tych kosztów. Mamy nadzieję, że wobec powszechnej dostępności Internetu, dostępność rozwiązań wzorcowych i testów na stronie internetowej Olimpiady: www.oi.edu.pl będzie wystarczająca. Dla tych egzemplarzy książki wszelkie odniesienia w tekście książki do „płyty” należy rozumieć, jako odniesienia do materiałów, które są dostępne na tej stronie. Oczywiście do wydawnictwa tegorocznego sprzedawanego przez Olimpiadę będą dołączane płyty.

W kolejnych latach zamierzamy z wydawania płyt zrezygnować, gdyż sądzimy, że w porównaniu z dostępnością materiałów w Internecie, będzie to rozwiązanie anachroniczne i zbytecznie podnoszące koszty. Jeśli uważasz Drogi Czytelniku, że się mylimy, napisz nam o tym proszę: sekretariat@oi.edu.pl. Zależy nam by służyć Ci możliwie najlepiej.

Wszystkim młodym czytelnikom tej książeczki, przyszłym olimpijczykom, życzę pójścia w ślady starszych kolegów. Jestem przekonany, że materiały, które oddajemy do Waszych rąk pomogą osiągnąć ten cel.

Krzysztof Diks

Sprawozdanie z działań XIII Olimpiady Informatycznej w roku szkolnym 2005/2006

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. Olimpiada działa zgodnie z Rozporządzeniem Ministra Edukacji Narodowej i Sportu z dnia 29 stycznia 2002 roku w sprawie organizacji oraz sposobu przeprowadzenia konkursów, turniejów i olimpiad (Dz. U. 02.13.125). Obecnie organizatorem Olimpiady Informatycznej jest Uniwersytet Wrocławski.

ORGANIZACJA ZAWODÓW

W roku szkolnym 2005/2006 odbyły się zawody XIII Olimpiady Informatycznej. Olimpiada Informatyczna jest trójstopniowa. Rozwiązaniem każdego zadania zawodów I, II i III stopnia jest program napisany w jednym z ustalonych przez Komitet Główny Olimpiady języków programowania lub plik z wynikami. Zawody I stopnia miały charakter otwartego konkursu dla uczniów wszystkich typów szkół młodzieżowych.

17 października 2005 r. rozesłano plakaty zawierające zasady organizacji zawodów I stopnia oraz zestaw 5 zadań konkursowych do 3721 szkół i zespołów szkół młodzieżowych ponadgimnazjalnych oraz do wszystkich kuratorów i koordynatorów edukacji informatycznej. Zawody I stopnia rozpoczęły się dnia 24 października 2005 r. Ostatecznym terminem nadsyłania prac konkursowych był 21 listopada 2005 roku.

Zawody II i III stopnia były dwudniowymi sesjami stacjonarnymi, poprzedzonymi jednodniowymi sesjami próbnymi. Zawody II stopnia odbyły się w sześciu okręgach: Warszawie, Wrocławiu, Toruniu, Katowicach, Krakowie i Rzeszowie oraz w Sopocie w dniach 12–14.02.2006 r., natomiast zawody III stopnia odbyły się w ośrodku firmy Combidata Poland S.A. w Sopocie, w dniach 28.03–01.04.2006 r.

Uroczystość zakończenia XIII Olimpiady Informatycznej odbyła się w dniu 01.04.2006 r. w Sali Posiedzeń Urzędu Miasta w Sopocie.

SKŁAD OSOBOWY KOMITETÓW OLIMPIADY INFORMATYCZNEJ

Komitet Główny

przewodniczący:
    dr hab. Krzysztof Diks, prof. UW (Uniwersytet Warszawski)
zastępcy przewodniczącego:
    prof. dr hab. Maciej M. Sysło (Uniwersytet Wrocławski)
    prof. dr hab. Paweł Idziak (Uniwersytet Jagielloński)

sekretarz naukowy:
dr Marcin Kubica (Uniwersytet Warszawski)

kierownik Jury:
dr Krzysztof Stencel (Uniwersytet Warszawski)

kierownik organizacyjny:
Tadeusz Kuran (OEIiZK)

członkowie:
dr Piotr Chrząstowski-Wachtel (Uniwersytet Warszawski)
prof. dr hab. Zbigniew Czech (Politechnika Śląska)
dr Przemysława Kanarek (Uniwersytet Wrocławski)
mgr Anna Beata Kwiatkowska (IV LO im. T. Kościuszki w Toruniu)
dr hab. Krzysztof Loryś, prof. UWr (Uniwersytet Wrocławski)
prof. dr hab. Jan Madey (Uniwersytet Warszawski)
prof. dr hab. Jerzy Nawrocki (Politechnika Poznańska)
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 Mirosława Skowrońska (Uniwersytet Mikołaja Kopernika w Toruniu)
dr Andrzej Walat (OEIiZK)
mgr Tomasz Waleń (Uniwersytet Warszawski)

sekretarz Komitetu Głównego:
Monika Kozłowska-Zając (OEIiZK)

Komitet Główny mieści się w Warszawie przy ul. Nowogrodzkiej 73, w Ośrodku Edukacji Informatycznej i Zastosowań Komputerów.
Komitet Główny odbył 5 posiedzeń.

Komitety okręgowe

Komitet Okręgowy w Warszawie
przewodniczący:
dr Adam Malinowski (Uniwersytet Warszawski)
sekretarz:
Monika Kozłowska-Zając (OEIiZK)
członkowie:
dr Marcin Kubica (Uniwersytet Warszawski)
dr Andrzej Walat (OEIiZK)

Komitet Okręgowy mieści się w Warszawie przy ul. Nowogrodzkiej 73, w Ośrodku Edukacji Informatycznej i Zastosowań Komputerów.

Komitet Okręgowy we Wrocławiu
przewodniczący:
dr hab. Krzysztof Loryś, prof. UWr (Uniwersytet Wrocławski)

zastępca przewodniczącego:
    dr Helena Krupicka (Uniwersytet Wrocławski)
sekretarz:
    inż. Maria Woźniak (Uniwersytet Wrocławski)
członkowie:
    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. Adam Jakubowski (Uniwersytet Mikołaja Kopernika w Toruniu)
zastępca przewodniczącego:
    dr Mirosława Skowrońska (Uniwersytet Mikołaja Kopernika w Toruniu)
sekretarz:
    dr Barbara Klunder (Uniwersytet Mikołaja Kopernika w Toruniu)
członkowie:
    dr Andrzej Kurpiel (Uniwersytet Mikołaja Kopernika w Toruniu)
    mgr Anna Beata Kwiatkowska (IV Liceum Ogólnokształcące w Toruniu)

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

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

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

Komitet Okręgowy w Krakowie
przewodniczący:
    prof. dr hab. Paweł Idziak (Uniwersytet Jagielloński)
zastępca przewodniczącego:
    dr Maciej Ślusarek (Uniwersytet Jagielloński)
sekretarz:
    mgr Henryk Białek (Kuratorium Oświaty w Krakowie) członkowie:
    dr Marcin Kozik (Uniwersytet Jagielloński)

mgr Jan Pawłowski (Uniwersytet Jagielloński)
Grzegorz Gutowski (Uniwersytet Jagielloński)

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

Komitet Okręgowy w Rzeszowie

przewodniczący:
prof. dr hab. inż. Stanisław Paszczyński (Wyższa Szkoła Informatyki i Zarządzania w Rzeszowie)
zastępca przewodniczącego:
dr Marek Jaszuk (Wyższa Szkoła Informatyki i Zarządzania w Rzeszowie)
sekretarz:
mgr inż. Grzegorz Owsiany (Wyższa Szkoła Informatyki i Zarządzania w Rzeszowie)
członkowie:
mgr Czesław Wal (Wyższa Szkoła Informatyki i Zarządzania w Rzeszowie)
mgr inż. Dominik Wojtaszek (Wyższa Szkoła Informatyki i Zarządzania w Rzeszowie)
mgr inż. Piotr Błajdo (Wyższa Szkoła Informatyki i Zarządzania w Rzeszowie)
Maksymilian Knap (Wyższa Szkoła Informatyki i Zarządzania w Rzeszowie).

Siedzibą Komitetu Okręgowego w Rzeszowie jest Wyższa Szkoła Informatyki i Zarządzania w Rzeszowie, ul. Sucharskiego 2.

Jury Olimpiady Informatycznej

W pracach Jury, które nadzorował Krzysztof Diks, a którymi kierował Krzysztof Stencel, brali udział pracownicy, doktoranci i studenci Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego, Wydziału Matematyki i Informatyki Uniwersytetu Jagiellońskiego oraz Wydziału Informatyki i Zarządzania Politechniki Poznańskiej:

Szymon Acedański
Marek Cygan
Krzysztof Dulęba
Tomasz Idziaszek
Adam Iwanicki
Damian Koniecki
dr Łukasz Kowalik
Tomasz Malesiński
Marcin Michalski
Anna Niewiarowska
Paweł Parys
Marcin Pilipczuk
Jakub Radoszewski
Piotr Stańczyk
Piotr Stawiński
Bartosz Walczak
Szymon Wąsik
Marek Żylak

ZAWODY I STOPNIA

W XIII Olimpiadzie Informatycznej wzięło udział 1106 zawodników.

Decyzją Komitetu Głównego do zawodów zostało dopuszczonych 36 uczniów gimnazjów. Byli to uczniowie z następujących gimnazjów:

Gimnazjum nr 24 przy III LO	Gdynia	6 uczniów
Gimnazjum nr 16 ZSO nr 7	Szczecin	3 uczniów
Gimnazjum	Koziegłowy	2 uczniów
Publiczne Gimnazjum nr 1	Bełchatów	1 uczeń
Gimnazjum nr 36	Bytom	1 uczeń
Gimnazjum nr 3	Gdańsk	1 uczeń
Gdańskie Gimnazjum Lingwista	Gdańsk	1 uczeń
Gimnazjum nr 4	Głogów	1 uczeń
Zespół Szkół	Gostyń	1 uczeń
Gimnazjum nr 7	Konin	1 uczeń
Gimnazjum nr 1	Kraków	1 uczeń
Gimnazjum nr 52 Ojców Pijarów	Kraków	1 uczeń
Gimanzjum nr 18	Kraków	1 uczeń
Gimnazjum	Kurów	1 uczeń
Zespół Szkół	Mniów	1 uczeń
Gimnazjum Publiczne	Nowa Wieś Królewska	1 uczeń
Publiczne Gimnazjum nr 2	Puławy	1 uczeń
Gimnazjum	Skarżysko Kościelne	1 uczeń
Gimnazjum nr 5	Stalowa Wola	1 uczeń
Gimnazjum nr 29	Szczecin	1 uczeń
Gimnazjum nr 1	Tomaszów Lubelski	1 uczeń
Gimnazjum i Liceum Akademickie	Toruń	1 uczeń
Gimnazjum nr 5 im. Jana Pawła II	Warszawa	1 uczeń
Gimnazjum nr 13 im Stanisława Staszica	Warszawa	1 uczeń
Gimnazjum	Węgrów	1 uczeń
Publiczne Gimnazjum nr 1	Włoszczowa	1 uczeń
Gimnazjum nr 1	Wolbrom	1 uczeń
Gimnazjum nr 2 im. Adama Asnyka	Zielona Góra	1 uczeń

Kolejność województw pod względem liczby uczestników była następująca:

małopolskie	185 uczniów	łódzkie	53 uczniów
mazowieckie	145 uczniów	lubelskie	41 uczniów
pomorskie	126 uczniów	podlaskie	41 uczniów
śląskie	117 uczniów	świętokrzyskie	36 uczniów
dolnośląskie	78 uczniów	zachodniopomorskie	34 uczniów
podkarpackie	74 uczniów	opolskie	20 uczniów
kujawsko-pomorskie	66 uczniów	warmińsko-mazurskie	18 uczniów
wielkopolskie	61 uczniów	lubuskie	11 uczniów

W zawodach I stopnia najliczniej reprezentowane były szkoły:

V LO im. A. Witkowskiego	Kraków	86 uczniów
III LO im. Marynarki Wojennej RP	Gdynia	56 uczniów
XIV LO im. Stanisława Staszica	Warszawa	46 uczniów
I LO im. A. Mickiewicza	Białystok	19 uczniów
VI LO im. Jana i Jędrzeja Śniadeckich	Bydgoszcz	17 uczniów
VIII LO im. Marii Skłodowskiej–Curie	Katowice	17 uczniów
IV Liceum Ogólnokształcące	Toruń	17 uczniów
XIV LO im. Polonii Belgijskiej	Wrocław	17 uczniów
VIII LO im. Adama Mickiewicza	Poznań	14 uczniów
VI LO im. Wacława Sierpińskiego	Gdynia	13 uczniów
II LO im. Jana III Sobieskiego	Kraków	11 uczniów
I LO im. Stefana Żeromskiego	Kielce	10 uczniów
I LO im. Stanisława Staszica	Lublin	10 uczniów
Publiczne LO nr 2 im. Marii Konopnickiej	Opole	10 uczniów
III LO im. Adama Mickiewicza	Wrocław	10 uczniów
V LO im. Stefana Żeromskiego	Gdańsk	8 uczniów
I LO im. Mikołaja Kopernika	Łódź	8 uczniów
XXVII LO im. Tadeusza Czackiego	Warszawa	8 uczniów
VII LO im. Krzysztofa Kamila Baczyńskiego	Wrocław	8 uczniów
L LO im. Ruy Barbosa	Warszawa	7 uczniów
Gimnazjum nr 24 przy III LO	Gdynia	6 uczniów
II LO im. Mikołaja Kopernika	Mielec	6 uczniów
LO WSIZ	Rzeszów	6 uczniów
Zespół Szkół Licealnych	Słubice	6 uczniów
V Liceum Ogólnokształcące	Bielsko-Biała	5 uczniów
I LO im. Cypriana Kamila Norwida	Bydgoszcz	5 uczniów
I LO im. Edwarda Dembowskiego	Gliwice	5 uczniów
I LO im. Tadeusza Kościuszki	Gorzów Wlkp.	5 uczniów
I LO im. Stanisława Dubois	Koszalin	5 uczniów
LO Zakonu Pijarów	Kraków	5 uczniów
Zespół Szkół Komunikacji	Poznań	5 uczniów
Zespół Szkół Elektronicznych	Rzeszów	5 uczniów
VI Liceum ogólnokształcące	Rzeszów	5 uczniów
II LO im. Adama Mickiewicza	Słupsk	5 uczniów
I LO im. Bolesława Krzywoustego	Słupsk	5 uczniów
Zespół Szkół nr 4	Stalowa Wola	5 uczniów
Zespół Szkół im. Adama Mickiewicza	Strzyżów	5 uczniów
Gimnazjum i Liceum Akademickie	Toruń	5 uczniów
I LO im. Ziemi Kujawskiej	Włocławek	5 uczniów
X Liceum Ogólnokształcące	Wrocław	5 uczniów
LO im. Marii Skłodowskiej-Curie	Andrychów	4 uczniów
I LO im. Juliusza Słowackiego	Chorzów	4 uczniów
III LO im. Stefana Batorego	Chorzów	4 uczniów
II LO im. Romualda Traugutta	Częstochowa	4 uczniów
VII LO im. Mikołaja Kopernika	Częstochowa	4 uczniów

I LO im. Króla Władysława Jagiełły	Dębica	4 uczniów
I LO im. Mikołaja Kopernika	Gdańsk	4 uczniów
LO im. Mikołaja Kopernika	Jarosław	4 uczniów
I LO im. Tadeusza Kościuszki	Konin	4 uczniów
I LO im. Bartłomieja Nowodworskiego	Kraków	4 uczniów
I LO im. Mikołaja Kopernika	Krosno	4 uczniów
I LO im. Tadeusza Kościuszki	Legnica	4 uczniów
I LO im. Tadeusza Kościuszki	Łomża	4 uczniów
Zespół Szkół Zawodowych	Mszana Dolna	4 uczniów
I LO im. Jana Długosza	Nowy Sącz	4 uczniów
III Liceum Ogólnokształcące	Opole	4 uczniów
1 LO im. Bolesława Chrobrego	Piotrków Trybunalski	4 uczniów
III Liceum Ogólnokształcące	Rzeszów	4 uczniów
I LO im. Juliusza Słowackiego	Skarżysko-Kamienna	4 uczniów
III LO im. Adama Mickiewicza	Tarnów	4 uczniów
VIII LO im. Władysława IV	Warszawa	4 uczniów
XVIII LO im. Jana Zamoyskiego	Warszawa	4 uczniów
II LO im. Władysława Andersa	Chojnice	3 uczniów
I LO im. B. Krzywoustego	Głogów	3 uczniów
I LO im. Kr. St. Leszczyńskiego	Jasło	3 uczniów
V Liceum Ogólnokształcące	Jastrzębie Zdrój	3 uczniów
II LO im. Marii Konopnickiej	Katowice	3 uczniów
II LO im. Jana Śniadeckiego	Kielce	3 uczniów
II Liceum ogólnokształcące	Konin	3 uczniów
XI Liceum Ogólnokształcące	Kraków	3 uczniów
I LO im. Władysława Jagiełły	Krasnystaw	3 uczniów
I LO im. Stefana Żeromskiego	Lębork	3 uczniów
Zespół Szkół nr 1	Limanowa	3 uczniów
II LO im. Jana Zamoyskiego	Lublin	3 uczniów
XII LO im. Stanisława Wyspiańskiego	Łódź	3 uczniów
I LO im. Tadeusza Kościuszki	Myślenice	3 uczniów
ZS im. Zygmunta Chmielewskiego	Nałęczów	3 uczniów
LO im. Jana Pawła II	Niepołomice	3 uczniów
II LO im. Marii Konopnickiej	Nowy Sącz	3 uczniów
II LO	Olsztyn	3 uczniów
II LO im. Joachima Chreptowicza	Ostrowiec Świętokrzyski	3 uczniów
LO im. Stanisława Małachowskiego	Płock	3 uczniów
LO im. Piotra Skargi	Pułtusk	3 uczniów
VI LO im. Jana Kochanowskiego	Radom	3 uczniów
II LO im. Andrzeja Frycza Modrzewskiego	Rybnik	3 uczniów
I LO im. Komisji Edukacji Narodowej	Sanok	3 uczniów
LO im. Bolesława Prusa	Skierniewice	3 uczniów
I LO im. Marii Skłodowskiej-Curie	Sokołów Podlaski	3 uczniów
IX Liceum Ogólnokształcące	Szczecin	3 uczniów
Gimnazjum nr 16 w ZSO nr 7	Szczecin	3 uczniów
XIII Liceum Ogólnokształcące	Szczecin	3 uczniów

II LO im. Stanisława Staszica	Tarnowskie Góry	3 uczniów
Zespół Szkół Technicznych	Tarnów	3 uczniów
II LO im. Hugona Kołłątaja	Wałbrzych	3 uczniów
LXIV LO im. Stanisława Ignacego Witkiewicza	Warszawa	3 uczniów
LI LO im. Tadeusza Kościuszki	Warszawa	3 uczniów
XIII LO im. płk. Leopolda Lisa-Kuli	Warszawa	3 uczniów
VI LO im. Tadeusza Reytana	Warszawa	3 uczniów
II LO im. ks. Józefa Tishnera	Wodzisław Śląski	3 uczniów
LO im. Henryka Sienkiewicza	Września	3 uczniów

Najliczniej reprezentowane były miasta:

Kraków	124 uczniów	Dębica	4 uczniów
Warszawa	110 uczniów	Elbląg	4 uczniów
Gdynia	80 uczniów	Głogów	4 uczniów
Wrocław	49 uczniów	Jarosław	4 uczniów
Rzeszów	28 uczniów	Krosno	4 uczniów
Białystok	26 uczniów	Mszana Dolna	4 uczniów
Bydgoszcz	26 uczniów	Ostrowiec Świętokrzyski	4 uczniów
Gdańsk	25 uczniów	Siedlce	4 uczniów
Toruń	25 uczniów	Skarżysko-Kamienna	4 uczniów
Poznań	23 uczniów	Sosnowiec	4 uczniów
Katowice	22 uczniów	Wałbrzych	4 uczniów
Łódź	22 uczniów	Zabrze	4 uczniów
Szczecin	20 uczniów	Chojnice	3 uczniów
Lublin	17 uczniów	Gorlice	3 uczniów
Kielce	16 uczniów	Inowrocław	3 uczniów
Opole	15 uczniów	Jasło	3 uczniów
Tarnów	12 uczniów	Jastrzębie Zdrój	3 uczniów
Częstochowa	11 uczniów	Jelenia Góra	3 uczniów
Słupsk	10 uczniów	Krasnystaw	3 uczniów
Bielsko-Biała	9 uczniów	Kraśnik	3 uczniów
Gliwice	9 uczniów	Legionowo	3 uczniów
Chorzów	8 uczniów	Lębork	3 uczniów
Konin	8 uczniów	Limanowa	3 uczniów
Gorzów Wlkp.	7 uczniów	Myślenice	3 uczniów
Legnica	7 uczniów	Nałęczów	3 uczniów
Nowy Sącz	7 uczniów	Niepołomice	3 uczniów
Słubice	7 uczniów	Ostrów Wielkopolski	3 uczniów
Mielec	6 uczniów	Oświęcim	3 uczniów
Piotrków Trybunalski	6 uczniów	Pabianice	3 uczniów
Stalowa Wola	6 uczniów	Piła	3 uczniów
Wodzisław Śląski	6 uczniów	Pułtusk	3 uczniów
Koszalin	5 uczniów	Racibórz	3 uczniów
Łomża	5 uczniów	Radom	3 uczniów
Olsztyn	5 uczniów	Ruda Śląska	3 uczniów

Płock	5 uczniów	Rybnik	3 uczniów
Sieradz	5 uczniów	Sanok	3 uczniów
Skierniewice	5 uczniów	Sokołów Podlaski	3 uczniów
Strzyżów	5 uczniów	Szczecinek	3 uczniów
Włocławek	5 uczniów	Tarnowskie Góry	3 uczniów
Zielona Góra	5 uczniów	Wadowice	3 uczniów
Andrychów	4 uczniów	Września	3 uczniów
Cieszyn	4 uczniów	Zawiercie	3 uczniów

Zawodnicy uczęszczali do następujących klas:

do klasy I	gimnazjum	2 uczniów
do klasy II	gimnazjum	8 uczniów
do klasy III	gimnazjum	26 uczniów
do klasy I	szkoły średniej	163 uczniów
do klasy II	szkoły średniej	408 uczniów
do klasy III	szkoły średniej	481 uczniów
do klasy IV	szkoły średniej	12 uczniów

8 zawodników nie podało informacji do której klasy uczęszczają.

W zawodach I stopnia zawodnicy mieli do rozwiązania pięć zadań: „Krążki", „Okresy słów", „Profesor Szu", „Tetris 3D", „Żaby".

W wyniku zastosowania procedury sprawdzającej wykryto niesamodzielne rozwiązania. Komitet Główny, w zależności od indywidualnej sytuacji, nie brał tych rozwiązań pod uwagę lub dyskwalifikował zawodników, którzy je nadesłali.

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

- **KRA** — Krążki

	KRA	
	liczba zawodników	czyli
100 pkt.	474	42,9%
75–99 pkt.	76	6,8%
50–74 pkt.	47	4,2%
1–49 pkt.	368	33,3%
0 pkt.	128	11,6%
brak rozwiązania	13	1,2%

- **OKR** — Okresy słów

	OKR	
	liczba zawodników	czyli
100 pkt.	303	27,4%
75–99 pkt.	13	1,2%
50–74 pkt.	48	4,3%
1–49 pkt.	381	34,5%
0 pkt.	121	10,9%
brak rozwiązania	240	21,7%

- **PRO** — Profesor Szu

	PRO	
	liczba zawodników	czyli
100 pkt.	118	10,7%
75–99 pkt.	62	5,6%
50–74 pkt.	121	10,9%
1–49 pkt.	211	19,1%
0 pkt.	108	9,8%
brak rozwiązania	486	43,9%

- **TET** — Tetris 3D

	TET	
	liczba zawodników	czyli
100 pkt.	16	1,4%
75–99 pkt.	25	2,3%
50–74 pkt.	37	3,3%
1–49 pkt.	631	57,1%
0 pkt.	209	18,9%
brak rozwiązania	188	17,0%

- **ZAB** — Żaby

	ZAB	
	liczba zawodników	czyli
100 pkt.	19	1,7%
75–99 pkt.	73	6,6%
50–74 pkt.	84	7,6%
1–49 pkt.	218	19,7%
0 pkt.	134	12,1%
brak rozwiązania	578	52,3%

W sumie za wszystkie 5 zadań konkursowych:

SUMA	liczba zawodników	czyli
500 pkt.	6	0,5%
375–499 pkt.	74	6,7%
250–374 pkt.	210	19,0%
125–249 pkt.	253	22,9%
1–124 pkt.	479	43,3%
0 pkt.	84	7,6%

Wszyscy zawodnicy otrzymali informacje o uzyskanych przez siebie wynikach. Na stronie internetowej Olimpiady udostępnione były testy, na podstawie których oceniano prace.

ZAWODY II STOPNIA

Do zawodów II stopnia zakwalifikowano 360 zawodników, którzy osiągnęli w zawodach I stopnia wynik nie mniejszy niż 210 pkt.

Pięciu zawodników nie stawiło się na zawody. W zawodach II stopnia uczestniczyło 355 zawodników.

Zawody II stopnia odbyły się w dniach 12–14 lutego 2006 r. w sześciu stałych okręgach oraz w Sopocie:

- w Gliwicach — 33 zawodników z następujących województw:
    - małopolskie (2)
    - śląskie (31)
- w Krakowie — 60 zawodników z następujących województw:
    - małopolskie (60)
- w Rzeszowie — 38 zawodników z następujących województw:
    - lubelskie (7)
    - małopolskie (15)
    - podkarpackie (15)
    - świętokrzyskie (1)
- w Toruniu — 35 zawodników z następujących województw:
    - kujawsko-pomorskie (25)
    - mazowieckie (3)
    - warmińsko-mazurskie (3)
    - wielkopolskie (4)
- w Warszawie — 73 zawodników z następujących województw:

  - lubelskie (3)
  - łódzkie (2)
  - mazowieckie (45)
  - podlaskie (13)
  - świętokrzyskie (10)
- we Wrocławiu — 50 zawodników z następujących województw:
  - dolnośląskie (18)
  - łódzkie (5)
  - opolskie (9)
  - śląskie (3)
  - świętokrzyskie (1)
  - wielkopolskie (9)
  - zachodniopomorskie (5)
- w Sopocie — 66 zawodników z następujących województw:
  - pomorskie (60)
  - zachodniopomorskie (6)

W zawodach II stopnia najliczniej reprezentowane były szkoły:

V LO im. Augusta Witkowskiego	Kraków	56 uczniów
III LO im. Marynarki Wojennej RP	Gdynia	40 uczniów
XIV LO im. Stanisława Staszica	Warszawa	19 uczniów
VI LO im. Jana i Jędrzeja Śniadeckich	Bydgoszcz	14 uczniów
VIII LO im. Marii Skłodowskiej-Curie	Katowice	13 uczniów
I LO im. Adama Mickiewicza	Białystok	9 uczniów
VI LO im. Wacława Sierpińskiego	Gdynia	9 uczniów
II LO im. Marii Konopnickiej	Opole	7 uczniów
XIV LO im. Polonii Belgijskiej	Wrocław	7 uczniów
I LO im. Stefana Żeromskiego	Kielce	5 uczniów
III LO im. Adama Mickiewicza	Wrocław	5 uczniów
I LO im. Stanisława Dubois	Koszalin	4 uczniów
II LO im. Jana III Sobieskiego	Kraków	4 uczniów
I LO im. Stanisława Staszica	Lublin	4 uczniów
L LO im. Ruy Barbosa	Warszawa	4 uczniów
V Liceum Ogólnokształcące	Bielsko–Biała	3 uczniów
I LO im. Juliusza Słowackiego	Chorzów	3 uczniów
LO Zakonu Pijarów	Kraków	3 uczniów
I LO im. Mikołaja Kopernika	Łódź	3 uczniów
VIII LO im. Adama Mickiewicza	Poznań	3 uczniów
I LO im. Komisji Edukacji Narodowej	Sanok	3 uczniów
II LO im. Adama Mickiewicza	Słupsk	3 uczniów
IV LO im. Tadeusza Kościuszki	Toruń	3 uczniów
VIII LO im. Władysława IV	Warszawa	3 uczniów

Najliczniej reprezentowane były miasta:

Kraków	68 zawodników	Chorzów	4 zawodników
Gdynia	52 zawodników	Gorzów Wielkopolski	4 zawodników
Warszawa	40 zawodników	Koszalin	4 zawodników
Bydgoszcz	17 zawodników	Poznań	4 zawodników
Katowice	15 zawodników	Szczecin	4 zawodników
Wrocław	14 zawodników	Bielsko-Biała	3 zawodników
Białystok	11 zawodników	Konin	3 zawodników
Opole	9 zawodników	Łódź	3 zawodników
Kielce	7 zawodników	Olsztyn	3 zawodników
Rzeszów	6 zawodników	Płock	3 zawodników
Lublin	5 zawodników	Sanok	3 zawodników
Słupsk	5 zawodników	Zielona Góra	3 zawodników
Toruń	5 zawodników		

W dniu 12 lutego odbyła się sesja próbna, na której zawodnicy rozwiązywali nie liczące się do ogólnej klasyfikacji zadania „Szkoły” i „Magazyn”. W dniach konkursowych zawodnicy rozwiązywali zadania: „Metro”, „Najazd”, „Listonosz” oraz „Orka”, każde oceniane maksymalnie po 100 punktów.

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

- **SZK — próbne** — Szkoły

	SZK — próbne	
	liczba zawodników	czyli
100 pkt.	5	1,4%
75–99 pkt.	16	4,5%
50–74 pkt.	46	13,0%
1–49 pkt.	95	26,8%
0 pkt.	90	25,3%
brak rozwiązania	103	29,0%

- **MAG — próbne** — Magazyn

	MAG — próbne	
	liczba zawodników	czyli
100 pkt.	6	1,7%
75–99 pkt.	2	0,6%
50–74 pkt.	23	6,5%
1–49 pkt.	228	64,2%
0 pkt.	48	13,5%
brak rozwiązania	48	13,5%

- **MET** — Metro

	MET	
	liczba zawodników	czyli
100 pkt.	9	2,5%
75–99 pkt.	11	3,1%
50–74 pkt.	7	2,0%
1–49 pkt.	116	32,7%
0 pkt.	166	46,8%
brak rozwiązania	46	12,9%

- **NAJ** — Najazd

	NAJ	
	liczba zawodników	czyli
100 pkt.	17	4,8%
75–99 pkt.	3	0,8%
50–74 pkt.	20	5,6%
1–49 pkt.	62	17,5%
0 pkt.	202	56,9%
brak rozwiązania	51	14,4%

- **LIS** — Listonosz

	LIS	
	liczba zawodników	czyli
100 pkt.	3	1,1%
75–99 pkt.	8	2,2%
50–74 pkt.	0	0,0%
1–49 pkt.	93	26,1%
0 pkt.	219	61,6%
brak rozwiązania	32	9,0%

- **ORK** — Orka

	ORK	
	liczba zawodników	czyli
100 pkt.	4	1,1%
75–99 pkt.	0	0,0%
50–74 pkt.	2	0,6%
1–49 pkt.	219	61,7%
0 pkt.	113	31,8%
brak rozwiązania	17	4,8%

W sumie za wszystkie 4 zadania konkursowe rozkład wyników zawodników był następujący:

SUMA	liczba zawodników	czyli
400 pkt.	0	0%
300–399 pkt.	1	0,3%
200–299 pkt.	11	3,1%
100–199 pkt.	33	9,3%
1–99 pkt.	251	70,7%
0 pkt.	59	16,6%

Wszystkim zawodnikom przesłano informację o uzyskanych wynikach, a na stronie Olimpiady dostępne były testy, według których sprawdzano rozwiązania. Poinformowano też dyrekcję szkół o kwalifikacji uczniów do finałów XIII Olimpiady Informatycznej.

ZAWODY III STOPNIA

Zawody III stopnia odbyły się w ośrodku firmy Combidata Poland S.A. w Sopocie w dniach od 28 marca do 1 kwietnia 2006 r.

Do zawodów III stopnia zakwalifikowano 67 najlepszych uczestników zawodów II stopnia, którzy uzyskali wynik nie mniejszy niż 77 pkt.

Zawodnicy uczęszczali do szkół w następujących województwach:

pomorskie	21 uczniów	łódzkie	2 uczniów
małopolskie	10 uczniów	podkarpackie	2 uczniów
śląskie	8 uczniów	zachodniopomorskie	2 uczniów
kujawsko-pomorskie	7 uczniów	opolskie	1 uczeń
mazowieckie	6 uczniów	podlaskie	1 uczeń
dolnośląskie	3 uczniów	świętokrzyskie	1 uczeń
wielkopolskie	3 uczniów		

Niżej wymienione szkoły miały w finale więcej niż jednego zawodnika:

III LO im. Marynarki Wojennej RP	Gdynia	18 uczniów
V LO im. Augusta Witkowskiego	Kraków	9 uczniów
XIV LO im. Stanisława Staszica	Warszawa	4 uczniów
VI LO im. Jana i Jędrzeja Śniadeckich	Bydgoszcz	3 uczniów
V Liceum Ogólnokształcące	Bielsko-Biała	2 uczniów
VI LO im. Wacława Sierpińskiego	Gdynia	2 uczniów
XIII Liceum Ogólnokształcące	Szczecin	2 uczniów
III LO im. Adama Mickiewicza	Wrocław	2 uczniów

28 marca odbyła się sesja próbna, na której zawodnicy rozwiązywali nie liczące się do ogólnej klasyfikacji zadanie: „Tańce w kółkach”. W dniach konkursowych zawodnicy rozwiązywali zadania: „Estetyczny tekst”, „Kryształ”, „Palindromy”, „Zosia” i „Misie”, każde oceniane maksymalnie po 100 punktów.

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

- **TAN — próbne** — Tańce w kółkach

	TAN — próbne	
	liczba zawodników	czyli
100 pkt.	0	0,0%
75–99 pkt.	0	0,0%
50–74 pkt.	4	6,0%
1–49 pkt.	38	56,7%
0 pkt.	14	20,9%
brak rozwiązania	11	16,4%

- **EST** — Estetyczny tekst

	EST	
	liczba zawodników	czyli
100 pkt.	6	8,9%
75–99 pkt.	3	4,5%
50–74 pkt.	15	22,4%
1–49 pkt.	36	53,7%
0 pkt.	5	7,5%
brak rozwiązania	2	3,0%

- **KRY** — Kryształ

	KRY	
	liczba zawodników	czyli
100 pkt.	3	4,5%
75–99 pkt.	2	3,0%
50–74 pkt.	1	1,5%
1–49 pkt.	46	68,6%
0 pkt.	12	17,9%
brak rozwiązania	3	4,5%

- **PAL** — Palindromy

	PAL	
	liczba zawodników	czyli
100 pkt.	2	3,0%
75–99 pkt.	1	1,5%
50–74 pkt.	1	1,5%
1–49 pkt.	48	71,6%
0 pkt.	11	16,4%
brak rozwiązania	4	6,0%

- **ZOS** — Zosia

	ZOS	
	liczba zawodników	czyli
100 pkt.	1	1,5%
75–99 pkt.	5	7,5%
50–74 pkt.	10	14,9%
1–49 pkt.	21	31,3%
0 pkt.	14	20,9%
brak rozwiązania	16	23,9%

- **MIS** — Misie

	MIS	
	liczba zawodników	czyli
100 pkt.	12	17,9%
75–99 pkt.	2	3,0%
50–74 pkt.	4	6,0%
1–49 pkt.	17	25,4%
0 pkt.	23	34,3%
brak rozwiązania	9	13,4%

W sumie za wszystkie 5 zadań konkursowych rozkład wyników zawodników był następujący:

SUMA	liczba zawodników	czyli
500 pkt.	1	1,5%
375–499 pkt.	1	1,5%
250–374 pkt.	5	7,5%
125–249 pkt.	21	31,3%
1–124 pkt.	36	53,7%
0 pkt.	3	4,5%

W dniu 1 kwietnia 2006 roku, w Sali Posiedzeń Urzędu Miasta w Sopocie, ogłoszono wyniki finału XIII Olimpiady Informatycznej 2005/2006 i rozdano nagrody ufundowane przez: PROKOM Software S.A., Wydawnictwa Naukowo-Techniczne, Ogólnopolską Fundację Edukacji Komputerowej i Olimpiadę Informatyczną.

Poniżej zestawiono listę wszystkich laureatów i finalistów:

(1) **Filip Wolski**, III LO im. Marynarki Wojennej RP w Gdyni — laureat I miejsca, złoty medal, 500 pkt. (puchar — Olimpiada Informatyczna; notebook — PROKOM; roczny abonament na książki — WNT), Filip Wolski otrzymał także puchar ufundowany przez Olimpiadę Informatyczną za całokształt swoich osiągnięć, zarówno w krajowych jak i międzynarodowych Olimpiadach Informatycznych.

(2) **Jakub Kallas**, III LO im. Marynarki Wojennej RP w Gdyni — laureat I miejsca, złoty medal, 457 pkt. (notebook — PROKOM)

(3) **Marcin Andrychowicz**, XIV LO im. Stanisława Staszica w Warszawie — laureat II miejsca, srebrny medal, 296 pkt. (notebook — PROKOM)

(4) **Michał Pilipczuk**, XIV LO im. Stanisława Staszica w Warszawie — laureat II miejsca, srebrny medal, 253 pkt. (aparat cyfrowy — PROKOM i Olimpiada Informatyczna)

(5) **Wojciech Tyczyński**, VI LO im. J. J. Śniadeckich w Bydgoszczy — laureat II miejsca, srebrny medal, 251 pkt. (aparat cyfrowy — PROKOM i Olimpiada Informatyczna)

(6) **Mateusz Rukowicz**, III LO im. A. Mickiewicza we Wrocławiu — laureat II miejsca, srebrny medal, 250 pkt. (aparat cyfrowy — PROKOM i Olimpiada Informatyczna)

(7) **Wojciech Śmietanka**, III LO im. Marynarki Wojennej RP w Gdyni — laureat II miejsca, srebrny medal, 250 pkt. (aparat cyfrowy — PROKOM i Olimpiada Informatyczna)

(8) **Adam Gawarkiewicz**, X LO im. prof. S. Banacha w Toruniu — laureat II miejsca, srebrny medal, 243 pkt. (aparat cyfrowy — PROKOM i Olimpiada Informatyczna)

(9) **Maciej Klimek**, II Liceum Ogólnokształcące w Gorzowie Wielkopolskim — laureat II miejsca, srebrny medal, 240 pkt. (aparat cyfrowy — PROKOM i Olimpiada Informatyczna)

(10) **Robert Obryk**, V LO im. Augusta Witkowskiego w Krakowie — laureat II miejsca, srebrny medal, 240 pkt. (aparat cyfrowy — PROKOM i Olimpiada Informatyczna)

(11) **Piotr Szmigiel**, V LO im. Augusta Witkowskiego w Krakowie — laureat II miejsca, srebrny medal, 240 pkt. (aparat cyfrowy — PROKOM i Olimpiada Informatyczna)

(12) **Tomasz Kulczyński**, VI LO im. J. J. Śniadeckich w Bydgoszczy — laureat II miejsca, srebrny medal, 234 pkt. (odtwarzacz mp3 — PROKOM)

(13) **Piotr Świgoń**, I LO im. Jana Kasprowicza w Inowrocławiu — laureat II miejsca, srebrny medal, 229 pkt. (odtwarzacz mp3 — PROKOM)

(14) **Bartłomiej Wołowiec**, V Liceum Ogólnokształcące w Bielsku-Białej — laureat II miejsca, srebrny medal, 221 pkt. (odtwarzacz mp3 — PROKOM)

(15) **Bartosz Janiak**, I LO im. Mikołaja Kopernika w Łodzi — laureat III miejsca, brązowy medal, 205 pkt. (odtwarzacz mp3 — PROKOM)

(16) **Marcin Skotniczny**, V LO im. Augusta Witkowskiego w Krakowie — laureat III miejsca, brązowy medal, 192 pkt. (odtwarzacz mp3 — PROKOM)

(17) **Łukasz Wołochowski**, III LO im. Marynarki Wojennej RP w Gdyni — laureat III miejsca, brązowy medal, 189 pkt. (odtwarzacz mp3 — PROKOM)

(18) **Marian Marek Kędzierski**, XIII Liceum Ogólnokształcące w Szczecinie — laureat III miejsca, brązowy medal, 174 pkt. (odtwarzacz mp3 — PROKOM)

(19) **Marcin Kurczych**, I LO im. Stefana Żeromskiego w Kielcach — laureat III miejsca, brązowy medal, 173 pkt. (odtwarzacz mp3 — PROKOM)

(20) **Szymon Wrzyszcz**, II LO im. Jana III Sobieskiego w Grudziądzu — laureat III miejsca, brązowy medal, 165 pkt. (odtwarzacz mp3 — PROKOM)

(21) **Juliusz Kopczewski**, 2 Społeczne LO w Warszawie — laureat III miejsca, brązowy medal, 155 pkt. (odtwarzacz mp3 — PROKOM)

(22) **Maciej Łaszcz**, III LO im. Marynarki Wojennej RP w Gdyni — laureat III miejsca, brązowy medal, 155 pkt. (odtwarzacz mp3 — PROKOM)

(23) **Marek Wróbel**, V LO im. Augusta Witkowskiego w Krakowie — laureat III miejsca, brązowy medal, 135 pkt. (odtwarzacz mp3 — OFEK)

(24) **Błażej Osiński**, VI LO im. Jana i Jędrzeja Śniadeckich w Bydgoszczy — laureat III miejsca, brązowy medal, 133 pkt. (odtwarzacz mp3 — OFEK)

(25) **Radosław Kożuch**, V LO im. Augusta Witkowskiego w Krakowie — laureat III miejsca, brązowy medal, 132 pkt. (odtwarzacz mp3 — OFEK)

(26) **Jacek Migdał**, V Liceum Ogólnokształcące w Bielsku-Białej — laureat III miejsca, brązowy medal, 128 pkt. (odtwarzacz mp3 — OFEK)

(27) **Kamil Nowosad**, I LO im. Edwarda Dembowskiego w Gliwicach — laureat III miejsca, brązowy medal, 127 pkt. (odtwarzacz mp3 — OFEK)

(28) **Marek Białowąs**, III LO im. Marynarki Wojennej RP w Gdyni — laureat III miejsca, brązowy medal, 125 pkt. (odtwarzacz mp3 — OFEK)

(29) **Mirosław Michalski**, III LO im. Marynarki Wojennej RP w Gdyni — laureat III miejsca, brązowy medal, 119 pkt. (odtwarzacz mp3 — OFEK)

(30) **Przemysław Witek**, I LO im. Karola Miarki w Żorach — laureat III miejsca, brązowy medal, 111 pkt. (odtwarzacz mp3 — OFEK)

Lista pozostałych finalistów w kolejności alfabetycznej:

- **Marcin Babij**, IX Liceum Ogólnokształcące we Wrocławiu
- **Wojciech Baranowski**, III LO im. Marynarki Wojennej RP w Gdyni
- **Jarosław Błasiok**, Zespół Szkół (Gimnazjum) w Gostyniu
- **Michał Chruszcz**, I LO im. Komisji Edukacji Narodwej w Sanoku
- **Elżbieta Dłutowska**, III LO im. Marynarki Wojennej RP w Gdyni
- **Tomasz Dubrownik**, III LO im. Marynarki Wojennej RP w Gdyni
- **Artur Dwornik**, I LO im. Tadeusza Kościuszki w Koninie
- **Przemysław Gajda**, V LO im. Augusta Witkowskiego w Krakowie
- **Adrian Galewski**, Zespół Szkół Technicznych w Wodzisławiu Śląskim
- **Bartosz Gęza**, III LO im. Marynarki Wojennej RP w Gdyni

- **Tomasz Gogacz**, III LO im. Marynarki Wojennej RP w Gdyni
- **Piotr Gurgul**, V LO im. Augusta Witkowskiego w Krakowie
- **Kamil Herba**, XIII Liceum Ogólnokształcące w Szczecinie
- **Michał Jastrzębski**, XIV LO im. Stanisława Staszica w Warszawie
- **Michał Kijewski**, III LO im. Marynarki Wojennej RP w Gdyni
- **Marek Kiszkis**, III LO im. Marynarki Wojennej RP w Gdyni
- **Marcin Kościelnicki**, I LO im. Juliusza Slowackiego w Chorzowie
- **Artur Koniński**, II Liceum ogólnokształcące w Koninie
- **Maciej Kruk**, V LO im. Augusta Witkowskiego w Krakowie
- **Krzysztof Krygiel**, IV LO im. Tadeusza Kościuszki w Toruniu
- **Wojciech Jakub Leja**, LO WSIiZ w Rzeszowie
- **Wojciech Łowiec**, III LO im. Marynarki Wojennej RP w Gdyni
- **Marek Marczykowski**, XIV LO im. Stanisława Staszica w Warszawie
- **Paweł Młocek**, LO im. Marii Skłodowskiej-Curie w Andrychowie
- **Przemysław Pietrzkiewicz**, II LO im. Marii Konopnickiej w Katowicach
- **Andrzej Redlarski**, Gdańskie Liceum Autonomiczne w Gdańsku
- **Marcin Sobczyk**, V LO im. Augusta Witkowskiego w Krakowie
- **Patryk Spanily**, III LO im. Marynarki Wojennej RP w Gdyni
- **Paweł Jędrzej Sroka**, III LO im. A. Mickiewicza we Wrocławiu
- **Juliusz Stasiewicz**, I LO im. A. Mickiewicza w Białymstoku
- **Bartosz Szreder**, VI LO im. Wacława Sierpińskiego w Gdyni
- **Tomasz Śniatowski**, II LO im. Marii Konopnickiej w Opolu
- **Paweł Wiejacha**, VI LO im. Wacława Sierpińskiego w Gdyni
- **Mateusz Włoch**, III LO im. Marynarki Wojennej RP w Gdyni
- **Przemysław Zych**, LO im. Bolesława Prusa w Skierniewicach
- **Tomasz Żołnowski**, LO im. Stanisława Małachowskiego w Płocku
- **Artur Żylinski**, III LO im. Marynarki Wojennej RP w Gdyni

Wszyscy laureaci i finaliści otrzymali książki ufundowane przez Wydawnictwa Naukowo-Techniczne.

Ogłoszono komunikat o powołaniu reprezentacji Polski na:

- Międzynarodową Olimpiadę Informatyczną IOI'2006 — Meksyk, Merida, 13–20 sierpnia 2006 r.
  (1) Filip Wolski
  (2) Jakub Kallas
  (3) Marcin Andrychowicz
  (4) Michał Pilipczuk

  rezerwowi:

  (5) Wojciech Tyczyński
  (6) Mateusz Rukowicz
  (7) Wojciech Śmietanka
- Olimpiadę Informatyczną Krajów Europy Środkowej CEOI'2006 — Chorwacja, Vrsar, 1–8 lipca 2006 r.
  (1) Filip Wolski
  (2) Jakub Kallas
  (3) Marcin Andrychowicz
  (4) Michał Pilipczuk

  rezerwowi:

  (5) Wojciech Tyczyński
  (6) Mateusz Rukowicz
  (7) Wojciech Śmietanka
- Bałtycką Olimpiadę Informatyczną BOI'2006 — Finlandia, Heinola, 18–22 maja 2006 r.
  (1) Jakub Kallas
  (2) Marcin Andrychowicz
  (3) Wojciech Śmietanka
  (4) Maciej Klimek
  (5) Robert Obryk
  (6) Tomasz Kulczyński

  rezerwowi:

  (7) Łukasz Wołochowski

    (8) Marcin Kurczych

- Obóz czesko-polsko-słowacki, Warszawa, 12–18 czerwca 2006 r.:
  - członkowie reprezentacji oraz zawodnicy rezerwowi powołani na Międzynarodową Olimpiadę Informatyczną (IOI'2006) i Olimpiadę Informatyczną Krajów Europy Środkowej (CEOI'2006).
- obóz rozwojowo-treningowy im. A. Kreczmara, 23–30 lipca 2006 r.:
  - reprezentanci na międzynarodowe zawody informatyczne, zawodnicy rezerwowi oraz wszyscy laureaci i finaliści, którzy uczęszczają do klas niższych niż maturalna.

Sekretariat wystawił łącznie 30 zaświadczeń o uzyskaniu tytułu laureata i 37 zaświadczeń o uzyskaniu tytułu finalisty XIII Olimpiady Informatycznej.

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

- Paweł Afelt (II Społeczne Liceum Ogólnokształcące w Warszawie)
  - Juliusz Kopczewski — laureat III miejsca
- Wiesława Amietszajew (II Liceum Ogólnokształcące w Koninie)
  - Artur Koniński — finalista
- Agnieszka Antas-Kucypera (IX Liceum Ogólnokształcące we Wrocławiu)
  - Marcin Babij — finalista
- Krzysztof Błasiok (Gostyń)
  - Jarosław Błasiok — finalista
- Ireneusz Bujnowski (I Liceum Ogólnokształcące im. Adama Mickiewicza w Białymstoku)
  - Juliusz Stasiewicz — finalista
- Czesław Drozdowski (XIII Liceum Ogólnokształcące w Szczecinie)
  - Marian Marek Kędzierski — laureat III miejsca
  - Kamil Herba — finalista
- Marek Dwurznik (Liceum Ogólnokształcące im. Bolesława Prusa w Skierniewicach)
  - Przemysław Zych — finalista
- Andrzej Dyrek (V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie)
  - Robert Obryk — laureat II miejsca
  - Marcin Skotniczny — laureat III miejsca

  - Marek Wróbel — laureat III miejsca
  - Przemysław Gajda — finalista
  - Marcin Sobczyk — finalista
  - Radosław Kożuch — laureat III miejsca

- Mirosława Firszt (IV Liceum Ogólnokształcące w Toruniu)
  - Krzysztof Krygiel — finalista
- Janusz Grabowski (Liceum Ogólnokształcące im. Stanisława Małachowskiego w Płocku)
  - Tomasz Żołnowski — finalista
- Henryk Gurgul (Akademia Górniczo-Hutnicza w Krakowie)
  - Piotr Gurgul — finalista
- Grzegorz Gutowski (V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie)
  - Maciej Kruk — finalista
- Adam Herman (I Liceum Ogólnokształcące im. K. Miarki w Żorach)
  - Przemysław Witek — laureat III miejsca
- Piotr Kita (I Liceum Ogólnokształcące im. KEN w Sanoku)
  - Michał Chruszcz — finalista
- Kamil Kloch (V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie)
  - Piotr Szmigiel — laureat II miejsca
  - Piotr Gurgul — finalista
- Anna Kowalska (V Liceum Ogólnokształcące w Bielsku-Białej
  - Bartłomiej Wołowiec — laureat II miejsca
  - Jacek Migdał — laureat III miejsca
- Jarosław Kruszyński (I Liceum Ogólnokształcące im. Jana Kasprowicza w Inowrocławiu)
  - Piotr Świgoń — laureat II miejsca
- Janusz Kurczych (Bank Spółdzielczy w Kielcach)
  - Marcin Kurczych — laureat III miejsca
- Piotr Kaźmierczyk (II Liceum Ogólnokształcące w Gorzowie Wielkopolskim)
  - Maciej Klimek laureat II miejsca

- Wojciech Kwaśny (I Liceum Ogólnokształcące im. Juliusza Słowackiego w Chorzowie)
    - Marcin Kościelnicki — finalista
- Ryszard Lisoń (II Liceum Ogólnokształcące w Opolu)
    - Tomasz Śniatowski — finalista
- Paweł Mateja (I Liceum Ogólnokształcące im. Mikołaja Kopernika w Łodzi)
    - Bartosz Janiak — laureat III miejsca
- Grzegorz Owsiany (Wyższa Szkoła Informatyki i Zarządzania w Rzeszowie)
    - Wojciech Jakub Leja — finalista
- Marcin Pilipczuk (student Uniwersytetu Warszawskiego)
    - Michał Pilipczuk — laureat II miejsca
- Ryszard Popardowski (Zespół Szkół Technicznych w Wodzisławiu Śląskim)
    - Adrian Galewski — finalista
- Włodzimierz Raczek (V Liceum Ogólnokształcące w Bielsku-Białej)
    - Jacek Migdał — laureat III miejsca
    - Bartłomiej Wołowiec — laureat II miejsca
- Hanna Stachera (XIV Liceum Ogólnokształcące im. Stanisława Staszica w Warszawie)
    - Marcin Andrychowicz — laureat II miejsca
- Henryk Szantula (II Liceum Ogólnokształcące im. Marii Konopnickiej w Katowicach)
    - Przemysław Pietrzkiewicz — finalista
- Ryszard Szubartowski (III Liceum Ogólnokształcące im. Marynarki Wojennej RP w Gdyni)
    - Filip Wolski — laureat I miejsca
    - Jakub Kallas — laureat I miejsca
    - Wojciech Śmietanka — laureat II miejsca
    - Łukasz Wołochowski — laureat III miejsca
    - Maciej Łaszcz — laureat III miejsca
    - Marek Białowąs — laureat III miejsca
    - Mirosław Michalski — laureat III miejsca
    - Wojciech Baranowski — finalista
    - Elżbieta Dłutowska — finalista

  - Tomasz Dubrownik — finalista
  - Bartosz Gęza — finalista
  - Tomasz Gogacz — finalista
  - Michał Kijewski — finalista
  - Marek Kiszkis — finalista
  - Wojciech Łowiec — finalista
  - Patryk Spanily — finalista
  - Bartosz Szreder — finalista
  - Paweł Wiejacha — finalista
  - Mateusz Włoch — finalista
  - Artur Żylinski — finalista
- Joanna Śmigielska (XIV Liceum Ogólnokształcące im. St. Staszica w Warszawie)
  - Michał Jastrzębski — finalista
  - Marek Marczykowski — finalista
- Paweł Walter (V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie)
  - Robert Obryk — laureat II miejsca
  - Marcin Skotniczny — laureat III miejsca
- Iwona Waszkiewicz (VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich w Bydgoszczy)
  - Wojciech Tyczyński — laureat II miejsca
  - Tomasz Kulczyński — laureat II miejsca
  - Błażej Osiński — laureat III miejsca
- Paweł Woźniak (Liceum Ogólnokształcące im. Marii Skłodowskiej-Curie w Andrychowie)
  - Paweł Młocek — finalista

Zgodnie z decyzją Komitetu Głównego z dn. 31 marca 2006 nauczyciele i opiekunowie naukowi laureatów i finalistów otrzymają nagrody pieniężne.

Warszawa, 09 czerwca 2006 roku

Regulamin Olimpiady Informatycznej

§1 WSTĘP

Olimpiada Informatyczna jest olimpiadą przedmiotową powołaną przez Instytut Informatyki Uniwersytetu Wrocławskiego, w dniu 10 grudnia 1993 roku. Olimpiada działa zgodnie z Rozporządzeniem Ministra Edukacji Narodowej i Sportu z dnia 29 stycznia 2002 roku w sprawie organizacji oraz sposobu przeprowadzenia konkursów, turniejów i olimpiad (Dz. U. 02.13.125). Organizatorem Olimpiady Informatycznej jest Uniwersytet Wrocławski. W organizacji Olimpiady Uniwersytet Wrocławski współdziała ze środowiskami akademickimi, zawodowymi i oświatowymi działającymi w sprawach edukacji informatycznej.

§2 CELE OLIMPIADY

(1) Stworzenie motywacji dla zainteresowania nauczycieli i uczniów informatyką.

(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 i inne międzynarodowe zawody informatyczne.

§3 ORGANIZACJA OLIMPIADY

(1) Olimpiadę przeprowadza Komitet Główny Olimpiady Informatycznej, zwany dalej Komitetem.

(2) Olimpiada jest trójstopniowa.

(3) W Olimpiadzie mogą brać indywidualnie udział uczniowie wszystkich typów szkół ponadgimnazjalnych i szkół średnich dla młodzieży, dających możliwość uzyskania matury.

(4) W Olimpiadzie mogą również uczestniczyć — za zgodą Komitetu Głównego — uczniowie szkół podstawowych, gimnazjów, zasadniczych szkół zawodowych i szkół zasadniczych.

(5) Zawody I stopnia mają charakter otwarty i polegają na samodzielnym rozwiązywaniu przez uczestnika zadań ustalonych dla tych zawodów oraz przekazaniu rozwiązań w podanym terminie, w miejsce i w sposób określony w „Zasadach organizacji zawodów", zwanych dalej Zasadami.

(6) Zawody II stopnia są organizowane przez komitety okręgowe Olimpiady lub instytucje upoważnione przez Komitet Główny.

(7) 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. Zawody poprzedzone są sesją próbną, której rezultaty nie liczą się do wyników zawodów.

(8) Liczbę uczestników kwalifikowanych do zawodów II i III stopnia ustala Komitet Główny i podaje ją w Zasadach.

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

(10) Rozwiązaniem zadania zawodów I, II i III stopnia są, zgodnie z treścią zadania, dane lub program. Program powinien być napisany w języku programowania i środowisku wybranym z listy języków i środowisk ustalanej przez Komitet Główny i ogłaszanej w Zasadach.

(11) Rozwiązania są oceniane automatycznie. Jeśli rozwiązaniem zadania jest program, to jest on uruchamiany na testach z przygotowanego zestawu.
Podstawą oceny jest zgodność sprawdzanego programu z podaną w treści zadania specyfikacją, poprawność wygenerowanego przez program wyniku oraz czas działania tego programu. Jeśli rozwiązaniem zadania jest plik z danymi, wówczas ocenia się poprawność danych i na tej podstawie przyznaje punkty.

(12) Rozwiązania zespołowe, niesamodzielne, niezgodne z „Zasadami organizacji zawodów" lub takie, co do których nie można ustalić autorstwa,nie będą oceniane.

(13) Każdy zawodnik jest zobowiązany do zachowania w tajemnicy swoich rozwiązań w czasie trwania zawodów.

(14) W szczególnie rażących wypadkach łamania Regulaminu i Zasad Komitet Główny może zdyskwalifikować zawodnika.

(15) Komitet Główny przyjął następujący tryb opracowywania zadań olimpijskich:

  (a) Autor zgłasza propozycję zadania, które powinno być oryginalne i nieznane, do sekretarza naukowego Olimpiady.

(b) Zgłoszona propozycja zadania jest opiniowana, redagowana, opracowywana wraz z zestawem testów, a opracowania podlegają niezależnej weryfikacji. Zadanie, które uzyska negatywną opinię może zostać odrzucone lub skierowane do ponownego opracowania.

(c) Wyboru zestawu zadań na zawody dokonuje Komitet Główny, spośród zadań, które zostały opracowane i uzyskały pozytywną opinię.

(d) Wszyscy uczestniczący w procesie przygotowania zadania są zobowiązani do zachowania tajemnicy do czasu jego wykorzystania w zawodach lub ostatecznego odrzucenia.

§4 KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

(1) Komitet Główny jest odpowiedzialny za poziom merytoryczny i organizację zawodów. Komitet składa corocznie organizatorowi sprawozdanie z przeprowadzonych zawodów.

(2) W skład Komitetu wchodzą nauczyciele akademiccy, nauczyciele szkół ponadgimnazjalnych i ponadpodstawowych oraz pracownicy oświaty związani z kształceniem informatycznym.

(3) Komitet wybiera ze swego grona Prezydium na kadencję trzyletnią. Prezydium 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 dokonuje zmian w swoim składzie za zgodą Organizatora.

(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 skład reprezentacji na Międzynarodową Olimpiadę Informatyczną i inne międzynarodowe zawody informatyczne.

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

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

(9) Do organizacji zawodów II stopnia w miejscowościach, których nie obejmuje żaden komitet okręgowy, Komitet powołuje komisję zawodów co najmniej trzy miesiące przed terminem rozpoczęcia zawodów.

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

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

(12) Komitet zatwierdza plan finansowy dla każdej edycji Olimpiady na pierwszym posiedzeniu Komitetu w nowym roku szkolnym.

(13) Komitet przyjmuje sprawozdanie finansowe z każdej edycji Olimpiady w ciągu czterech miesięcy od zakończenia danej edycji.

(14) Komitet ma siedzibę w Ośrodku Edukacji Informatycznej i Zastosowań Komputerów w Warszawie. Ośrodek wspiera Komitet we wszystkich działaniach organizacyjnych zgodnie z Deklaracją z dnia 8 grudnia 1993 roku przekazaną Organizatorowi.

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

(16) 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.

(17) 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.

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

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

§6 PRZEBIEG OLIMPIADY

(1) Komitet rozsyła do szkół wymienionych w § 3.3 oraz kuratoriów oświaty i koordynatorów edukacji informatycznej treści zadań I stopnia wraz z Zasadami.

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

(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 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) Laureaci i finaliści Olimpiady otrzymują z informatyki lub technologii informacyjnej celującą roczną (semestralną) ocenę klasyfikacyjną.

(2) Laureaci i finaliści Olimpiady są zwolnieni z egzaminu maturalnego z informatyki. Uprawnienie to przysługuje także wtedy, gdy przedmiot nie był objęty szkolnym planem nauczania danej szkoły.

(3) Uprawnienia określone w p. 1. i 2. przysługują na zasadach określonych w rozporządzeniu MENiS z dnia 7 września 2004 r. w sprawie warunków i sposobu oceniania, klasyfikowania i promowania uczniów i słuchaczy oraz przeprowadzania sprawdzianów i egzaminów w szkołach publicznych. (Dz. U. z 2004 r. Nr 199, poz. 2046, §§ 18 i 56).

(4) Laureaci i finaliści Olimpiady mają ułatwiony lub wolny wstęp do tych szkół wyższych, których senaty podjęły uchwały w tej sprawie, zgodnie z przepisami ustawy z dnia 12 września 1990 r. o szkolnictwie wyższym, na zasadach zawartych w tych uchwałach (Dz. U. z 1990 r. Nr 65 poz. 385, Art. 141).

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

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

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

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

§8 FINANSOWANIE OLIMPIADY

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

§9 PRZEPISY KOŃCOWE

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

(2) Komitet zatwierdza sprawozdanie merytoryczne z przeprowadzonej edycji Olimpiady w ciągu 3 miesięcy po zakończeniu zawodów III stopnia i przedstawia je Organizatorowi i Ministerstwu Edukacji Narodowej i Sportu.

(3) Niniejszy regulamin może być zmieniony przez Komitet tylko przed rozpoczęciem kolejnej edycji zawodów Olimpiady po zatwierdzeniu zmian przez Organizatora.

Warszawa, 3 września 2005 r.

Komitet Główny Olimpiady Informatycznej

Zasady organizacji zawodów w roku szkolnym 2005/2006

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 2005/2006.

§1 WSTĘP

Olimpiada Informatyczna jest olimpiadą przedmiotową powołaną przez Instytut Informatyki Uniwersytetu Wrocławskiego, w dniu 10 grudnia 1993 roku. Olimpiada działa zgodnie z Rozporządzeniem Ministra Edukacji Narodowej i Sportu z dnia 29 stycznia 2002 roku w sprawie organizacji oraz sposobu przeprowadzenia konkursów, turniejów i olimpiad (*Dz. U.* 02.13.125). Organizatorem Olimpiady Informatycznej jest Uniwersytet Wrocławski. W organizacji Olimpiady Uniwersytet Wrocławski współdziała ze środowiskami akademickimi, zawodowymi i oświatowymi działającymi w sprawach edukacji informatycznej.

§2 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 ponadgimnazjalnych i szkół średnich dla młodzieży dających możliwość uzyskania matury. W Olimpiadzie mogą również uczestniczyć — za zgodą Komitetu Głównego — uczniowie szkół podstawowych, gimnazjów, zasadniczych szkół zawodowych i szkół zasadniczych.

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

(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 300 uczestników, których rozwiązania zadań I stopnia zostaną ocenione najwyżej; do zawodów III stopnia —

60 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, zajętych miejscach i przyznanych nagrodach oraz składzie polskiej reprezentacji na Międzynarodową Olimpiadę Informatyczną i inne międzynarodowe zawody informatyczne są ostateczne.

(9) Terminarz zawodów:

- zawody I stopnia — 24.10–21.11.2005 r.

  ogłoszenie wyników:
  - w witrynie Olimpiady — 16.12.2005 r.,
  - pocztą — 29.12.2005 r.
- zawody II stopnia — 14–16.02.2006 r.

  ogłoszenie wyników:
  - w witrynie Olimpiady — 24.02.2006 r.
  - pocztą — 03.03.2006 r.
- zawody III stopnia — 28.03–01.04.2006 r.

§3 WYMAGANIA DOTYCZĄCE ROZWIĄZAŃ ZADAŃ ZAWODÓW I STOPNIA

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

- Poprzez witrynę Olimpiady o adresie: `www.oi.edu.pl` do godziny 12:00 (w południe) dnia 21 listopada 2005 r. Komitet Główny nie ponosi odpowiedzialności za brak możliwości przekazania rozwiązań przez witrynę w sytuacji nadmiernego obciążenia lub awarii serwisu. Odbiór przesyłki zostanie potwierdzony przez Komitet Główny zwrotnym listem elektronicznym (prosimy o zachowanie tego listu). Brak potwierdzenia może oznaczać, że rozwiązanie nie zostało poprawnie zarejestrowane. W tym przypadku zawodnik powinien przesłać swoje rozwiązanie przesyłką poleconą za pośrednictwem zwykłej poczty. Szczegóły dotyczące sposobu postępowania przy przekazywaniu zadań i związanej z tym rejestracji będą podane w witrynie.
- Pocztą, przesyłką poleconą, na adres:

Olimpiada Informatyczna
Ośrodek Edukacji Informatycznej i Zastosowań Komputerów
ul. Nowogrodzka 73
02-006 Warszawa
tel. (0-22) 626-83-90

w nieprzekraczalnym terminie nadania do 21 listopada 2005 r. (decyduje data stempla pocztowego). Prosimy o zachowanie dowodu nadania przesyłki.

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

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

(3) Rozwiązania zespołowe, niesamodzielne, niezgodne z „Zasadami organizacji zawodów” lub takie, co do których nie można ustalić autorstwa, nie będą oceniane.

(4) Każdy zawodnik jest zobowiązany do zachowania w tajemnicy swoich rozwiązań w czasie trwania zawodów.

(5) Rozwiązanie każdego zadania, które polega na napisaniu programu, składa się z (tylko jednego) pliku źródłowego; imię i nazwisko uczestnika muszą być podane w komentarzu na początku każdego programu.

(6) Nazwy plików z programami w postaci źródłowej muszą być takie jak podano w treści zadania. Nazwy tych plików muszą mieć następujące rozszerzenia zależne od użytego języka programowania:

Pascal	`.pas`
C	`.c`
C++	`.cpp`

(7) Programy w *C/C++* będą kompilowane w systemie Linux za pomocą kompilatora GCC/G++ v. 3.4.4.
Programy w Pascalu będą kompilowane w systemie Linux za pomocą kompilatora FreePascal v. 2.0.0. Wybór polecenia kompilacji zależy od podanego rozszerzenia pliku w następujący sposób (np. dla zadania *abc*):

Dla `c`	`gcc -O2 -static abc.c -lm`
Dla `cpp`	`g++ -O2 -static abc.cpp -lm`
Dla `pas`	`ppc386 -O2 -XS -Xt abc.pas`

Pakiety instalacyjne tych kompilatorów (i ich wersje dla DOS/Windows) są dostępne w witrynie Olimpiady `www.oi.edu.pl`.

(8) Program powinien odczytywać dane wejściowe ze standardowego wejścia i zapisywać dane wyjściowe na standardowe wyjście, chyba że dla danego zadania wyraźnie napisano inaczej.

(9) Należy przyjąć, że dane testowe są bezbłędne, zgodne z warunkami zadania i podaną specyfikacją wejścia.

(10) Uczestnik korzystający z poczty zwykłej przysyła:

- Nośnik (dyskietkę lub CD-ROM) w standardzie dla komputerów PC, zawierający:
  - spis zawartości nośnika oraz dane osobowe zawodnika w pliku nazwanym `SPIS.TXT`,
  - do każdego rozwiązanego zadania — program źródłowy lub plik z danymi.

  Na nośniku nie powinno być żadnych podkatalogów.
- Wypełniony dokument zgłoszeniowy (dostępny w witrynie internetowej Olimpiady). Należy podać adres elektroniczny. Podanie adresu jest niezbędne do wzięcia udziału w procedurze reklamacyjnej opisanej w punktach 14, 15 i 16.

(11) Uczestnik korzystający z witryny Olimpiady postępuje zgodnie z instrukcjami umieszczonymi w witrynie.

(12) W witrynie Olimpiady znajdują się *Odpowiedzi na pytania zawodników* dotyczące Olimpiady. Ponieważ *Odpowiedzi* mogą zawierać ważne informacje dotyczące toczących się zawodów wszyscy uczestnicy Olimpiady proszeni są o regularne zapoznawanie się z ukazującymi się odpowiedziami. Dalsze pytania należy przysyłać poprzez witrynę Olimpiady. Komitet Główny może nie udzielić odpowiedzi na pytanie z ważnych przyczyn, m.in. gdy jest ono niejednoznaczne lub dotyczy sposobu rozwiązania zadania.

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

(14) Od dnia 5 grudnia 2005 r. poprzez witrynę Olimpiady każdy zawodnik będzie mógł zapoznać się ze wstępną oceną swojej pracy. Wstępne oceny będą dostępne jedynie w witrynie Olimpiady i tylko dla osób, które podały adres elektroniczny.

(15) Do dnia 9 grudnia 2005 r. (włącznie) poprzez witrynę Olimpiady każdy zawodnik będzie mógł zgłaszać uwagi do wstępnej oceny swoich rozwiązań. Reklamacji nie podlega jednak dobór testów, limitów czasowych, kompilatorów i sposobu oceny.

(16) Reklamacje złożone po 9 grudnia 2005 r. nie będą rozpatrywane.

§4 UPRAWNIENIA I NAGRODY

(1) Laureaci i finaliści Olimpiady otrzymują z informatyki lub technologii informacyjnej celującą roczną (semestralną) ocenę klasyfikacyjną.

(2) Laureaci i finaliści Olimpiady są zwolnieni z egzaminu maturalnego z informatyki. Uprawnienie to przysługuje także wtedy, gdy przedmiot nie był objęty szkolnym planem nauczania danej szkoły.

(3) Uprawnienia określone w p. 1. i 2. przysługują na zasadach określonych w rozporządzeniu MENiS z dnia 7 września 2004 r. w sprawie warunków i sposobu oceniania, klasyfikowania i promowania uczniów i słuchaczy oraz przeprowadzania sprawdzianów i egzaminów w szkołach publicznych. (Dz. U. z 2004 r. Nr 199, poz. 2046, §§ 18 i 56) wraz z późniejszymi zmianami zawartymi w Rozporządzeniu MENiS z dnia 16 czerwca 2005 r. (Dz. U. z 2005 r. Nr 108, poz. 905).

(4) Laureaci i finaliści Olimpiady mają ułatwiony lub wolny wstęp do tych szkół wyższych, których senaty podjęły uchwały w tej sprawie, zgodnie z przepisami ustawy z dnia 12 września 1990 r. o szkolnictwie wyższym, na zasadach zawartych w tych uchwałach (Dz.U. z 2005 r. Nr 164 poz. 1365).

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

(6) Komitet Główny ustala skład reprezentacji Polski na XVIII Międzynarodową Olimpiadę Informatyczną w 2006 roku na podstawie wyników zawodów III stopnia i regulaminu tej Olimpiady Międzynarodowej.

(7) Komitet Główny może przyznać nagrodę nauczycielowi lub opiekunowi naukowemu, który przygotował laureata lub finalistę Olimpiady.

(8) 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 VII Obozie Naukowo-Treningowym im. Antoniego Kreczmara, który odbędzie się w czasie wakacji 2006 r.

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

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

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

Witryna Olimpiady: `www.oi.edu.pl`

Zawody I stopnia

opracowania zadań

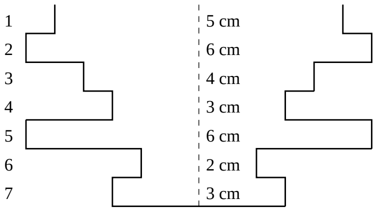
Jakub Radoszewski
Treść zadania, Opracowanie

Michał Adamaszek
Program

OI, Etap I

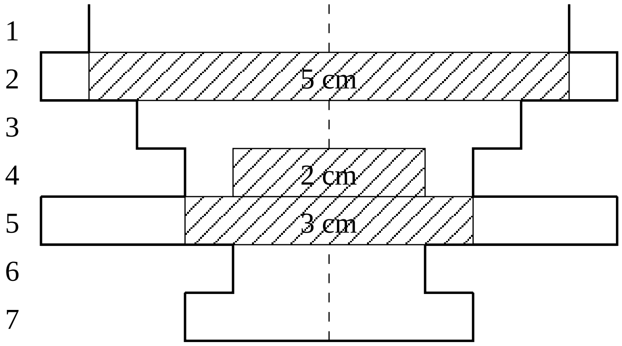
Krążki

Mały Jaś dostał od rodziców na urodziny nową zabawkę, w której skład wchodzą rurka i krążki. Rurka ma nietypowy kształt — mianowicie jest to połączenie pewnej liczby walców (o takiej samej grubości) z wyciętymi w środku (współosiowo) okrągłymi otworami różnej średnicy. Rurka jest zamknięta od dołu, a otwarta od góry. Na poniższym rysunku przedstawiono przykładową taką rurkę, złożoną z walców, w których wycięto otwory o średnicach kolejno: 5 cm, 6 cm, 4 cm, 3 cm, 6 cm, 2 cm i 3 cm.



Krążki w zabawce Jasia są walcami o różnych średnicach i takiej samej grubości, co walce tworzące rurkę.

Jaś wymyślił sobie następującą zabawę. Mając do dyspozycji pewien zestaw krążków zastanawia się, na jakiej głębokości zatrzymałby się ostatni z nich, gdyby wrzucał je kolejno do rurki centralnie (czyli dokładnie w jej środek). Dla przykładu, gdyby wrzucić do powyższej rurki krążki o średnicach kolejno 3 cm, 2 cm i 5 cm, to otrzymalibyśmy następującą sytuację:



Jak widać, każdy kolejny krążek po wrzuceniu spada dopóki się nie zaklinuje (czyli nie oprze się o walec, w którym wycięty jest otwór o mniejszej średnicy niż średnica krążka) albo nie natrafi na przeszkodę w postaci innego krążka lub dna rurki.

Ponieważ zabawa ta jest trudna dla małego Jasia, to ciągle prosi swoich rodziców o pomoc. A jako że rodzice Jasia nie lubią takich zabaw intelektualnych, to poprosili Ciebie — znajomego programistę — o napisanie programu, który zamiast nich będzie udzielał odpowiedzi Jasiowi.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia schemat rurki i opis krążków, jakie Jaś będzie wrzucał do rurki,*
- *wyznaczy głębokość, na jakiej zatrzyma się ostatni wrzucony przez Jasia krążek,*
- *wypisze wynik na standardowe wyjście.*

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby całkowite n *i* m ($1 \leqslant n, m \leqslant 300\,000$)*, oddzielone pojedynczym odstępem i oznaczające wysokość rurki Jasia (liczbę walców wchodzących w jej skład) i liczbę krążków, które zamierza wrzucić do rurki. Drugi wiersz wejścia zawiera* n *liczb całkowitych* $r_1, r_2, \ldots, r_n$ ($1 \leqslant r_i \leqslant 1\,000\,000\,000$ *dla* $1 \leqslant i \leqslant n$) *oddzielonych pojedynczymi odstępami i oznaczających średnice otworów wyciętych w kolejnych (od góry) walcach tworzących rurkę. Trzeci wiersz wejścia zawiera* m *liczb całkowitych* $k_1, k_2, \ldots, k_m$ ($1 \leqslant k_j \leqslant 1\,000\,000\,000$ *dla* $1 \leqslant j \leqslant m$) *oddzielonych pojedynczymi odstępami i oznaczających średnice kolejnych krążków, które Jaś zamierza wrzucić do rurki.*

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać jedną liczbę całkowitą, oznaczającą głębokość zatrzymania się ostatniego krążka. Jeżeli krążek ten w ogóle nie wpadnie do rurki, to odpowiedzią powinna być liczba 0.

Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
2
```

Rozwiązanie

Rozwiązanie o złożoności czasowej $O(n + m \log n)$

Najprostszym rozwiązaniem jest symulacja sytuacji opisanej w zadaniu — każdy krążek spada, dopóki nie zaklinuje się bądź nie natrafi na przeszkodę w postaci innego krążka lub dna rurki. Symulację spadania jednego krążka możemy przeprowadzić w czasie $O(n)$ — po prostu przeglądamy rurkę od góry, poszukując miejsca, w którym zatrzyma się dany krążek. Całkowita złożoność takiego rozwiązania to $O(nm)$. Jest ona zdecydowanie za duża jak na ograniczenia z zadania.

Kluczem do rozwiązania zadania jest przyspieszenie powyższej symulacji — musimy znaleźć poziom, na którym zatrzyma się dany krążek w czasie krótszym niż $O(n)$. Oczywiście bierzemy pod uwagę jedynie poziomy powyżej pozycji ostatnio wrzuconego krążka. Następny krążek może dolecieć do określonego poziomu, jeżeli wcześniej nie zaklinuje się, czyli jeżeli wszystkie walce na wyższych poziomach mają średnice niemniejsze od średnicy krążka. To sformułowanie jest równoważne temu, że *minimum* ze średnic wszystkich walców położonych powyżej danego poziomu musi być niemniejsze od średnicy krążka. Jeżeli na początku policzymy minima $m_1, m_2, \ldots, m_n$ dla wszystkich poziomów (można to zrobić w czasie $O(n)$), to w czasie stałym uzyskujemy odpowiedź na pytanie, czy krążek może dolecieć na dany poziom. Co ważniejsze, obliczony ciąg wartości $m_1, m_2, \ldots, m_n$ jest niemalejący i w celu znalezienia najniższego poziomu i, dla którego m_i jest niemniejsze od średnicy wrzucanego krążka, można zastosować wyszukiwanie binarne. To oznacza, że pozycję końcową jednego krążka umiemy znaleźć w czasie $O(\log n)$, a zatem całe rozwiązanie ma złożoność $O(n + m \log n)$.

Poniżej zamieszczamy pseudokod tego rozwiązania. Pełny kod można znaleźć na dysku dołączonym do książeczki.

```
program Rozwiązanie pierwsze;
{ Liczymy ciąg minimów $m_i$ }
$m_1$ := $r_1$;
for $i$ := 2 to $n$ do
  $m_i$ := $\min(m_{i-1}, r_i)$;
{ Poszukiwanie pozycji krążków }
poprzedni := $n+1$; { pierwszy krążek może się zatrzymać }
                    { najdalej na dnie rurki }
for $j$ := 1 to $m$ do
begin  { Wyszukiwanie binarne }
  $a$ := 0, $b$ := poprzedni $- 1$;
  while $a < b$ do
  begin { Uwaga! Wyszukujemy binarnie wśród liczb całkowitych! }
    $c$ := $\lfloor \frac{a+b}{2} \rfloor + 1$;
    if $m_c < k_j$ then
      { Tak daleko krążek nie doleci }
      $b$ := $c - 1$;
    else
      { Tutaj krążek może dolecieć }
      $a$ := $c$;
  end
  poprzedni := $a$;
  if poprzedni $= 0$ then
    return 0;
end
return poprzedni;
```

Rozwiązanie o złożoności czasowej $O(n+m)$

Co prawda powyższe rozwiązanie przechodzi wszystkie testy, jednak możemy je jeszcze usprawnić. Zamiast wyszukiwać pozycję krążka binarnie wykorzystamy proste wyszukiwanie liniowe, z tym, że rozpoczniemy je od pozycji poprzedniego krążka (dla pierwszego krążka — od dna rurki). Począwszy od tej pozycji będziemy badać kolejne, coraz płytsze, do momentu aż znajdziemy poziom i o wystarczająco dużej wartości m_i — będzie to poziom, na którym zatrzyma się dany krążek.

Mimo, że jeden krok takiego wyszukiwania może w najgorszym przypadku wymagać przejrzenia nawet $O(n)$ poziomów, to łatwo zauważyć, że każdy poziom analizujemy *co najwyżej* raz. Jeśli bowiem w trakcie poszukiwania pozycji krążka rozważamy pewien poziom, to oznacza, że dany krążek zatrzyma się na nim lub ponad nim i poszukując pozycji pozostałych krążków będziemy już rozważać tylko poziomy leżące powyżej. Stąd wynika, że sumaryczna złożoność czasowa algorytmu wynosi $O(n+m)$ (stały czas na rozważanie każdego poziomu oraz stały czas na rozważenie każdego krążka).

Z analizy tego prostego algorytmu można wysnuć ciekawy wniosek: algorytm, którego dowolny krok może mieć pesymistycznie dużą złożoność, nie musi być wolnym algorytmem. Jego *całkowita* złożoność może okazać się istotnie lepsza niż iloczyn pesymistycznej złożoności jednego jego kroku i liczby wszystkich kroków (choć oczywiście nie jest to regułą).

Oto pseudokod tego rozwiązania (pełny kod można znaleźć na dysku dołączonym do książeczki):

```
program Rozwiązanie drugie
{ Liczymy ciąg minimów $m_i$ }
$m_1$ := $r_1$;
for $i$ := 2 to $n$ do
  $m_i$ := $\min(m_{i-1}, r_i)$;
{ Poszukiwanie pozycji krążków }
poprzedni := $n+1$; { pierwszy krążek może się zatrzymać }
                    { najdalej na dnie rurki }
for $j$ := 1 to $m$ do
begin
  { Wyszukiwanie liniowe od pozycji poprzedniego krążka }
  $a$ := poprzedni $- 1$;
  while $(a > 0)$ and $(m_a < k_j)$ do
    $a$ := $a - 1$;
  poprzedni := $a$;
  if poprzedni $= 0$ then
    return 0;
end
return poprzedni;
```

Testy

Zadanie testowane było na zestawie 11 danych. Dwa z nich stanowiły grupy po dwa testy — jeden miał odpowiedź równą 0 (czyli nie wszystkie krążki mieściły się w rurce), a drugi dodatnią.

Nazwa	n	m	Opis
kra1a.in	50	51	za dużo krążków, odpowiedź 0
kra1b.in	100	60	rurka zwęża się
kra2.in	353	71	losowe nieregularności w rurce
kra3a.in	100	25	odpowiedź 0
kra3b.in	1000	600	rurka zwęża się ku górze
kra4.in	7100	1457	długie sekwencje jednakowych średnic
kra5.in	50 001	20 043	rurka na przemian rozszerza się i zwęża
kra6.in	95 000	63 000	miejscowe przewężenia w rurce
kra7.in	120 001	97 003	równomierna szerokość rurki z dwiema przeszkodami
kra8.in	250 002	90 001	rurka rozszerza się losowo ku górze
kra9.in	300 000	90 000	rurka najpierw rozszerza się, a potem zwęża
kra10.in	300 000	240 301	rurka na przemian rozszerza się i zwęża
kra11.in	300 000	280 001	rurka rozszerza się losowo ku górze

Szymon Acedański
Treść zadania, Opracowanie

Karol Cwalina
Program

OI, Etap I

Okresy słów

Napisem *nazywamy każdy skończony ciąg małych liter alfabetu angielskiego. W szczególności może to być ciąg pusty. Zapis* $A = BC$ *oznacza, że* A *jest napisem powstałym przez sklejenie napisów* B *i* C *(w tej kolejności). Napis* P *jest* **prefiksem** *napisu* A*, jeżeli istnieje taki napis* B*, że* $A = PB$*. Inaczej mówiąc, prefiksy* A *to początkowe fragmenty* A*. Jeśli dodatkowo* $P \neq A$ *oraz* P *nie jest napisem pustym, to mówimy, że* P *jest* **prefiksem właściwym** A*.*

Napis Q *jest* **okresem** A*, jeśli* Q *jest prefiksem właściwym* A *oraz* A *jest prefiksem (niekoniecznie właściwym) napisu* QQ*. Przykładowo, napisy* `abab` *i* `ababab` *są okresami napisu* `abababa`*.* **Maksymalnym okresem** *napisu* A *nazywamy najdłuższy z jego okresów, lub napis pusty, jeśli* A *nie posiada okresu. Dla przykładu, maksymalnym okresem napisu* `ababab` *jest* `abab`*. Maksymalnym okresem* `abc` *jest napis pusty.*

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia długość napisu oraz napis,*
- *wyznaczy sumę długości maksymalnych okresów wszystkich jego prefiksów,*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu standardowego wejścia zapisana jest jedna liczba całkowita k *(*$1 \leqslant k \leqslant 1\,000\,000$*) — długość napisu. W kolejnym wierszu znajduje się ciąg dokładnie* k *małych liter alfabetu angielskiego — napis.*

Wyjście

W pierwszym i jedynym wierszu standardowego wyjścia Twój program powinien zapisać jedną liczbę — sumę długości maksymalnych okresów wszystkich prefiksów napisu zadanego na wejściu.

Przykład

Dla pliku wejściowego `okr.in`*:*

```
8
babababa
```

poprawnym wynikiem jest plik wyjściowy `okr.out`*:*

```
24
```

Rozwiązanie

Niniejsze zadanie zawiera kolejny z serii problemów związanych z algorytmami tekstowymi. Na dodatek ograniczenia na dane wejściowe pozwalają domyślać się, że oczekiwane rozwiązanie ma złożoność liniową. Stąd rodzi się podejrzenie, że pomocnym może okazać się algorytm Knutha, Morrisa, Pratta (KMP), który już niejednokrotnie przydawał się w rozwiązaniach problemów wymagających analizy napisów. Jest to słuszne podejrzenie, ale zacznijmy od początku. Spróbujmy najpierw znaleźć szybką metodę na wyznaczenie maksymalnego okresu całego słowa.

Zauważmy, że zdefiniowany w treści zadania *maksymalny okres* rzadko jest tym, co zazwyczaj uważamy za okres słowa. Otóż maksymalny okres, jeśli w ogóle istnieje, ma długość będącą co najmniej połową długości słowa:

$$\underbrace{\text{słowookresowesłowookresowesłowookresowe}}_{\textit{maksymalny okres}}\text{słowookresowe}$$

Rys. 1: Maksymalny okres ma długość będącą co najmniej połową długości słowa

Warto także zaznaczyć, że nie zawsze maksymalny okres jest wielokrotnością standardowo rozumianego, minimalnego okresu:

$$\underbrace{\text{słowookresowesłowookresowesłowookresowesłowookre}}_{\textit{maksymalny okres}}\text{s}$$

Rys. 2: Maksymalny okres nie musi być wielokrotnością minimalnego okresu

W powyższych przykładach maksymalny okres zajmuje większą część słowa. Skoncentrujmy się teraz na pozostałej części. Jest to początkowy fragment „drugiej kopii” maksymalnego okresu — musi być on jednocześnie prefiksem, jak i sufiksem słowa.

Definicja 1 *Prefikso-sufiksem* słowa nazywamy każdy jego prefiks, który jest jednocześnie jego sufiksem.

Łatwo spostrzec, że to, co zostaje po odcięciu maksymalnego okresu to po prostu *minimalny niepusty prefikso-sufiks* słowa. Co więcej, jeżeli długość najkrótszego prefikso-sufiksu słowa jest niewiększa od połowy długości słowa, to po odcięciu tego prefikso-sufiksu zawsze otrzymujemy maksymalny okres słowa.

$$\text{słowookresowesłowookresowesłowookresowe}\underbrace{\text{słowookresowe}}_{\textit{minimalny prefikso-sufiks}}$$

$$\text{słowookresowesłowookresowesłowookresowesłowookre}\underbrace{\text{s}}_{\textit{minimalny prefikso-sufiks}}$$

Rys. 3: Pozostałość po obcięciu maksymalnego okresu

Jak wyznaczyć ten prefikso-sufiks? Możemy próbować skorzystać z funkcji prefiksowej $p(i)$, znanej z algorytmu KMP (jego opis można znaleźć na przykład w książce [13] czy [18]). Funkcja ta, dla danej liczby i, zwraca długość *maksymalnego właściwego prefikso-sufiksu* i-literowego prefiksu całego słowa. Zwróćmy uwagę na te z jej własności, które będą pomocne przy znajdowaniu *minimalnego prefikso-sufiksu.*

1. Każdy prefikso-sufiks słowa jest prefiksem maksymalnego prefikso-sufiksu (ponieważ jest prefiksem całego słowa).
2. Analogicznie, każdy prefikso-sufiks słowa jest sufiksem maksymalnego prefikso-sufiksu.
3. Z tego wniosek, że każdy prefikso-sufiks jest prefikso-sufiksem maksymalnego prefikso-sufiksu i dalej, każdy prefikso-sufiks słowa jest prefikso-sufiksem dowolnego dłuższego prefikso-sufiksu tego słowa.
4. Niech n oznacza długość słowa. Aby znaleźć minimalny prefikso-sufiks, iterujemy funkcję prefiksową p, otrzymując kolejno: $p(n)$ — długość maksymalnego prefikso-sufiksu, $p(p(n))$ — długość krótszego prefikso-sufiksu, $p(p(p(n)))$ — długość jeszcze krótszego, itd. W ten sposób dostajemy w kolejności malejącej długości wszystkich prefikso-sufiksów słowa, więc ostatnia niezerowa wartość jest długością minimalnego niepustego prefikso-sufiksu.

Stosując opisaną metodę otrzymujemy algorytm, który działa w czasie $O(n^2)$.

My jednak chcielibyśmy skonstruować coś szybszego. Przyjrzyjmy się zatem dokładniej funkcji p. Może uda nam się szybko wyznaczyć podobną funkcję, nazwijmy ją s, która dla danego argumentu i zwraca długość *minimalnego* prefikso-sufiksu i-literowego prefiksu słowa wejściowego? Okazuje się to możliwe, jeśli najpierw policzymy funkcję p. Wtedy możemy skorzystać z następującej własności.

$$s(i) = \begin{cases} p(i) & \text{gdy } p(p(i)) = 0 \\ s(p(i)) & \text{w przeciwnym przypadku} \end{cases} \tag{1}$$

W pierwszym przypadku istnieje tylko jeden prefikso-sufiks, więc jest on najkrótszy. W drugim przypadku, gdy słowo ma więcej niż jeden prefikso-sufiks, to szukany najkrótszy prefikso-sufiks jest najkrótszym prefikso-sufiksem najdłuższego prefikso-sufiksu słowa.

Korzystając z funkcji p oraz z powyższej zależności możemy w czasie $O(n)$ obliczyć wartości $s(i)$ dla $i = 1,\ldots,n$. Można łatwo pokazać, że jeżeli dane słowo ma właściwy niepusty prefikso-sufiks dłuższy niż połowa danego słowa, to słowo to posiada również niepusty prefikso-sufiks niedłuższy niż połowa tego słowa. Stąd jeżeli słowo posiada niepusty najkrótszy prefikso-sufiks, to prefiks powstały przez jego odcięcie będzie maksymalnym okresem słowa. Obserwacja ta prowadzi nas do wniosku, jeśli dla pewnego $i = 1,\ldots,n$ zachodzi $s(i) = 0$, to maksymalnym okresem prefiksu długości i jest słowo puste, a w przeciwnym wypadku słowo długości $i - s(i)$. Sumując tak zdefiniowane wartości otrzymujemy ostateczny wynik.

Testy

Do oceny zostało użytych 14 testów, przedstawionych w poniższej tabeli.

Nazwa	**n**	**Opis**
okr 1 .in	10	prosty test poprawnościowy
okr 2 .in	36	prosty test poprawnościowy
okr 3 .in	441	$(\mathtt{a}^{k-1}\mathtt{b})^k$ z nieco zmienioną ostatnią ćwiartką
okr 4 .in	1 024	$(\mathtt{a}^{k-1}\mathtt{b})^k$, zmiany j.w.
okr 5 .in	501 264	$(\mathtt{a}^{k-1}\mathtt{b})^k$, zmiany j.w.
okr 6 .in	970 225	$(\mathtt{a}^{k-1}\mathtt{b})^k$, zmiany j.w.
okr 7 .in	100 387	w^k dla losowego wzorca w, zmiany j.w.
okr 8 .in	299 809	w^k dla losowego wzorca w, zmiany j.w.
okr 9 .in	599 976	w^k dla losowego wzorca w, zmiany j.w.
okr 10 .in	949 721	w^k dla losowego wzorca w, zmiany j.w.
okr 11 .in	129 962	$(w^k v)^{k^2}$ dla losowych w i v
okr 12 .in	331 332	$(w^k v)^{k^2}$ dla losowych w i v
okr 13 .in	614 619	$(w^k v)^{k^2}$ dla losowych w i v
okr 14 .in	923 387	$(w^k v)^{k^2}$ dla losowych w i v

Jakub Radoszewski
Treść zadania, Opracowanie

Michał Adamaszek
Program

OI, Etap I

Profesor Szu

W mieście Bajtion ma swoją siedzibę bajtocki uniwersytet. Oprócz głównego gmachu, uniwersytet ma do dyspozycji n domków dla pracowników naukowych. Domki połączone są jednokierunkowymi drogami, może jednak być wiele dróg łączących dwa domki, istnieją także drogi łączące gmach uniwersytetu z domkami (droga może łączyć także pewien obiekt z nim samym). Bajtion został tak skonstruowany, żeby żadne drogi nie przecinały się w miejscach innych niż domki lub gmach (ale mogą przebiegać mostami i tunelami); ponadto każda droga zaczyna się w pewnym domku lub w gmachu i kończy się w domku lub w gmachu. Wiadomo ponadto, że istnieje co najmniej jedna droga łącząca pewien domek z gmachem uniwersytetu.

Pewnego razu uniwersytet zapragnął zatrudnić u siebie znanego specjalistę informatyki teoretycznej — profesora Szu. Jak wielu wielkich naukowców, profesor Szu ma dziwny zwyczaj; otóż każdego dnia lubi dojeżdżać do gmachu uniwersytetu inną trasą (będącą drogą bądź układem dróg, z których każda następna zaczyna się w domku, w którym kończy się poprzednia; trasa może przechodzić przez ten sam domek bądź główny gmach uniwersytetu wielokrotnie). Profesor dwie trasy uważa za różne, jeżeli różnią się chociaż jedną wykorzystaną drogą (przy czym kolejność dróg jest ważna, a dwie różne drogi łączące te same domki uważa on za różne).

Znając schemat połączeń między domkami Bajtionu, pomóż uniwersytetowi znaleźć domek, z którego istnieje najwięcej różnych tras do gmachu uniwersytetu (zamieszkawszy w tym domku, profesor Szu będzie chciał najdłużej pracować na uczelni) — jeżeli takich domków jest więcej niż jeden, to podaj wszystkie z nich. Jeżeli przy tym z jakiegoś domku istnieje więcej niż $36\,500$ tras do gmachu, to zakładamy, że profesor może tam zamieszkać na zawsze (jako że nie może żyć nieskończenie długo, a 100 lat to dość bezpieczna granica).

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia schemat połączeń między domkami Bajtionu,*
- *wyznaczy domki, w których profesor Szu mógłby mieszkać najdłużej, oraz najdłuższy czas jego zamieszkiwania,*
- *wypisze wynik na standardowe wyjście.*

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby całkowite n oraz m $(1 \leqslant n, m \leqslant 1\,000\,000)$ oddzielone pojedynczym odstępem i oznaczające odpowiednio liczbę domków i liczbę dróg w Bajtionie (domki są ponumerowane liczbami od 1 do n, a umownie nadajemy gmachowi uniwersytetu numer $n+1$). W wierszach o numerach od 2 do $m+1$ znajdują się pary liczb całkowitych a_i, b_i $(1 \leqslant a_i, b_i \leqslant n+1$ dla $1 \leqslant i \leqslant m)$ oddzielone pojedynczymi odstępami i oznaczające odpowiednio numer domku, w którym zaczyna się, i numer domku, w którym kończy się i–ta droga.

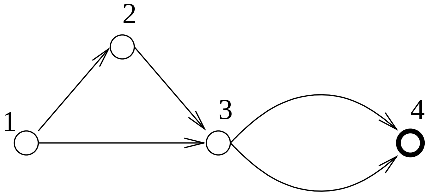
Wyjście

Pierwszy wiersz wyjścia powinien zawierać maksymalną liczbę dni jaką profesor Szu może mieszkać w Bajtionie lub jedno słowo „`zawsze`*”, jeżeli ta liczba przekracza 36 500 dni. W drugim wierszu powinna znajdować się liczba domków, zamieszkanie w których zapewnia profesorowi okres pobytu podany w pierwszym wierszu wyjścia. W trzecim wierszu powinny znaleźć się numery wszystkich takich domków, oddzielone pojedynczymi odstępami i podane w kolejności rosnącej. Wszystkie domki, w których profesor może zamieszkać na zawsze uważamy przy tym za jednakowo dobre.*

Przykład

Dla danych wejściowych:

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

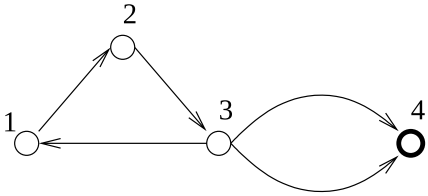


poprawnym wynikiem jest:

```
4
1
1
```

Natomiast dla danych:

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



poprawnym wynikiem jest:

```
zawsze
3
1 2 3
```

Rozwiązanie

Analiza problemu

Na wstępie zauważmy, że sytuację opisaną w treści zadania możemy przedstawić jako graf skierowany, którego wierzchołkami są domki i gmach uniwersytetu, a krawędziami — jednokierunkowe drogi. Graf ten może być multigrafem, to znaczy mogą w nim występować krawędzie wielokrotne i pętle.

Na pytanie postawione w treści zadania będziemy mogli łatwo odpowiedzieć, jeżeli uda nam się każdy wierzchołek przyporządkować do jednej z następujących grup:

- *wierzchołków zerowych* — czyli takich, z których nie da się dojść do gmachu uniwersytetu,
- *wierzchołków skończonych* — są to wierzchołki, z których istnieje skończenie wiele tras do gmachu uniwersytetu,
- *wierzchołków nieskończonych* — z których do gmachu uniwersytetu prowadzi nieskończenie wiele tras; wierzchołek jest nieskończony tylko wtedy, gdy można z niego osiągnąć cykl, z którego da się dojść do gmachu uniwersytetu (można wtedy skonstruować nieskończenie wiele tras, które przechodzą przez ten cykl $0, 1, 2, \ldots$ razy) — gdyby taki cykl nie istniał, to każda trasa do gmachu mogłaby zawierać co najwyżej n różnych wierzchołków, czyli tras byłoby skończenie wiele.

Dla każdego wierzchołka skończonego trzeba także policzyć liczbę różnych tras, prowadzących z niego do gmachu. Na jej podstawie dzielimy dalej wierzchołki skończone na *skończone małe*, czyli takie, dla których liczba tras jest niewiększa niż 36 500, oraz *skończone duże* — czyli pozostałe.

Po dokonaniu takiej klasyfikacji wierzchołków wynik jest prawie gotowy. Jeżeli istnieje jakikolwiek wierzchołek nieskończony lub wierzchołek skończony duży, to należy wypisać odpowiedź „`zawsze`" i podać wszystkie takie wierzchołki. W pozostałych przypadkach znajdujemy maksimum z wartości przypisanych wierzchołkom skończonym małym i wypisujemy wszystkie wierzchołki, dla których liczba różnych tras do gmachu uniwersytetu jest równa temu maksimum. Ten krok jesteśmy w stanie wykonać w czasie $O(n)$. Ponieważ w zadaniu jest założenie, że istnieje przynajmniej jeden domek, z którego można dojść do gmachu uniwersytetu, więc opisane przypadki wyczerpują wszystkie możliwości. Pozostaje więc tylko sklasyfikować wierzchołki — na tym skupimy się w dalszej części rozwiązania.

Klasyfikacja wierzchołków

Wierzchołki zerowe

Najłatwiej będzie nam zidentyfikować wierzchołki zerowe. Z definicji są to takie wierzchołki, z których nie jest osiągalny gmach uniwersytetu; bezpośrednie sprawdzenie tej własności (na przykład poprzez przeszukiwanie grafu wszerz czy w głąb startując kolejno z każdego z tych wierzchołków) nie jest jednak wystarczająco szybkie — ma złożoność czasową aż $O(n \cdot (n+m))$.

Szybszy algorytm uzyskamy, rozważając osiągalność w nieco zmienionym grafie. Jeżeli z pewnego wierzchołka v nie da się w grafie G osiągnąć gmachu g, to w grafie odwróconym G^T (utworzonym z G przez zmianę zwrotu wszystkich krawędzi na przeciwny) z gmachu g nie da się przejść do wierzchołka v. Wynika to stąd, że każda trasa z g do v w grafie G^T odpowiada trasie z v do g w wyjściowym grafie. Wystarczy więc „odwrócić" graf G w czasie $O(n+m)$, a następnie, wykonując tylko jedno przeszukiwanie grafu G^T z wierzchołka g w czasie $O(n+m)$, wyznaczyć wierzchołki nieosiągalne z gmachu, czyli wierzchołki zerowe.

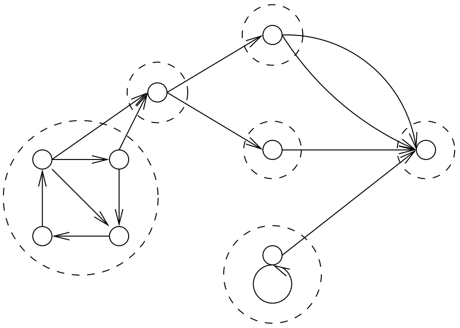
W dalszym opisie algorytmu będziemy więc pomijać wierzchołki zerowe, ponieważ nie mogą one stanowić rozwiązania ani wchodzić w skład interesujących nas dróg prowadzących do gmachu.

Graf silnie spójnych składowych

Przedstawimy dwa sposoby klasyfikacji wierzchołków na skończone i nieskończone. W pierwszym korzystamy z dość zaawansowanych pojęć, co sprawia, że algorytm jest niełatwy w implementacji. Jest jednak elegancki w zapisie i można go łatwo przystosować do rozwiązywania innych, podobnego typu zadań. W drugim algorytmie wykorzystujemy tylko elementarne pojęcia, jest więc stosunkowo prosty w opisie i implementacji, jednakże jego poprawność jest mniej oczywista i nie uogólnia się on łatwo na inne podobne problemy.

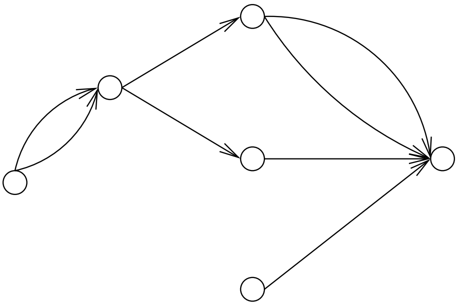
Rozpocznijmy od przypomnienia pojęcia silnie spójnej składowej w grafie.

Definicja 1 Powiemy, że dwa wierzchołki w_1 i w_2 należą do tej samej *silnie spójnej składowej* grafu, jeżeli w grafie istnieje trasa z wierzchołka w_1 do w_2 oraz z wierzchołka w_2 istnieje trasa do w_1.



Rys. 1: Przykładowy graf z zaznaczonymi silnie spójnymi składowymi

W książce [18] można znaleźć algorytm podziału grafu na silnie spójne składowe, którego złożoność czasowa wynosi $O(n+m)$. Po wyznaczeniu takich składowych, możemy graf G „skompresować” tworząc graf G', którego wierzchołkami są obliczone składowe; ponadto w grafie G' istnieje krawędź z wierzchołka v_1 do v_2 (dla $v_1 \neq v_2$), jeżeli w grafie G istniały połączone krawędzią wierzchołki w_1 i w_2, należące do silnie spójnych składowych odpowiadających v_1 i v_2. Konstrukcja G' ściśle zgodna z powyższą definicją nie jest prosta do wykonania w czasie $O(n+m)$ — trudno wykryć i wyeliminować krawędzie wielokrotne. Jednakże w naszym przypadku usuwanie ich nie jest konieczne, a wręcz mogłoby prowadzić do błędnych wyliczeń liczby ścieżek. Dlatego możemy skorzystać z szybkiej wersji algorytmu konstrukcji grafu skompresowanego, w wyniku której otrzymamy graf G', w którym z wierzchołka v_1 do v_2 prowadzi tyle krawędzi, ile jest w grafie G różnych krawędzi (w_1, w_2) takich, że w_1 należy do składowej odpowiadającej v_1, a w_2 — do składowej v_2.



Rys. 2: Graf silnie spójnych składowych, odpowiadający grafowi z poprzedniego rysunku

Zauważmy, że G' jest grafem acyklicznym (grafy acykliczne skierowane często oznaczamy skrótem DAG). Gdyby bowiem zawierał cykl $v_1 \to v_2 \to \ldots \to v_k \to v_1$, to z każdego wierzchołka należącego do składowej v_1 dałoby się dojść do każdego wierzchołka ze składowej v_2 i odwrotnie. A to oznaczałoby, że v_1 i v_2 tworzyłyby jedną silnie spójną składową.

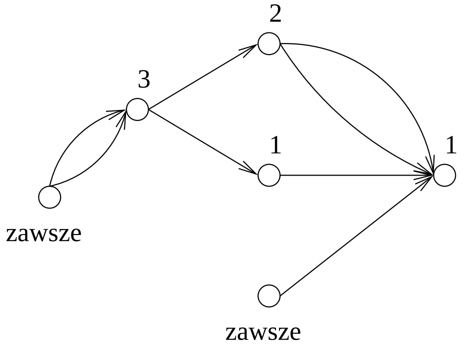
Odnotujmy także jeszcze jedno istotne spostrzeżenie. Jeżeli silnie spójna składowa grafu G zawiera co najmniej jedną krawędź łączącą jej wierzchołki (w szczególności może być to pętla łącząca wierzchołek z samym sobą), to każdy wierzchołek należący do tej składowej należy do pewnego cyklu w grafie G. Z drugiej strony, wierzchołki grafu G nienależące do żadnego cyklu tworzą jednoelementowe silnie spójne składowe, niezawierające żadnych krawędzi grafu G.

Wierzchołki skończone i nieskończone

Załóżmy, że mamy już zbudowany graf G'. Korzystając z poczynionych spostrzeżeń możemy stwierdzić, że każdy niezerowy wierzchołek grafu G należący do silnie spójnej składowej zawierającej przynajmniej jedną krawędź, należy do jakiegoś cyklu, z którego można dojść do gmachu, a tym samym jest wierzchołkiem nieskończonym.

Aby wyznaczyć liczbę ścieżek prowadzących do gmachu dla wszystkich pozostałych wierzchołków będziemy przeglądać wierzchołki G' w kolejności odwrotnej do topologicznej. Oznacza to, że przed przejrzeniem wierzchołka v grafu G' musimy odwiedzić wszystkie wierzchołki, do których prowadzą z v krawędzie — jest to możliwe, gdyż G' jest acykliczny. Pierwsza w tym porządku jest składowa zawierająca gmach uniwersytetu. Jeżeli zawiera ona krawędź, to z gmachu (oraz ze wszystkich wierzchołków tworzących z nim silnie spójną składową) do gmachu możemy przejść na nieskończenie wiele sposobów. W przeciwnym razie istnieje dokładnie jedna ścieżka (długości zero) prowadząca z gmachu do gmachu i składowa gmachu składa się z tego jedynego wierzchołka.

Obliczmy liczbę ścieżek dla pozostałych wierzchołków G' w wyznaczonym porządku. Jeśli kolejny wierzchołek jest składową zawierającą krawędź, to znamy odpowiedź — wszystkie wierzchołki grafu G tworzące tę składową są wierzchołkami nieskończonymi. W przeciwnym razie składowa zawiera dokładnie jeden wierzchołek grafu G i liczba tras łączących go z gmachem jest równa sumie wyznaczonych wyników dla wszystkich wierzchołków G', do których prowadzą krawędzie z tego wierzchołka (zauważmy, że w rozumowaniu tym istotne jest pozostawienie wielokrotnych krawędzi łączących różne silnie spójne składowe). Jeżeli wynikiem dla któregokolwiek z tych wierzchołków jest nieskończoność, to i cała suma jest nieskończona. Jeżeli suma jest skończona, ale przekracza 36 500, to możemy przyjąć, że wynik także wynosi nieskończoności (mamy bowiem do czynienia z wierzchołkiem skończonym dużym, którego nie musimy odróżniać od nieskończonego).



Rys. 3: Poprzedni graf z wynikami wyznaczonymi opisaną metodą

Pierwszy algorytm

Jesteśmy już gotowi do zapisania pseudokodu pierwszego rozwiązania zadania:

```
function Profesor($g$,$G$)
{ Wyznacza wynik dla każdej silnie spójnej składowej grafu $G$. }
begin
   { Wyznaczanie wierzchołków zerowych. }
   Wyznacz graf odwrócony $G^T$;
   visited := BFS($g$,$G^T$); { Przeszukaj graf $G^T$, zaczynając od gmachu. }
   { Właściwy algorytm, w którym będziemy eliminować wierzchołki zerowe. }
   Wyznacz graf silnie spójnych składowych $G'$;
   foreach $v$ in $G'$ w kolejności odwrotnej do topologicznej do
   begin
      wynik[$v$] := 0;
      if ($\exists w \in v$ : not visited[$w$]) then
         continue; { Wierzchołek zerowy $w$ zawarty w składowej $v$. }
      if (silnie spójna składowa $v$ zawiera krawędź) then
         wynik[$v$] := $\infty$;
      else
      begin
         foreach $e = (v,w) \in G'$ do
            wynik[$v$] := wynik[$v$] + wynik[$w$]; { $a + \infty = \infty + a = \infty$ }
         if (wynik[$v$] $> 36500$) then
            wynik[$v$] := $\infty$;
      end
   end
   return wynik;
end
```

Przeanalizujmy złożoność czasową przedstawionego rozwiązania. Jak już pisaliśmy, kroki algorytmu w wierszach 4-8 można wykonać w czasie $O(n+m)$. W głównej pętli algorytmu (instrukcje 9-23) dla każdej silnie spójnej składowej rozważamy wszystkie krawędzie wychodzące z niej w grafie G' (każdą tylko raz) — liczba przeanalizowanych krawędzi będzie więc co najwyżej taka, jak liczba krawędzi w grafie G (może być mniejsza, gdyż pomijamy krawędzie wychodzące z wierzchołków zerowych). Ostatecznie pętla ma więc złożoność $O(n+m)$ i taka jest też złożoność czasowa całego algorytmu.

Drugie rozwiązanie

Pierwsze rozwiązanie prezentuje ogólny schemat postępowania stosowany w przypadku rozwiązywania wielu problemów grafowych — z grafu konstruujemy graf silnie spójnych składowych, a następnie wykorzystujemy jego acykliczność przeglądając wierzchołki „grupami”. Niestety konstrukcja tego grafu jest stosunkowo skomplikowana, zatem poniżej przedstawimy inne rozwiązanie — ideologicznie podobne, acz nie odwołujące się do pojęcia silnie spójnych składowych.

Przypomnijmy, że potrafimy łatwo wyznaczyć wierzchołki zerowe grafu, więc w dalszych rozważaniach zakładamy, że ich w grafie nie ma (przypomnimy sobie o nich dopiero w pseudokodzie rozwiązania). Zastanówmy się teraz, jak wygląda część grafu zawierająca wyłącznie wierzchołki skończone. Musi to być oczywiście graf acykliczny — z żadnego wierzchołka skończonego nie można dojść do żadnego cyklu. Aby go zlokalizować, możemy rozpocząć od wierzchołków o zerowym stopniu wyjściowym (czyli takich, z których nie wychodzi żadna krawędź). Oczywiście należą one do poszukiwanego grafu — usuwamy je więc z oryginalnego grafu wraz z prowadzącymi do nich krawędziami. Jako kolejne usuwamy (wraz z krawędziami wchodzącymi) wierzchołki, z których krawędzie prowadziły jedynie do już wybranych, czyli wierzchołki mające w tej chwili zerowy stopień wyjściowy. Iterujemy opisany proces do momentu, gdy w grafie nie będzie już żadnego wierzchołka o stopniu wyjściowym równym 0. W ten sposób w grafie pozostaną jedynie wierzchołki nieskończone.

Do uzasadnienia poprawności powyższej procedury klasyfikacji wierzchołków wystarczą następujące spostrzeżenia.

- Po pierwsze w trakcie procesu nie usuniemy z grafu wierzchołka w, jeśli jest on nieskończony. Z każdego wierzchołka nieskończonego istnieje bowiem ścieżka do jakiegoś cyklu, a my naszą procedurą nie jesteśmy w stanie usunąć z grafu żadnego cyklu — który wierzchołek miałby zostać usunięty jako pierwszy? Nie możemy także usunąć żadnego wierzchołka należącego do ścieżki prowadzącej od w do cyklu, bo każdy wierzchołek na tej ścieżce ma cały czas krawędź wychodzącą.

- Po drugie pokażmy, że algorytm usuwa z grafu wszystkie wierzchołki skończone. W tym celu zauważmy, że każdy nieusunięty wierzchołek w ma co najmniej jedną krawędź wychodzącą. Wobec tego startując z wierzchołka w i przechodząc po dowolnych krawędziach w końcu dojdziemy do wierzchołka, w którym już w trakcie tego „spaceru” byliśmy — wynika to z zasady szufladkowej Dirichlet'a. To dowodzi, że z wierzchołka w doszliśmy do pewnego cyklu, czyli że wierzchołek ten jest nieskończony.

Warto zauważyć, że powyższy proces rozpoznawania wierzchołków skończonych jest analogiczny do sortowania topologicznego stosowanego również w pierwszym algorytmie. Jest on jednak przeprowadzany bardziej „elementarnymi” środkami, stąd tak istotny jest powyższy dowód pozwalający przekonać się o jego poprawności.

Drugi algorytm

Możemy teraz dokładnie zapisać algorytm. Będziemy w nim sukcesywnie obliczać wartości tablicy *wynik*, zawierającej wykrytą dla danego wierzchołka liczbę ścieżek prowadzących z niego do gmachu. Na początku przypiszmy gmachowi uniwersytetu wynik równy 1 (być może powinien on być równy nieskończoność, ale do tego jeszcze dojdziemy — na razie znamy tylko jedna ścieżkę), a wszystkim pozostałym wierzchołkom przyporządkujmy tymczasowo 0. Znajdowanie wszystkich wierzchołków skończonych wykonamy za pomocą kolejki — najpierw umieścimy w niej wszystkie wierzchołki, z których początkowo nie wychodzi żadna krawędź. Potem wyjmując wierzchołek z kolejki będziemy usuwać go z grafu i jednocześnie sprawdzać, czy wskutek tego żaden nowy wierzchołek nie powinien do kolejki trafić. Ponadto będziemy zwiększać wartości *wynik* dla wszystkich wierzchołków, z których prowadzi do niego jakakolwiek krawędź, o liczbę dróg prowadzących przez usuwany wierzchołek. Ponieważ w czasie usuwania wierzchołków i aktualizacji liczby ścieżek przeglądamy graf w porządku odwrotnym do „topologicznego” (pamiętając, że nasz graf może zawierać cykle, a więc porządek topologiczny nie jest w nim właściwie zdefiniowany), więc cały czas utrzymujemy graf odwrócony G^T. Po obliczeniu wyniku dla wierzchołków skończonych w grafie pozostają wierzchołki zerowe — nieosiągalne z gmachu uniwersytetu, oraz wierzchołki nieskończone — nierozważane w trakcie znajdowania wierzchołków skończonych.

Po tak dokładnym opisie możemy przedstawić implementację rozwiązania:

```
function Profesor2(g,G)
{ Wyznacza wynik dla każdego wierzchołka grafu G. }
begin
   { Wyznaczanie wierzchołków zerowych. }
   Wyznacz graf odwrócony $G^T$;
   visited := BFS(g,$G^T$); { Przeszukaj graf $G^T$, zaczynając od gmachu. }
   { Inicjacja zmiennych. }
   foreach v in G do
      wynik[v] := 0;
   wynik[g] := 1;
   rozwazone := $\emptyset$;
   kol := $\emptyset$ { Prosta kolejka FIFO. }
   foreach v in G do
      if outdeg[v] = 0 then
         Insert(v,kol);
   { Główna pętla algorytmu }
   while (kol $\neq \emptyset$) do
   begin
      v := Pop(kol);
      Insert(v,rozwazone);
```

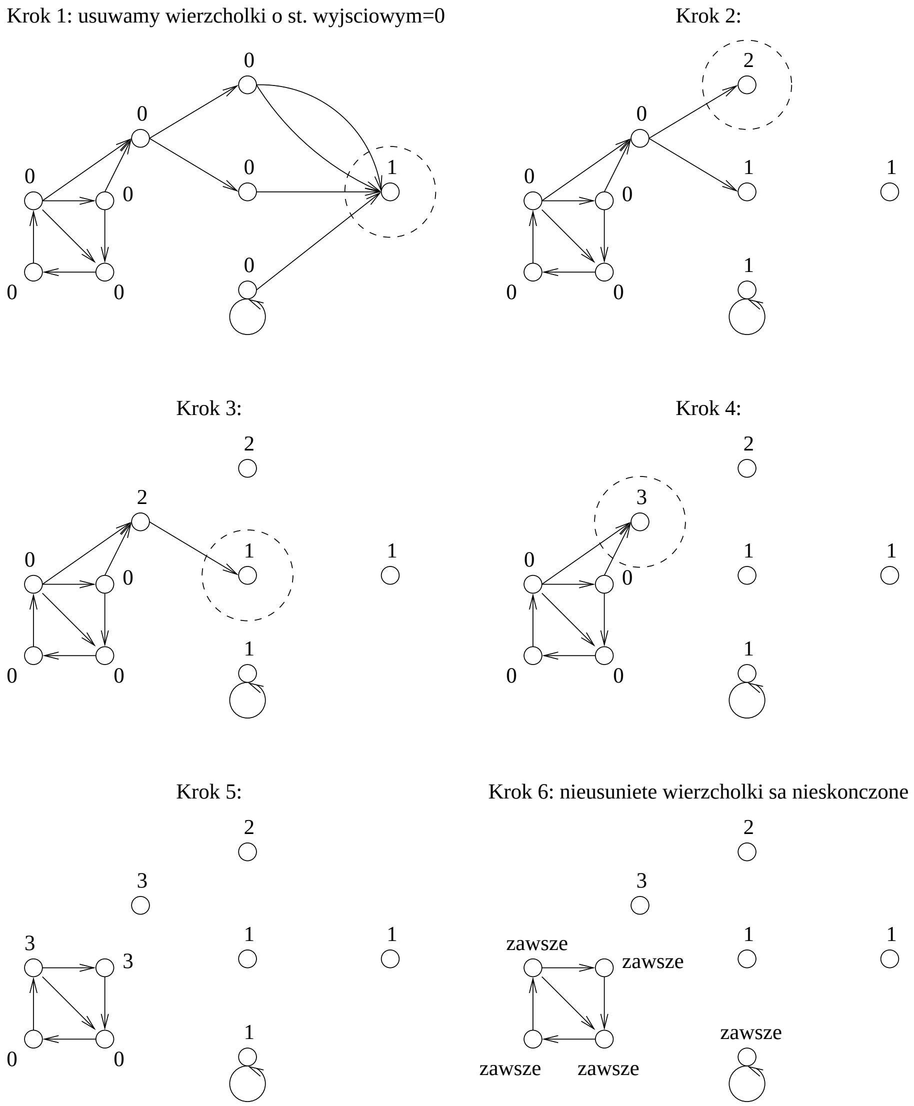
```
21:       if (not $visited[v]$) then
22:          continue; { Nie rozważamy wierzchołków zerowych. }
23:       foreach $e = (u,v) \in G$ do
24:       begin
25:          $wynik[u]$ := $wynik[u] + wynik[v]$;
26:          if ($wynik[u] > 36500$) then
27:             $wynik[u]$ := $\infty$;
28:          Usuń krawędź $e$ z grafu $G$;
29:          if ($outdeg[u] = 0$) then
30:             $Insert(u,kol)$;
31:       end
32:    end
33:    { Nierozważone wierzchołki są nieskończone. }
34:    foreach $v$ in $G$ do
35:       if ($v \notin rozwazone$) then
36:          { Wszystkie wierzchołki zerowe znajdują się w zbiorze rozwazone. }
37:          $wynik[v]$ := $\infty$;
38:    return $wynik$;
39: end
```

Pozostało nam jeszcze oszacować czas działania zaprezentowanego rozwiązania. Łatwo zauważyć, że wszystkie kroki poza główną pętlą (wiersze 16-32) mają złożoność mieszczącą się w ograniczeniu $O(n+m)$. W głównej pętli także rozważamy każdy wierzchołek i każdą krawędź co najwyżej raz, cały algorytm ma więc rzeczywiście złożoność czasową $O(n+m)$.

Powyższe rozwiązanie było rozwiązaniem wzorcowym; jego implementacja znajduje się na dysku dostarczonym do książeczki, a przykład jego działania można zobaczyć na rysunku 4.

Gorsze i błędne rozwiązania

Nieoptymalne wykonanie któregokolwiek kroku algorytmu zazwyczaj prowadziło do rozwiązania o złożoności $O(n \cdot (n+m))$. Nie był to jednak główny powód utraty punktów przez zawodników. Podstawową trudnością była poprawna implementacja rozwiązania z uwzględnieniem wszystkich przypadków szczególnych (na przykład istnienie cyklu zawierającego gmach czy jednoczesne występowanie wierzchołków nieskończonych i skończonych dużych). Dodatkowym utrudnieniem był limit pamięciowy równy 32 MB, który wymuszał oszczędne gospodarowanie pamięcią.



Rys. 4: Działanie rozwiązania drugiego na przykładowym grafie

Testy

Zadanie testowane było na zestawie 10 danych testowych. W poniższej tabeli zamieszczone są przybliżone wartości parametrów dla testów (liczby wierzchołków i krawędzi) oraz krótkie charakterystyki zawartych w nich grafów. Tylko dla trzech punktowanych testów (o numerach 1, 6 i 10) odpowiedzią nie było „`zawsze`”.

Nazwa	**n**	**m**	**Opis**
pro1.in	10	13	prosty graf bez cykli
pro2.in	30	53	dwa rozłączne cykle, wierzchołek skończony i acykliczny podgraf z więcej niż 36 500 trasami do gmachu
pro3.in	22	42	cykl z zerowych wierzchołków, niezerowe wierzchołki tworzące graf acykliczny, niezerowy cykl
pro4.in	100	601	pętla w gmachu, połowa wierzchołków zerowa
pro5.in	13 000	48 500	mały cykl i jeden wierzchołek z wynikiem 36 501
pro6.in	80 000	96 000	graf acykliczny z długimi trasami i kilkoma dodatkowymi krawędziami
pro7.in	100 000	1 000 000	dosyć gęsty graf acykliczny oraz cykle: z wierzchołków zerowych i niezerowych
pro8.in	10 000	1 000 000	losowy, bardzo gęsty graf acykliczny i cykl
pro9.in	1 000 000	1 000 000	prawie losowy test
pro10.in	1 000 000	1 000 000	bardzo duży graf acykliczny i cykle z wierzchołków zerowych

Jakub Radoszewski	**Jakub Radoszewski, Marcin Pilipczuk**	**Marcin Pilipczuk**
Treść zadania	**Opracowanie**	**Program**

OI, Etap I

Tetris 3D

Autorzy gry Tetris postanowili stworzyć nową, trójwymiarową wersję gry, w której prostopadłościenne klocki będą opadać na prostokątną platformę. Podobnie jak w przypadku zwykłej, dwuwymiarowej wersji gry, klocki mają opadać osobno, w pewnej ustalonej kolejności. Dany klocek opada, dopóki nie natrafi na przeszkodę w postaci platformy albo innego, już stojącego klocka, a wtedy się zatrzymuje (w pozycji, w jakiej opadał) i pozostaje na swoim miejscu do końca gry.

Autorzy nowej gry postanowili jednak zmienić charakter gry, ze zręcznościowej na grę logiczną. Znając kolejność opadania klocków na płaszczyznę i tory ich lotu, gracz będzie musiał podać wysokość najwyżej położonego punktu w układzie powstałym po opadnięciu wszystkich klocków. Wszystkie klocki opadają pionowo w dół i nie obracają się w trakcie opadania. Dla ułatwienia wprowadźmy na platformie układ współrzędnych kartezjańskich o początku w jednym z jej narożników i osiach równoległych do jej boków.

Napisz program, który zautomatyzuje sprawdzanie, czy gracz udzielił poprawnej odpowiedzi.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opisy kolejno opadających klocków,*
- *wyznaczy najwyższy punkt układu klocków po zakończeniu ich opadania,*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu wejścia znajdują się trzy liczby całkowite D, S *i* N ($1 \leqslant N \leqslant 20\,000$, $1 \leqslant D, S \leqslant 1\,000$)*, oddzielone pojedynczymi odstępami i oznaczające odpowiednio: długość i szerokość platformy oraz liczbę klocków, które na nią opadną. W następnych* N *wierszach występują opisy kolejnych klocków, po jednym w wierszu.*

Każdy opis klocka składa się z pięciu liczb całkowitych: d, s, w, x *oraz* y ($1 \leqslant d$, $0 \leqslant x$, $d+x \leqslant D$, $1 \leqslant s$, $0 \leqslant y$, $s+y \leqslant S$, $1 \leqslant w \leqslant 100\,000$)*, reprezentujących klocek o długości* d *szerokości* s *i wysokości* w*. Klocek ten będzie opadał na platformę ścianą o wymiarach* $d \times s$*, przy czym długość i szerokość klocka będą równoległe odpowiednio do długości i szerokości platformy. Wierzchołki rzutu klocka na platformę będą miały współrzędne:* (x,y), $(x+d,y)$, $(x,y+s)$ *i* $(x+d,y+s)$*.*

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać dokładnie jedną liczbę całkowitą, oznaczającą wysokość najwyższego punktu w układzie klocków po zakończeniu ich opadania.

Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
6
```

Rozwiązanie

Tetris 2D

Przy rozwiązywaniu wielu problemów trójwymiarowych dobrym pomysłem jest rozważenie odpowiadających im problemów dwuwymiarowych i następnie uogólnienie rozwiązania na trzy wymiary. Dokładnie tak postąpimy w przypadku niniejszego zadania — na początek rozważymy problem, w którym prostokąty spadają na prostą. Aby przeprowadzić symulację tego procesu, wystarczy dla każdej liczby całkowitej x (będziemy ją utożsamiać z punktem $(x,0)$ na prostej OX) pamiętać maksymalną wysokość (oznaczmy ją przez M_x), do jakiej sięga prostokąt, którego rzut na prostą OX zawiera punkt x. Tak więc dla prostokąta spadającego na odcinek $(x,x+d)$, symulację wykonujemy w dwóch krokach:

- w pierwszej kolejności obliczamy, na jakiej wysokości prostokąt zatrzyma się, wyznaczając M — maksimum z wartości M_y dla $y \in (x,x+d)$; rozważamy przedział otwarty, ponieważ nasze prostokąty nie mają brzegu,
- następnie wszystkim punktom z z przedziału $[x,x+d]$ przypisujemy wartość $M_z = M + w$ równą właśnie obliczonemu maksimum, powiększonemu o wysokość prostokąta; wartość tę przypisujemy również punktom leżącym na brzegu podstawy prostokąta, gdyż jeżeli inny prostokąt będzie zawierał je we wnętrzu podstawy, to się na nich zatrzyma.

Opisaną wyżej procedurę można wykonać bardzo prosto w czasie $O(DN)$. Uogólnienie tego rozwiązania na przypadek trójwymiarowy ma już złożoność czasową $O(DSN)$, czyli zdecydowanie za dużą jak na ograniczenia z zadania.

Spróbujmy więc trochę udoskonalić przedstawione rozwiązanie dwuwymiarowe. Aby zwiększyć jego efektywność, zastosujemy popularną strukturę danych — drzewo przedziałowe.

Drzewa przedziałowe

Struktura i własności

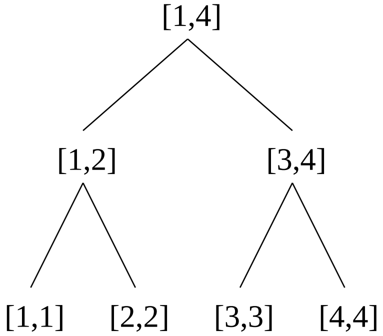
Rozważmy przedział $[1,2^k]$, gdzie k jest liczbą całkowitą nieujemną. *Drzewem przedziałowym* nazwiemy pełne drzewo binarne (czyli drzewo, w którym każdy wierzchołek, oprócz liścia, ma dokładnie dwoje dzieci), w którym:

- korzeń reprezentuje cały przedział $[1,2^k]$, zawierający 2^k elementów,

- dzieci korzenia reprezentują przedziały $[1, 2^{k-1}]$ i $[2^{k-1}+1, 2^k]$,
- dzieciom wierzchołka reprezentującego przedział $[a, b]$ są przypisane połówki $[a, \frac{a+b}{2}]$ i $[\frac{a+b}{2}+1, b]$ tego przedziału,
- liściom są przypisane przedziały jednoelementowe (długości zero).

Będziemy rozróżniać długość przedziału ($b-a$ dla $[a, b]$) od liczby jego elementów, czyli liczby liczb całkowitych w nim zawartych ($b-a+1$ dla przedziału $[a, b]$, gdy a i b są liczbami całkowitymi).

Jeżeli z wierzchołka v_1 można dojść do wierzchołka v_2 ścieżką prowadzącą tylko w górę drzewa, to powiemy, że v_2 jest *przodkiem* v_1 lub że v_1 jest *potomkiem* v_2. W szczególności każdy wierzchołek drzewa jest zarówno swoim przodkiem, jak i potomkiem. Ponadto będziemy mówili, że korzeń znajduje się na *głębokości* zero, jego dzieci na głębokości jeden itd. *Głębokością drzewa* nazwiemy liczbę wierzchołków na ścieżce od korzenia do liścia o maksymalnej głębokości.



Rys. 1: Przykładowe drzewo przedziałowe o głębokości 3

Przedziały przypisane wierzchołkom drzewa przedziałowego nazywamy *przedziałami bazowymi*. Zanim zastosujemy drzewa przedziałowe do rozwiązania zadania, pokażemy kilka ich własności.

Obserwacja 1 *Głębokość drzewa przedziałowego dla przedziału* $[1, 2^k]$ *jest równa* $k+1$, *a całkowita liczba wierzchołków drzewa jest równa* $2 \cdot 2^k - 1$.

Rozmiar drzewa przedziałowego jest więc proporcjonalny do rozmiaru przedziału, a wysokość — do logarytmu z rozmiaru przedziału.

Dowód Korzeń reprezentuje przedział o 2^k elementach. Jego dzieciom przypisane są połówki tego przedziału, zawierające po 2^{k-1} elementów. Dalej, na kolejnym poziomie mamy łącznie 4 wierzchołki, każdy związany z przedziałem zawierającym 2^{k-2} elementów itd. Liczba elementów w przedziale maleje przy przejściu o jeden poziom w głąb dwukrotnie, a zatem na k-tym poziomie otrzymujemy przedziały jednoelementowe — $2^{k-k} = 1$. Stąd widzimy, że drzewo przedziałowe ma głębokość $k+1$.

Oznaczmy przez S łączną liczbę wierzchołków w drzewie. Ponieważ na kolejnych poziomach drzewa liczba wierzchołków podwaja się, a startujemy od jednego korzenia i kończymy na 2^k liściach, więc S wyraża się jako suma ciągu geometrycznego i jest równe:

$$S = 1+2+2^2+\ldots+2^k$$

$$\begin{aligned} &= (1-2^{k+1})/(1-2) \\ &= 2\cdot 2^k - 1 \end{aligned}$$

Jeżeli oznaczymy przez $D = 2^k$ liczbę elementów w przedziale $[1,2^k]$ zawiązanym z korzeniem drzewa, to rozmiar drzewa jest rzędu $O(2\cdot D-1) = O(D)$, a jego wysokość — $O(k+1) = O(\log D)$. ■

Drzewa przedziałowe możemy stosować do przechowywania informacji o odcinkach lub punktach zawartych w przedziale $[1,2^k]$. W typowym schemacie działania algorytmu wykorzystującego drzewo przedziałowe z każdym punktem przedziału $[1,2^k]$ związana jest pewna wartość, a wykonywane są operacje dwojakiego typu:

- zmiana wartości wszystkich punktów zawartych w danym przedziale,
- zapytanie o wartości punktów zawartych w danym przedziale.

Liście służą wówczas do zapisu informacji o punktach, wierzchołki na poziomie wyższym — do zapisu informacji o przedziałach bazowych długości 1, na kolejnym — o przedziałach bazowych długości 2 itd. Aby zapisać w drzewie informację o przedziale innym niż bazowy, musimy najpierw rozłożyć go na sumę parami rozłącznych przedziałów bazowych. Warto zatem zauważyć, iż:

Obserwacja 2 *Każdy przedział zawarty w* $[1,2^k]$ *można rozłożyć na* $O(\log D)$ *rozłącznych przedziałów bazowych.*

Dowód Aby udowodnić powyższą obserwację, przedstawimy rekurencyjny algorytm konstrukcji odpowiedniego rozkładu. Załóżmy, że rozkładamy przedział $[a,b]$ i rozpocznijmy proces rozkładu w korzeniu drzewa, związanym z przedziałem $[1,2^k]$. W każdym kroku algorytmu:

- Na początku znajdujemy się w wierzchołku reprezentującym pewien przedział bazowy $[u,v]$.
- Jeżeli przedział $[u,v]$ jest w całości zawarty w przedziale $[a,b]$, to dołączamy $[u,v]$ do rozkładu i kończymy bieżące wywołanie rekurencyjne.
- Jeżeli lewy syn bieżącego wierzchołka (odpowiadający przedziałowi $[u,\frac{u+v}{2}]$) jest nierozłączny z $[a,b]$, to schodzimy rekurencyjnie do tego syna.
- Podobnie, jeżeli prawy syn ($[\frac{u+v}{2}+1,v]$) przecina się niepusto z przedziałem $[a,b]$, to schodzimy do niego rekurencyjnie.

Udowodnijmy poprawność powyższego algorytmu. Po pierwsze zauważmy, że dla dowolnego przedziału bazowego $[u,v]$, jeśli P jest wierzchołkiem związanym z tym przedziałem, to:

- wszystkie przedziały związane z potomkami P są zawarte w $[u,v]$ — powstają z jego podziału;
- $[u,v]$ jest zawarty we wszystkich przedziałach związanych z przodkami P — powstał z ich podziału;

- oprócz przodków i potomków P nie ma w drzewie przedziałowym innych wierzchołków, których przedziały mają niepusty przekrój z $[u,v]$.

Aby uzasadnić ostatni podpunkt, rozważmy wierzchołek Q niebędący potomkiem P i znajdujący się na niemniejszej głębokości w drzewie niż P. Niech R będzie przodkiem Q leżącym na tej samej wysokości w drzewie, co wierzchołek P — oczywiście $P \neq R$. Ponieważ łatwo widać, że przedziały związane z wierzchołkami leżącymi na tej samej głębokości są rozłączne, więc przedziały wierzchołków P i R są rozłączne, a tym samym przedziały wierzchołków Q i P także są rozłączne.

W przedstawionym algorytmie nigdy nie weźmiemy do rozkładu pary wierzchołków, z których jeden jest przodkiem drugiego. Stąd każdy element przedziału $[a,b]$ będzie zawierał się w co najwyżej jednym przedziale bazowym. Z drugiej strony, jeżeli rozważamy w algorytmie przedział bazowy $[u,v]$ nierozłączny z $[a,b]$, to żaden punkt $x \in [a,b] \cap [u,v]$ nie zostanie pominięty przez algorytm (jeżeli $[u,v]$ nie jest zawarty w $[a,b]$, to i tak jeden z synów $[u,v]$ będzie zawierał x, a zatem nastąpi wywołanie rekurencyjne dla tego syna). To kończy dowód poprawności algorytmu.

Pokażemy teraz, że powyższy algorytm ma złożoność czasową $O(\log D)$, co implikuje, że rozmiar otrzymanego rozkładu jest także $O(\log D)$. W tym celu zauważmy, że na każdym poziomie głębokości drzewa istnieją co najwyżej dwa wierzchołki, dla których zostanie wykonane wywołanie rekurencyjne procedury dla co najmniej jednego z dzieci. Przypuśćmy, że istnieją trzy takie wierzchołki: P_1, P_2 i P_3 (ich przedziały to odpowiednio $[u_1,v_1]$, $[u_2,v_2]$ i $[u_3,v_3]$). Dla ustalenia uwagi przyjmijmy, że P_1 „leży na lewo” od P_2, a P_2 „leży na lewo” od P_3, to znaczy $v_1 < u_2$ i $v_2 < u_3$. Ponieważ każdy z przedziałów $[u_1,v_1]$, $[u_2,v_2]$ i $[u_3,v_3]$ przecina się z $[a,b]$ i żaden nie jest zawarty w $[a,b]$ (tylko w takim przypadku dla odpowiedniego wierzchołka P_i wykonujemy wywołanie rekurencyjne dla jednego z dzieci), stąd istnieją punkty $x_1,x_3 \in [a,b], x_2 \notin [a,b]$, takie że $x_i \in [u_i,v_i]$ dla $i = 1,2,3$. Z przyjętego założenia o wzajemnym położeniu przedziałów wynika, że $x_1 \leqslant x_2 \leqslant x_3$, co stanowi sprzeczność z tym, że tylko $x_2 \notin [a,b]$.

Z właśnie udowodnionego spostrzeżenia wynika, że na każdym poziomie drzewa odwiedzimy co najwyżej 4 wierzchołki (gdyż zejdziemy rekurencyjnie w dół z co najwyżej dwóch wierzchołków na poziomie o jeden wyższym), a zatem liczba wszystkich odwiedzonych w trakcie działania algorytmu wierzchołków nie przekroczy $4 \cdot (k+1) = O(\log D)$. ■

Odnotujmy jeszcze jedno spostrzeżenie, które przyda się nam później. W powyższym algorytmie odwiedzamy wszystkie takie wierzchołki, które są przodkami jakichkolwiek wierzchołków z rozkładu przedziału $[a,b]$ (wynika to z faktu, że algorytm działa zgodnie ze schematem „od korzenia do liści”), czyli wszystkie takie wierzchołki, które odpowiadają przedziałom zawierającym jakiekolwiek przedziały z rozkładu.

Zastosowanie drzewa przedziałowego

Drzewo przedziałowe wykorzystamy do szybszej symulacji dwuwymiarowej gry Tetris. Będziemy w nim przechowywać informacje o przedziałach odpowiadających rzutom poziomych boków prostokątów na prostą, na którą spadają. Przy czym dla każdego dolnego boku wykorzystamy drzewo do obliczenia wysokości, na jakiej zatrzyma się podczas spadania ten odcinek (i cały prostokąt), po czym umieścimy w drzewie informację o górnym boku, by odnotować fakt, że odpowiednia przestrzeń została zajęta przez prostokąt. W każdym węźle drzewa będziemy zapisywać dwojakiego rodzaju informacje:

- maksimum z wysokości dotychczas wstawionych przedziałów, których rozkłady zawierają przedział bazowy odpowiadający danemu wierzchołkowi drzewa — oznaczmy tę wartość przez M;
- maksimum z wartości M dla wszystkich potomków danego węzła — tę wartość oznaczymy m.

Opiszemy teraz, w jaki sposób efektywnie obliczać powyższe wartości i jak z nich korzystać w symulacji gry. Początkowo wartości M i m dla każdego wierzchołka drzewa ustawiamy na 0.

Opadanie prostokąta — przypadek bazowy Na początek spróbujemy przeprowadzić symulację spadania prostokąta P o dolnej podstawie odpowiadającej przedziałowi bazowemu $[a,b]$. Prostokąt ten zatrzyma się na górnym boku $[u,v]$ pewnego innego, wcześniej zrzuconego prostokąta — rozważmy rozkład $[u,v]$ na przedziały bazowe $[u_i,v_i]$. Zauważmy, że co najmniej jeden z tych przedziałów przecina się niepusto z $[a,b]$ (gdyż zatrzymał opadający prostokąt o podstawie $[a,b]$), ale także co najwyżej jeden taki przedział ma punkt wspólny z $[a,b]$ (gdyż żadne dwa przedziały spośród $[u_i,v_i]$ nie przecinają się i konstruując rozkład $[u,v]$ nigdy nie rozkładamy przedziału bazowego całkowicie zawartego w $[u,v]$ na mniejsze przedziały bazowe). Oznaczmy szukany jedyny przedział o powyższej własności przez $[u_k,v_k]$. Zachodzi dokładnie jedna z poniższych sytuacji:

1. $[u_k,v_k]$ jest zawarty w $[a,b]$ (węzeł drzewa odpowiadający $[u_k,v_k]$ jest potomkiem węzła odpowiadającego $[a,b]$),
2. $[u_k,v_k]$ zawiera przedział $[a,b]$ (węzły odpowiadające przedziałom mają odwrotne ułożenie niż w poprzednim punkcie: węzeł $[a,b]$ jest potomkiem $[u_k,v_k]$).

Stąd wysokość, na jaką opadnie prostokąt P, możemy wyznaczyć licząc maksimum z wysokości wszystkich przedziałów bazowych, zawartych w $[a,b]$ lub zawierających $[a,b]$. Pierwszą z tych możliwości możemy rozpatrzyć, biorąc po prostu wartość m z wierzchołka odpowiadającego przedziałowi $[a,b]$ (wynika to wprost z definicji m), drugą zaś, licząc maksimum z wartości M na ścieżce od korzenia do węzła związanego z $[a,b]$ — tam są wszystkie przedziały zawierające $[a,b]$.

Opadanie prostokąta — przypadek ogólny Rozważmy teraz przypadek, gdy $[a,b]$ nie musi być przedziałem bazowym. Można zauważyć, że wysokość, na jakiej zatrzyma się prostokąt P, wyraża się jako maksimum z wysokości, na jakie mogłyby opaść prostokąty, których podstawami są przedziały bazowe $[a_i,b_i]$, powstałe z rozkładu $[a,b]$. W takim razie, na podstawie poprzednich rozważań wynik dla $[a,b]$ możemy policzyć jako maksimum z:

1. wartości m odpowiadających wierzchołkom związanym z $[a_i,b_i]$,
2. wartości M z przodków wierzchołków odpowiadających $[a_i,b_i]$.

Przypomnijmy, że w algorytmie rozkładu przedziału na przedziały bazowe znajdujemy wszystkie wierzchołki z pierwszej grupy, a po drodze przechodzimy przez wszystkie wierzchołki z drugiej grupy. Stąd modyfikując nieznacznie algorytm możemy wyliczyć szukane maksimum równe wysokości opadnięcia prostokąta H — złożoność czasowa tej procedury wynosi $O(\log D)$.

Zapisanie prostokąta w drzewie Po opadnięciu górny brzeg prostokąta zatrzyma się zatem na wysokości $H+h$, gdzie h to wysokość samego prostokąta. Powinniśmy więc zaktualizować wartości M i m w węzłach drzewa, by zapisać informację o odcinku $[a,b]$ znajdującym się na wysokości $H+h$. Zgodnie z definicją, musimy poprawić wartości M dla wszystkich przedziałów bazowych z rozkładu przedziału $[a,b]$; wierzchołki im odpowiadające można otrzymać w wyniku wykonania algorytmu rozkładu $[a,b]$. Wskutek modyfikacji tych wartości M, zmianie mogą ulec wartości m pewnych wierzchołków — mogą to być wyłącznie wierzchołki, będące przodkami wierzchołków odpowiadających $[a_i,b_i]$. Widzimy więc, że wszystkie wierzchołki, których wartości m lub M trzeba zmienić, zostaną skonstruowane przez algorytm rozkładu przedziału na przedziały bazowe. Stąd wszystkie konieczne aktualizacje wartości M i m można wykonać w złożoności czasowej $O(\log D)$.

Drobny szczegół Pozostaje jeszcze do rozpatrzenia pewien drobny, ale istotny szczegół. Prostokąty, z którymi mamy do czynienia, nie zawierają brzegu, w związku z czym właściwie powinniśmy za podstawę prostokąta uznać przedział otwarty (a,b), zawierający tylko punkty całkowite $a+1, a+2, \ldots, b-1$. Gdybyśmy bowiem za podstawę wzięli przedział domknięty $[a,b]$, to moglibyśmy uznać, iż prostokąt zatrzyma się na innym, z którym jedynie się styka (na przykład prostokąty o podstawach $[a,b]$ i $[b,c]$). Z drugiej strony, zawężenie odcinków do najbliższych liczb całkowitych spowodowałoby inne błędy — wówczas na przykład prostokąt o podstawie $(a,a+1)$ nie blokowałby żadnego innego — do czego oczywiście również nie możemy dopuścić. Rozwiązanie tej kłopotliwej sytuacji zasygnalizowaliśmy już na początku. Otóż szukając przeszkód, które mogą zatrzymać prostokąt o podstawie (a,b) będziemy rozważać przeszkody w przedziale $[a+1,b-1]$. W ten sposób uwzględnimy fakt, że prostokąt nie może zatrzymać się na innym, z którym styka się tylko brzegiem. Z kolei wpisując do drzewa informacje, nad którymi punktami prostokąt zajmuje przestrzeń, będziemy jego podstawę traktować jak odcinek domknięty $[a,b]$, co pozwoli nam zaznaczyć, że prostokąt ten zablokuje wszystkie prostokąty, które zawierają punkty z tego przedziału jako wewnętrzne.

Powyższe modyfikacje nie zmieniają złożoności czasowej procedury symulacji spadania prostokąta, więc ostatecznie złożoność całego algorytmu możemy oszacować przez $O(N\log D)$, co daje istotną poprawę w stosunku do naiwnego algorytmu o złożoności $O(ND)$. W kolejnym rozdziale to właśnie ten algorytm będziemy starali się uogólnić na przypadek trójwymiarowy.

Tetris 3D

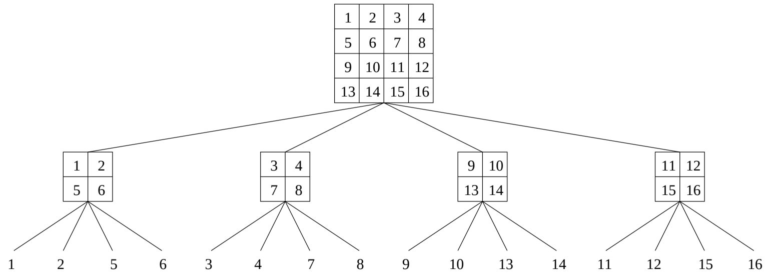
Poszukując rozwiązania dla przypadku trójwymiarowego przeanalizujemy kilka różnych sposobów uogólnienia rozwiązania dwuwymiarowego.

Wiele drzew przedziałowych

Możemy sprowadzić przypadek trójwymiarowy do dwuwymiarowego, dzieląc platformę o podstawie rozmiaru $D \times S$ na S platform o podstawie $D \times 1$ (lub D platform o podstawie $1 \times S$). Dla każdej z tych platform możemy zbudować drzewo przedziałowe i każdy spadający klocek o wymiarach $d \times s \times w$ potraktować jako s identycznych prostokątów o rozmiarach $d \times w$, spadających na odpowiednie proste. Wynikiem dla danego klocka jest maksimum z wartości uzyskanych dla prostokątów. Całkowita złożoność symulacji spadania dla jednego klocka to wówczas $O(S \log D)$ (lub symetrycznie $O(D \log S)$), co daje całkowitą złożoność czasową rozwiązania $O(NS \log D)$. Nie jest to jednak wystarczająco szybko jak na ograniczenia z zadania.

Drzewo czwórkowe

Możemy też próbować zbudować strukturę dwuwymiarową analogiczną do drzewa przedziałowego, w której będziemy przechowywać informacje o prostokątach. Takim rozwiązaniem jest *drzewo czwórkowe*, które pozwala reprezentować prostokąty zawarte w ustalonym kwadracie o wymiarach $2^k \times 2^k$. Korzeń tego drzewa odpowiada całemu kwadratowi. Z kolei dzieci korzenia — ma ich dokładnie 4 — odpowiadają kwadratom, powstałym przez podział dużego kwadratu na cztery równe ćwiartki; każde z dzieci ma również 4 dzieci, odpowiadających jego ćwiartkom itd. Przykład drzewa czwórkowego dla $k = 2$ jest pokazany na rys. 2.

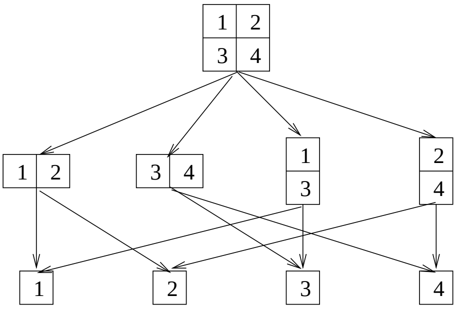


Rys. 2: Przykładowe drzewo czwórkowe

Zanim zaczniemy zapisywać w drzewie czwórkowym informacje potrzebne do symulacji gry, zastanówmy się, na ile kwadratów „bazowych” może zostać rozłożony dany prostokąt. Dla przykładu, prostokąt mający kształt paska o rozmiarze 1×2^k trzeba rozłożyć wyłącznie na kwadraty jednostkowe, do czego potrzebujemy aż 2^k kwadratów! To oznacza, że po adaptacji algorytmu dwuwymiarowego do trzech wymiarów z użyciem drzewa czwórkowego otrzymamy rozwiązanie, które w pesymistycznym przypadku będzie wymagać wykonania co najmniej ND lub NS operacji. Jest to rozwiązanie dalekie od satysfakcjonującego.

Graf prostokątowy

Właściwym uogólnieniem na przypadek trójwymiarowy jest struktura, która już nie jest drzewem, ale acyklicznym grafem skierowanym (określanym często skrótem DAG). Korzeń grafu reprezentuje kwadrat o bokach $2^k \times 2^k$. Wierzchołek ten ma czwórkę dzieci, z których dwójka reprezentuje prostokąty powstałe w wyniku podziału kwadratu symetralną pionową, a dwa pozostałe — symetralną poziomą. W kolejnych wierzchołkach grafu przeprowadzamy analogiczny podział prostokąta związanego z wierzchołkiem. Prostokąty, które odpowiadają poszczególnym wierzchołkom grafu, nazywamy *prostokątami bazowymi*. Całą strukturę określamy mianem *grafu prostokątowego* — na rysunku 3 jest przedstawiony przykład takiego grafu dla $k = 2$.



Rys. 3: Przykładowy graf prostokątowy

Graf prostokątowy ma następujące własności, analogiczne do odpowiednich własności drzewa przedziałowego.

- Każdy prostokąt będący produktem kartezjańskim dwóch jednowymiarowych przedziałów bazowych, z których jeden jest podprzedziałem przedziału $[1, D]$, a drugi podprzedziałem $[1, S]$, jest pewnym prostokątem bazowym i odwrotnie, każdy prostokąt bazowy to produkt kartezjański dwóch przedziałów bazowych.

- Głębokość grafu prostokątowego jest równa $O(\log D \log S)$, gdzie D i S to wymiary prostokąta bazowego zawartego w korzeniu, który zresztą nie musi być nawet kwadratem. Wynika to stąd, że przechodząc od rodzica do dziecka zmniejszamy dwukrotnie jeden z wymiarów prostokąta związanego z wierzchołkiem, więc po $\log D \log S$ krokach oba wymiary muszą być już równe 1.

- Graf zawiera $O(DS)$ wierzchołków, gdyż tyle jest właśnie prostokątów bazowych utworzonych z par jednowymiarowych przedziałów bazowych.

Podobnie jak w przypadku jednowymiarowym (drzewa przedziałowego i odcinka), także dowolny prostokąt możemy rozłożyć na prostokąty bazowe. Dobry podział otrzymamy, rozbijając boki prostokąta na przedziały (odcinki) bazowe i tworząc prostokąty bazowe — iloczyny kartezjańskie par przedziałów bazowych. Metoda ta daje nam rozbicie prostokąta P o wymiarach $d \times s$ ($d \leqslant D$, $s \leqslant S$) na $O(\log D \log S)$ prostokątów bazowych. Jeżeli będziemy chcieli zastosować graf prostokątowy analogicznie, jak drzewo przedziałowe, to musimy jeszcze zastanowić się, ile jest w grafie prostokątów bazowych, zawierających w sobie jakiekolwiek prostokąty bazowe, powstałe z rozkładu prostokąta P. Ich liczbę możemy oszacować zauważając, że wszystkie muszą być produktami kartezjańskimi przedziałów bazowych, zawierających przedziały bazowe z jednowymiarowych rozkładów boków P. Skądinąd wiemy, że takich prostokątów jest w każdym wymiarze logarytmicznie wiele, stąd liczbę szukanych prostokątów bazowych także możemy oszacować przez $O(\log D \log S)$.

Algorytm wzorcowy

Co prawda powyższe spostrzeżenie ogranicza liczbę prostokątów bazowych w rozkładzie konkretnego prostokąta i nawet w sensie matematycznym pozwala je wskazać, potrzebujemy jednak algorytmu, który pozwoli nam efektywnie zidentyfikować je w grafie bez odwoływania się do drzew przedziałowych dla boków prostokąta. Przedstawimy algorytm rekurencyjny analogiczny do algorytmu rozkładu na przedziały bazowe w drzewie przedziałowym.

Załóżmy, że rozkładamy prostokąt $P = [a,b] \times [c,d]$ i rozpocznijmy proces rozkładu w korzeniu grafu, z którym związany jest prostokąt $[1,2^k] \times [1,2^k]$. W każdym kroku algorytmu:

- Na początku znajdujemy się w wierzchołku q reprezentującym pewien prostokąt bazowy $Q = [u,v] \times [x,y]$.
- Jeżeli w trakcie rozkładania P byliśmy już wcześniej w wierzchołku q, to kończymy to wywołanie rekurencyjne.
- Jeżeli prostokąt Q jest w całości zawarty w P, to dokładamy Q do rozkładu i kończymy to wywołanie rekurencyjne.
- Jeżeli nie zachodzi $[u,v] \subseteq [a,b]$, to dokonujemy podziału prostokąta Q w tym wymiarze i wykonujemy zejście rekurencyjne do tych spośród dzieci q, których prostokąty (odpowiednio $[u, \frac{u+v}{2}] \times [x,y]$ oraz $[\frac{u+v}{2}+1, v] \times [x,y]$) nie są rozłączne z P.
- Analogicznie postępujemy w drugim wymiarze: jeżeli nie zachodzi $[x,y] \subseteq [c,d]$, to wykonujemy zejście rekurencyjne do tych spośród dzieci q, których prostokąty (odpowiednio $[u,v] \times [x, \frac{x+y}{2}]$ oraz $[u,v] \times [\frac{x+y}{2}+1, y]$) nie są rozłączne z P.

Widzimy, że powyższy algorytm został tak zaprojektowany, żeby konstruował dokładnie taki rozkład prostokąta, jak produkt kartezjański rozkładów jednowymiarowych — w każdym kroku rozcinamy aktualny prostokąt bazowy wzdłuż każdej osi, jeżeli odpowiedni bok prostokąta nie jest przedziałem bazowym i zostałby podzielony w algorytmie w przypadku jednowymiarowym. Tak więc wszystkie prostokąty, które odwiedzimy po drodze, są postaci $[u',v'] \times [x',y']$, gdzie $[u',v']$ to jeden z przedziałów odwiedzanych w trakcie jednowymiarowego rozkładu odcinka $[a,b]$, a $[x',y']$ to przedział odwiedzany w trakcie jednowymiarowego rozkładu $[c,d]$. Ponieważ w algorytmie unikamy wielokrotnego zagłębiania rekurencyjnego z tego samego wierzchołka, więc jego złożoność jest proporcjonalna do liczby odwiedzonych różnych wierzchołków, czyli wynosi $O(\log D \log S)$.

Wiedząc, że potrafimy efektywnie rozłożyć prostokąt na „umiarkowaną” liczbę prostokątów bazowych w równie umiarkowanym czasie, zastanówmy się, jakie informacje powinniśmy umieścić w grafie, by móc symulować spadanie klocków. Będzie to trochę więcej danych, niż w przypadku gry dwuwymiarowej. W każdym wierzchołku grafu zapiszemy:

- maksimum z wartości dotychczas wstawionych prostokątów, których rozkłady zawierają prostokąt bazowy odpowiadający danemu wierzchołkowi grafu (oznaczenie: M),

- maksimum z wartości M dla wszystkich potomków danego węzła (oznaczenie: m); obie wartości m i M są analogiczne jak w przypadku drzewa przedziałowego,
- maksimum z wartości M wszystkich potomków danego węzła o takim samym rzucie na pierwszy wymiar (oznaczenie: m_1),
- maksimum z wartości M wszystkich potomków danego węzła o takim samym rzucie na drugi wymiar (oznaczenie: m_2).

W powyższym opisie, rzutem prostokąta $[a,b] \times [c,d]$ na pierwszy wymiar jest przedział $[a,b]$, a na drugi — przedział $[c,d]$. Wartości M, m, m_1 i m_2 na początku algorytmu ustawiamy na 0.

Opiszemy teraz, w jaki sposób efektywnie obliczać powyższe wartości i jak z nich korzystać poszukując wysokości, na jakiej zatrzyma się kolejny spadający klocek.

Przypadek bazowy Podobnie jak w przypadku drzewa przedziałowego, na początek przeprowadzimy symulację spadania klocka K o dolnej podstawie odpowiadającej prostokątowi bazowemu $P = [a,b] \times [c,d]$. Klocek ten zatrzyma się na górnej podstawie $Q = [u,v] \times [x,y]$ pewnego innego klocka; rozważmy rozkład $[u,v] \times [x,y]$ na prostokąty bazowe. Co najmniej jeden z tych prostokątów przecina się niepusto z P; oznaczmy dowolny z prostokątów bazowych o tej własności przez $Q_k = [u_k,v_k] \times [x_k,y_k]$. Zachodzi jedna z poniższych sytuacji:

1. $Q_k \subseteq P$ (węzeł odpowiadający Q_k jest potomkiem węzła odpowiadającego P) — przypadek analogiczny jak w wersji dwuwymiarowej;
2. $P \subseteq Q_k$ (węzeł odpowiadający P jest potomkiem węzła odpowiadającego Q_k) — także przypadek analogiczny jak w wersji dwuwymiarowej;
3. $[a,b] \subset [u_k,v_k]$, ale $[x_k,y_k] \subset [c,d]$ — czyli P zawiera się w pierwszym wymiarze w Q_k,a w drugim wymiarze zachodzi zależność odwrotna — wtedy węzeł odpowiadający Q_k jest potomkiem o tym samym pierwszym wymiarze pewnego węzła $[u_k,v_k] \times [x',y']$, występującego przy rozkładaniu P na prostokąty bazowe,
4. $[c,d] \subset [x_k,y_k]$, ale $[u_k,v_k] \subset [a,b]$ — czyli P zawiera się w drugim wymiarze w Q_k,a w pierwszym wymiarze zachodzi zależność odwrotna — wtedy węzeł odpowiadający Q_k jest potomkiem o tym samym drugim wymiarze pewnego węzła $[u',v'] \times [x_k,y_k]$, występującego przy rozkładaniu P na prostokąty bazowe.

Policzenie wysokości zatrzymania klocka K wymaga uwzględnienia każdej z powyższych sytuacji. Pierwsze dwa przypadki rozpatrujemy podobnie jak w zadaniu dwuwymiarowym: pierwszy odpowiada wzięciu wartości m z wierzchołka odpowiadającego prostokątowi P, drugi zaś maksimum z wartości M wszystkich przodków P w grafie. Wysokość zatrzymania wynikającą z sytuacji opisywanej w trzecim przypadku, znajdujemy wyliczając maksimum z wartości m_1 wszystkich napotkanych w procesie rozkładu P prostokątów bazowych, których pierwszy wymiar zawiera $[a,b]$, a drugi jest zawarty w $[c,d]$. Przypadek czwarty rozpatrujemy analogicznie, używając wartości m_2.

Przypadek ogólny Jesteśmy już gotowi do przeprowadzenia symulacji spadania klocka K, którego podstawa P jest dowolnym prostokątem. Podobnie jak w dwóch wymiarach, rozkładamy P na prostokąty bazowe P_i i symulujemy oddzielnie opadanie każdego z nich. Ostateczną wysokość, na jakiej osiądzie cały klocek, wyznaczamy jako maksimum wyników obliczonych dla poszczególnych prostokątów z rozkładu, czyli maksimum z:

1. wartości m odpowiadających wierzchołkom związanym z P_i,
2. wartości M z przodków wierzchołków odpowiadających P_i,
3. wartości m_1 z tych spośród przodków wierzchołków P_i, które w drugim wymiarze są zawarte w P_i,
4. wartości m_2 z tych spośród przodków wierzchołków P_i, które w pierwszym wymiarze są zawarte w P_i.

Ponieważ w procedurze rozkładu na prostokąty bazowe odwiedzimy wszystkie wierzchołki z powyższych grup 1 i 2, a grupy 3 i 4 są podzbiorami grupy 2, to widzimy, że wysokość H, na jakiej zatrzyma się klocek K, możemy policzyć w łącznej złożoności czasowej $O(\log D \log S)$.

Aktualizacja grafu Górna podstawa klocka K po opadnięciu będzie znajdować się na wysokości $H+h$, gdzie h to wysokość klocka. Wartość tę wykorzystamy do aktualizacji informacji zapisanych w grafie w tych miejscach, gdzie jest to potrzebne. Aktualizacje M i m przeprowadzamy podobnie jak w przypadku dwuwymiarowym: wartość M we wszystkich wierzchołkach odpowiadających P_i, a wartość m — we wszystkich ich przodkach. Wartość m_1 aktualizujemy w tych wierzchołkach spośród napotkanych w algorytmie rozkładu, które w pierwszym wymiarze są zawarte w P. Każdy z tych wierzchołków ma potomka o tym samym pierwszym wymiarze, który jest jednocześnie jednym z wierzchołków odpowiadających pewnemu spośród prostokątów P_i. Co więcej, każdy z przodków jakiegokolwiek z prostokątów P_i o tym samym pierwszym wymiarze zostanie odwiedzony w algorytmie rozkładu, a zatem taki sposób aktualizacji jest wystarczający, aby wszystkie wierzchołki grafu miały poprawne wartości m_1. Analogiczną procedurę przeprowadzamy dla wartości m_2 i drugiego wymiaru. Całą aktualizację wykonujemy w czasie $O(\log D \log S)$, gdyż podobnie jak przy symulacji spadania, wszystkie wierzchołki grafu, w których dokonujemy zmian, napotkamy w procedurze rozkładu P na prostokąty bazowe.

Rozwiązanie z użyciem drzewa przedziałowego ma zatem złożoność czasową $O(N \log D \log S + DS)$, gdzie $O(DS)$ to czas budowy struktury grafu prostokątowego w pamięci. Dokładny opis samej budowy grafu pozostawiamy Czytelnikowi jako ćwiczenie.

Inne rozwiązania

Zadanie Tetris 3D można próbować rozwiązać stosując zupełnie inny pomysł. Dla każdego spadającego klocka można przeglądać wszystkie, które opadły poprzednio, i sprawdzać, czy ich rzuty na platformę przecinają się z aktualnie spadającym. Wysokość, na jakiej zatrzyma się klocek, wyznaczamy wówczas jako maksimum z wysokości takich kolidujących klocków. Tego typu rozwiązanie ma pesymistycznie złożoność $O(N^2)$ i nie bierzemy w nim w ogóle pod uwagę niedużych ograniczeń na długość i szerokość platformy.

Teoretycznie jest to rozwiązanie zbyt wolne, jak na ograniczenia z zadania. Jednakże prawdziwą sztuką było ułożenie na tyle dobrych testów, żeby móc równocześnie wyeliminować bardzo efektywnie zaimplementowane (i czasami niepoprawne) wersje powyższej symulacji i nieefektywne trójwymiarowe uogólnienia drzewa przedziałowego. Oceniając po liczbie punktów uzyskanych przez zawodników, sztuka ta jurorom udała się, a za zadanie Tetris 3D było najtrudniej zdobyć maksymalną liczbę punktów w pierwszym etapie XIII Olimpiady Informatycznej.

Testy

Zadanie testowane było na 14 zestawach danych testowych, które zawierały łącznie aż 55 testów.

Nazwa	**D**	**S**	**N**	**Opis**
tet1.in	20	20	20	mały test poprawnościowy
tet2a-7a.in	1 000	1 000	20 000	maksymalne testy losowe
tet8a-10a.in	512	512	20 000	duże testy złożone tylko z prostokątów bazowych
tet11a-14a.in	1 000	1 000	20 000	losowe klocki o długości lub szerokości 30-60, a o drugim wymiarze 1 000
tet2b-10b.in	10000	1 000	20 000	testy maksymalne z 19 000 klockami 1 × 1 i 1 000 dużych klocków
tet11b-14b.in	1 000	1 000	20 000	testy maksymalne z 18 000 klockami 1 × 1 i 2 000 dużych klocków
tet2c-10c.in	1 000	1 000	1 000	1 000 losowych klocków o dużej powierzchni
tet11c-14c.in	100	100	100	niewielkie poprawnościowe testy losowe
tet2d-7d.in	1 000	1 000	1 000	testy maksymalne z klockami o jednym boku długości lub szerokości 1, a drugim około 1 000
tet8d-10d.in	100	100	100	niewielkie testy losowe poprawnościowe
tet2e-7e.in	1 000	1 000	20 000	losowe klocki o długości lub szerokości 30-60, a o drugim wymiarze 1 000

Piotr Stańczyk
Treść zadania, Opracowanie

Piotr Stańczyk
Program

OI, Etap I

Żaby

W Bajtocji zapanowała klęska żab — niszczą one wszystkie uprawy. Rolnik Bajtazar postanowił walczyć z żabami za pomocą specjalnych odstraszaczy, które rozmieścił w wybranych miejscach na swoim polu. Każda żaba, przemieszczając się z jednego miejsca do drugiego, stara się omijać odstraszacze w jak największej odległości, tzn. maksymalizuje odległość od najbliższego odstraszacza.

Pole Bajtazara ma kształt prostokąta. Żaby skaczą po polu równolegle do boków pola, przemieszczając się w każdym pojedynczym skoku o jedną jednostkę. Odległość od odstraszaczy jest rozumiana jako minimum z odległości od poszczególnych odstraszaczy dla wszystkich pozycji, na których żaba się znajdowała.

Bajtazar wie skąd i dokąd żaby się najczęściej przemieszczają i eksperymentuje z różnymi rozmieszczeniami odstraszaczy. Poprosił Cię o pomoc, chce abyś napisał program obliczający dla zadanego rozstawienia odstraszaczy, w jakiej odległości od odstraszaczy żaby mogą przedostawać się po jego polu z jednego miejsca do drugiego.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia rozmiar pola, rozmieszczenie odstraszaczy oraz pozycję początkową i końcową żaby,*
- *wyznaczy maksymalną odległość od najbliższego odstraszacza, przy jakim żaba będzie musiała przejść na swojej drodze,*
- *wypisze kwadrat znalezionej odległości na standardowe wyjście.*

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby całkowite: w_x i w_y oddzielone pojedynczym odstępem — szerokość i długość pola ($2 \leqslant w_x, w_y \leqslant 1\ 000$). Drugi wiersz wejścia zawiera cztery liczby całkowite: p_x, p_y, k_x i k_y oddzielone pojedynczymi odstępami; (p_x, p_y) to początkowe położenie żaby, (k_x, k_y) to końcowe położenie żaby ($1 \leqslant p_x, k_x \leqslant w_x$, $1 \leqslant p_y, k_y \leqslant w_y$). Trzeci wiersz wejścia zawiera jedną liczbę całkowitą n — liczbę odstraszaczy rozmieszczonych na polu ($1 \leqslant n \leqslant w_x \cdot w_y$). Kolejne n wierszy zawiera współrzędne kolejnych odstraszaczy. Wiersz $i+3$ dla $1 \leqslant i \leqslant n$ zawiera dwie liczby całkowite x_i i y_i oddzielone pojedynczym odstępem — są to współrzędne i-tego odstraszacza ($1 \leqslant x_i \leqslant w_x$, $1 \leqslant y_i \leqslant w_y$). Każdy odstraszacz znajduje się w innym miejscu i żaden z nich nie znajduje się w punkcie (p_x, p_y) ani (k_x, k_y).

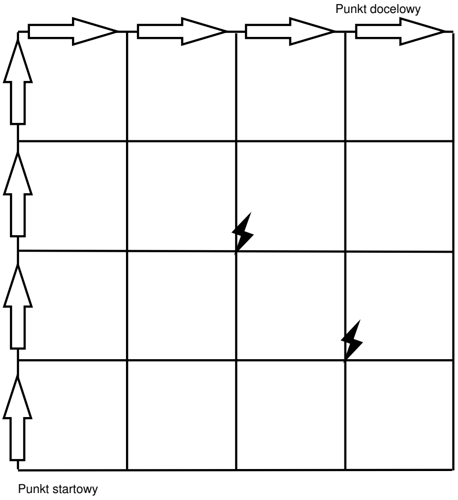
Wyjście

W pierwszym i jedynym wierszu wyjścia powinna znaleźć się jedna liczba całkowita, kwadrat maksymalnej odległości, na jaką żaba musi zbliżyć się do najbliższego odstraszacza. Jeżeli żaba nie może uniknąć wskoczenia bezpośrednio na odstraszacz, to wynikiem jest 0.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*

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



Optymalna droga żaby.

Rozwiązanie

Wprowadzenie

Rozpocznijmy od wprowadzenia terminologii, którą będziemy stosować w opisie rozwiązania. *Pozycją* nazwiemy dowolne miejsce na polu Bajtazara, na którym może znaleźć się żaba (na polu o wymiarach w_x na w_y jest dokładnie $w_x \cdot w_y$ pozycji). Poprzez *odległość danej pozycji od odstraszaczy* rozumieć będziemy minimum z odległości tej pozycji od wszystkich odstraszaczy.

Rozwiązanie zadania podzielimy na dwie niezależne części:

- wyznaczenie dla każdej pozycji odległości od odstraszaczy (faza I),
- wyznaczenie drogi żaby maksymalizującej odległość od odstraszaczy na podstawie odległości pozycji od odstraszaczy (faza II).

Powstaje pytanie, czy taki podział pracy nie spowoduje, że obliczenia będą trwały dłużej niż to konieczne. Jeżeli uda nam się skonstruować algorytm działający w czasie zbliżonym do $w_x \cdot w_y$, to na pewno nie będzie to algorytm zły. Wynika to stąd, że już same dane dla zadania mogą mieć rozmiar $w_x \cdot w_y$ (Bajtazar mógł rozstawić aż tyle odstraszaczy), a każdy — nawet najlepszy — algorytm wymaga przynajmniej przeczytania tych danych.

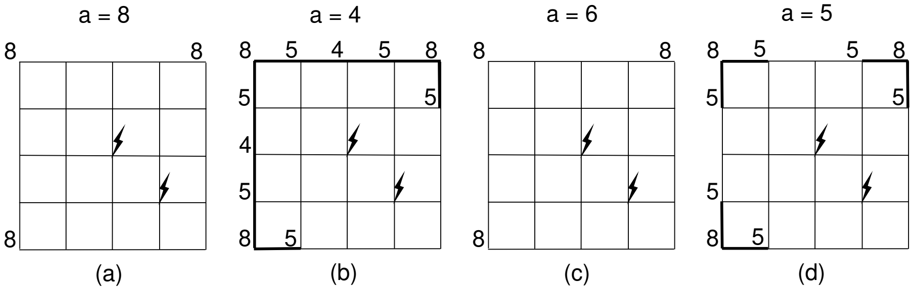
W kolejnych rozdziałach przedstawimy możliwe sposoby realizacji obu faz. Rozpoczniemy dość nietypowo — od fazy drugiej.

Faza II

Ścieżką będziemy nazywali sekwencję sąsiednich pozycji, pomiędzy którymi żaba może bezpośrednio przeskakiwać, z których pierwsza jest pozycją początkową żaby, natomiast ostatnia — pozycją końcową. Przez *odległość ścieżki od odstraszaczy* będziemy rozumieli najmniejszą odległość pozycji należącej do ścieżki od odstraszaczy. Zakładając, że dla każdej pozycji jest już policzona odległość od odstraszaczy, należy wyznaczyć ścieżkę, której odległość od odstraszaczy jest możliwie największa — kwadrat tej odległości, oznaczmy go s_{max}, jest poszukiwanym wynikiem zadania.

Wartość s_{max} jest liczbą całkowitą niemniejszą od 0 (wtedy najlepsza ścieżka musi przejść przez pozycję zawierającą odstraszacz) oraz nie większą od $w_x^2 + w_y^2$ (Bajtazar postawił na polu co najmniej jeden odstraszacz). Na pytanie, czy wartość s_{max} jest niemniejsza od zadanej wartości a można udzielić odpowiedzi w czasie $O(w_x \cdot w_y)$. Jeśli bowiem taka ścieżka istnieje, to musi ona składać się z pól, których odległość od odstraszaczy jest niemniejsza od a. Aby ją znaleźć (lub stwierdzić, że nie istnieje), skonstruujmy graf G_a o zbiorze wierzchołków odpowiadającym pozycjom o odległościach od odstraszaczy niemniejszych niż a. Krawędziami w grafie G_a połączymy takie pozycje, pomiędzy którymi żaba może bezpośrednio przeskoczyć. Ścieżka o odległości co najmniej a od odstraszaczy istnieje wtedy i tylko wtedy, gdy wierzchołki reprezentujące pozycję początkową oraz końcową żaby są połączone ścieżką w grafie G_a (lub inaczej — należą do tej samej składowej spójności). Sprawdzenia tego można dokonać przy użyciu prostego algorytmu przeszukiwania grafu wszerz bądź w głąb. Przeszukiwanie grafu o $O(w_x \cdot w_y)$ wierzchołkach i krawędziach wykonujemy właśnie w takim czasie.

Zapewne Czytelnik spotkał się z grą, której celem jest odgadnięcie liczby naturalnej x wymyślonej przez przeciwnika, przy użyciu pytań postaci „czy x jest mniejsze od a?” (wśród informatyków grę tę nazywa się zazwyczaj *wyszukiwaniem binarnym*). Wykorzystując opisany powyżej algorytm, można wyznaczyć poszukiwaną wartość s_{max} w analogiczny sposób. Początkowo wiemy, że wartość ta znajduje się w przedziale $[0, w_x \cdot w_y]$. Następnie, sprawdzając jak ma się s_{max} do środka przedziału, zawężamy dalsze poszukiwania do przedziału o połowę mniejszego. W ten sposób, zadając logarytmiczną liczbę pytań, można zrealizować drugą fazę algorytmu w czasie $O(w_x \cdot w_y \cdot \log(w_x \cdot w_y))$. Przykład procesu wyszukiwania został przedstawiony na rysunku 1.



Rys. 1: Wyszukiwanie binarne wartości s_{max} dla przykładowych danych z treści zadania (kwadraty odległości poszczególnych pozycji zostały umieszczone przy pozycjach, o ile są niemniejsze od rozpatrywanego w kolejnym kroku ograniczenia a). **(a)** Maksymalna odległość dwóch pól wynosi 16, zatem wyszukiwanie binarne wykonywane jest na przedziale $[0\ldots16]$. Dla ograniczenia 8, pozycja początkowa (lewa-dolna) oraz końcowa (prawa-górna) nie są połączone — rozwiązanie musi znajdować się w takim razie w przedziale $[0\ldots7]$ **(b)** Dla kolejnej testowanej wartości $a=4$, pozycja początkowa i końcowa są połączone, więc rozwiązanie musi mieścić się w przedziale $[4\ldots7]$. **(c)** Dla $a=6$ pozycje znowu nie są połączone — rozwiązanie znajduje się w przedziale $[4\ldots5]$. **(d)** Dla $a=5$ po raz kolejny pozycje nie są połączone, zatem wynikiem końcowym jest 4.

Inne możliwe rozwiązania

Istnieją również inne sposoby realizacji drugiej fazy algorytmu. Dla przykładu przedstawimy dwie z nich — obie sprowadzają analizowany problem do zagadnienia grafowego.

Pierwszym ze sposobów — dość często implementowanym przez uczestników XIII Olimpiady Informatycznej — było przedstawienie problemu w postaci grafu G, którego wierzchołkami są pozycje i krawędzie łączą te pozycje, pomiędzy którymi żaba może bezpośrednio przeskoczyć. Po skonstruowaniu grafu G, do wyznaczenia poszukiwanej ścieżki można wykorzystać modyfikację algorytmu Dijkstry (patrz [18]). W oryginalnej wersji tego algorytmu mamy graf, którego krawędziom są przypisane długości, i wyliczamy długości najkrótszych ścieżek do wszystkich wierzchołków grafu z zadanego źródła (długość ścieżki rozumiana jest jako suma długości krawędzi na niej leżących). W naszym problemie wagi są przypisane wierzchołkom, a nie krawędziom, natomiast poszukiwanym wynikiem jest ścieżka między wierzchołkiem początkowym a docelowym, dla której najmniejsza waga wierzchołka do niej należącego jest maksymalna. Zadanie możemy rozwiązać modyfikując algorytm Dijkstry tak, by obliczać dla każdego wierzchołka v najmniejszą wagę w wierzchołka na ścieżce prowadzącej z wierzchołka początkowego do v. Wówczas wierzchołki musimy odwiedzać w kolejności nierosnących wartości w. Implementacja tego algorytmu (przy użyciu kopca jako kolejki priorytetowej dla wierzchołków) ma złożoność proporcjonalną $O(w_x \cdot w_y \cdot \log(w_x \cdot w_y))$.

Drugi sposób jest modyfikacją algorytmu Kruskala wyznaczania najlżejszych drzew rozpinających w grafie (patrz [18]). W oryginalnym algorytmie rozpoczynamy działanie od grafu zawierającego jedynie wierzchołki, a następnie przeglądamy krawędzie w kolejności niemalejących wag, dodając do grafu te, które łączą różne spójne składowe grafu. Kończymy w momencie, gdy graf zawiera dokładnie jedną spójną składową. Algorytm możemy zmodyfikować tak, by budować graf stopniowo dodając nie krawędzie, lecz wierzchołki (wraz ze wszystkimi incydentnymi do nich krawędziami) w kolejności nierosnących odległości od odstraszaczy. Ponadto obliczenia kończymy w chwili, gdy pozycja początkowa i końcowa żaby znajdą się w tej samej składowej. Kwadrat odległości od odstraszaczy ostatniego dodanego do grafu wierzchołka jest wówczas poszukiwaną wartością s_{max}. Czas działania tego algorytmu (ze względu na konieczność posortowania wierzchołków) to ponownie $O(w_x \cdot w_y \cdot \log(w_x \cdot w_y))$.

Faza I

Prostym i nasuwającym się od razu sposobem wyznaczenia wyniku dla pierwszej fazy jest obliczenie dla każdej pozycji odległości od wszystkich odstraszaczy, a następnie wybranie najmniejszej z tych wartości. Podejście takie jest bardzo proste w implementacji, jednak wyjątkowo nieefektywne — przy maksymalnej możliwej liczbie odstraszaczy jego złożoność czasowa to $O(w_x^2 \cdot w_y^2)$. Wielu zawodników zauważyło, że taki algorytm nie jest wystarczający i starało się wymyślić efektywniejszą metodę. My także, mając na względzie fakt, iż druga faza algorytmu została zrealizowana w czasie $O(w_x \cdot w_y \cdot \log(w_x \cdot w_y))$, skupimy się na poszukiwaniu rozwiązania o niegorszej złożoności.

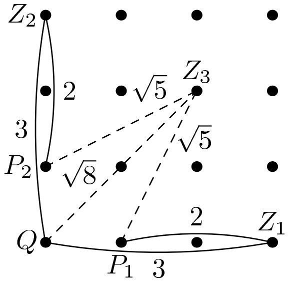
Rozwiązania błędne

Jednym z możliwych pomysłów jest, podobnie jak to zostało zrobione w drugiej fazie, potraktowanie pola Bajtazara jako grafu G, w którym wierzchołki oznaczają pozycje na polu, a krawędzie — skoki żaby. Na takim grafie możemy wykonać algorytm przeszukiwania wszerz, rozpoczynając od wszystkich pozycji odstraszaczy jednocześnie. Pozycjom tym przypisujemy odległość zero, a następnie przeglądając kolejne wierzchołki przypisujemy im obliczone minimalne odległości od odstraszaczy. Algorytm taki ma złożoność czasową $O(w_x \cdot w_y)$ i jest poprawny w przypadku stosowania metryki miejskiej do wyznaczania odległości między pozycjami, jednak w przypadku metryki Euklidesowej, z jaką mamy do czynienia w zadaniu, podejście takie nie jest poprawne (w metryce miejskiej definiujemy odległość między pozycjami a i b jako $|a_x - b_x| + |a_y - b_y|$, a w metryce Euklidesowej, jako $(a_x - b_x)^2 + (a_y - b_y)^2$).

Kolejnym podejściem może być zastąpienie przeszukiwania wszerz algorytmem Dijkstry. Dla każdej pozycji obliczymy i zapiszemy odległość od najbliższego odstraszacza oraz pozycję najbliższego odstraszacza (pozycja źródłowa).

- Na początku dla każdej pozycji P zawierającej odstraszacz zapisujemy odległość równą 0 i pozycję źródłową równą P; pozostałym pozycjom przypisujemy wstępnie odległości $+\infty$.
- Wszystkie wierzchołki z wyznaczoną odległością 0 wstawiamy do kolejki priorytetowej,
- Przetwarzamy pozycje z kolejki, w kolejności niemalejących odległości. Dla pozycji P, której źródłem jest pozycja Z, odwiedzamy każdego sąsiada S pozycji P (czyli pozycje, na które żaba może bezpośrednio przeskoczyć z P). Jeśli odległość pozycji S od Z jest mniejsza niż dotychczasowa odległość S od jej źródła, to przypisujemy S nowe źródło Z i odpowiednio zmieniamy odległość.

Powyższy algorytm został zaprogramowany przez wielu zawodników. Nie jest on jednak poprawny, co można łatwo wykazać. Niech $Z_1(4,1)$, $Z_2(1,4)$, $Z_3(3,3)$ będą pozycjami odstraszaczy na polu Bajtazara. Przedstawiony algorytm dla pozycji $P_1(2,1)$ wyznaczy źródło Z_1 i odległość 2, dla pozycji $P_2(1,2)$ źródłem będzie Z_2, a odległością 2. Dla obu pól P_1 i P_2 odległość do odstraszacza Z_3 wynosi $\sqrt{5}$ i jest większa od 2. Pozycja Q jest sąsiadem P_1 i P_2, zatem źródło dla pozycji Q zostanie ustalone na Z_1 lub na Z_2, a co za tym idzie odległość od odstraszaczy będzie ustalona na 3. Tymczasem odległość Q od Z_3 jest mniejsza i wynosi $\sqrt{8}$.



Rys. 2: Przykładowa sytuacja, dla której zmodyfikowany algorytm Dijkstry daje złą odpowiedź

Dziwna funkcja g

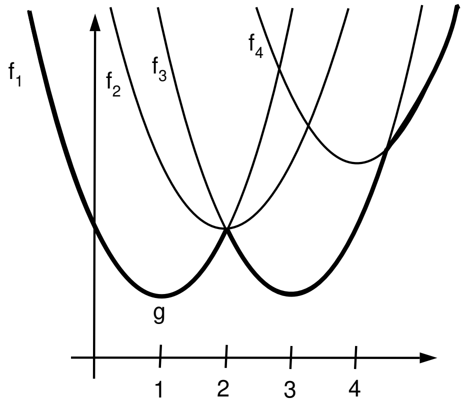
Wszystkie powyższe kłopoty wynikały z nietypowości metryki, która jest zastosowana w zadaniu. Okazuje się jednak, że istnieje stosunkowo prosty (implementacyjnie, acz nie koniecznie koncepcyjnie) sposób wyznaczenia wyniku dla fazy pierwszej w złożoności czasowej $O(w_x \cdot w_y)$.

Rozpatrzmy trochę inny problem od tego, który musimy rozwiązać. Niech dana będzie funkcja $b : \{1,2,\ldots,N\} \to R$. Naszym zadaniem jest wyznaczenie funkcji $g : \{1,2,\ldots,N\} \to R$, zadanej wzorem:

$$g(x) = \min_{n \in \{1,2,\ldots,N\}} ((x-n)^2 + b(n))$$

Przykładowo, dla $N = 4$, $b(1) = 1$, $b(2) = 2$, $b(3) = 1$ i $b(4) = 3$ wartości poszukiwanej funkcji g są następujące: $g(1) = 1$, $g(2) = 2$, $g(3) = 1$, $g(4) = 2$

Po rozszerzeniu dziedziny funkcji g do zbioru liczb rzeczywistych, wyznaczanie wartości $g(x)$ można traktować jako znajdowanie najmniejszej wartości rodziny funkcji kwadratowych. Wykres funkcji g powstaje z fragmentów paraboli rozpatrywanych funkcji kwadratowych. Wykresy dwóch funkcji kwadratowych $f_1(x) = (x-a_1)^2 + b_1$ oraz $f_2(x) = (x-a_2)^2 + b_2$, $a_1 < a_2$ mają dokładnie jeden punkt przecięcia $c_{(1,2)} = (c_x, c_y)$ (łatwo go wyznaczyć z równania $(x-a_1)^2 + b_1 = (x-a_2)^2 + b_2$). Funkcja f_1 ma mniejsze wartości od funkcji f_2 dla $x < c_x$, natomiast dla $x > c_x$ sytuacja jest odwrotna. Zatem wykres funkcji g zdefiniowanej jako minimum dwóch funkcji f_1 i f_2 składa się z fragmentu paraboli funkcji f_1 położonego na lewo od punktu $c_{1,2}$ (czyli $g(x) = f_1(x)$ dla $x \leqslant c_x$) oraz z fragmentu paraboli f_2 położonego na prawo od tego punktu (czyli $g(x) = f_2(x)$ dla $x \geqslant c_x$). Łatwo zauważyć, że funkcja g wyznaczona przez k funkcji f_i ($1 \leqslant i \leqslant k$) składa się z co najwyżej k fragmentów paraboli (niektóre parabole mogą wcale nie wchodzić w skład wykresu g). Przykład funkcji g wyznaczonej przez rodzinę czterech funkcji kwadratowych jest przedstawiony na rysunku 3).



Rys. 3: Postać poszukiwanej funkcji g wyznaczonej przez rodzinę funkcji f_1, f_2, f_3 i f_4

Załóżmy, że mamy wyznaczoną funkcję $g = min(f_1, f_2, \ldots, f_k)$, gdzie $f_1(x) = (x-a_1)^2 + b_1$, $f_2(x) = (x-a_2)^2 + b_2$, …, $f_k(x) = (x-a_k)^2 + b_k$, $a_1 < a_2 < \ldots < a_k$. Funkcję g możemy przedstawić jako ciąg funkcji ją wyznaczających, przeplatanych punktami przecięcia sąsiednich funkcji — reprezentację tę oznaczymy $R(g)$:

$$R(g) = [f_{w_1}, c_{(w_1,w_2)}, f_{w_2}, c_{(w_2,w_3)}, \ldots, c_{(w_{l-1},w_l)}, f_{w_l}]$$

Przykładowo, reprezentacja funkcji g z rysunku 3 wygląda następująco:

$$R(g) = [f_1, (2,2), f_3, (4.5, 3.25), f_4]$$

Tak zapisaną funkcję g możemy łatwo modyfikować dodając kolejne funkcje z rodziny f. Aby dodać funkcję $f_{l+1}(x) = (x - a_{l+1})^2 + b_{l+1}$, gdzie $a_l < a_{l+1}$ (dla uproszczenia oznaczeń załóżmy przez chwilę, że $R(g) = [f_1, c_{1,2}, f_2, \ldots, f_{l-1}, c_{l-1,l}, f_l]$), wystarczy usuwać z reprezentacji $R(g)$ od końca kolejne funkcje $f_i = f_l, f_{l-1}, \ldots$ tak długo, dopóki punkt przecięcia usuwanej funkcji f_i z funkcją f_{l+1} znajduje się na lewo od punktu przecięcia f_i z funkcją f_{i-1}. Następnie należy dodać na końcu reprezentacji punkt przecięcia f_i z f_{l+1} oraz funkcję f_{l+1}.

Wyznaczenie reprezentacji funkcji g zdefiniowanej przez k funkcji z rodziny f zajmuje czas $O(k)$, gdyż każda funkcja z rodziny f jest raz wstawiana do reprezentacji i co najwyżej raz z niej usuwana (oczywiście przy założeniu, że funkcje z rodziny f mamy uporządkowane według wartości a_i). Także wyznaczenie wartości funkcji g dla kolejnych argumentów $1, 2, \ldots, k$ może zostać w prosty sposób wykonane w czasie $O(k)$ przy użyciu jej reprezentacji $R(g)$.

Poprawne rozwiązanie

Rozpatrzmy teraz pole Bajtazara, którego wymiary wynoszą $w_x \times 1$. Zauważmy, że zdefiniowaną niżej funkcję f_i możemy interpretować, jako kwadrat odległości od odstraszacza stojącego na pozycji $(i,1)$ (jeśli tam jest jakiś odstraszacz) dla wszystkich pozycji $(x,1)$, gdzie $1 \leqslant x \leqslant w_x$,

$$f_i(x) = (x-1)^2 + o_i,$$

o ile zdefiniujemy $o_i = 0$, jeśli na pozycji $(i,1)$ stoi odstraszacz, oraz $o_i = \infty$ w przeciwnym wypadku.
Kwadrat odległości pozycji (x,1) od najbliższego odstraszacza, to z kolei wartość funkcji:

$$g(x) = min(f_1(x), f_2(x), \ldots, f_{w_x}(x)).$$

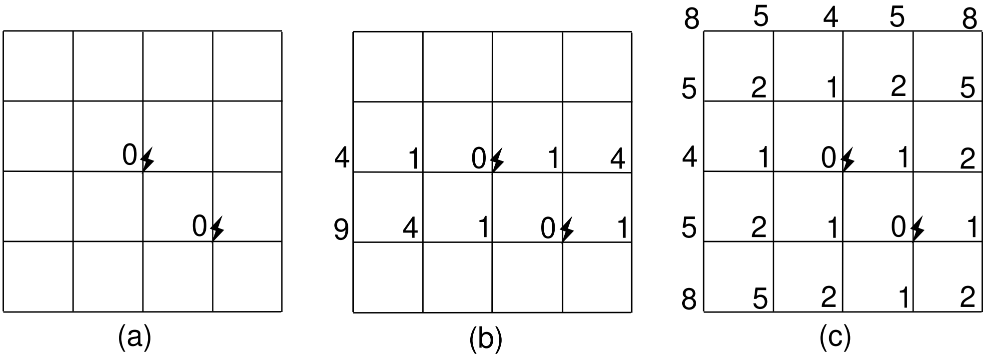
Z poprzedniego rozdziału wiemy, że wartości te dla argumentów $x = 1,2,\ldots,w_x$ potrafimy wyznaczyć w czasie $O(w_x)$.

Obliczenie wyniku w przypadku pola dwuwymiarowego nie wymaga już dużo pracy. Wystarczy najpierw dla każdego wiersza pola Bajtazara wyznaczyć powyższą metodą odległości od odstraszaczy znajdujących się w tym wierszu. W następnym kroku powtórzymy takie same obliczenia dla kolumn, wychodząc od wartości wyliczonych w poprzednim kroku. Jeśli dla pewnej kolumny wartości wyliczone dla pozycji w niej występujących podczas pierwszego przebiegu będą równe $c_1, c_2, \ldots, c_{w_y}$, to rodzinę funkcji f definiujemy jako $f_i = (x-i)^2 + c_i$ (dla $1 \leqslant i \leqslant w_y$) i dalej:

$$g(x) = min(f_1(x), f_2(x), \ldots, f_{w_y})$$

Na rysunku 4 zaprezentowane jest działanie algorytmu na przykładzie z treści zadania. Wyznaczenie odległości wszystkich pozycji na polu Bajtazara od odstraszaczy poprzez wyznaczenie funkcji g jest realizowane w łącznym czasie $O(w_x \cdot w_y)$.

Powyższa metoda z pewnością wymaga dowodu poprawności. Weźmy zatem dowolną pozycję (p_x, p_y), dla której najbliższy odstraszacz jest umieszczony na pozycji (o_x, o_y). W pierwszym kroku algorytmu, przetwarzając wiersze pola, wyznaczymy dla pozycji (p_x, o_y) wartość $(o_x - p_x)^2$ (gdyby wyznaczona wartość była mniejsza, to oznaczałoby to istnienie bliższego pozycji (p_x, o_y) odstraszacza w wierszu o_y, ale taki odstraszacz byłby bliższy pozycji (p_x, p_y) niż odstraszacz (o_x, o_y)). W drugim kroku, przetwarzając kolumny, dla pozycji (p_x, p_y) wyznaczymy (na podstawie wartości $(o_x - p_x)^2$ dla pozycji (p_x, o_y)) wartość $(o_x - p_x)^2 + (o_y - p_y)^2$, czyli faktycznie kwadrat odległości między pozycją (p_x, p_y) a najbliższym odstraszaczem (znów mniejszy wynik oznaczałby istnienie bliższego niż (o_x, o_y) odstraszacza).



Rys. 4: Proces wyznaczania odległości pól od odstraszaczy. **(a)** Początkowe wartości (nieskończoności nie zostały zaznaczone). **(b)** Wyznaczenie odległości od odstraszaczy w wierszach. **(c)** Wyznaczenie odległości od odstraszaczy w kolumnach

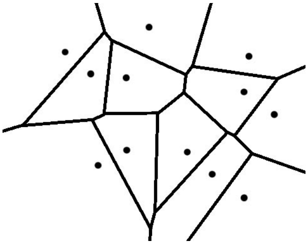
Testy

Zadanie było testowane na zestawie 20 danych testowych. Połowa testów to testy poprawnościowe, mające na celu zweryfikowanie poprawności zastosowanych przez zawodników algorytmów. Reszta testów to testy poprawnościowo-wydajnościowe. Poniższa tabela zawiera listę testów wraz z ich podstawową charakterystyką — wymiarami pola w_x i w_y oraz liczbą odstraszaczy n:

Nazwa	w_x	w_y	n	Opis
zab1.in	4	4	3	test poprawnościowy
zab2.in	10	10	6	test poprawnościowy
zab3.in	10	10	15	test poprawnościowy
zab4.in	50	10	20	test poprawnościowy
zab5.in	11	50	15	test poprawnościowy
zab6.in	50	50	160	test poprawnościowy
zab7.in	50	50	199	test poprawnościowy
zab8.in	50	39	334	test poprawnościowy
zab9.in	50	50	350	test poprawnościowy
zab10.in	31	32	213	test poprawnościowy
zab11.in	400	400	120	test wydajnościowo-poprawnościowy
zab12.in	1 000	200	100	test wydajnościowy
zab13.in	200	1 000	50	test wydajnościowy
zab14.in	1 000	400	20 000	test wydajnościowy
zab15.in	800	200	200	test wydajnościowy
zab16.in	1 000	1000	23	test wydajnościowo-poprawnościowy
zab17.in	999	999	10 016	test wydajnościowo-poprawnościowy
zab18.in	999	999	76 151	test wydajnościowo-poprawnościowy
zab19.in	1 000	1 000	50 000	test wydajnościowo-poprawnościowy
zab20.in	1 000	1 000	200 003	test wydajnościowo-poprawnościowy

Ciekawostka

Gdyby rozpatrywać poruszanie się żaby w zadaniu w sposób ciągły (żaba nie wykonuje skoków, lecz przesuwa się płynnie pomiędzy punktem startowym a docelowym), to zadanie można by rozwiązać w złożoności $O(n \cdot \log n)$, gdzie n jest liczbą odstraszaczy. Rozwiązanie takie jest możliwe dzięki zastosowaniu tzw. *Diagramów Voronoia* (można przeczytać o nich więcej w książce [33] lub [34]). Diagram taki można wyznaczyć dla dowolnego zbioru n punktów na płaszczyźnie $P_1, \ldots, P_n$ i stanowi on podział płaszczyzny na n rozłącznych obszarów o kształcie wielokątów wypukłych. Obszar zawierający punkt P_i składa się ze wszystkich takich punktów płaszczyzny, których odległość od P_i jest niewiększa od ich odległości od $P_1, \ldots, P_{i-1}, P_{i+1}, \ldots, P_n$.



Rys. 5: Przykładowy *Diagram Voronoia* wyznaczony dla zbioru 12 punktów.

Wyznaczenie *Diagramu Voronoia* jest możliwe w czasie $O(n \cdot \log n)$, jednak implementacja algorytmu jest bardzo skomplikowana. Po wyznaczeniu diagramu można potraktować go jako graf, w którym wierzchołkami są punkty przecięcia odcinków diagramu, natomiast krawędziami grafu — odcinki diagramu. Każdej krawędzi należy przydzielić wagę równą odległości odpowiadającego odcinka od najbliższego odstraszacza (jest to zadanie proste, gdyż najbliższym punktem każdego odcinka jest ten punkt, którego obszar odcinek wyznacza), a następnie postępować podobnie, jak w drugiej fazie przedstawionego wcześniej algorytmu — zastosować algorytm Dijkstry do wyznaczenia końcowego wyniku. Jako początkowe (końcowe) wierzchołki działania algorytmu można przyjąć na przykład wszystkie wierzchołki wielokąta diagramu, w którym zawarta jest pozycja początkowa (końcowa) ruchu żabki; uprzednio należy ustawić wagi przypisane tym wierzchołkom, wynikające z konieczności przejścia do nich z pozycji początkowej (końcowej). Diagram Voronoia ma co najwyżej $2n-5$ wierzchołków i $3n-6$ krawędzi, zatem cały algorytm (wliczając czas konstrukcji diagramu oraz wykonania algorytmu Dijkstry) działa w czasie $O(n \cdot \log n)$.

Niniejszy skrótowy opis stanowi tylko zarys algorytmu; po dowody przedstawionych w nim faktów i algorytm konstrukcji diagramu Voronoi odsyłamy do wymienionej powyżej literatury.

Zawody II stopnia

opracowania zadań

Jakub Radoszewski **Marek Cygan, Łukasz Kowalik** **Paweł Parys**
Treść zadania **Opracowanie** **Program**

OI, Etap II, dzień próbny, 14.02.2006

Szkoły

W Bajtocji znajduje się n szkół, z których każda ma przypisany numer m, $1 \leqslant m \leqslant n$*. Poprzedni król Bajtocji w ogóle nie zwracał uwagi na porządek w numeracji szkół, pozwalając każdej nowowybudowanej szkole wybrać dla siebie dowolny numer z przedziału od 1 do n. Przez to w kraju mogło powstać wiele szkół o tym samym numerze, a niektóre numery z przedziału od 1 do n mogły nie zostać wykorzystane.*

Nowy król Bajtocji postanowił przywrócić porządek i dokonać takiego przenumerowania szkół, żeby teraz każdy numer był wykorzystany dokładnie raz. Niestety nie jest to proste zadanie, gdyż większość szkół niechętnie poddałaby się zmianie numeru.

Król wysłał swoich informatorów do poszczególnych szkół, aby dowiedzieli się, na jak dużą zmianę numeru poszczególne szkoły mogą się zgodzić. Co więcej, każda szkoła określiła koszt c dokonania zmiany swojego numeru o 1. Stąd całkowity koszt dokonania zmiany numeru danej szkoły wynosi $c \cdot |m - m'|$*, gdzie m oznacza stary, a* m' *nowy numer danej szkoły. Oczywiście liczba* m' *musi się mieścić w podanym wcześniej przedziale tolerancji dla danej szkoły.*

Król — dostawszy wyżej opisane informacje — chciałby się dowiedzieć, czy jest możliwy do przywrócenia porządek w numeracji szkół (zakładając przestrzeganie przedziałów tolerancji wszystkich szkół), a jeżeli tak, to jaki jest minimalny koszt takiego przenumerowania. Dlatego poprosił Ciebie — nadwornego informatyka — o podanie mu tej informacji na podstawie danych o szkołach, które Ci dostarczył.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia aktualne numery szkół w Bajtocji, ich przedziały tolerancji dotyczące zmiany numeru oraz koszty zmiany numerów poszczególnych szkół o 1,*
- *sprawdzi, czy jest możliwe przenumerowanie szkół spełniające wszystkie wymienione wcześniej warunki i jeżeli tak — obliczy minimalny koszt takiej zmiany,*
- *wypisze wynik na standardowe wyjście.*

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę całkowitą n $(1 \leqslant n \leqslant 200)$*, oznaczającą liczbę szkół w Bajtocji. W kolejnych n wierszach znajdują się opisy poszczególnych szkół. Wiersz o numerze* $i+1$ $(1 \leqslant i \leqslant n)$ *zawiera cztery liczby całkowite* m_i, a_i, b_i *oraz* k_i $(1 \leqslant a_i \leqslant m_i \leqslant b_i \leqslant n,\ 1 \leqslant k_i \leqslant 1\ 000)$*, oddzielone pojedynczymi odstępami. Liczby te oznaczające odpowiednio: aktualny numer i-tej szkoły, początek i koniec przedziału tolerancji i-tej szkoły odnośnie zmiany numeru (jest to przedział domknięty, czyli nowy numer i-tej szkoły* m_i' *musi spełniać nierówność* $a_i \leqslant m_i' \leqslant b_i$*) oraz koszt zamiany numeru i-tej szkoły o 1.*

Wyjście

Jeżeli przenumerowanie szkół spełniające podane wyżej warunki jest możliwe, program powinien wypisać jedną liczbę całkowitą k oznaczającą najmniejszy możliwy koszt zmiany numeracji. W przeciwnym przypadku na wyjściu powinno zostać wypisane słowo `NIE`.

Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
9
```

Rozwiązanie

Rozwiązanie wzorcowe

Skojarzenia

Sformułujmy zadanie w terminach teorii grafów. *Grafem dwudzielnym* nazywamy graf, w którym wierzchołki można podzielić na dwie rozłączne grupy, takie że krawędzie przebiegają jedynie pomiędzy wierzchołkami z różnych grup. Każdemu egzemplarzowi naszego problemu odpowiada następujący graf dwudzielny $G = (V, E)$:

- $V = S \cup N$, gdzie $S = \{s_1, \ldots, s_n\}$ oraz $N = \{1, \ldots, n\}$ oznaczają odpowiednio szkoły oraz numery, które zamierzamy przypisać szkołom.
- Krawędź między szkołą s_i oraz numerem k istnieje, jeżeli dana szkoła może zaakceptować ten numer, czyli zachodzi $a_i \leqslant k \leqslant b_i$. Wagą każdej krawędzi jest koszt odpowiedniego przypisania.

Skojarzeniem w grafie G nazywamy podzbiór $M \subseteq E$ taki, że żadne dwie krawędzie z M nie mają wspólnego wierzchołka. Dla każdego podzbioru $F \subseteq E$ określimy *wagę* F, jako sumę wag krawędzi należących do F — będziemy ją oznaczać $waga(F)$. Zadanie sprowadza się zatem do znalezienia minimalnej wagi skojarzenia o rozmiarze n.

Niech M będzie skojarzeniem w grafie G. Definiujemy graf skierowany $D(G, M)$ następująco:

- Wierzchołki i krawędzie w grafie $D(G, M)$ są takie same jak w grafie G, ale wszystkie krawędzie grafu $D(G, M)$ są skierowane.

- Jeżeli w grafie G istnieje krawędź o wadze w łącząca wierzchołki $v_1 \in S$ i $v_2 \in N$ oraz krawędź ta należy do skojarzenia M, wówczas w grafie $D(G,M)$ jest ona zorientowana z wierzchołka v_2 do v_1 i ma wagę $-w$ (ujemną!).
- Jeżeli w grafie G istnieje krawędź o wadze w łącząca wierzchołki $v_1 \in S$ i $v_2 \in N$ oraz krawędź ta nie należy do skojarzenia M, wówczas w grafie $D(G,M)$ jest ona zorientowana z wierzchołka v_1 do v_2 i ma wagę w.

Ścieżką rozszerzającą w grafie $D(G,M)$ nazywamy ścieżkę rozpoczynającą się w wierzchołku $u \in S$ i kończącą się w wierzchołku $v \in N$ taką, że żaden z wierzchołków u, v nie jest pokrywany przez skojarzenie M.

Opis algorytmu

Przedstawimy teraz algorytm służący do znajdowania skojarzenia maksymalnego rozmiaru i posiadającego minimalną wagę.

Zdefiniujemy operator różnicy symetrycznej dla zbiorów:

$$A \oplus B = (A \cup B) - (A \cap B) = (A - B) \cup (B - A).$$

Niech M_0 oznacza puste skojarzenie. Przez długość ścieżki będziemy rozumieć sumę wag jej krawędzi.

Algorytm wykonuje co najwyżej n kroków. W k-tym kroku obliczamy skojarzenie M_k (w skład którego wchodzi k krawędzi) w następujący sposób. Najpierw znajdujemy najkrótszą ścieżkę rozszerzającą P_k w grafie $D(G,M_{k-1})$ (powrócimy jeszcze do tego, jak to zrobić). Wyznaczamy skojarzenie $M_k = M_{k-1} \oplus P_k$, czyli zaliczamy do M_k krawędzie, które należą do M_{k-1} lub do P_k, ale nie do obu równocześnie. Jeżeli w którymś momencie okaże się, że ścieżka rozszerzająca nie istnieje, oznacza to, że szukane skojarzenie nie istnieje.

Należy jeszcze zwrócić uwagę na jedną rzecz: w grafie z wagami, w którym niektóre wagi są ujemne, mogą istnieć dowolnie krótkie ścieżki. Zdarza się to, gdy w grafie tym istnieje cykl o ujemnej wadze. Wtedy pojęcie „najkrótsza ścieżka” nie jest dobrze zdefiniowane. Jednak w naszym zadaniu ścieżka rozszerzająca nie może przechodzić 2 razy przez ten sam wierzchołek, ponieważ wierzchołek ten musiałby być incydentny z dwiema krawędziami skojarzenia, co jest niemożliwe.

Dlaczego to działa?

Teraz musimy pokazać, że powyższy algorytm zawsze znajduje skojarzenie o minimalnej wadze.

Przypuśćmy, że w grafie G istnieje skojarzenie rozmiaru $s \leqslant n$. Skorzystamy z indukcji matematycznej i pokażemy, że dla każdego $k = 1,\ldots,s$ algorytm znajdzie skojarzenie M_k zawierające k krawędzi oraz posiadające minimalną wagę spośród wszystkich skojarzeń rozmiaru k.

Rozważmy k-ty krok algorytmu. Z założenia indukcyjnego przyjmujemy, że skojarzenie M_{k-1} znalezione w poprzednim kroku algorytmu posiada $k-1$ krawędzi i ma ono minimalną wagę wśród skojarzeń o tej mocy.

Niech M'_k będzie skojarzeniem w G rozmiaru k o minimalnej wadze. Jak wygląda zbiór krawędzi $M_{k-1} \oplus M'_k$? Składa się on wyłącznie z cykli i ścieżek, gdyż stopień każdego wierzchołka jest w nim niewiększy niż 2. Dodatkowo, w każdej ze ścieżek liczby krawędzi ze zbioru M'_k i M_{k-1} mogą się różnić co najwyżej o 1 (ponieważ występują one na przemian). Ponieważ zbiór M'_k posiada o jedną krawędź więcej niż M_{k-1}, zatem musi istnieć co najmniej jedna ścieżka, w której liczba krawędzi ze zbioru M'_k jest o 1 większa niż liczba krawędzi z M_{k-1}. Oznaczmy taką ścieżkę przez P. Zauważmy, że $M'_k \oplus P$ oraz $M_{k-1} \oplus P$ są skojarzeniami, posiadającymi odpowiednio $k-1$ i k krawędzi.

Lemat 1 *Krawędzie z M'_k nienależące do P mają taką samą sumę wag, jak krawędzie z M_{k-1} nienależące do P. Inaczej mówiąc:*

$$waga(M'_k \setminus P) = waga(M_{k-1} \setminus P)$$

Dowód Załóżmy, że powyższa teza nie zachodzi. Wtedy musi zachodzić dokładnie jeden z warunków:

- $waga(M'_k \setminus P) > waga(M_{k-1} \setminus P)$, zatem

  $waga(M'_k) - waga(M'_k \cap P) > waga(M_{k-1} \setminus P))$, korzystając z rozłączności zbiorów

  $waga(M'_k) > waga((M_{k-1} \setminus P) \cup (M'_k \cap P))$ i dalej

  $waga(M'_k) > waga(M_{k-1} \oplus P)$, co oznacza sprzeczność, gdyż M'_k jest skojarzeniem liczności k o minimalnej wadze

- rozważając analogicznie drugi przypadek otrzymujemy

  $waga(M'_k \setminus P) < waga(M_{k-1} \setminus P)$, zatem

  $waga(M'_k \setminus P) < waga(M_{k-1}) - waga(M_{k-1} \cap P)$, co oznacza

  $waga((M'_k \setminus P) \cup (M_{k-1} \cap P)) < waga(M_{k-1})$ i dalej

  $waga(M'_k \oplus P) < waga(M_{k-1})$, co doprowadza do sprzeczności, gdyż M_{k-1} jest skojarzeniem liczności $k-1$ o minimalnej wadze

■

Teraz zauważmy, że P jest ścieżką rozszerzającą w grafie $D(G, M_{k-1})$, a z powyższego lematu wynika, że dla wag krawędzi z grafu $D(G, M_{k-1})$ zachodzi równość: $waga(M'_k) = waga(M_{k-1}) + waga(P)$ (jest to odpowiednio przekształcona postać równości z lematu). W k-tym kroku algorytmu znajdujemy pewną ścieżkę rozszerzającą — oznaczmy ją P_{alg}, i przyjmujemy $M_k = M_{k-1} \oplus P_{alg}$. Ponieważ znaleziona ścieżka jest najkrótszą ścieżką rozszerzającą, zatem $waga(P_{alg}) \leqslant waga(P)$. Otrzymujemy więc, że $waga(M_k) = waga(M_{k-1}) + waga(P_{alg}) \leqslant waga(M_{k-1}) + waga(P) = waga(M'_k)$. Zatem M_k ma minimalną wagę spośród wszystkich skojarzeń rozmiaru k.

Implementacja

W zadaniu wystarczy zatem zaimplementować algorytm znajdowania najkrótszych ścieżek rozszerzających w grafach postaci $D(G, M_k)$. Ponieważ w tych grafach mogą znajdować się krawędzie z ujemnymi wagami, nie możemy w tym celu posłużyć się algorytmem Dijkstry. Zamiast tego użyjemy algorytmu Bellmana-Forda[1], który pozwala znaleźć najkrótszą ścieżkę

---

[1] Zobacz np. w: [18]

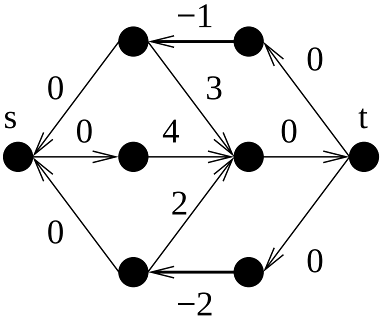
rozszerzającą w grafie z dowolnymi wagami, pod warunkiem, że nie występują w nim cykle o ujemnych wagach. Sprawdźmy, czy nasz graf spełnia ten warunek.

Przypuśćmy, że w grafie $D(G,M_k)$ występuje cykl C o ujemnej wadze. Wówczas dokładnie połowa krawędzi tego cyklu należy do skojarzenia M_k. Ponieważ po modyfikacjach wag w grafie $D(G,M_k)$ wszystkie krawędzie należące do skojarzenia mają wagi ujemne, a pozostałe krawędzie mają wagi dodatnie, zatem suma oryginalnych wag krawędzi należących do skojarzenia jest większa niż suma wag pozostałych krawędzi. Wtedy jednak $M_k \oplus C$ jest skojarzeniem rozmiaru k takim, że $waga(M_k \oplus C) < waga(M_k)$. Jednak to nie może zachodzić, ponieważ udowodniliśmy już, że M_k posiada minimalną wagę spośród wszystkich skojarzeń rozmiaru k.

Zatem możemy użyć algorytmu Bellmana-Forda. Jego złożoność czasowa wynosi $O(mn)$ dla grafu z m krawędziami oraz n wierzchołkami. Ponieważ liczba krawędzi może być co najwyżej rozmiaru $O(n^2)$, złożoność algorytmu Bellmana-Forda to $O(n^3)$, zatem złożoność czasowa całego algorytmu wynosi $O(n^4)$.

Aby znaleźć ścieżkę powiększającą w grafie $D(G,M)$ musimy dodać do grafu dwa dodatkowe wierzchołki. Jeden z nich będziemy nazywali źródłem, a drugi ujściem (terminologia pochodzi z zagadnień szukania maksymalnego przepływu). Źródło będziemy oznaczać jako wierzchołek s, a ujście jako wierzchołek t. Do grafu $D(G,M)$ dodajmy następujące krawędzie o wadze 0:

- pomiędzy wierzchołkiem s a każdym wierzchołkiem $v \in S$, jeśli wierzchołek v nie jest incydentny z żadną krawędzią należącą do skojarzenia, to krawędź jest skierowana od wierzchołka s do wierzchołka v, w przeciwnym wypadku krawędź jest skierowana w kierunku wierzchołka s,
- pomiędzy wierzchołkiem t, a każdym wierzchołkiem $v \in N$, jeśli wierzchołek v nie jest incydentny z żadną krawędzią należącą do skojarzenia, to krawędź jest skierowana od wierzchołka v do wierzchołka t, w przeciwnym wypadku krawędź jest skierowana w kierunku wierzchołka v.



Rys. 1: Przykładowy graf ze skojarzeniem o liczności 2

Powstały graf będziemy oznaczać $D'(G,M)$. Nietrudno zauważyć, że ścieżka powiększająca w grafie $D(G,M)$ odpowiada ścieżce w grafie $D'(G,M)$ prowadzącej od wierzchołka s do wierzchołka t. Po znalezieniu każdej nowej ścieżki powiększającej, odwracamy znajdujące się na niej krawędzie i zmieniamy ich wagi na przeciwne.

Inne poprawne rozwiązania

Wiadomo, że maksymalne skojarzenie w grafie może zostać znalezione za pomocą algorytmów znajdujących maksymalny przepływ w sieci. Również problem znajdowania skojarzenia o minimalnej wadze spośród wszystkich skojarzeń danego rozmiaru może zostać sprowadzony do znajdowania przepływu o danym rozmiarze i minimalnym koszcie. Aby rozwiązać zadanie, można więc zmodyfikować odpowiednio algorytm Forda-Fulkersona. W ten sposób otrzymamy algorytm o takiej samej złożoności czasowej ($O(n^4)$) jak algorytm wzorcowy.

Szybsze rozwiązania

Istnieją również szybsze algorytmy znajdowania skojarzenia o minimalnej wadze (jak również przepływu o minimalnym koszcie). Możemy zmodyfikować nasz algorytm, aby do znajdowania najkrótszych ścieżek powiększających używać algorytmu Dijkstry, zamiast algorytmu Bellmana-Forda. Dzięki temu otrzymamy algorytm o złożoności czasowej $O(n(m+n^2)) = O(n^3)$.

Wiadomo, że algorytm Dijkstry możemy stosować tylko i wyłącznie dla grafów o nieujemnych wagach krawędzi (wymyślenie przykładu grafu z ujemnymi wagami krawędzi, dla którego algorytm Dijkstry znajdzie niepoprawne rozwiązanie pozostawiamy Czytelnikowi jako ćwiczenie), a w grafie $D'(G,M)$ krawędzie należące do skojarzenia mogą mieć ujemne wagi. Aby pokonać tę trudność wprowadźmy pojęcie potencjału. Potencjał wierzchołka v będziemy oznaczać jako $\pi(v)$. Dla każdej krawędzi w grafie $D'(G,M)$ zdefiniujmy nową wagę $waga'(v_1,v_2) = waga(v_1,v_2) + \pi(v_1) - \pi(v_2)$. Zauważmy, że dla każdej ścieżki P z wierzchołka s do wierzchołka t zachodzi:

$$waga'(P) = waga(P) + \pi(s) - \pi(t),$$

czyli dla zadanego potencjału wagi te różnią się jedynie o stały składnik. Wynika z tego, że najkrótsza ścieżka powiększająca dla nowych wag jest również najkrótszą ścieżką powiększającą dla poprzednich kosztów.

Możemy sobie zadać pytanie, co daje nam wprowadzenie potencjału π. Otóż możemy tak dobrać jego wartości, aby nowe wagi były zawsze nieujemne; w tym celu dla każdej krawędzi (v_1,v_2) musi zachodzić:

$$waga(v_1,v_2) + \pi(v_1) - \pi(v_2) \geqslant 0,$$

czyli

$$\pi(v_2) \leqslant \pi(v_1) + waga(v_1,v_2).$$

Zaobserwujmy, że nierówność tę spełnia (jako potencjał) funkcja odległości od wierzchołka s. Możemy ten fakt wykorzystać w następujący sposób: początkowo potencjał każdego wierzchołka przyjmiemy równy 0, podczas wyszukiwania nowej ścieżki powiększającej będziemy obliczać odległość każdego wierzchołka od źródła s i podczas wyszukiwania następnej ścieżki powiększającej użyjemy tych odległości jako nowej funkcji potencjału. Należy sobie jeszcze zadać pytanie, czy podczas odwracania krawędzi funkcja potencjału nie straci swoich właściwości. Okazuje się, że każda krawędź (v_1, v_2) leżąca na najkrótszej ścieżce powiększającej spełnia następującą równość:

$$\pi(v_2) = \pi(v_1) + waga(v_1, v_2)$$

zatem zarówno przed jak i po odwróceniu tej krawędzi jej waga będzie wynosić 0. Zwrot pozostałych krawędzi nie ulega zmianom. Ostatecznie nowe wagi wszystkich krawędzi będą nieujemne, co pozwala nam na użycie algorytmu Dijkstry do wyszukiwania najkrótszych ścieżek powiększających.

Testy

Zadanie testowane było na 15 zestawach danych testowych, które zawierały łącznie 20 testów. W poniższej tabeli n oznacza liczbę szkół w teście, a k — przedział w jakim mieszczą się wszystkie stopnie wierzchołków reprezentujących szkoły w grafie (obrazuje to gęstość grafu).

Nazwa	n	k	Opis
szk1a.in	3	$1 \div 2$	bardzo mały test z odpowiedzią NIE
szk1b.in	10	$2 \div 10$	bardzo mały test
szk3.in	5	$1 \div 4$	test losowy
szk4.in	16	$1 \div 15$	test losowy
szk5a.in	12	$1 \div 12$	test losowy
szk5b.in	200	$10 \div 10$	tylko 10 dozwolonych numerów szkół
szk7.in	15	$1 \div 3$	test losowy
szk8.in	60	$1 \div 3$	test losowy
szk9.in	120	$1 \div 2$	test losowy
szk10a.in	30	$1 \div 30$	test losowy
szk10b.in	70	$10 \div 20$	test losowy z odpowiedzią NIE
szk12.in	100	$10 \div 20$	test losowy
szk13.in	200	$2 \div 200$	test losowy
szk14a.in	200	$200 \div 200$	graf pełny dwudzielny
szk14b.in	200	$1 \div 10$	test losowy z odpowiedzią NIE
szk16.in	200	$37 \div 200$	wszystkie szkoły mają ten sam numer początkowy
szk17.in	200	$1 \div 10$	test losowy
szk18a.in	200	$100 \div 100$	test losowy
szk18b.in	200	$199 \div 199$	jeden numer nie jest dopuszczalny dla żadnej ze szkół
szk20.in	200	$50 \div 100$	graf o specyficznej strukturze

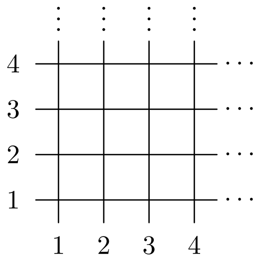
Zbigniew Czech
Treść zadania, Opracowanie

Łukasz Kowalik
Program

OI, Etap II, dzień próbny, 14.02.2006

Magazyn

Ulice w Bajtomieście tworzą prostopadłą siatkę — prowadzą ze wschodu na zachód lub z północy na południe. Ulice północ-południe są ponumerowane od 1 do 500 000 000 w kolejności z zachodu na wschód. Podobnie ulice wschód-zachód są ponumerowane od 1 do 500 000 000 w kolejności z południa na północ. Każda ulica północ-południe przecina każdą ulicę wschód-zachód i odwrotnie, każda ulica wschód-zachód przecina każdą północ-południe. Odległość między dwiema sąsiednimi ulicami północ-południe, a także sąsiednimi ulicami wschód-zachód jest równa jednemu kilometrowi.


4
3
2
1
1
2
3
4

W mieście znajduje się k sklepów, a każdy sklep jest położony przy skrzyżowaniu ulic. Kupiec Bajtazar dostarcza towary do każdego z k sklepów, przy czym część sklepów odwiedza kilka razy dziennie. Bajtazar postanowił wybudować magazyn, z którego dostarczałby towary do sklepów. Magazyn powinien być położony przy skrzyżowaniu ulic. Ciężarówka dostarczająca towary w trakcie jednego kursu może odwiedzić tylko jeden sklep — wyjeżdża z magazynu, dostarcza towar do sklepu i wraca do magazynu. Ciężarówka zawsze jedzie najkrótszą trasą z magazynu do sklepu i z powrotem. Odległość między punktami (x_i,y_i) i (x_j,y_j) jest równa

$$max\{|x_i-x_j|,|y_i-y_j|\}.$$

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis rozmieszczenia sklepów oraz ile razy dziennie towary są dostarczane do poszczególnych sklepów,*
- *wyznaczy takie położenie magazynu, żeby łączna odległość pokonywana przez ciężarówkę każdego dnia była jak najmniejsza,*
- *wypisze wynik na standardowe wyjście.*

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą n ($1 \leqslant n \leqslant 100\,000$), oznaczającą liczbę sklepów w Bajtomieście.

Kolejne n wierszy wejścia zawiera opisy sklepów. Wiersz $i+1$-wszy zawiera trzy liczby całkowite x_i, y_i i t_i ($1 \leqslant x_i,y_i \leqslant 500\,000\,000$, $1 \leqslant t_i \leqslant 1\,000\,000$), oddzielone pojedynczymi odstępami. Ten opis oznacza, że i-ty sklep jest położony na skrzyżowaniu x_i-tej ulicy północ-południe i y_i-tej ulicy wschód-zachód i ciężarówka codziennie dojeżdża do tego sklepu t_i razy.

Wyjście

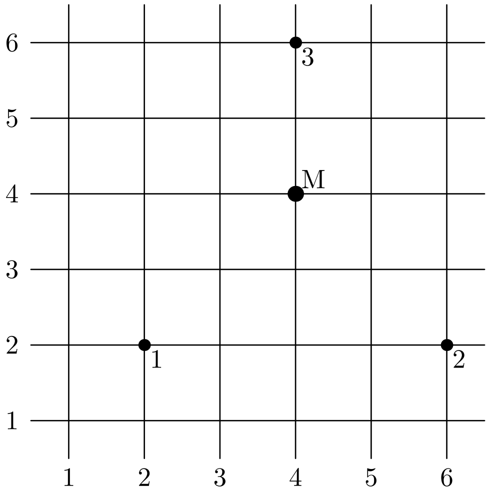
Pierwszy i jedyny wiersz wyjścia powinien zawierać dwie liczby całkowite x_m oraz y_m, oddzielone pojedynczym odstępem i opisujące położenie magazynu jako skrzyżowanie x_m-tej ulicy północ-południe i y_m-tej ulicy wschód-zachód. Jeżeli istnieje wiele poprawnych wyników, Twój program powinien wypisać dowolny z nich.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*

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

Poniższy rysunek przedstawia sytuację z przykładowego wejścia. Ponumerowane punkty oznaczają odpowiednie sklepy. Punkt M oznacza optymalne położenie magazynu.


6
5
4
3
2
1
1
2
3
4
5
6
3
M
1
2

Rozwiązanie

Analiza zadania

Niech $S = \{1,2,\ldots,n\}$ będzie zbiorem sklepów. Niech sklep $i \in S$ ma współrzędne (x_i, y_i) i niech t_i będzie codzienną liczbą pobytów Bajtazara w tym sklepie. Naszym zadaniem jest znalezienie punktu $M = (x_M, y_M)$, dla którego odległość D_m pokonywana przez ciężarówkę każdego dnia jest jak najmniejsza:

$$D_m = \sum_{i \in S} t_i \cdot d_m(M,i),$$

gdzie funkcja $d_m(i,j) = \max\{|x_i - x_j|, |y_i - y_j|\}$ określa odległość między punktami i oraz j.

W ogólnym przypadku, do mierzenia odległości służą funkcje o wartościach nieujemnych zwane metrykami.

Definicja 1 *Metrykę* definiujemy jako dowolną funkcję nieujemną $d(i,j)$, opisującą odległość między punktami i oraz j. Metryka musi mieć następujące własności:

- $d(i,i) = 0$ oraz $d(i,j) > 0$, jeżeli $i \neq j$,
- $d(i,j) = d(j,i)$,
- $d(i,j) \leqslant d(i,k) + d(k,j)$, dla dowolnego punktu k.

Metryka z treści zadania jest *metryką maksimum*. Inną znaną metryką jest *metryka miejska*, zwana też *metryką taksówkową* lub *metryką Manhattan*, zdefiniowana wzorem $d_t(i,j) = |x_i - x_j| + |y_i - y_j|$. Zauważmy, że nasze zadanie byłoby prostsze, gdybyśmy zamiast metryki maksimum mogli użyć metryki miejskiej. Mamy bowiem:

$$D_t = \sum_{i \in S} t_i \cdot d_t(M,i) = \sum_{i \in S} t_i \cdot (|x_M - x_i| + |y_M - y_i|) = \sum_{i \in S} t_i \cdot |x_M - x_i| + \sum_{i \in S} t_i \cdot |y_M - y_i|.$$

Tak więc, aby znaleźć optymalne położenie magazynu, możemy osobno znaleźć współrzędną x_M, dla której suma $\sum_{i \in S} t_i \cdot |x_M - x_i|$ jest minimalna, i osobno współrzędną y_M, dla której minimalna jest suma$\sum_{i \in S} t_i \cdot |y_M - y_i|$. Tym samym zadanie redukuje się do przypadku jednowymiarowego. W dalszym ciągu omówimy metodę rozwiązania przypadku jednowymiarowego dla metryki miejskiej, a następnie pokażemy, jak zastosować ją dla metryki maksimum.

Przypadek jednowymiarowy

Niech $T_L(x)$ (odpowiednio $T_R(x)$) będzie łączną liczbą pobytów Bajtazara we wszystkich sklepach $i \in S$, dla których zachodzi $x_i \leqslant x$ (odpowiednio $x_i > x$) dla pewnej odciętej x, tzn.:

$$T_L(x) = \sum_{i \in S:\, x_i \leqslant x} t_i \quad \text{oraz} \quad T_R(x) = \sum_{i \in S:\, x_i > x} t_i.$$

Rozważmy funkcję kosztu $c(x) = \sum_{i \in S} t_i \cdot |x - x_i|$. Łatwo zauważyć, że dla dowolnej odciętej x mamy:

$$c(x+1) = c(x) + T_L(x) - T_R(x).$$

Stąd widać, że wartość funkcji c maleje przy przejściu od x do $x+1$, kiedy $T_L(x) < T_R(x)$, i rośnie, kiedy $T_L(x) > T_R(x)$. Ponieważ funkcja T_L jest niemalejąca, a funkcja T_R — nierosnąca, więc znajdując punkt x, w którym występuje największa „równowaga" pomiędzy wartościami T_L i T_R, znajdziemy miejsce, w którym koszt c jest minimalny. Dokładniej, jeśli znajdziemy odciętą x_i, dla której $T_L(x_i) = T_R(x_i)$, to funkcja c przyjmuje minimalną wartość we wszystkich punktach przedziału $[x_i, x_j]$, gdzie j jest kolejnym po i sklepem w posortowanym ciągu. Jeśli T_L i T_R nie są sobie równe w żadnym punkcie, a x_i jest minimalną wartością, dla której $T_L(x_i) > T_R(x_i)$, to x_i jest jedyną odciętą, dla której koszt c jest minimalny.

Powyższe spostrzeżenia pozwalają nam rozwiązać przypadek jednowymiarowy w osi x (i podobnie w osi y) w czasie $O(n \log n)$, stosując szybki algorytm sortowania, np. sortowanie przez scalanie lub kopcowanie. Alternatywnie można zastosować algorytmy wyznaczające medianę zbioru liczb, np. algorytm Hoare'a — można w ten sposób uzyskać algorytm o złożoności $O(n)$.

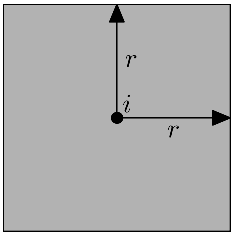
Przypadek dwuwymiarowy

Skonstruujmy rozwiązanie dla przypadku dwuwymiarowego (nadal dla metryki miejskiej) z rozwiązań obliczonych oddzielnie dla obu osi. Jak zauważyliśmy, rozwiązaniem przypadku jednowymiarowego jest albo jeden punkt, albo cały przedział punktów. Zbiór optymalnych rozwiązań przypadku dwuwymiarowego to iloczyn kartezjański rozwiązań przypadków jednowymiarowych. Ostatecznie w wyniku otrzymujemy więc albo pojedynczy punkt, albo odcinek, albo prostokąt złożony z rozwiązań optymalnych.

Zmiana metryki

Definicja 2 Dla danej metryki d, punktu i oraz nieujemnej liczby r, definiujemy *kulę domkniętą* $B_r(i)$ jako zbiór punktów, znajdujących się w odległości co najwyżej r od punktu i. Punkt i nazywamy *środkiem*, a r *promieniem* kuli $B_r(i)$.

Kula domknięta $B_r(i)$ w metryce maksimum to kwadrat o środku w punkcie i oraz długości boku $2r$ (rys. 1). Kula domknięta $B_r(i)$ w metryce miejskiej to także kwadrat o środku w punkcie i, tyle że obrócony o 45° i o długości przekątnej $2r$ (rys. 2). Boki obróconego kwadratu mają długość $\frac{2r}{\sqrt{2}}$.


r
i
r

Można zauważyć, że kula w metryce maksimum, obrócona o 45° i przeskalowana przez współczynnik $\frac{1}{\sqrt{2}}$ daje kulę o tym samym środku i promieniu, ale w metryce miejskiej. To spostrzeżenie jest pożyteczne wobec prawdziwości następującego twierdzenia:

Twierdzenie 1 *Niech d_1 i d_2 będą metrykami. Jeżeli dla każdego punktu i oraz promienia r, kula o promieniu r oraz środku w punkcie i w metryce d_1 jest taka sama, jak kula o promieniu r oraz środku w punkcie i w metryce d_2, to te dwie metryki są takie same. Oznacza to, że $d_1(i,j) = d_2(i,j)$ dla każdej pary punktów i, j.*

Dowód Niech $B_r^1(i)$ (odpowiednio $B_r^2(i)$) oznacza kulę o promieniu r oraz środku w punkcie i w metryce d_1 (odpowiednio d_2). Niech i, j będą dowolnymi punktami. Przyjmijmy $r = d_1(i,j)$. Wówczas $j \in B_r^2(i)$, a stąd $d_2(i,j) \leqslant r$. Załóżmy, że $d_2(i,j) < r$. Wówczas zachodzi $j \in B^1_{d_2(i,j)}(i)$. To implikuje, że $r = d_1(i,j) \leqslant d_2(i,j) < r$, co daje sprzeczność. Stąd ostatecznie $d_2(i,j) = r$. ■

Tak więc zamiast używać metryki maksimum dla danego zbioru punktów S możemy dokonać obrotu tych punktów o kąt 45°, przeskalować przez współczynnik $\frac{1}{\sqrt{2}}$ i używać metryki miejskiej. Z powyższego twierdzenia wynika poprawność tego postępowania; aby je lepiej zrozumieć, przeprowadzimy jeszcze dowód niekorzystający z ogólnych własności kul w różnych metrykach. Dowolny punkt o współrzędnych (x,y) obrócony o kąt 45° wokół punktu $(0,0)$ ma współrzędne $(\frac{x-y}{\sqrt{2}}, \frac{x+y}{\sqrt{2}})$. Jeżeli dodatkowo przeskalujemy współrzędne przez współczynnik $\frac{1}{\sqrt{2}}$, to otrzymamy punkt $(\frac{x-y}{2}, \frac{x+y}{2})$. Niech $A = (x_A, y_A)$ i $B = (x_B, y_B)$ będą dowolnymi punktami. Oznaczmy $a = x_A - x_B$ oraz $b = y_A - y_B$. Po obrocie i przeskalowaniu otrzymujemy punkty $A' = (\frac{x_A - y_A}{2}, \frac{x_A + y_A}{2})$ i $B' = (\frac{x_B - y_B}{2}, \frac{x_B + y_B}{2})$. Wówczas odległość w metryce miejskiej jest równa

$$d_t(A',B') = \left|\frac{x_A - y_A - x_B + y_B}{2}\right| + \left|\frac{x_A + y_A - x_B - y_B}{2}\right| = \frac{|a-b| + |a+b|}{2}.$$

Bez straty ogólności możemy przyjąć, że $a \geqslant 0$ (w przeciwnym razie punkty możemy zamienić). Załóżmy, że $|a| \geqslant |b|$. Wówczas $d_m(A,B) = a$ i $d_t(A',B') = \frac{a-b+a+b}{2} = a$. Stąd $d_m(A,B) = d_t(A',B')$. Załóżmy teraz, że $|a| < |b|$. Wówczas $d_m(A,B) = |b|$. Jeżeli $b \geqslant 0$, to $d_t(A',B') = \frac{-a+b+a+b}{2} = b = |b|$. Jeżeli zaś $b < 0$, to $d_t(A',B') = \frac{a-b-a-b}{2} = -b = |b|$. Ponownie mamy $d_m(A,B) = d_t(A',B')$.

Rozwiązania zadania

Rozwiązanie wzorcowe

Opierając się na powyższych rozważaniach można zaproponować następujące rozwiązanie zadania. Najpierw obracamy i przeskalowujemy współrzędne wszystkich sklepów. Zamiast używać współczynnika przeskalowania $\frac{1}{\sqrt{2}}$ używamy współczynnika $\frac{2}{\sqrt{2}}$. Wówczas punkt o współrzędnych (x,y) zostaje przekształcony do punktu o współrzędnych $(x-y, x+y)$. Otrzymane współrzędne są liczbami całkowitymi, co upraszcza obliczenia. Dwa punkty położone w odległości d od siebie w metryce maksimum, po przekształceniu znajdują się w odległości $2d$ w metryce miejskiej. Mimo że wszystkie odległości zostały podwojone, optymalne położenie magazynu w metryce miejskiej odpowiada optymalnemu położeniu w metryce maksimum.

Po wykonaniu przekształcenia otrzymujemy jak gdyby nowe miasto, w którym odległość między dowolnymi dwoma równoległymi sąsiednimi ulicami jest równa dwa kilometry. Nazwijmy to miasto Nowym Bajtomiastem. W Nowym Bajtomieście znajdujemy najlepsze położenie magazynu w metryce miejskiej.

Po uzyskaniu optymalnego położenia magazynu (x,y) w Nowym Bajtomieście, przekształcamy je do punktu o współrzędnych $(\frac{x+y}{2},\frac{y-x}{2})$ w Bajtomieście. Jeżeli współrzędne x i y są obie parzyste lub obie nieparzyste, to otrzymujemy punkt położony na skrzyżowaniu w Bajtomieście będący poszukiwanym rozwiązaniem. W przeciwnym razie otrzymujemy punkt w środku kwadratu utworzonego przez 4 ulice i 4 skrzyżowania. W takim przypadku jako rozwiązanie zadania wybieramy najlepsze z tych czterech skrzyżowań. Dokonując wyboru nie trzeba liczyć kosztu dostarczenia towaru dla magazynu położonego w każdym z tych czterech skrzyżowań. Wystarczy obliczyć wzrost kosztu dostawy zakładając przemieszczenie magazynu z punktu $(\frac{x+y}{2},\frac{y-x}{2})$ do każdego z tych skrzyżowań.

Uzasadnimy, że takie postępowanie jest poprawne. Oczywiście nasze cztery skrzyżowania w Bajtomieście odpowiadają czterem skrzyżowaniom, sąsiadującym z punktem (x,y) w Nowym Bajtomieście, tzn. skrzyżowaniom $(x+1,y),(x,y+1),(x-1,y),(x,y-1)$. Jeżeli zbiór optymalnych położeń magazynu w Nowym Bajtomieście nie jest pojedynczym punktem, to jedno z tych sąsiednich skrzyżowań jest także optymalnym położeniem magazynu. Tym samym odpowiadające mu skrzyżowanie w Bajtomieście jest optymalne. Jeśli optymalne położenie magazynu w Nowym Bajtomieście, $s_0 = (x,y)$, jest pojedynczym punktem, to oczywiście, aby uzyskać równie dobry koszt rozwiązania w Bajtomieście, musielibyśmy postawić magazyn pomiędzy sąsiednimi ulicami — tego nam jednak nie wolno. Pokażemy, że wybrana lokalizacja odpowiada drugiemu co do jakości rozwiązaniu z Nowego Bajtomiasta, a więc najlepszemu możliwemu w Bajtomieście. Wystarczy pokazać, że ten „vicelider” jest wśród skrzyżowań $(x+1,y),(x,y+1),(x-1,y),(x,y-1)$.

Załóżmy, że s jest poszukiwanym położeniem vicelidera (czyli jest pewnym skrzyżowaniem w Nowym Bajtomieście). Oczywiście najkrótsza droga z s_0 do s przechodzi przez jedno z czterech sąsiadujących z s_0 skrzyżowań. Przemieszczenie magazynu od s wzdłuż tej drogi w kierunku s_0 powoduje, że łączny koszt dostarczenia towarów pozostaje ten sam lub maleje (wynika to stąd, że w ten sposób w każdym wymiarze nie oddalamy się od położenia s_0). A zatem przyjmując jako nowe położenie magazynu odpowiedniego sąsiada s_0 zamiast s uzyskujemy rozwiązanie niegorsze niż w przypadku s.

Implementacja rozwiązania wzorcowego znajduje się na dysku dołączonym do książeczki w plikach `mag.cpp`, `mag1.cpp` oraz `mag2.pas`.

Inne rozwiązania

Jednym z prostych sposobów rozwiązania zadania jest obliczenie kosztu dostarczenia towaru dla wszystkich możliwych położeń magazynu i wybranie najlepszego położenia. Program działający według tego sposobu zawarty jest w pliku `mags0.cpp`.
Inna możliwość wynika z obserwacji, iż funkcja kosztu osiąga w punkcie optymalnym swoje minima zarówno w osi x, jak i y. Wybieramy więc dowolnie punkt siatki s jako położenie magazynu i obliczamy dla niego wartość funkcji kosztu. Następnie obliczamy koszty

dla wszystkich skrzyżowań sąsiednich z punktem s, włączając skrzyżowania sąsiadujące z punktem s po przekątnych. Jeśli istnieje sąsiednie skrzyżowanie o mniejszej w stosunku do s wartości kosztu, to przemieszczamy magazyn do tego skrzyżowania (nowe s) i kontynuujemy postępowanie. Jeśli skrzyżowanie sąsiednie o mniejszej wartości kosztu nie istnieje, to znaczy, że osiągnęliśmy optimum. Program skonstruowany zgodnie z takim postępowaniem zawarty jest w pliku `mags1.cpp`.

Rozwiązania błędne

Jak wskazano powyżej, jednym z możliwych rozwiązań jest poszukiwanie optimum przez badanie sąsiadów bieżącego punktu s i przemieszczanie magazynu do punktu sąsiedniego o najmniejszym koszcie. W pliku `magb0.cpp` zawarty jest program, w którym uwzględnia się jedynie sąsiadów leżących wzdłuż osi x i y, tzn. pomija się sąsiadów leżących po przekątnych. Prowadzi to do błędnych wyników dla niektórych danych testowych.

Źródłem złych rozwiązań mogą być także błędy arytmetyki zmiennopozycyjnej, jeśli stosuje się typy `float` lub `real` (plik `magb1.cpp`).

Testy

Zadanie testowane było na zestawie 15 danych testowych.

Nazwa	n	Opis
mag1.in	3	prosty test poprawnościowy
mag2.in	5	prosty test sprawdzający poprawność
mag3.in	10	współrzędne sklepów to (x,x) dla $x = 1, \ldots, 10$
mag4.in	100	wszystkie sklepy położone w dwóch rogach kwadratu
mag5.in	100	test losowy, sklepy położone na brzegach
mag6.in	100	test losowy
mag7.in	100	test losowy, sklepy położone na brzegach
mag8.in	200	test losowy
mag9.in	30 000	test losowy, sklepy położone na brzegach
mag10.in	70 000	test losowy, sklepy położone na brzegach
mag11.in	70 000	test losowy
mag12.in	90 000	test losowy
mag13.in	100 000	test losowy
mag14.in	100 000	test losowy
mag15.in	100 000	test losowy

Karol Cwalina
Treść zadania, Opracowanie

Szymon Wąsik
Program

OI, Etap II, dzień pierwszy, 15.02.2006

Metro

W pewnym mieście od długiego czasu zmagano się z budową metra. Przy tym źle gospodarowano środkami, nie doszacowano kosztów budowy i zapomniano przewidzieć pieniądze na zakup pociągów. W rezultacie zbudowano wiele stacji, ale wydrążono tylko część zaplanowanych tuneli — ledwie wystarczających do tego, żeby pomiędzy każdymi dwiema stacjami istniała możliwość przejazdu. Liczba tuneli jest o 1 mniejsza od liczby zbudowanych stacji, ponadto wszystkie tunele są dwukierunkowe. Za pozostałe środki udało się kupić zaledwie kilka pociągów.

Chcąc ratować twarz, dyrekcja metra zwróciła się do Ciebie z prośbą o opracowanie tras pociągów w taki sposób, by możliwie najwięcej stacji znalazło się na trasach linii metra. Każdy pociąg musi jeździć po ustalonej trasie. Trasy muszą być proste, tzn. nie mogą się rozgałęziać (żadne trzy tunele zbiegające się na jednej stacji nie mogą jednocześnie leżeć na tej samej trasie). Kilka tras może natomiast przebiegać przez tę samą stację lub ten sam tunel.

Zadanie

Zadanie polega na napisaniu programu, który:

- *wczyta ze standardowego wejścia opis sieci tuneli oraz liczbę tras pociągów metra, które należy zaplanować;*
- *obliczy maksymalną liczbę stacji, jakie mogą się znaleźć na wymaganej liczbie tras metra;*
- *zapisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu wejścia zapisane są dwie liczby całkowite n *i* l ($2 \leqslant n \leqslant 1\,000\,000$, $0 \leqslant l \leqslant n$) *oddzielone pojedynczym odstępem. Liczba* n *to liczba stacji, a* l *to liczba tras pociągów, które należy zaplanować. Stacje są ponumerowane od 1 do* n*.*

W każdym z kolejnych $n-1$ *wierszy znajdują się po dwie różne liczby całkowite oddzielone pojedynczym odstępem. Liczby* $1 \leqslant a_i, b_i \leqslant n$ *znajdujące się w* $i+1$*-szym wierszu są numerami stacji połączonych przez* i*-ty tunel.*

Wyjście

W pierwszym i jedynym wierszu wyjścia należy zapisać jedną liczbę całkowitą równa maksymalnej liczbie stacji, jakie mogą znaleźć się na trasach pociągów.

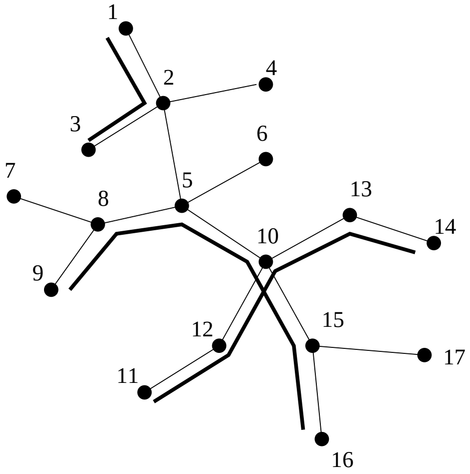
Przykład

Dla danych wejściowych:

```
17 3
1 2
3 2
2 4
5 2
5 6
5 8
7 8
9 8
5 10
10 13
13 14
10 12
12 11
15 17
15 16
15 10
```

poprawnym wynikiem jest:

```
13
```



Na rysunku przedstawiono sieć tuneli z zaznaczonymi trasami metra w jednym z optymalnych układów.

Rozwiązanie

Rozwiązanie

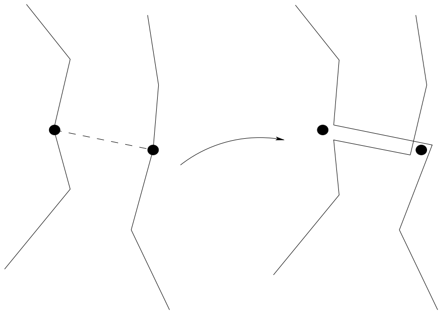
Oczywiście sieć tuneli tworzy *drzewo*[1]. Naszym zadaniem jest znalezienie jego optymalnego (zawierającego maksymalną liczbę wierzchołków) *podgrafu*, który można zbudować z l ścieżek. Zauważmy, że możemy ograniczyć się do poszukiwania *poddrzewa* o tej własności. Wynika to stąd, że jeśli mamy rozwiązanie optymalne, które nie jest spójnym podgrafem drzewa tuneli, to potrafimy je łatwo do takiej postaci przekształcić, nie zmniejszając liczby stacji obsługiwanych przez metro. Wystarczy splatać po dwie trasy tak, jak jest to przedstawione na rysunku 1, aż do uzyskania spójności grafu.

A zatem w n wierzchołkowym drzewie mamy znaleźć maksymalnie liczne poddrzewo, które można pokryć siecią l linii metra. Wydaje się intuicyjnym, że takie pokrycie jest możliwe w przypadku każdego poddrzewa o $2l$ liściach — aby się o tym przekonać, wykażemy następujące

Twierdzenie 1 (Zasadnicze Twierdzenie o Budowie Metra) *Każde drzewo o co najwyżej* $2l$ *liściach da się pokryć* l *liniami metra.*

---

[1] Jest to wniosek z powszechnie znanej charakteryzacji drzewa jako spójnego grafu, w którym liczba krawędzi jest o 1 mniejsza od liczby wierzchołków.



Rys. 1: Splecenie ścieżek

Dowód Na początek przyjmijmy definicję: **skrzyżowaniem** nazywamy każdy węzeł drzewa mający co najmniej trzech sąsiadów.

Aby udowodnić twierdzenie posłużymy się indukcją ze względu na liczbę liści w drzewie.

Jeśli są w nim co najwyżej dwa liście, twierdzenie jest oczywiście prawdziwe. W przeciwnym przypadku w drzewie występuje przynajmniej jedno skrzyżowanie.

Jeśli jest ono tylko jedno, to drzewo ma postać gwiazdy: nie więcej niż $2l$ ścieżek rozchodzi się promieniście z jednego węzła. W takiej sytuacji l linii metra niewątpliwie wystarczy.

Gdy z kolei w drzewie są co najmniej dwa skrzyżowania, jedną z linii metra zaplanujmy tak, by zawierała dwa liście i przynajmniej dwa skrzyżowania. Zmodyfikujmy też drzewo poprzez wyrzucenie z niego kawałków linii metra, łączących liście wybrane do linii z najbliższymi im skrzyżowaniami (w ten sposób usuwamy część wierzchołków i krawędzi, które na pewno zostają pokryte przez wybraną linię metra).

W efekcie liczba liści zmniejszyła się o dwa (w miejscu skrzyżowań nie powstały liście!), co pozwala zastosować założenie indukcyjne. ■

Tym samym zadanie sprowadza się do znalezienia maksymalnie licznego poddrzewa o co najwyżej $2l$ liściach w danym n wierzchołkowym drzewie.

Rozwiązanie autorskie

Ponieważ wybór podzbioru liści drzewa jednoznacznie określa pewne jego poddrzewo, aby znaleźć żądane poddrzewo możemy wybrać jeden jego liść, a następnie dobrać $2l-1$ kolejnych liści. Równoważnie możemy myśleć o dobraniu $2l-1$ ścieżek, prowadzących do liści i odchodzących od już wybranego fragmentu poddrzewa (początkowo jest nim wybrany liść, a później także wszystkie węzły leżące na dobranych ścieżkach).

Każdą ścieżkę w drzewie, której jedynym węzłem należącym do wybranego poddrzewa jest jej koniec, będziemy dalej nazywać **odrostem** tego **poddrzewa**.

W przypadku, gdy $l=1$, wystarczy znaleźć najdłuższą ścieżkę odchodzącą od wybranego liścia, z tym, że wcześniej jako startowy liść trzeba wybrać koniec pewnej spośród najdłuższych ścieżek w drzewie.

Okazuje się, że tę metodę można zastosować także dla $l>1$ i poprawny jest następujący algorytm:

```
$S$ := {koniec pewnej najdłuższej ścieżki w drzewie};
```

```
for $i$ := 1 to min(|{liście}| − 1, 2l − 1) do
    $S$ := $S \cup \{$najdłuższy odrost poddrzewa $S\}$;
return $S$;
```

Aby przekonać się o poprawności powyższej metody rozważmy dwa układy linii metra w naszym drzewie: niech P_{alg} będzie układem stworzonym przez algorytm, zaś P pewnym układem optymalnym, przy czym układy będziemy utożsamiać ze zbiorami ich węzłów.

Niech O będzie pierwszym w kolejności odrostem wybranym przez algorytm takim, że $O \not\subset P$:

- jeśli $O \cap P = \emptyset$, to dowolną ścieżkę O' w układzie P, łączącą liść z najbliższym mu skrzyżowaniem w P, która nie jest zawarta w P_{alg}, można zastąpić odrostem O — taka ścieżka istnieje, bo P jest optymalny i różny od P_{alg}, więc $P \not\subset P_{alg}$. Z definicji O jako najdłuższego odrostu mamy $|O| \geqslant |O'|$;
- jeśli $O \cap P \neq \emptyset$, to istnieje ostatni w stronę liścia O węzeł, wspólny dla O i P. Wówczas dowolną wychodzącą z tego węzła ścieżkę należącą do P można zastąpić przez odpowiedni fragment odrostu O. Z definicji O fragment ten nie jest krótszy niż owa ścieżka.

W obu przypadkach znaleźliśmy inne rozwiązanie optymalne P', które zawiera odrost O oraz wszystkie odrosty wcześniej znalezione przez algorytm.

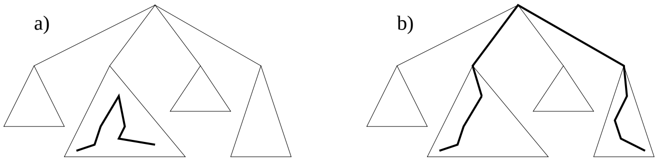
Po co najwyżej $2l-1$ krokach powyższego rozumowania znajdziemy rozwiązanie optymalne P_{opt}, zawierające wszystkie odrosty, czyli takie, że $P_{alg} \subset P_{opt}$. Ponieważ rozwiązanie to zawiera wszystkie liście danego drzewa zawarte w P_{alg}, a P_{alg} zawiera maksymalną możliwą liczbę liści, więc $P_{alg} = P_{opt}$.

Szczegóły implementacyjne

Pozostaje tylko wspomnieć o tym, jak znaleźć koniec którejś z najdłuższych ścieżek oraz jak znajdować najdłuższe odrosty.

Z pierwszym z zadań można uporać się za pomocą przeszukiwania w głąb. Dla ustalenia uwagi wybierzmy dowolny wierzchołek drzewa i nazwijmy go **korzeniem** drzewa. W drzewie ukorzenionym możemy dla każdego węzła zdefiniować jego wysokość — jest to wysokość poddrzewa, którego ten węzeł jest korzeniem. Dla każdego węzła v o wysokości h istnieje ścieżka tej długości łącząca v z jednym z liści jego poddrzewa — ścieżkę tę nazwiemy **pniem** w poddrzewie wierzchołka v. Najdłuższa ścieżka w drzewie:

- albo w całości jest zawarta w którymś z poddrzew korzenia (rys. 2a);
- albo jest zbudowana z korzenia i z pni dwóch poddrzew korzenia o największej wysokości, ewentualnie jednego pnia, jeśli korzeń ma tylko jedno poddrzewo (rys. 2b).



Rys. 2: Położenia najdłuższej ścieżki względem korzenia drzewa

Z kolei znajdowanie najdłuższych odrostów można sprowadzić do wybierania węzłów o największej wysokości — najdłuższy odrost to pień niewybranego dotychczas do metra węzła o największej wysokości wraz z krawędzią łączącą ten węzeł z pokrytym już fragmentem metra.

Poczynione powyżej spostrzeżenia prowadzą do następującego algorytmu:

```
$S$ := {koniec pewnej najdłuższej ścieżki w drzewie};
ukorzeń drzewo w wybranym liściu;
wyznacz wysokości wszystkich węzłów oraz pień dla każdego z nich;
$Kol$ := {sąsiad wybranego liścia};
for $i$ := 1 to min(|{liście}| − 1, 2$l$ − 1) do begin
    $v$ := element $Kol$ o największej wysokości;
    $S$ := $S \cup$ {pień dla węzła $v$};
    $Kol$ := $Kol \cup$ {sąsiedzi węzłów z pnia węzła $v$} − {$v$};
end;
return $S$;
```

Wykorzystując kopiec do reprezentacji kolejki priorytetowej *Kol*, można otrzymać implementację powyższego algorytmu, działającą w czasie $O(n \log n)$.

Można jednak wyznaczone wysokości posortować zawczasu w czasie liniowym za pomocą sortowania kubełkowego, a następnie przejrzeć węzły w kolejności malejących wysokości, co prowadzi do poniższego rozwiązania liniowego.

```
$S$ := {koniec pewnej najdłuższej ścieżki w drzewie};
ukorzeń drzewo w wybranym liściu;
wyznacz wysokości wszystkich węzłów oraz pień dla każdego z nich;
posortuj węzły względem ich wysokości;
$ile\_liści$ := 1;
while $ile\_liści$<min(|{liście}|, 2$l$) do begin
    $v$ := kolejny węzeł w kolejności malejących wysokości;
    if $v \in S$ then continue;
    $S$ := $S \cup$ {pień węzła $v$};
    $ile\_liści$ := $ile\_liści$ + 1;
end;
return $S$;
```

Zaprezentowany powyżej algorytm był podstawą dla rozwiązania, które przyjęto jako wzorcowe podczas zawodów drugiego etapu: jego kod znajduje się w plikach `met.cpp` oraz `met1.pas`.

Rozwiązanie wzorcowe

Zaprezentujemy teraz rozwiązanie, które nie zostało wybrane jako wzorcowe na czas drugiego etapu Olimpiady tylko dlatego, iż nie było ono wtedy znane twórcy niniejszego opracowania. Ze względu jednak na swoją elegancję i prostotę, zasługuje na miano rozwiązania wzorcowego.

Definicja 1 Na początek przyjmijmy następującą definicję *warstwy*:

- liście drzewa tworzą warstwę pierwszą;
- węzły z nieokreślonej jeszcze warstwy, których wszyscy sąsiedzi, prócz co najwyżej jednego, należą do warstw o numerach $\leqslant j-1$, tworzą warstwę numer j.

Podamy teraz dwa proste fakty o warstwach.

Fakt 1 Ścieżki wychodzące z każdego węzła, w prawie każdym kierunku (prócz być może jednego) są malejące w sensie numerów warstw ich węzłów.

Fakt 2 Z każdego węzła warstwy j wychodzi malejąca (w powyższym sensie) ścieżka długości j.

Fakt 1 wynika z tego, że dla każdego węzła co najwyżej jeden sąsiad nie należy do warstwy o niższym numerze. Fakt 2 zaś z tego, że każdy węzeł wewnętrzny ma sąsiada, który należy do warstwy o numerze o jeden mniejszym.

Zauważmy, że na każdej linii metra są co najwyżej 2 węzły z każdej warstwy: gdyby bowiem znalazły się trzy węzły z jednej warstwy, otrzymalibyśmy sprzeczność z faktem 1 dla tego z węzłów, który znajduje się pomiędzy pozostałymi dwoma. Tym samym liczba stacji obsługiwanych przez l linii metra jest niewiększa niż

$$\sum_{j=1}^{\infty} min(2l, |\{\text{warstwa nr j}\}|).$$

Okazuje się, że szukana maksymalna liczba obsługiwanych stacji jest równa tej sumie.

Podobnie jak w przypadku rozwiązania autorskiego, główną trudność stanowi pokazanie poprawności proponowanego rozwiązania.

Jeśli liczba liści drzewa jest $\leqslant 2l$, to na mocy Zasadniczego Twierdzenia o Budowie Metra można zaproponować układ linii, pokrywający całe drzewo, czyli podany wzór jest poprawny.

Załóżmy teraz, że liczba liści jest większa niż $2l$. Niech i będzie najmniejszym numerem warstwy o mocy $\leqslant 2l$:

- każdemu węzłowi warstwy i-tej przypiszmy jeden węzeł z warstwy $(i-1)$-szej;
- z pozostałych węzłów warstwy $(i-1)$-szej wybierzmy dowolne tak, by razem wybrać z tej warstwy $2l$ węzłów.

Na mocy faktu 2 wybranym węzłom z $(i-1)$-szej warstwy odpowiadają parami rozłączne ścieżki długości $(i-1)$. Razem z elementami warstw o numerach $\geqslant i$ tworzą one poddrzewo o $2l$ liściach, które na mocy Zasadniczego Twierdzenia o Budowie Metra można pokryć l liniami. Z drugiej strony liczba węzłów tego drzewa jest równa ograniczeniu górnemu, określonemu przez wzór, co dowodzi poprawności proponowanego algorytmu.

A oto jego pseudokod:

```
wynik := 0;
warstwa := {liście};
while $warstwa \neq \emptyset$ do begin
```

```
    $wynik$ := $wynik$ + min($2l$,$warstwa$);
    $nowa\_warstwa$ := $\emptyset$;
    for $v \in warstwa$ do begin
        $w$ := nieodwiedzony jeszcze sąsiad $v$;
        $w.ilu\_sąsiadów$ := $w.ilu\_sąsiadów - 1$;
        if $w.ilu\_sąsiadów = 1$ then
            $nowa\_warstwa$ := $nowa\_warstwa \cup \{w\}$;
    end
    $warstwa$ := $nowa\_warstwa$;
end;
return $wynik$;
```

O wyższości rozwiązania wzorcowego nad autorskim

Przedstawione powyżej rozumowanie powstało z inspiracji rozwiązaniami stworzonymi w czasie zawodów przez niektórych olimpijczyków.

Dzięki swojej prostocie pozwala ono na stworzenie bardzo szybkiego programu: najszybsze rozwiązanie stworzone przez zawodników było implementacją właśnie tego algorytmu i było szybsze od implementacji rozwiązania autorskiego.

Kolejną widoczną jego zaletą jest dużo mniejsze zapotrzebowanie na pamięć, gdyż nie ma konieczności wykonywania pamięciożernego rekurencyjnego przeszukiwania w głąb, potrzebnego dla znalezienia pierwszego węzła.

Dla przykładu, w zapisanej w pliku `met3.cpp` implementacji algorytmu wzorcowego są wykorzystane zaledwie trzy n-elementowe tablice liczb 32-bitowych i kilka zmiennych. Dzięki trickowemu określaniu nieodwiedzonego sąsiada węzła v jest ona istotnie szybsza zarówno od implementacji rozwiązania autorskiego, jak i od rozwiązań zawodników.

Inne rozwiązania

Mając w pamięci opisane powyżej rozwiązania, można dostrzec przynajmniej dwie kategorie „innych rozwiązań”: mniej efektywne implementacje poprawnych algorytmów oraz niepoprawne heurystyki, będące jedynie ich przybliżeniami.

Rozwiązania mniej efektywne

Do tej grupy możemy zaliczyć implementację rozwiązania autorskiego, wykorzystującą kolejkę priorytetową — odpowiedni program jest zapisany w pliku `met2.c`. Rozwiązanie to działa w czasie $O(n\log n)$.

Inne rozwiązanie o nieoptymalnej złożoności zapisane jest w pliku `mets1.cpp`. W programie tym posłużono się wielokrotnym przeszukiwaniem wszerz w celu znalezienia najdłuższej ścieżki w drzewie, co zaowocowało algorytmem działającym w czasie $O(n^2)$.

Rozwiązania niepoprawne

Przykładem rozwiązania niepoprawnego może być prosta heurystyka zakładająca, że wszystkie węzły wewnętrzne zostaną pokryte przez linie metra. Przypuszczenie to prowadzi do formuły $|\{\text{węzły wewnętrzne}\}| + \min(2l, |\{\text{liście}\}|)$, która wykazuje nieprzypadkowe podobieństwo do wzoru opisującego rozwiązanie wzorcowe.

Oprócz tego istnieje zapewne wiele innych rozwiązań niepoprawnych, ale ich poszukiwanie nie wydaje się być wartościową formą spędzania czasu. Dlatego też poprzestaniemy na tym jednym.

Testy

Do oceny rozwiązań zawodników użyto następującego zestawu testów:

Nazwa	n	l	wynik
met1a.in	15	2	11
met1b.in	15	3	13
met1c.in	2	2	2
met1d.in	19	2	12
met2a.in	13	3	13
met2b.in	2	0	0
met2c.in	16	2	13
met3.in	15	2	13
met4a.in	2 821	600	2 610
met4b.in	5 656	704	4 230

Nazwa	n	l	wynik
met5a.in	33 375	240	31 616
met5b.in	70 023	9996	55 099
met6a.in	52 500	1 400	52 341
met6b.in	119 008	14 101	86 156
met7.in	200 000	47 000	193 969
met8a.in	315 001	70 000	297 228
met8b.in	874 641	123 456	680 918
met9.in	980 001	240 000	969 882
met10.in	991 520	1 200	991 432
met11a.in	1 000 000	30 000	342 789
met11b.in	1 000 000	2	1 000 000

A oto krótka charakteryzacja poszczególnych grup testów:

- `met[1-3].in` to proste testy poprawnościowe,
- `met[4,5].in` to średniej wielkości testy, na których przestają sobie radzić rozwiązania kwadratowe,
- `met[6-8].in` to duże testy wydajnościowe, zaliczane przez rozwiązania liniowe,
- `met[9-11].in` to testy o maksymalnych i prawie maksymalnych rozmiarach, sprawdzające pewne niuanse, jak na przykład możliwość utrzymywania zbyt dużej liczby zmiennych rekurencyjnych w procedurze wyznaczającej najdłuższą ścieżkę.

Jakub Radoszewski
Treść zadania, Opracowanie

Marcin Pilipczuk
Program

OI, Etap II, dzień pierwszy, 15.02.2006

Najazd

Stało się — Trójkątowie najechali Bajtocję! Bajtocja leży na wyspie i zajmuje całą jej powierzchnię. Wyspa ta ma kształt wielokąta wypukłego (tzn. takiego wielokąta, którego każdy kąt wewnętrzny jest mniejszy od 180°). W Bajtocji znajduje się pewna liczba fabryk oprogramowania. Każda fabryka przynosi pewne stałe zyski lub straty.

Trójkątowie postanowili opanować taki fragment Bajtocji, który:

- *ma kształt trójkąta, którego wierzchołkami są pewne trzy różne wierzchołki wielokąta-wyspy,*
- *przyniesie im jak największe zyski, tzn. suma zysków i strat przynoszonych przez fabryki znajdujące się na opanowanym terytorium będzie możliwie jak największa.*

Przyjmujemy, że jeżeli fabryka leży na brzegu lub w wierzchołku opanowanego terytorium, to należy do niego. Terytorium, które nie zawiera żadnej fabryki przynosi oczywiście zysk równy 0.

Król Bajtocji, Bajtazar, zastanawia się, jak duże straty dla gospodarki kraju może przynieść najazd Trójkątów. Pomóż mu i napisz program, który wyznaczy sumę zysków i strat przynoszonych przez fabryki, które chcą opanować Trójkątowie.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis kształtu Bajtocji i położenie fabryk,*
- *wyznaczy maksymalną sumę zysków i strat przynoszonych przez fabryki znajdujące się w obrębie trójkąta, którego wierzchołkami są pewne trzy różne wierzchołki wielokąta-wyspy.*
- *wypisze wynik na standardowe wyjście.*

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę całkowitą n ($3 \leqslant n \leqslant 600$), oznaczającą liczbę wierzchołków wielokąta-wyspy. Kolejnych n wierszy wejścia zawiera po dwie liczby całkowite x_j i y_j ($-10\,000 \leqslant x_j, y_j \leqslant 10\,000$), oddzielone pojedynczym odstępem i oznaczające współrzędne x i y kolejnych wierzchołków wyspy, podane w kolejności zgodnej z kierunkiem ruchu wskazówek zegara. Wiersz $n+2$-gi zawiera jedną liczbę całkowitą m ($1 \leqslant m \leqslant 10\,000$), oznaczającą liczbę fabryk. Kolejne m wierszy zawiera po trzy liczby całkowite x'_i, y'_i i w_i ($-10\,000 \leqslant x'_i, y'_i \leqslant 10\,000$, $-100\,000 \leqslant w_i \leqslant 100\,000$), oddzielone pojedynczymi odstępami i oznaczające odpowiednio: współrzędne x i y i-tej fabryki, oraz zysk (dla $w_i \geqslant 0$) bądź stratę (dla $w_i < 0$), którą ta fabryka przynosi. Każda fabryka leży na wyspie-wielokącie, tzn. wewnątrz niej lub na jej brzegu. Kilka fabryk może leżeć w tym samym miejscu, tj. mieć te same współrzędne.

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać jedną liczbę całkowitą, oznaczającą maksymalną sumę zysków i strat przynoszonych przez fabryki znajdujące się w obrębie trójkąta, którego wierzchołkami są pewne trzy różne wierzchołki wielokąta-wyspy. Może się zdarzyć, że liczba ta będzie ujemna.

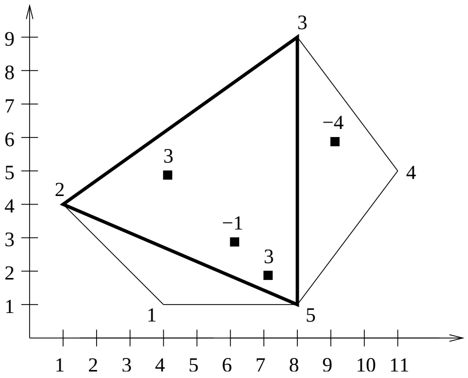
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
5
```



Rozwiązanie

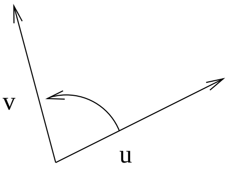
Rozwiązanie siłowe

Zacznijmy od konstrukcji jakiegokolwiek rozwiązania zadania. Najprostszy pomysł, jaki przychodzi na myśl, to przeanalizowanie każdego dopuszczalnego trójkąta (jest ich $O(n^3)$) i zsumowanie zysków/strat wszystkich fabryk w nim zawartych. Dostajemy w ten sposób algorytm, który można nazwać naiwnym, działający w czasie $O(n^3 m)$, o ile … umiemy w czasie stałym sprawdzić, czy dany punkt należy do danego trójkąta.

Jest to pytanie z zakresu geometrii obliczeniowej, czyli dziedziny matematyki i informatyki, która zajmuje się geometrią z algorytmicznego punktu widzenia. Zagadnienia z tej dziedziny są na ogół łatwe do rozwiązania dla człowieka (wystarczy narysować trójkąt oraz punkt i wynik widać „gołym okiem”), ale o wiele trudniejsze do zapisania w postaci algorytmu, gdy trzeba posługiwać się odpowiednimi wzorami matematycznymi. Ponadto często trudno tak zapisać algorytm, żeby działał poprawnie dla wszystkich przypadków szczególnych. Wiąże się to między innymi z ograniczoną dokładnością działań na typach zmiennoprzecinkowych na komputerze. Ich używanie może uniemożliwić na przykład bezbłędne sprawdzenie, czy punkt należy do danego odcinka (w tym celu trzeba porównać dwie liczby rzeczywiste, których reprezentacja w komputerze może być przybliżona), stąd zawsze warto próbować pozostać w dziedzinie liczb całkowitych, na których obliczenia są dokładne.

W geometrii obliczeniowej bardzo istotną rolę odgrywa *iloczyn wektorowy* (oznaczany $\vec{u} \times \vec{v}$). Przypomnijmy jego najważniejsze własności dla wektorów na płaszczyźnie (więcej o tym działaniu można przeczytać w dowolnej książce o geometrii obliczeniowej, np. [34], a także w [18]):

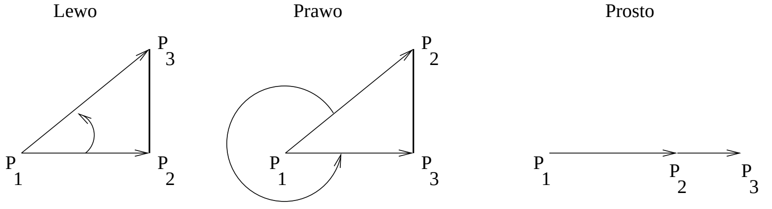
1. Iloczyn wektorowy dwóch wektorów jest także wektorem, którego kierunek jest prostopadły do płaszczyzny, w której zawarte są oba te wektory, zwrot jest wyznaczony z reguły prawej dłoni (śruby prawoskrętnej), a wartość równa jest $|\vec{u} \times \vec{v}| = |\vec{u}| \cdot |\vec{v}| \cdot \sin\alpha$, gdzie α jest kątem skierowanym między wektorami $\vec{u}$ i $\vec{v}$ (zauważmy, że tak zdefiniowana wartość jest liczbą ze znakiem, czyli uwzględnia zwrot — będziemy z tego korzystać w dalszej części opisu).



Rys. 1: Kąt skierowany

2. Z powyższej własności wynika, że wartość bezwzględna wartości iloczynu wektorowego jest równa dwukrotności pola trójkąta, zawartego między wektorami $\vec{u}$ i $\vec{v}$.

3. Na podstawie pierwszej własności możemy sprawdzić, czy przechodząc przez łamaną łączącą kolejno punkty P_1, P_2 i P_3 (będziemy ją oznaczać $P_1 \rightarrow P_2 \rightarrow P_3$) skręcamy w punkcie P_2 w prawo ($|\overrightarrow{P_1P_2} \times \overrightarrow{P_1P_3}| < 0$), w lewo ($|\overrightarrow{P_1P_2} \times \overrightarrow{P_1P_3}| > 0$) czy idziemy prosto ($|\overrightarrow{P_1P_2} \times \overrightarrow{P_1P_3}| = 0$). W pseudokodach w niniejszym opracowaniu przy badaniu kierunku skrętu będziemy posługiwać się funkcją $\text{Skręt}(P_1, P_2, P_3)$, o wartościach ze zbioru $\{Lewo, Prawo, Prosto\}$.



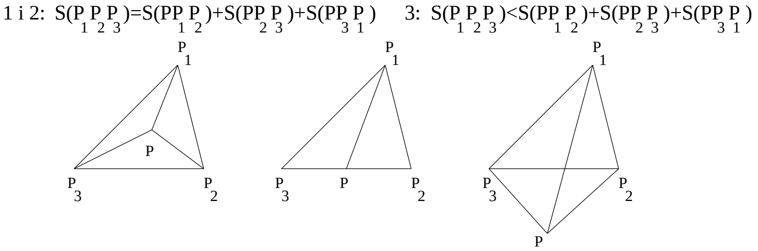
Rys. 2: Sprawdzanie kierunku skrętu

4. Jeżeli $\vec{u} = [x_1, y_1]$, a $\vec{v} = [x_2, y_2]$, to $|\vec{u} \times \vec{v}| = x_1y_2 - x_2y_1$. Jest to bardzo ważna własność, która pozwala łatwo obliczać wartość iloczynu wektorowego i dodatkowo upewnia nas, że liczba ta jest całkowita, o ile tylko współrzędne wektorów są całkowitoliczbowe.

Z użyciem iloczynu wektorowego możemy już łatwo określić, czy punkt P leży wewnątrz trójkąta $\triangle P_1P_2P_3$: wystarczy sprawdzić, czy

$$S(\triangle P_1P_2P_3) = S(\triangle PP_1P_2) + S(\triangle PP_2P_3) + S(\triangle PP_3P_1),$$

gdzie $S(\triangle ABC)$ oznacza pole trójkąta $\triangle ABC$.

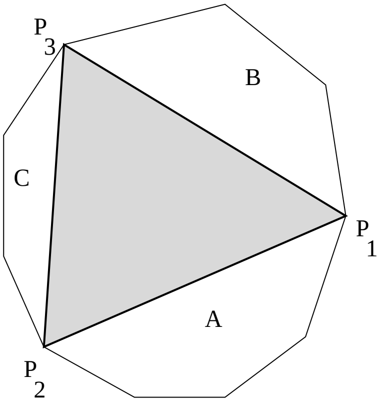


Rys. 3: Sprawdzanie, czy punkt należy do trójkąta

To kończy opis algorytmu naiwnego. Mimo nikłej trudności tego rozwiązania, można było za nie zdobyć w czasie zawodów 30 punktów, gdyż w pełni poprawna implementacja wymagała pewnej wiedzy z zakresu geometrii obliczeniowej.

Szybsze rozwiązanie

Złożoność czasowa powyższego rozwiązania nie jest wystarczająca przy ograniczeniach z zadania. W kolejnym rozwiązaniu najpierw wykonamy obliczenia, które potem umożliwią szybsze znajdowanie sumy zysków/strat fabryk zawartych w danym trójkącie. Najistotniejszym spostrzeżeniem, prowadzącym do tego rozwiązania jest przedstawienie wyniku dla dopuszczalnego trójkąta, jako różnicy sumy wartości wszystkich fabryk i sum wartości fabryk zawartych w trzech półpłaszczyznach otwartych, ograniczonych przez boki tego trójkąta (oznaczonych na rysunku przez A, B i C):



Rys. 4: Półpłaszczyzny A, B i C

Aby zrozumieć, dlaczego waga powyższego spostrzeżenia jest tak duża, zastanówmy się, ile jest różnych takich półpłaszczyzn. Zauważmy, że każda z półpłaszczyzn jest wyznaczona jednoznacznie przez wektor złożony z dwóch wierzchołków wielokąta-wyspy, a zatem różnych półpłaszczyzn jest $O(n^2)$. Dla ustalenia uwagi oznaczmy przez (P_1, P_2) półpłaszczyznę złożoną z punktów P takich, że przechodząc przez $P_1 \rightarrow P_2 \rightarrow P$ skręcamy w lewo (na przykład na powyższym rysunku $A = (P_1, P_2)$). Teraz możemy sformułować rozwiązanie:

1. Najpierw dla każdej półpłaszczyzny (P_1, P_2) wyznaczamy sumę wartości fabryk w niej zawartych — oznaczmy tę wartość przez $f[P_1, P_2]$. Możemy to uczynić obliczając

dla każdego punktu-fabryki odpowiedni iloczyn wektorowy. Ten krok ma złożoność czasową $O(n^2m)$, właśnie dzięki niewielkiej liczbie różnych półpłaszczyzn.

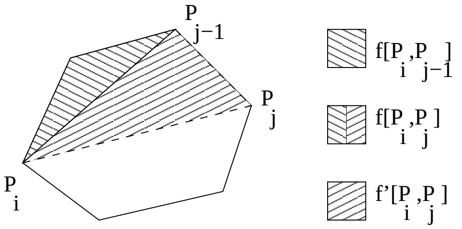
2. Następnie wynik dla trójkąta $\triangle P_1P_2P_3$ wyznaczamy w czasie stałym, jako $S - f[P_1,P_2] - f[P_2,P_3] - f[P_3,P_1]$, gdzie S jest sumą wartości wszystkich fabryk. Złożoność czasowa tego kroku to $O(n^3)$.

Całkowita złożoność czasowa tego rozwiązania to $O(n^2(n+m))$. Pozwalało ono uzyskać 70 punktów, jako że zawierało pewien pomysł natury algorytmicznej, który — jak się za chwilę okaże — ulepszony stanowi już rozwiązanie wzorcowe.

Rozwiązania wzorcowe

Kluczem do rozwiązania wzorcowego jest przyspieszenie pierwszej fazy wyżej opisanego algorytmu. Odtąd będziemy w opisie zakładali, że wierzchołki wielokąta są ponumerowane w kolejności zgodnej z kierunkiem ruchu wskazówek zegara: $P_1,P_2,\ldots,P_n$. Ponadto przyjmujemy, że numeracja wierzchołków jest cykliczna, to znaczy $P_0 = P_n$, $P_{n+1} = P_1$ itd.

Będziemy starali się policzyć wartości $f'[P_i,P_j]$, zdefiniowane jako $f'[P_i,P_j] = f[P_i,P_j] - f[P_i,P_{j-1}]$. Odpowiadają one sumom wartości fabryk, zawartych w jednostronnie otwartych trójkątach (trójkąt o wierzchołkach P_i, P_{j-1} oraz P_j, niezawierający boku $\overline{P_iP_j}$ w dalszej części będziemy oznaczać symbolem $\angle P_iP_{j-1}P_j$).



Rys. 5: Ilustracja f i f'

Na podstawie tych wartości będziemy w stanie w złożoności czasowej $O(n^2)$ policzyć wartości f — dla każdego punktu P_i wyznaczymy wszystkie wartości $f[P_i,*]$ na podstawie równości:

- $f[P_i,P_i] = f[P_i,P_{i+1}] = 0$,
- $f[P_i,P_j] = f[P_i,P_{j-1}] + f'[P_i,P_j]$ dla $j > i+1$,

Zastanówmy się więc jak policzyć wartości f'. Skupimy się na wyznaczeniu wszystkich wartości $f'[P_i,*]$ dla ustalonego punktu P_i. Opiszemy dwie metody wykonania tego podzadania.

Sposób pierwszy

W danym kroku przeanalizujemy wszystkie fabryki i dla każdej z nich stwierdzimy, w którym trójkącie $\angle P_iP_{j-1}P_j$ się znajduje — oczywiście każda fabryka należy do co najwyżej jednego takiego trójkąta. Zauważmy, że fabryka F położona na wyspie nie należy do żadnego z podanych trójkątów wtedy i tylko wtedy, gdy należy do odcinka $\overline{P_{i-1}P_i}$. Możemy to sprawdzić obliczając jeden iloczyn wektorowy. W pozostałych przypadkach możemy zidentyfikować odpowiedni trójkąt za pomocą wyszukiwania binarnego, w którym będziemy dla danej przekątnej $\overline{P_iP_j}$ sprawdzać, w jakim kierunku skręcamy przechodząc przez $P_i \to P_j \to F$.

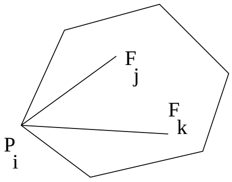
Powyższy krótki opis pozwala już na ułożenie pseudokodu tego rozwiązania:

```
{ Wyliczanie $f'[P_i,*]$. }
for $j := 1$ to $m$ do
begin
   if (Skręt($P_i$,$P_{i-1}$,$F_j$)= Prosto) then
      { Fabryka należy do odcinka $\overline{P_{i-1}P_i}$ }
      continue;
   { Dla fabryki $F_j$ szukamy $a$, takiego że $F_j \in \angle P_iP_{a-1}P_a$. }
   $a := i+1$, $b := i+n-1$;
   while ($a \neq b$) do
   begin
      $c := \lfloor \frac{a+b}{2} \rfloor$;
      if (Skręt($P_i$,$P_c$,$F_j$)$\in \{Prawo, Prosto\}$) then
         $a := c+1$
      else
         $b := c$;
   end
   $f'[P_i,P_a] := f'[P_i,P_a]$+wartość($F_j$);
end
```

Złożoność czasowa powyższego kodu to oczywiście $O(m \log n)$. Wykorzystując tę metodę możemy skonstruować rozwiązanie całego zadania o złożoności $O(nm \log n + n^3)$. Jest to pierwsze z rozwiązań wzorcowych.

Sposób drugi

Zamiast stosować wyszukiwanie binarne, możemy dla danego punktu P_i posortować wszystkie fabryki po kącie nachylenia odcinków, których końcami są punkt P_i i dana fabryka. Dokładniej, powiemy że $F_j < F_k$, jeżeli przechodząc przez $F_j \to P_i \to F_k$ skręcamy w lewo.



Rys. 6: Kryterium sortowania

W tej samej kolejności mamy już posortowane wierzchołki wielokąta — kolejność zgodna z kierunkiem ruchu wskazówek zegara jest właśnie takim porządkiem. Teraz możemy połączyć te dwa posortowane zbiory, czyli wykonać krok podobny do fazy scalania w algorytmie sortowania przez scalanie (ang. Merge Sort). W trakcie tego kroku kolejne fabryki są przydzielane do pierwszego trójkąta, a kiedy jakaś fabryka przestaje się w nim mieścić, to zaczynamy przydzielanie do drugiego trójkąta itd. Zauważmy, że do sprawdzenia przynależności punktu do trójkąta wystarczy nam tym razem jedno użycie iloczynu wektorowego.

Pseudokod tego rozwiązania jest jeszcze łatwiejszy niż poprzedniego:

```
{ Wyliczanie $f'[P_i,*]$. }
Sortowanie zbioru $\{F_1,\ldots,F_m\}$ w porządku kątowym;
$j := 1, a := i+2$;
while ($a \neq i$) do
begin
  while ($j \leqslant m$ and $Skręt(P_i,P_a,F_j) = Lewo$) do
  begin
    { Fabryka należy do trójkąta $\angle\ P_iP_{a-1}P_a$]. }
    $f'[P_i,P_a] := f'[P_i,P_a] + wartość(F_j)$;
    $j := j+1$;
  end
  $a := a+1$; { Przypominamy, że numeracja wierzchołków jest cykliczna. }
end
```

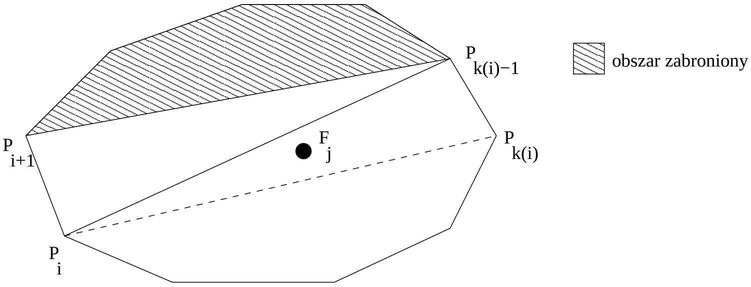
Sortowanie przy zastosowaniu szybkiego algorytmu ma złożoność $O(m\log m)$, faza scalania wymaga czasu liniowego względem sumy liczb fabryk i wierzchołków wielokąta. A zatem złożoność powyższej procedury to $O(n+m\log m)$, a złożoność czasowa całego algorytmu to $O(nm\log m+n^3)$. Otrzymujemy więc metodę nieznacznie gorszą od poprzedniej.

Ponieważ nie sposób odróżnić powyższe rozwiązania w procesie testowania, to oba zostały uznane za wzorcowe. Implementacje obu rozwiązań znajdują się na dysku dołączonym do książeczki.

Rozwiązanie szybsze od wzorcowego

Okazuje się, że istnieje szybsze rozwiązanie zadania — pomysł przedstawili sami zawodnicy olimpiady. Idea rozwiązania polega na szybszym wykonaniu fazy obliczania wartości f (a właściwie f'). Tym razem dla to danej fabryki F_j i wszystkich punktów P_i będziemy wyznaczali punkty $P_{k(i)}$, takie że F_j należy do $\angle P_iP_{k(i)-1}P_{k(i)}$.

Rozpoczynamy od wyznaczenia wartości $k(1)$, na przykład za pomocą najprostszego, liniowego algorytmu, analizując kolejne punkty $P_2, P_3, \ldots$ Następnie zauważamy, że dla kolejnych punktów P_i wartość $k(i)$ będzie tylko rosła (ponownie przypominamy o cyklicznej numeracji punktów). Na rysunku 7 pokazujemy, że rzeczywiście $P_{k(i+1)}$ nie może być wierzchołkiem, leżącym na brzegu wielokąta między wierzchołkami P_i oraz $P_{k(i)-1}$:



Rys. 7: Dowód stwierdzenia

A zatem możemy wyznaczać kolejne wartości $k(i)$ zwiększając o jeden wskaźnik, który na samym końcu tego procesu musi znaleźć się w punkcie $k(1)$, czyli wykonamy co najwyżej n ruchów. W ten sposób wszystkie wartości $k(i)$ dla fabryki F_j znajdujemy w czasie $O(n)$. Obliczenie wszystkich wartości f' ma złożoność równą $O(nm)$, czyli lepszą niż w przypadku rozwiązania wzorcowego, a całe rozwiązanie ma złożoność $O(nm+n^3) = O(n(m+n^2))$.

Na koniec zilustrujemy powyższy opis pseudokodem rozwiązania, który już tradycyjnie jest krótszy od pseudokodu poprzedniego rozwiązania:

```
1:  { Dla danej fabryki $F_j$ wyznaczamy $P_{k(i)}$ dla $i = 1, 2, \ldots, n$. }
2:  $k := 1$;
3:  for $i := 1$ to $n$ do
4:  begin
5:    { W pętli 7-8 zarówno wyliczamy $k(1)$ w złożoności $O(n)$, }
6:    { jak i pozostałe wartości $k(i)$ w łącznej złożoności liniowej. }
7:    while ($k \neq i$ and Skręt($P_i$,$P_k$,$F_j$)$\neq$ Lewo) do
8:      $k := k + 1$; { Przypominamy, że numeracja wierzchołków jest cykliczna. }
9:    if ($k \neq i$) then
10:     $f'[P_i, P_k] := f'[P_i, P_k]$+wartość($F_j$);
11: end
```

Testy

Zadanie testowane było na zestawie 10 danych testowych, z których pierwszych 7 było grupami złożonymi z dwóch testów. Większość stanowiły testy losowe, a pozostałe były testami „specyficznymi”, które charakteryzowały się tym, że kolejne trójkąty były na przemian zyskowne i stratne, co pomagało eliminować rozwiązania, polegające na znajdowaniu pewnego maksimum lokalnego w zbiorze trójkątów.

Nazwa	**n**	**m**	**Opis**
naj1a.in	64	200	test losowy
naj1b.in	64	200	test specyficzny
naj2a.in	64	200	test losowy

Nazwa	n	m	Opis
naj2b.in	64	200	test specyficzny
naj3a.in	64	200	test losowy
naj3b.in	64	200	test specyficzny
naj4a.in	200	5 000	test losowy
naj4b.in	200	500	test specyficzny
naj5a.in	200	5 000	test losowy
naj5b.in	200	500	test specyficzny
naj6a.in	200	5 000	test losowy
naj6b.in	200	500	test specyficzny
naj7a.in	200	5 000	test losowy
naj7b.in	200	500	test specyficzny
naj8.in	600	10 000	test losowy
naj9.in	600	10 000	test losowy
naj10.in	600	10 000	test losowy

Tomasz Waleń
Treść zadania, Opracowanie

Jakub Pawlewicz
Program

OI, Etap II, dzień drugi, 16.02.2006

Listonosz

Listonosz Bajtazar codziennie musi odwiedzić wszystkie ulice swojego rejonu i dostarczyć listy. Wszystkie ulice są jednokierunkowe i łączą (parami różne) skrzyżowania. Parę skrzyżowań mogą łączyć co najwyżej dwie ulice: jedna w jednym, a druga w drugim kierunku. Skrzyżowania są ponumerowane od 1 do n.

Bajtazar rozpoczyna i kończy trasę w centrali poczty Bajtockiej, przy skrzyżowaniu nr 1. Od dawien dawna Bajtazar sam wybierał trasę, którą obchodził swój rejon, jednak ostatnio dyrekcja poczty wydała nowe rozporządzenie, ograniczające swobodę wyboru tras. Każdemu listonoszowi przydzielono pewien zestaw fragmentów trasy — zbiór sekwencji skrzyżowań. Bajtazar musi wybrać taką trasę, która:

- *prowadzi każdą ulicą dokładnie raz,*
- *zawiera w sobie każdą z zadanych sekwencji (jako spójny podciąg),*
- *rozpoczyna się i kończy na skrzyżowaniu nr 1,*

Niestety, możliwe jest, że dyrekcja wydała rozporządzenie, dla którego nie istnieje trasa Bajtazara spełniająca wymogi, np. wymaga ono w jednej ze swoich sekwencji pójścia drogą, która nie istnieje. Pomóż Bajtazarowi i napisz program, który sprawdzi czy poprawna trasa istnieje, a jeśli tak, to wyznaczy ją.

Zadanie

Napisz program który:

- *wczyta ze standardowego wejścia opis ulic i przydzielone sekwencje,*
- *sprawdzi czy Bajtazar może obejść swój rejon, tak żeby odwiedzić każdą ulicę dokładnie raz oraz spełnić wszystkie zalecenia dyrekcji,*
- *wypisze na standardowe wyjście znalezioną trasę lub stwierdzi, że taka trasa nie istnieje.*

Wejście

W pierwszym wierszu standardowego wejścia zapisanie są dwie liczby całkowite n i m oddzielone pojedynczym odstępem, $2 \leqslant n \leqslant 50\,000$, $1 \leqslant m \leqslant 200\,000$, odpowiednio liczba skrzyżowań i ulic. W kolejnych m wierszach znajdują się opisy ulic: dwie liczby całkowite a, b, oddzielone pojedynczym odstępem, $1 \leqslant a,b \leqslant n$, $a \neq b$, oznaczają, że ze skrzyżowania a do b prowadzi (jednokierunkowa) ulica. Każda (uporządkowana) para a,b pojawia się w danych co najwyżej raz. W kolejnym wierszu zapisana jest liczba t, $0 \leqslant t \leqslant 10\,000$, oznaczająca liczbę nakazanych sekwencji. W kolejnych t wierszach zapisane są opisy sekwencji. Opis sekwencji składa się z liczby k, $2 \leqslant k \leqslant 200\,000$, oraz ciągu $v_1,\ldots,v_k$ numerów skrzyżowań. Liczby w wierszu są pooddzielane pojedynczymi odstępami. Sumaryczna długość wszystkich sekwencji nie przekracza $1\,000\,000$.

Wyjście

Twój program powinien wypisać w pierwszym wierszu wyjścia:

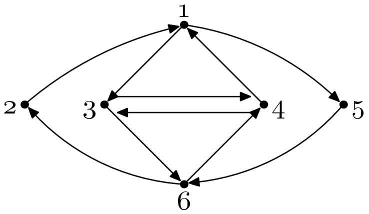
- `TAK` — *jeśli istnieje trasa spełniająca warunki zadania,*
- `NIE` — *jeśli taka trasa nie istnieje.*

W przypadku odpowiedzi `TAK`*, w kolejnych wierszach należy zapisać opis znalezionej trasy. Jeśli jest wiele takich tras, można wypisać dowolną z nich. Opis powinien składać się z sekwencji skrzyżowań kolejno odwiedzanych na trasie listonosza —* $m+1$ *liczb:* $v_1, \ldots, v_{m+1}$*, każdej wypisanej w osobnym wierszu, takich że:*

- $v_1 = v_{m+1} = 1$,
- v_i *i* v_{i+1} *(dla* $1 \leqslant i \leqslant m$*) są połączone ulicami,*
- *każda ulica występuje na liście dokładnie raz,*
- *trasa zawiera, jako spójne podciągi, wszystkie nakazane przez dyrekcję sekwencje.*

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`6 10`	`TAK`
`1 5`	`1`
`1 3`	`3`
`4 1`	`4`
`6 4`	`3`
`3 6`	`6`
`3 4`	`4`
`4 3`	`1`
`5 6`	`5`
`6 2`	`6`
`2 1`	`2`
`4`	`1`
`3 1 5 6`	
`3 3 4 3`	
`4 4 3 6 4`	
`3 5 6 2`	



Rozwiązanie

Przedstawmy sytuację opisaną w zadaniu jako graf, w którym wierzchołkami są skrzyżowania, a krawędziami — ulice. Zadanie polega na wyznaczeniu cyklu przechodzącego przez każdą krawędź grafu dokładnie raz, który spełnia zadane dodatkowe warunki: przez wybrane sekwencje krawędzi musimy przejść w określonym porządku.

Jeśli zbiór sekwencji byłby pusty, zadanie wymagałoby jedynie wyznaczenia cyklu Eulera, co potrafimy zrobić szybko i prosto (patrz na przykład [27] bądź [18]). W dalszej części opisu pokażemy jak zmodyfikować graf, by problem sprowadzić do znajdowania zwykłego cyklu Eulera.

Rozwiązanie dla jednej sekwencji

Rozpocznijmy od przypadku, gdy mamy tylko jedną sekwencję, czyli $t = 1$. Wówczas możemy zmodyfikować graf, usuwając z niego wszystkie krawędzie sekwencji i wprowadzając w zamian jedną krawędź — łączącą początek pierwszej krawędzi sekwencji z końcem ostatniej. W ten sposób każdy cykl Eulera w zmodyfikowanym grafie będzie odpowiadał cyklowi Eulera w grafie oryginalnym zawierającemu sekwencję. Jedynym (być może) skomplikowanym punktem powyższej procedury jest konieczność zapamiętania, że nowa krawędź zastępuje całą sekwencję krawędzi z oryginalnego grafu. Ta informacja będzie nam potrzebna przy wypisywaniu wyniku.

Schemat takiego rozwiązania możemy zapisać algorytmicznie:

- niech $G = (V,E)$ oznacza graf skrzyżowań, a jedyna zadana sekwencja ma postać $p_1,\ldots,p_k$.
- usuń z E krawędzie z sekwencji: $\{(p_i,p_{i+1}) : 1 \leqslant i < k\}$,
- dodaj do E krawędź (p_1,p_k) z etykietą $p_1,\ldots,p_k$,
- oblicz cykl Eulera w zmodyfikowanym grafie.

Ten sam sposób można zastosować w przypadku, gdy mamy wiele sekwencji, które są krawędziowo rozłączne. Każdą z nich zastępujemy krawędzią i wyznaczamy w zmienionym grafie cykl Eulera.

Ponieważ cykl Eulera można obliczyć w czasie $O(|V| + |E|)$, więc całe rozwiązanie wymaga czasu liniowego względem rozmiaru danych wejściowych.

Rozwiązanie dla ogólnego przypadku

Pozostaje nam do rozważenia przypadek, gdy zadane sekwencje mają wspólne krawędzie. W pierwszej kolejności sprawdzimy, czy sekwencje nie są ze sobą sprzeczne (zauważmy na przykład, że nie jest możliwe jednoczesne spełnienie sekwencji $(1,2,3)$ i $(1,2,4)$). Użyjemy w tym celu dwóch tablic *nast* i *poprz* indeksowanych krawędziami grafu. W $nast[e]$ zapiszemy informację, jaką krawędź musimy przejść bezpośrednio po krawędzi e, jeśli takie ograniczenie wynika z zadanych sekwencji. Analogicznie wypełnimy tablicę *poprz*.

```
for $e \in E$ do $nast[e] := \mathbf{nil}, poprz[e] := \mathbf{nil}$;
for $s := 1$ to $t$ do begin
```

```
  { Niech $p_1, \ldots, p_k$ oznacza $s$–tą sekwencję }
  for $i := 1$ to $k-2$ do
    $e_1 := (p_i, p_{i+1})$; $e_2 := (p_{i+1}, p_{i+2})$;
    if $e_1 \notin E$ or $e_2 \notin E$ then
      return ”BŁĄD - krawędzi nie ma w grafie”;
    if $nast[e_1] =$ nil then $nast[e_1] := e_2$
    else if $nast[e_1] \neq e_2$ then return ”BŁĄD - sprzeczne sekwencje”;
    if $poprz[e_2] =$ nil then $poprz[e_2] := e_1$
    else if $poprz[e_2] \neq e_1$ then return ”BŁĄD - sprzeczne sekwencje”;
  end;
end;
return ”OK”;
```

Jeśli powyższa procedura zwróci błąd, to nie istnieje cykl zawierający wszystkie zadane sekwencje. Jeśli procedura zakończy się sukcesem, to możemy z tablic *nast*/*poprz* przygotować nowy zestaw sekwencji — równoważny oryginalnemu, ale już krawędziowo rozłączny. Wystarczy „odczytać” ścieżki zapisane na przykład w tablicy *nast* — są to zsumowane zadane sekwencje.

Trudności techniczne

Jakkolwiek rozwiązanie zadania nie jest zbyt trudne do wymyślenia, jednak poprawna jego implementacja może sprawiać pewne trudności.

Pierwszym problemem do rozwiązania jest przygotowanie tablic indeksowanych krawędziami grafu. Można do tego celu wykorzystać strukturę `map` z biblioteki STL, jednak w takim przypadku dostęp do jednej pozycji tablicy wymaga czasu $O(\log n)$. Innym rozwiązaniem jest „ponumerowanie” krawędzi przez przypisanie im liczb z zakresu $1, \ldots, |E|$. Taką numerację możemy uzyskać, sortując krawędzie jako pary wierzchołków — ze względu na ograniczony zbiór wierzchołków możemy wykonać to w czasie liniowym. Jeśli każdą krawędź sekwencji uzupełnimy o jej numer, późniejsze odwołania do tablic *nast*/*poprz* można wykonać w czasie $O(1)$.

Innym problemem są przypadki szczególne. Aby poprawnie rozwiązać zadanie, trzeba uniknąć wielu pułapek:

- Graf spójny po modyfikacji może składać się z więcej niż jednej spójnej składowej. Jeśli co najmniej dwie z nich zawierają krawędzie, to rozwiązanie nie istnieje.
- Problemem są także cykle powstałe przez zsumowanie sekwencji. Jeśli krawędzie z tablic *nast*/*poprz* tworzą cykl (na przykład, jak dla sekwencji $S_1 = (1,2,3)$ i $S_2 = (2,3,1,2)$), to:
  - jeśli ten cykl nie zawiera wszystkich krawędzi, to rozwiązanie na pewno nie istnieje;
  - jeśli cykl zawiera wszystkie krawędzie grafu, to trzeba jeszcze sprawdzić, czy ograniczenia nałożone przez sekwencje pozwalają listonoszowi zacząć i zakończyć trasę w centrali poczty.

Kolejną trudnością jest konieczność zapisu „długich” etykiet krawędzi stanowiących skróty ścieżek oryginalnego grafu oraz ich odczytywanie podczas wypisywania odpowiedzi.

Testy

Zadanie testowane było na następującym zestawie danych testowych.

Nazwa	n	m	t	∑k	Opis
lis1a.in	3	3	2	4	na sprawdzanie warunku Eulera (odp. NIE)
lis1b.in	4	6	2	4	prosty test poprawnościowy
lis2a.in	5	7	2	6	sprzeczne sekwencje (odp. NIE)
lis2b.in	11	20	3	10	na sklejanie ścieżek
lis3a.in	7	9	2	6	sprzeczne sekwencje (odp. NIE)
lis3b.in	7	14	4	12	koniec ścieżki w początku ścieżki
lis4a.in	5	6	2	8	ścieżki po sklejeniu dają cykl (odp. NIE)
lis4b.in	7	12	10	24	na sklejanie ścieżek
lis5a.in	7	9	2	6	rozspójnienie grafu po skróceniu sekwencji (odp. NIE)
lis5b.in	20	100	50	200	test losowy
lis6.in	100	500	50	200	test losowy
lis7.in	500	2 000	1 000	10 000	test losowy
lis8.in	2 000	10 000	10 000	50 000	test losowy
lis9.in	10 000	50 000	9 999	200 000	test losowy
lis10.in	49 999	60000	100	400	test losowy
lis11.in	450	199 998	500	4 000	test losowy
lis12.in	300	30 000	10 000	100 000	test losowy
lis13.in	40 000	200 000	5	1 000 000	test losowy
lis14.in	50 000	200 000	10 000	1 000 000	test losowy

Marcin Kubica
Treść zadania, Opracowanie

Marek Cygan
Program

OI, Etap II, dzień drugi, 16.02.2006

Orka

Rolnik Bajtazar chce zaorać pole w kształcie prostokąta. Bajtazar może zacząć od zaorania jednej skiby z dowolnego boku pola, potem może zaorać jedną skibę z dowolnego boku niezaoranej części pola itd., aż całe pole będzie zaorane. Po zaoraniu każdej kolejnej skiby, niezaorana część pola ma kształt prostokąta. Skiby mają szerokość 1, a długość i szerokość pola wyrażają się liczbami całkowitymi m *i* n*.*

Niestety Bajtazar do orki ma tylko jedną słabowitą szkapę. Gdy szkapa zacznie orać skibę, to nie zatrzymuje się, aż zaorze ją do końca. Bajtazar musi uważać, jeżeli zaoranie skiby będzie dla szkapy zbyt wielkim wysiłkiem, to szkapa padnie. Po zaoraniu każdej kolejnej skiby szkapa może odpocząć i nabrać sił. Nie wszystkie miejsca na polu są tak samo trudne do zaorania. Bajtazar dokładnie zna swoje pole i dokładnie wie jak trudno się orze w każdym miejscu.

Podzielmy pole na $m \times n$ *kwadratów jednostkowych. Kwadraty będziemy identyfikować za pomocą ich współrzędnych* (i,j)*, dla* $1 \leqslant i \leqslant m$ *i* $1 \leqslant j \leqslant n$*. Każdemu z kwadratów jest przypisany jego współczynnik trudności orki — nieujemna liczba całkowita. Współczynnik trudności orki kwadratu o współrzędnych* (i,j) *będziemy oznaczać przez* $t_{i,j}$*. Dla każdej skiby suma współczynników trudności orki kwadratów tworzących skibę nie może przekroczyć pewnej ustalonej stałej* k *— w przeciwnym przypadku szkapa padnie.*

Bajtazar stoi przed trudnym zadaniem. Przed zaoraniem każdej skiby musi zdecydować z którego boku niezaoranej części pola ją zaorać, tak żeby szkapa nie padła. Z drugiej strony, chciałby żeby było jak najmniej skib.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia liczby* k*,* m *i* n*, oraz współczynniki trudności orki,*
- *wyznaczy, w jaki sposób Bajtazar powinien zaorać pole,*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu standardowego wejścia znajdują się trzy dodatnie liczby całkowite: k*,* m *i* n*, pooddzielane pojedynczymi odstępami,* $1 \leqslant k \leqslant 200\,000\,000$*,* $1 \leqslant m \leqslant 2\,000$*,* $1 \leqslant n \leqslant 2\,000$*. W kolejnych* n *wierszach znajdują się współczynniki trudności orki. Wiersz* $j+1$ *zawiera współczynniki* $t_{1,j}, t_{2,j}, \ldots, t_{m,j}$*, pooddzielane pojedynczymi odstępami,* $0 \leqslant t_{i,j} \leqslant 100\,000$*.*

Wyjście

Twój program powinien wypisać na standardowe wyjście jedną liczbę całkowitą: minimalną liczbę skib powstałych po zaoraniu pola zgodnie z podanymi warunkami. Możesz założyć, że dla danych wejściowych, pole zawsze da się zaorać zgodnie z warunkami podanymi w zadaniu.

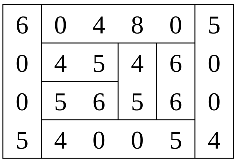
Przykład

Dla danych wejściowych:

```
12 6 4
6 0 4 8 0 5
0 4 5 4 6 0
0 5 6 5 6 0
5 4 0 0 5 4
```

poprawnym wynikiem jest:

```
8
```



6	0	4	8	0	5
0	4	5	4	6	0
0	5	6	5	6	0
5	4	0	0	5	4

Powyższy rysunek przedstawia przykładowy sposób zaorania pola.

Rozwiązanie

Rozwiązanie wykładnicze

Jako pierwsze przedstawimy rozwiązanie proste, lecz nieefektywne. Zauważmy, że po zaoraniu każdej kolejnej skiby pole ma kształt prostokąta złożonego z całych kwadratów jednostkowych. Zaoranie każdej skiby to zmniejszenie tego prostokąta przez usunięcie skrajnej kolumny lub wiersza. Widać więc, że problem ma charakter rekurencyjny: zaoranie całego pola sprowadza się do właściwego wyboru pierwszej skiby i zaorania pozostałego pola w optymalny sposób.

Niemniej jednak, rozwiązywanie zadania poprzez zwykłe rekurencyjne przeszukiwanie z nawrotami (ang. *back-tracking*) możliwych sekwencji skib byłoby błędem. Złożoność czasowa takiego algorytmu jest wykładnicza — wystarczy rzut oka na ograniczenia wielkości danych, żeby przekonać się, że takie rozwiązanie nie mogłoby działać w sensownym czasie.

Rozwiązanie o złożoności czasowej $O(n^2 \cdot m^2)$

Proste spostrzeżenie pozwala znacznie ulepszyć poprzedni algorytm. Wystarczy zauważyć, że wiele różnych początkowych sekwencji skib prowadzi do takich samych pozostałych niezaoranych fragmentów pola i jest tylko $O(n^2 \cdot m^2)$ możliwych różnych niezaoranych prostokątów. Możemy więc rozwiązać zadanie stosując programowanie dynamiczne: dla każdego prostokąta zawartego w polu obliczamy minimalną liczbę skib potrzebnych do jego zaorania. Prostokąty przeglądamy w kolejności od mniejszych do większych i dla każdego rozważamy wszystkie skrajne skiby. Wybieramy taką, którą szkapa może zaorać i po zaoraniu której pozostaje prostokąt wymagający zaorania najmniejszej liczby skib.

Rozwiązanie problemu uzyskujemy po wyznaczeniu wyniku dla największego prostokąta, czyli dla całego pola.

Opiszmy przedstawiony algorytm bardziej formalnie. Przyjmijmy najpierw, że dla $i < 1$, $i > m$, $j < 1$ lub $j > n$ mamy $t_{i,j} = 0$. Oznaczmy przez $S[i, j, i', j']$ (dla $1 \leqslant i \leqslant m$, $1 \leqslant j \leqslant n$) minimalną liczbę skib potrzebnych do zaorania prostokąta, który w dolnym lewym rogu ma kwadrat o współrzędnych (i, j), a w górnym prawym rogu kwadrat o współrzędnych (i', j'). Szukana przez nas wartość to $S[1,1,m,n]$. Liczba skib potrzebna do zaorania danego prostokąta jest określona następująco. Jeśli $i > i'$ lub $j > j'$, to przyjmujemy, że $S[i, j, i', j'] = 0$, natomiast w przeciwnym przypadku:

$$S[i,j,i'j'] = 1 + \min \begin{cases} S[i+1,j,i',j'] & \text{jeśli} \quad t_{i,j} + t_{i,j+1} + \cdots + t_{i,j'} \leqslant k \\ S[i,j+1,i',j'] & \text{jeśli} \quad t_{i,j} + t_{i+1,j} + \cdots + t_{i',j} \leqslant k \\ S[i,j,i'-1,j'] & \text{jeśli} \quad t_{i',j} + t_{i',j+1} + \cdots + t_{i',j'} \leqslant k \\ S[i,j,i',j'-1] & \text{jeśli} \quad t_{i,j'} + t_{i+1,j'} + \cdots + t_{i',j'} \leqslant k \end{cases}$$

Duże znaczenie dla efektywności tego algorytmu ma sposób liczenia sum współczynników trudności dla określonych skib prostokąta. Dla uproszenia sumę taką będziemy nazywać *wagą* skiby. Jeżeli będziemy liczyć wagi wprost, sumując współczynniki trudności kolejnych kwadratów jednostkowych, to obliczenie pojedynczej wartości $S[i, j, i'j']$ będzie wymagać czasu rzędu $O(m+n)$. W ten sposób otrzymamy algorytm o łącznej złożoności czasowej rzędu $O(m^2 \cdot n^2 \cdot (m+n))$ i złożoności pamięciowej rzędu $O(m^2 \cdot n^2)$.

Wagi możemy jednak obliczać znacznie efektywniej. Wystarczy, że dla każdej pary współrzędnych $0 \leqslant i \leqslant m$, $0 \leqslant j \leqslant n$ wyznaczymy wartości:

$$W[i,j] = t_{1,j} + t_{2,j} + \cdots + t_{i,j}$$

$$K[i,j] = t_{i,1} + t_{i,2} + \cdots + t_{i,j},$$

co można zrobić w czasie rzędu $O(n \cdot m)$, przeglądając współczynniki trudności kwadratów jednostkowych w wierszach oraz w kolumnach, jak w poniższej procedurze.

```
var W,K: array [0..m,0..n] of integer;
begin
  for $i = 0$ to $m$ do begin
    $W[i,0] := 0$;
    $K[i,0] := 0$
  end;
  for $j = 0$ to $n$ do begin
    $W[0,j] := 0$;
    $K[0,j] := 0$
  end;
  for $i = 1$ to $m$ do
    for $j = 1$ to $n$ do begin
      $W[i,j] := W[i-1,j] + t_{i,j}$;
      $K[i,j] := K[i,j-1] + t_{i,j}$
    end
end
```

Wyliczone wartości W i K pozwalają nam w czasie stałym wyznaczyć wagę każdej skiby i do obliczenia tablicy S możemy zastosować następujący wzór:

$$S[i,j,i'j'] = 1 + \min \begin{cases} S[i+1,j,i',j'] & \text{jeśli} \quad K[i,j'] - K[i,j-1] \leqslant k \\ S[i,j+1,i',j'] & \text{jeśli} \quad W[i',j] - W[i-1,j] \leqslant k \\ S[i,j,i'-1,j'] & \text{jeśli} \quad K[i',j'] - K[i',j-1] \leqslant k \\ S[i,j,i',j'-1] & \text{jeśli} \quad W[i',j'] - W[i-1,j'] \leqslant k \end{cases}$$

Wykorzystanie tego wzoru w metodzie programowania dynamicznego daje w efekcie algorytm, którego pseudokod wygląda następująco:

```
var $S$ : array $[1..m, 1..n]$ of integer;
begin
  for $i = m$ downto 1 do
    for $i' = 1$ to $m$ do
      for $j = n$ downto 1 do
        for $j' = 1$ to $n$ do begin
          if $i > i'$ or $j > j'$ then $S[i,j,i',j'] := 0$
          else begin
            $S[i,j,i',j'] := \infty$;
            if $K[i,j'] - K[i,j-1] \leqslant k$ then
              $S[i,j,i',j'] := \min(S[i,j,i',j'], S[i+1,j,i',j'] + 1)$;
            if $W[i',j] - W[i-1,j] \leqslant k$ then
              $S[i,j,i',j'] := \min(S[i,j,i',j'], S[i,j+1,i',j'] + 1)$;
            if $K[i',j'] - K[i',j-1] \leqslant k$ then
              $S[i,j,i',j'] := \min(S[i,j,i',j'], S[i,j,i'-1,j'] + 1)$;
            if $W[i',j'] - W[i-1,j'] \leqslant k$ then
              $S[i,j,i',j'] := \min(S[i,j,i',j'], S[i,j,i',j'-1] + 1)$;
          end
        end
end
```

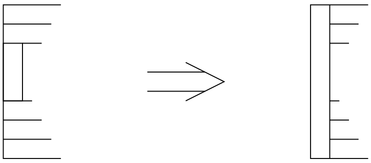
W rezultacie możemy rozwiązać zadanie w czasie $O(m^2 \cdot n^2)$, wykorzystując pamięć tego samego rzędu. Powyższy algorytm został zaimplementowany w programach `orkb1.cpp` i `orkb2.pas`. Taką samą złożoność możemy uzyskać udoskonalając przeszukiwanie z nawrotami poprzez spamiętywanie wyników podproblemów.

Rozwiązanie o złożoności czasowej $O((n+m)^3)$

Zadanie można rozwiązać jeszcze lepiej. W tym celu musimy dokładniej przeanalizować możliwe układy skib. Zauważmy, że w dowolnym zaoraniu pola mamy n skib poziomych lub m skib pionowych. Wynika to stąd, że każde odcięcie skiby zmniejsza długość jednej pary boków pola o 1, a ostatnia skiba musi mieć przynajmniej jedną parę boków długości 1. Jeśli są to boki pionowe (czyli skiba jest pozioma), to trzeba zaorać n skib poziomych, by doprowadzić do tej sytuacji — takie zaoranie nazwiemy *poziomym*. Analogicznie, jeśli ostatnia skiba jest pionowa, to również zaoranie będziemy nazywać *pionowym*. Gdy ostatnia skiba ma wymiary 1×1, zaoranie możemy nazwać zarówno poziomym, jak i pionowym. Co więcej, jeżeli pole w ogóle da się zaorać (a w zadaniu rozważamy tylko takie pola), to istnieje dla niego zarówno zaoranie poziome, jak i pionowe. Wynika to stąd, że zaoranie poziome można przekształcić w pionowe dzieląc ostatnią skibę na skiby pionowe o wymiarach 1×1. Analogicznie można uzupełnić zaoranie pionowe do poziomego.

Możemy więc poszukać oddzielnie optymalnego zaorania poziomego oraz optymalnego zaorania pionowego i wybrać ten z wyników, który zawiera mniej skib. W dalszej części skupimy się na poszukiwaniu optymalnego zaorania pionowego, czyli zawierającego m skib pionowych i minimalną liczbę skib poziomych. Dla wygody skibę, której waga nie przekracza k, a więc szkapa może ją zaorać, nazwiemy *dopuszczalną*. Dodatkowo pisząc o boku pola (lewym, prawym, górnym lub dolnym), będziemy mieli na myśli skrajną skibę przylegającą do tego boku prostokąta.

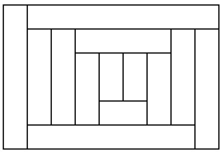
Fakt 1 *Jeżeli lewy bok pola jest skibą dopuszczalną, to istnieje optymalne zaoranie pola zaczynające się od tej skiby. Analogiczne stwierdzenie jest prawdziwe dla prawego boku pola.*



Rys. 1: Zaoranie skrajnie lewej skiby w rozwiązaniu optymalnym.

Dowód W rozwiązaniu optymalnym mamy m skib pionowych, więc istnieje wśród nich skiba pionowa zawarta w lewym boku pola. Skoro cała skrajnie lewa skiba pola jest dopuszczalna, to możemy od niej rozpocząć oranie pola. Dalszą część orki prowadzimy jak poprzednio, z tym że część skib poziomych będzie krótsza o jeden kwadrat. Ilustruje to rys. 1. Łączna liczba skib nie zwiększy się. ■

Korzystając z powyższej własności rozwiązania optymalnego możemy część skib wybierać w sposób zachłanny. Jeżeli lewy lub prawy bok pola jest skibą dopuszczalną, to orkę rozpoczynamy odpowiednio od lewego lub prawego boku. Co jednak należy zrobić, jeżeli nie możemy zaorać (nie zamęczając szkapy) ani lewego, ani prawego boku pola, ale zarówno górny, jak i dolny bok pola są skibami dopuszczalnymi? Który z nich wybrać?



Rys. 2: Rozłączne obszary górnych i dolnych skib poziomych.

W zaoraniu pionowym, skiby poziome tworzą dwa rozłączne obszary: jeden złożony z górnych, a drugi z dolnych skib — ilustruje to rys. 2. Rozważmy na chwilę decyzyjną wersję problemu z zadania: dla danych liczb g i d interesuje nas, czy można zaorać pole pionowo tak, żeby było co najwyżej g górnych skib poziomych i co najwyżej d dolnych skib poziomych. Następujący fakt pozwala znaleźć odpowiedź na tak postawione pytanie metodą zachłanną.

Fakt 2 *Jeżeli istnieje pionowe zaoranie pola złożone z g górnych skib poziomych i d dolnych skib poziomych, $g > 0$ oraz górny bok pola jest skibą dopuszczalną, to istnieje zaoranie pionowe spełniające podane ograniczenia, zaczynające się od zaorania górnej skiby. Analogiczne stwierdzenie dla dolnej skiby zachodzi, gdy $d > 0$ oraz dolny bok pola jest skibą dopuszczalną.*

Dowód Rozważmy zaoranie spełniające założenia pierwszej części faktu. Ponieważ $g > 0$, więc istnieje przynajmniej jedna skiba pozioma górna, czyli istnieje też skiba pozioma zawarta w górnym boku prostokąta. Skoro bok ten jest skibą dopuszczalną, to możemy zmodyfikować rozważane zaoranie wybierając jako pierwszą skibę górny bok. Resztę pola możemy zaorać jak poprzednio, więc liczba skib poziomych górnych i dolnych pozostanie niezmieniona, a jedynie niektóre skiby pionowe będą krótsze.

Analogiczną modyfikację można przeprowadzić w drugim przypadku, gdy $d > 0$ i dolny bok jest skibą dopuszczalną.

■

Korzystając z wykazanych własności możemy zaproponować następujący algorytm sprawdzania, czy istnieje zaoranie spełniające zadane ograniczenia g i d. Kolejne skiby do zaorania będziemy wybierać zgodnie z następującymi zasadami:

- jeżeli lewy lub prawy bok pola jest skibą dopuszczalną, to wybieramy tę skibę, a w przeciwnym przypadku
- jeżeli dolny bok pola jest skibą dopuszczalną i nie przekroczyliśmy jeszcze limitu d, to wybieramy tę skibę, w przeciwnym przypadku
- jeżeli górny bok pola jest skibą dopuszczalną i nie przekroczyliśmy limitu g, to wybieramy dolny bok, w pozostałym przypadku
- widzimy, że nie da się zaorać pola przy danych ograniczeniach g i d.

Żeby móc szybko stwierdzić, czy określony bok pola jest skibą dopuszczalną, możemy posłużyć się opisanymi wcześniej tablicami K i W. Wówczas sprawdzenie, czy istnieje sposób zaorania pola przy zadanych ograniczeniach g i d, przedstawione w poniższej procedurze *test*, wymaga czasu rzędu $O(n+m)$.

```
function test($g,d$ : integer): boolean;
var $i,i',j,j'$ : integer;
begin
  $i := 1$; $j := 1$; $i' := m$; $j' := n$;
  while $i \leqslant i'$ and $j \leqslant j'$ do begin
    { lewa skiba }
    if $K[i,j'] - K[i,j-1] \leqslant k$ then $i := i+1$ else
    { prawa skiba }
    if $K[i',j'] - K[i',j-1] \leqslant k$ then $i' := i'-1$ else
```

```
          { dolna skiba }
          if $d > 0$ and $W[i',j'] - W[i-1,j'] \leqslant k$ then begin
             $j' := j' - 1$; $d := d - 1$
          { górna skiba }
          end else if $g > 0$ and $W[i',j] - W[i-1,j] \leqslant k$ then begin
             $j := j + 1$; $g := g - 1$
          end else begin
             { porażka }
             test := false; exit
          end
       end;
       test := true
    end
```

Przypomnijmy, że naszym celem jest znalezienie takich wartości g i d, dla których zaoranie pola jest możliwe, a ich suma jest minimalna. Najprościej jest sprawdzić wszystkie możliwe pary tych liczb. Biorąc pod uwagę, że jest ich $O(n^2)$, obliczenia zajmą czas $O(n^2(n+m))$. Dodatkowo musimy jeszcze znaleźć optymalne zaoranie poziome, czyli uzyskujemy rozwiązanie o złożoności czasowej $O((m+n)^3)$. Algorytm ten został zaimplementowany w programach `orks3.cpp` i `orks4.pas`.

Rozwiązanie wzorcowe o złożoności czasowej $O((n+m)^2)$

Poszukując optymalnego zaorania pionowego nie musimy badać wszystkich par g i d! Oznaczmy przez $f(g)$ najmniejszą wartość d, dla której jest możliwe zaoranie pola przy ustalonym ograniczeniu g na liczbę skib poziomych górnych:

$$f(g) = \min\{d : \mathit{test}(g,d)\}$$

Wówczas szukany wynik to:

$$m + \min_{g \in \{1,\ldots,n-1\}} (g + f(g))$$

Zauważmy, że funkcja f jest nierosnąca. Pozwala nam to policzyć powyższy wynik zadając $O(n+m)$ zapytań o wartość $\mathit{test}(x,y)$.

```
var $wyn, g, d$ : integer;
begin
   $wyn := \infty$; $g := 0$; $d := n - 1$;
   while $g < n$ and $d \geqslant 0$ do
      if $test(g,d)$ then begin
         $wyn := \min(wyn, m + g + d)$;
         $d := d - 1$
      end else $g := g + 1$;
   return $wyn$
end
```

W podanym algorytmie każdy ze wskaźników d i g zostanie zmieniony co najwyżej n razy, więc otrzymaliśmy rozwiązanie zadania o złożoności czasowej $O((n+m)^2)$ i pamięciowej $O(n \cdot m)$.

Testy

Testy użyte do oceny rozwiązań były podzielone na 10 grup. Grupy 1–5 zawierają po jednym teście. Grupy 6–10 zawierają po dwa testy: test wydajnościowy przedstawiający pole, na którym znajduje się trudny do zaorania teren w kształcie litery „H" (lub obróconej litery „H"), oraz test losowy. Ów trudny do zaorania teren w kształcie litery „H", wraz z odpowiednio dobraną wytrzymałością szkapy, wymagał dużej liczby skib. Wielkości testów przedstawiono w poniższej tabelce:

Nazwa	m	n	k	**Opis**
ork1.in	9	10	163	prosty test poprawnościowy
ork2.in	36	32	8 723	prosty test poprawnościowy
ork3.in	50	50	2 547	prosty test poprawnościowy
ork4.in	450	400	2 100 678	test losowy
ork5.in	614	590	3 672 684	test losowy
ork6a.in	1 300	1 305	200 000	test wydajnościowy „H"
ork6b.in	700	700	35 531 618	test losowy
ork7a.in	1 220	1 190	290 000	test wydajnościowy „H"
ork7b.in	1 300	1 250	64 048 256	test losowy
ork8a.in	1 611	1 613	100 000	test wydajnościowy „H"
ork8b.in	1 599	1 601	72 709 332	test losowy
ork9a.in	2 000	2 000	270 000	test wydajnościowy „H"
ork9b.in	2 000	2 000	100 771 608	test losowy
ork10a.in	1 998	2 000	250 000	test wydajnościowy „H"
ork10b.in	1 999	2 000	100 328 366	test losowy

Zawody III stopnia

opracowania zadań

Krzysztof Sikora **Paweł Parys**
Treść zadania, Opracowanie Program

OI, Etap III, dzień próbny, 28.03.2006

Tańce w kółkach

Do pewnego przedszkola chodzi n dzieci, które codziennie ustawiają się w k kółek i tańczą. W każdym kółku tańczy co najmniej l dzieci. Dwa ustawienia dzieci uważamy za różne, jeżeli pewne dziecko w jednym ustawieniu ma innego sąsiada po swojej prawej stronie niż w drugim.

Twoim zadaniem jest obliczenie liczby wszystkich różnych ustawień modulo 2005. Jeżeli nie ma ustawień spełniających opisane warunki, poprawnym wynikiem jest 0.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia liczby n, k oraz l,*
- *obliczy liczbę $d' = d \bmod 2005$, gdzie d jest liczbą różnych ustawień dzieci ($d \bmod 2005$ oznacza resztę z dzielenia d przez 2005),*
- *wypisze d' na standardowe wyjście.*

Wejście

Pierwszy i jedyny wiersz standardowego wejścia zawiera trzy liczby całkowite oddzielone pojedynczymi odstępami: n — liczba dzieci ($3 \leqslant n \leqslant 1\,000\,000\,000$), k — liczba kółek ($1 \leqslant k \leqslant n$) oraz l — minimalna liczba dzieci w każdym kółku ($2 \leqslant l \leqslant n$).

Wyjście

Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać jedną liczbę: $d \bmod 2005$.

Przykład

Dla danych wejściowych:

```
7 2 3
```

poprawnym wynikiem jest:

```
420
```

Rozwiązanie

Rozwiązanie podstawowe

Niech $S(n,k,l)$ oznacza liczbę ustawień n dzieci w k kółkach, w każdym przynajmniej l dzieci. Dla $n \geqslant k \cdot l$ oraz $k > 0$ mamy własność:

$$S(n,k,l) = (n-1) \cdot S(n-1,k,l) + (n-1) \cdots (n-l+1) \cdot S(n-l,k-1,l) \tag{1}$$

Można ją wytłumaczyć w następujący kombinatoryczny sposób. Spójrzmy na pewne ustalone dziecko d. Wtedy wszystkie ustawienia dzieci możemy podzielić na dwie rozłączne grupy, zależnie od rozmiaru kółka, w którym stoi dziecko d:

- ustawienia, w których dziecko d znajduje się w kółku o rozmiarze większym niż l. Pozostałe $n-1$ dzieci można ustawić w kółkach na $S(n-1,k,l)$ sposobów. W każdym takim układzie możemy umieścić dziecko d na prawo od dowolnego z pozostałych $n-1$ dzieci, uzyskując ustawienie, w którym dziecko d występuje w kółku o rozmiarze przynajmniej $l+1$. To daje nam $(n-1) \cdot S(n-1,k,l)$ układów — pierwszy składnik sumy,

- ustawienia, w których dziecko d znajduje się w kółku o rozmiarze l. Pozostałe $l-1$ dzieci do tego kółka możemy wybrać na $\binom{n-1}{l-1}$ sposobów i ustawić je w kółko na $(l-1)!$ sposobów. Pozostałe $n-l$ dzieci mogą być ustawione na $S(n-l,k-1,l)$ sposobów. To daje nam $(n-1) \cdots (n-l+1) \cdot S(n-l,k-1,l)$ układów — drugi składnik sumy.

Dla $n < k \cdot l$ oraz dla $k = 0$ i $n > 0$ mamy $S(n,k,l) = 0$. Ponadto $S(0,0,l) = 1$. Wykorzystując równanie (1) i programowanie dynamiczne możemy łatwo rozwiązać zadanie. Na początku obliczamy iloczyny $(i-1) \cdots (i-l+1)$ dla każdego $i \leqslant n$. Następnie liczymy $S(i,j,l)$ dla wszystkich $i \leqslant n$, $j \leqslant k$, przy czym podczas wyznaczania wartości funkcji S dla i i j wykorzystujemy wcześniej policzone wartości funkcji S dla mniejszych argumentów. Wszystkie działania wykonujemy modulo 2005 stosując dobrze znane własności kongruencji stanowiące, iż jeśli

$$a \equiv b \pmod{p} \quad \text{oraz} \quad c \equiv d \pmod{p}$$

to

$$a + c \equiv b + d \pmod{p} \quad \text{oraz} \quad a \cdot c \equiv b \cdot d \pmod{p}.$$

Przedstawiony algorytm ma złożoność czasową $O(n \cdot (k+l))$, a pamięciową $O(n)$. Nieźle, jednak wystarczy to do zaliczenia tylko około połowy testów. Dla rozwiązania problemu dla większych n potrzebny jest szybszy program.

Rozwiązanie wzorcowe

Rozwiązanie podstawowe można przyspieszyć spostrzegając, że wśród wartości funkcji S modulo 2005 występuje wiele zer. Aby je skutecznie wyznaczyć wprowadzimy dodatkowe funkcje.

Alternatywne sformułowanie S

Dla nieujemnych liczb całkowitych $c_l, c_{l+1}, \ldots, c_n$ niech

$$A(c_l, c_{l+1}, \ldots, c_n) = c_l! \cdot l^{c_l} \cdot c_{l+1}! \cdot (l+1)^{c_{l+1}} \cdot \ldots \cdot c_n! \cdot n^{c_n}$$

oraz niech

$$B(c_l, c_{l+1}, \ldots, c_n) = \frac{n!}{A(c_l, c_{l+1}, \ldots, c_n)}$$

Wówczas zachodzi następująca równość:

$$S(n,k,l) = \sum B(c_l, c_{l+1}, \ldots, c_n), \tag{2}$$

gdzie suma przebiega po wszystkich ciągach $c_l, c_{l+1}, \ldots, c_n$ takich, że $\sum_{i=l}^{n} c_i = k$ oraz $\sum_{i=l}^{n} i \cdot c_i = n$.

Aby uzasadnić poprawność powyższej zależności przyjrzyjmy się jej bliżej. Zauważmy, że warunki nałożone na wartości $c_l, c_{l+1}, \ldots, c_n$ pozwalają nam interpretować c_i jako liczbę kółek rozmiaru i. Tak więc każdy ciąg $c_l, c_{l+1}, \ldots, c_n$ odpowiada konkretnej konfiguracji rozmiarów kółek, stąd odpowiadający mu składnik sumy (2) powinien oznaczać liczbę możliwych ustawień dzieci w tej konfiguracji. Upewnijmy się, że właściwie wyznaczyliśmy tę liczbę — obliczmy, na ile sposobów można rozmieścić dzieci w kółkach o podanych rozmiarach. Rozważmy następujący sposób formowania kółek: ustawmy wszystkie dzieci w szeregu (możemy to zrobić na $n!$ sposobów), a następnie podzielmy je kolejno na grupy: c_l grup po l dzieci, c_{l+1} grup po $l+1$ dzieci itd. — na koniec z każdej grupy utwórzmy kółko. Oczywiście przy takim podejściu każdą konfigurację tańca w kółkach o zadanych rozmiarach $c_l, c_{l+1}, \ldots, c_n$ możemy otrzymać wiele razy. Jak wiele? Po pierwsze w każdym kółku nie ma znaczenia, które dziecko jest pierwsze, więc każde ustawienie dzieci w kółkach liczymy $l^{c_l} \cdot (l+1)^{c_{l+1}} \cdot \ldots \cdot n^{c_n}$ razy. Dodatkowo także kolejność kółek tego samego rozmiaru nie ma znaczenia, więc musimy domnożyć tę liczbę przez $c_l! \cdot c_{l+1}! \cdot \ldots \cdot c_n!$. Ostatecznie otrzymujemy, że każde ustawienie powtarza się $A(c_l, c_{l+1}, \ldots, c_n)$ razy wśród wszystkich $n!$ szeregów, tak więc mamy $B(c_l, c_{l+1}, \ldots, c_n)$ różnych ustawień dla danych rozmiarów kółek.

Niezerowe konfiguracje

Postaramy się teraz odpowiedzieć na pytanie, kiedy dla liczby pierwszej p zachodzi nierówność $B(c_l, \ldots, c_n) \not\equiv 0 \pmod{p}$? Niech $X(m,p)$ będzie największym i takim, że $p^i | m$. Zauważmy, że $B(c_l, \ldots, c_n) \not\equiv 0 \pmod{p}$ tylko wtedy, gdy wszystkie potęgi p z licznika i z mianownika skracają się, a tak dzieje się tylko wówczas, gdy $X(A(c_l, \ldots, c_n), p) = X(n!, p)$. Wiemy, że

$$X(n!,p) = \left\lfloor \frac{n}{p} \right\rfloor + \left\lfloor \frac{n}{p^2} \right\rfloor + \left\lfloor \frac{n}{p^3} \right\rfloor + \cdots$$

Spróbujmy także oszacować $X(A(c_l, \ldots, c_n), p)$. W tym celu zauważmy, że:

$$\begin{aligned}
X(A(c_l, \ldots, c_n), p) &= X(c_l! \cdot l^{c_l} \cdot c_{l+1}! \cdot (l+1)^{c_{l+1}} \cdot \ldots \cdot c_n! \cdot n^{c_n}, p) \\
&= X(c_l! \cdot l^{c_l} \cdot c_{l+1}! \cdot (l+1)^{c_{l+1}} \cdot \ldots \cdot c_{p-1}! \cdot (p-1)^{c_{p-1}}, p) \\
&+ X(c_p! \cdot p^{c_p}, p) + \cdots + X(c_n! \cdot n^{c_n}, p)
\end{aligned}$$

Rozważymy odrębnie poszczególne składniki tej sumy.

Przypadek 1. Dla kółek rozmiaru $i = p$ mamy:

$$\begin{aligned} X(c_p! \cdot p^{c_p}, p) &= X(c_p!, p) + X(p^{c_p}, p) \\ &= c_p + \left\lfloor \frac{c_p}{p} \right\rfloor + \left\lfloor \frac{c_p}{p^2} \right\rfloor + \cdots \\ &= \left\lfloor \frac{c_p p}{p} \right\rfloor + \left\lfloor \frac{c_p p}{p^2} \right\rfloor + \left\lfloor \frac{c_p p}{p^3} \right\rfloor + \cdots \\ &= X((c_p p)!, p). \end{aligned} \tag{3}$$

Przypadek 2. Teraz przyjrzyjmy się kółkom o rozmiarze $i > p$. Jeśli $p \geqslant 3, j \geqslant 2$, to zachodzi nierówność $1 + j \leqslant p^{j-1}$. To daje nam $(1 + X(i,p)) \cdot p \leqslant i$ dla wszystkich $p \geqslant 3, i > p$ (dla $j = X(i,p) \geqslant 2$ działa podana nierówność, a pozostałe przypadki łatwo sprawdzić). Otrzymujemy wtedy:

$$\begin{aligned} X(c_i! \cdot i^{c_i}, p) &\leqslant c_i + X(c_i! \cdot i^{c_i}, p) \\ &= c_i + X(i,p) \cdot c_i + \left\lfloor \frac{c_i}{p} \right\rfloor + \left\lfloor \frac{c_i}{p^2} \right\rfloor + \cdots \\ &= \left\lfloor \frac{(1 + X(i,p))c_i \cdot p}{p} \right\rfloor + \left\lfloor \frac{c_i \cdot p}{p^2} \right\rfloor + \left\lfloor \frac{c_i \cdot p}{p^3} \right\rfloor + \cdots \\ &\leqslant \left\lfloor \frac{c_i i}{p} \right\rfloor + \left\lfloor \frac{c_i i}{p^2} \right\rfloor + \left\lfloor \frac{c_i i}{p^3} \right\rfloor + \cdots \\ &= X((c_i i)!, p). \end{aligned} \tag{4}$$

Przy czym równość może zachodzić jedynie wtedy, gdy $c_i = 0$.

Przypadek 3. Rozważmy łącznie pozostałe kółka rozmiaru mniejszego niż p. Załóżmy, że takie kółka mogą istnieć, czyli $l < p$. Niech $m = c_l l + c_{l+1}(l+1) + \ldots + c_{p-1}(p-1)$. Ponieważ $l \geqslant 2$, to zachodzi nierówność $2 \cdot (c_l + \cdots + c_{p-1}) \leqslant m$. Wstępnie zauważmy, że

$$X(c_l! \cdot l^{c_l} \cdot \ldots \cdot c_{p-1}! \cdot (p-1)^{c_{p-1}}, p) = X(c_l! \cdot \ldots \cdot c_{p-1}!, p).$$

Dalej możemy przeprowadzić następujące szacowanie:

$$\begin{aligned} & X(c_l! \cdots c_{p-1}!, p) \leqslant \left\lfloor \frac{m}{p} \right\rfloor - \left\lfloor \frac{m}{2p} \right\rfloor + X(c_l! \cdots c_{p-1}!, p) \\ = & \left\lfloor \frac{m}{p} \right\rfloor - \left\lfloor \frac{m}{2p} \right\rfloor + \left\lfloor \frac{c_l}{p} \right\rfloor + \cdots + \left\lfloor \frac{c_{p-1}}{p} \right\rfloor + \left\lfloor \frac{c_l}{p^2} \right\rfloor + \cdots + \left\lfloor \frac{c_{p-1}}{p^2} \right\rfloor + \cdots \\ \leqslant & \left\lfloor \frac{m}{p} \right\rfloor - \left\lfloor \frac{m}{2p} \right\rfloor + \left\lfloor \frac{c_l + \cdots + c_{p-1}}{p} \right\rfloor + \left\lfloor \frac{c_l + \cdots + c_{p-1}}{p^2} \right\rfloor + \cdots \\ \leqslant & \left\lfloor \frac{m}{p} \right\rfloor - \left\lfloor \frac{m}{2p} \right\rfloor + \left\lfloor \frac{m}{2p} \right\rfloor + \left\lfloor \frac{m}{p^2} \right\rfloor + \left\lfloor \frac{m}{p^3} \right\rfloor + \cdots \\ = & X(m!, p). \end{aligned} \tag{5}$$

Zauważmy, że dla $m < p$ zachodzi $\left\lfloor \frac{m}{p} \right\rfloor = \left\lfloor \frac{m}{2p} \right\rfloor = 0$, a w przeciwnym przypadku mamy $\left\lfloor \frac{m}{p} \right\rfloor > \left\lfloor \frac{m}{2p} \right\rfloor$ i wtedy na pewno w powyższym ciągu porównań nie zachodzi równość.

Podsumowując wszystkie trzy przypadki dostajemy oszacowanie:

$$\begin{aligned} X(A(c_l,\dots,c_n),p) &\leqslant X(m!,p)+X((c_p\cdot p)!,p)+\dots+X((c_n\cdot n)!,p) \\ &= X(m!\cdot(c_p\cdot p)!\cdot\ldots\cdot(c_n\cdot n)!,p) \end{aligned}$$

Dodatkowo łatwo zauważyć, że prawdziwa jest następująca prosta własność $X(n_1!\cdots n_s!,p)\leqslant X((n_1+\cdots+n_s)!,p)$, więc mamy

$$\begin{aligned} X(A(c_l,\dots,c_n),p) &\leqslant X((m+c_p\cdot p+\dots+c_n\cdot n)!,p) \\ &= X((c_l\cdot l+\dots+c_n\cdot n)!,p) \\ &= X(n!,p) \end{aligned}$$

Co więcej, z przeprowadzonych obliczeń wiemy, że powyższa nierówność może być równością tylko wtedy, gdy $m<p$ oraz $c_i=0$ dla $i>p$. Otrzymujemy stąd następujący wniosek:

Wniosek 1 (O niezerowości) *Niech m będzie liczbą dzieci w kółkach mniejszych niż p, to znaczy $m=c_l l+c_{l+1}(l+1)+\cdots+c_{p-1}(p-1)$. Jeśli $B(c_l,\dots,c_n)\not\equiv 0 \pmod p$, to $m<p$ oraz $c_i=0$ dla każdego $i>p$.*

Uproszczona rekurencja

Z wniosku 1 wynika, że poszukując niezerowych wartości $B(c_l,\dots,c_n)\not\equiv 0 \pmod p$ wystarczy rozważyć konfiguracje, w których nie ma kółek rozmiaru większego niż p i mniej niż p dzieci łącznie stoi w kółkach rozmiaru mniejszego niż p. To oznacza, że musimy sformować $c_p=\lfloor n/p\rfloor$ kółek rozmiaru p, a pozostałe $n \bmod p$ dzieci rozmieścić w kółkach rozmiaru mniejszego niż p. Widzimy więc, że zachodzi następujący wzór:

$$\begin{aligned} S(n,k,l) &\equiv \binom{n}{c_p\cdot p}\cdot\frac{(c_p\cdot p)!}{c_p!\cdot p^{c_p}}\cdot S(n \bmod p,k-c_p,l) \pmod p \\ &\equiv \frac{n!}{c_p!\cdot p^{c_p}\cdot(n \bmod p)!}\cdot S(n \bmod p,k-c_p,l) \pmod p, \end{aligned}$$

Pierwszy czynnik drugiego wiersza możemy jeszcze bardziej uprościć. Powołujemy się przy tym na następujące ogólnie znane twierdzenie:

Twierdzenie 2 (Wilson) *Dla dowolnej liczby pierwszej p prawdą jest, że*

$$(p-1)!\equiv -1 \pmod p$$

To oznacza, że dla każdego j mamy $\prod_{i=1}^{p-1}(j\cdot p+i)\equiv -1 \pmod p$. Stąd odpowiednio grupując czynniki $n!$ otrzymujemy:

$$n!=\underbrace{\left(\prod_{k=0}^{c_p-1}\prod_{i=1}^{p-1}(k\cdot p+i)\right)}_{(*)}\cdot\underbrace{p^{c_p}\cdot c_p!}_{(**)}\cdot\underbrace{\left(\prod_{k=1}^{n \bmod p}(c_p\cdot p+k)\right)}_{(***)}.$$

Zauważmy, że:

(*) równa się $(-1)^{c_p}$ z twierdzenia Wilsona,

(**) skraca się z mianownikiem,

(***) podzielona przez $(n \bmod p)!$ równa się 1, co wynika stąd, iż:

$$(c_p \cdot p + 1) \cdots (c_p \cdot p + n \bmod p) \equiv (n \bmod p)! \pmod{p} \tag{6}$$

i dalej, korzystając z faktu, że gdy

$$da \equiv db \pmod{c} \quad \text{oraz} \quad \mathrm{NWD}(d,c) = 1$$

to

$$a \equiv b \pmod{c},$$

po podzieleniu kongruencji (6) stronami przez $(n \bmod p)!$ otrzymujemy:

$$\frac{(c_p \cdot p + 1) \cdot (c_p \cdot p + 2) \cdots (c_p \cdot p + n \bmod p)}{(n \bmod p)!} \equiv 1 \pmod{p}.$$

Stąd ostatecznie otrzymujemy

$$\frac{n!}{c_p! \cdot p^{c_p} \cdot (n \bmod p)!} \equiv (-1)^{c_p} \pmod{p}$$

i możemy sformułować:

Wniosek 3 (Uproszczona rekurencja)

$$S(n,k,l) \equiv (-1)^{\lfloor n/k \rfloor} \cdot S(n \bmod p, k - \lfloor n/k \rfloor, l) \pmod{p}$$

Algorytm

Wykorzystując uproszczoną rekurencję możemy rozwiązać zadanie dużo szybciej niż poprzednimi metodami. Dostajemy algorytm działający w czasie $O(p^2)$, gdzie p jest liczbą pierwszą, modulo która chcemy otrzymać wynik. Co prawda 2005 nie jest liczbą pierwszą, gdyż rozkłada się na czynniki $2005 = 5 \cdot 401$. Zauważmy jednak, że 5 i 401 to liczby pierwsze i możemy obliczyć liczbę różnych ustawień dzieci modulo te liczby — oznaczmy je odpowiednio przez a i b. Chińskie twierdzenie o resztach mówi, że w zbiorze $\{0, 1, \ldots, 2004\}$ jest dokładnie jedna liczba c, taka że $c \equiv a \pmod{5}$ i $c \equiv b \pmod{401}$. Najprostszy sposób, aby ją znaleźć, to sprawdzić wszystkich kandydatów od 0 do 2004.

Testy

Zadanie testowane było na zestawie 20 danych testowych pogrupowanych w pary.

Nazwa	**n**	**k**	**l**	**Opis**
tan1a.in	82	3	18	test poprawnościowy
tan1b.in	1000	1000	1000	test poprawnościowy
tan2a.in	400	200	2	test poprawnościowy
tan2b.in	1 000	22	10	test poprawnościowy
tan3a.in	976	6	43	test poprawnościowy
tan3b.in	988	3	200	test poprawnościowy
tan4a.in	4	1	3	test poprawnościowy
tan4b.in	800	21	21	test poprawnościowy
tan5a.in	167	5	18	test poprawnościowy
tan5b.in	477	18	5	test poprawnościowy
tan6a.in	5 555	1 111	2	mały test wydajnościowy
tan6b.in	197 512	144	3	mały test wydajnościowy
tan7a.in	50 000	130	9	mały test wydajnościowy
tan7b.in	20 000	2 000	4	średni test wydajnościowy
tan8a.in	1 234 581	3 079	116	duży test wydajnościowy
tan8b.in	1 000 000 000	500 111 000	102	duży test wydajnościowy
tan9a.in	999 999 492	2 493 765	105	duży test wydajnościowy
tan9b.in	1 000 000 000	10 000	8	duży test wydajnościowy
tan10a.in	1 000 000 000	200 000 000	4	duży test wydajnościowy
tan10b.in	999 999 999	5 988 023	76	duży test wydajnościowy

Bartosz Walczak
Treść zadania, Opracowanie

Marcin Michalski
Program

OI, Etap III, dzień pierwszy, 29.03.2006

Estetyczny tekst

Rozważmy dowolny tekst złożony z n *słów ponumerowanych od* 1 *do* n*. Dowolny podział tego tekstu na* k *wierszy reprezentujemy za pomocą takiego ciągu liczb* $(a_1, a_2, \ldots, a_{k-1})$*, że słowa o numerach od* 1 *do* a_1 *znajdują się w pierwszym wierszu, słowa o numerach od* $a_1 + 1$ *do* a_2 *znajdują się w drugim wierszu itd., a słowa o numerach od* $a_{k-1} + 1$ *do* n *znajdują się w ostatnim,* k*-tym wierszu.*

Każde słowo ma określoną długość (wyrażoną liczbą znaków). Długość słowa o numerze x *oznaczamy przez* $\mathit{length}(x)$*. Ponadto każde dwa sąsiednie słowa w wierszu są oddzielone odstępem szerokości jednego znaku. Długością wiersza nazywamy sumę długości wszystkich słów w tym wierszu powiększoną o liczbę odstępów między nimi. Długość wiersza o numerze* w *oznaczamy przez* $\mathit{line}(w)$*. Oznacza to, że jeżeli w wierszu o numerze* w *znajdują się słowa o numerach od* i *do* j *włącznie, to długość tego wiersza wynosi:*

$$\mathit{line}(w) = \mathit{length}(i) + \mathit{length}(i+1) + \ldots + \mathit{length}(j) + (j - i)$$

Dla przykładu, rozważmy tekst złożony z 4 słów o długościach kolejno 4, 3, 2 i 5 oraz jego podział $(1,3)$ *na 3 wiersze. Wówczas długość pierwszego wiersza wynosi 4, drugiego — 6, a trzeciego — 5:*

```
XXXX    (1. wiersz)
XXX XX  (2. wiersz)
XXXXX   (3. wiersz)
```

Współczynnikiem estetyczności podziału danego tekstu na k *wierszy nazywamy liczbę wyrażoną wzorem:*

$$|\mathit{line}(1) - \mathit{line}(2)| + |\mathit{line}(2) - \mathit{line}(3)| + \ldots + |\mathit{line}(k-1) - \mathit{line}(k)|$$

W szczególności, jeżeli podział zajmuje tylko jeden wiersz, jego współczynnik estetyczności jest równy 0.

Im mniejszy jest współczynnik estetyczności, tym bardziej estetyczny jest dany podział. Rozpatrujemy tylko takie podziały, w których długość żadnego wiersza nie przekracza pewnej stałej liczby m*. Spośród wszystkich takich podziałów danego tekstu na dowolną liczbę wierszy poszukujemy podziału najbardziej estetycznego, czyli o minimalnym współczynniku estetyczności. W podanym powyżej przykładzie współczynnik estetyczności podziału jest równy 3 i jest to minimalna wartość współczynnika estetyczności dla* $m = 6$ *lub* $m = 7$*.*

Zadanie

Napisz program, który:

- *Wczyta ze standardowego wejścia liczby* m *i* n *oraz długości kolejnych słów.*
- *Wyznaczy minimalny współczynnik estetyczności dla tych podziałów, w których długość żadnego wiersza nie przekracza* m*.*
- *Wypisze wynik na standardowe wyjście.*

Wejście

Pierwszy wiersz standardowego wejścia zawiera liczby całkowite m i n, $1 \leqslant m \leqslant 1\,000\,000$, $1 \leqslant n \leqslant 2\,000$, oddzielone pojedynczym odstępem. Drugi i ostatni wiersz wejścia zawiera n liczb całkowitych będących długościami kolejnych słów, $1 \leqslant \mathsf{length}(i) \leqslant m$ dla $i = 1, 2, \ldots, n$, pooddzielanych pojedynczymi odstępami.

Wyjście

Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą: minimalny współczynnik estetyczności dla tych podziałów, w których długość żadnego wiersza nie przekracza m.

Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
3
```

Dla danych wejściowych:

```
4 2
1 2
```

poprawnym wynikiem jest:

```
0
```

Rozwiązanie

Rozpocznijmy od pewnych uproszeń zadania, które ułatwią prezentację rozwiązania. Po pierwsze zauważmy, że możemy zapomnieć o odstępach pomiędzy wyrazami w wierszu, jeśli zwiększymy o 1 długość każdego słowa oraz ograniczenie długości wiersza m. Wówczas długość wiersza możemy liczyć po prostu jako sumę długości występujących w nim wyrazów.

Wprowadźmy także następujące terminy: *minimalnym podziałem* danego tekstu będziemy nazywali minimalny współczynnik estetyczności podziału tego tekstu na wiersze, przy czym minimum bierzemy albo po wszystkich podziałach, albo po pewnych szczególnych podziałach (będzie to wynikało z kontekstu). Ponadto *prefiksem tekstu* będziemy nazywać tekst złożony z początkowych wyrazów tekstu, czyli ze słów o numerach $1, \ldots, k$ dla pewnego $1 \leqslant k \leqslant n$.

Proste rozwiązanie dynamiczne

Możemy teraz przystąpić do poszukiwania efektywnego rozwiązania problemu. Naturalną metodą, która przychodzi na myśl w przypadku zadań optymalizacyjnych podobnych do naszego, jest programowanie dynamiczne. Postępując zgodnie z tą techniką postaramy się wyznaczyć minimalne podziały wszystkich prefiksów danego tekstu, otrzymując ostatecznie minimalny podział całości.

Jednak aby wyznaczyć minimalny podział określonego prefiksu, nie wystarczy obliczyć wcześniej jedynie minimalne podziały dla wszystkich krótszych prefiksów. Dla każdego z nich musimy także uwzględnić różne możliwe długości ostatniego wiersza, by w zależności od tej wartości ustalić współczynnik estetyczności po dołożeniu kolejnego wiersza.

Niech więc $suma_x^y$ oznacza sumę długości słów o numerach $x,\dots,y$, a $wynik_x^y$ niech oznacza minimalny podział prefiksu złożonego ze słów $1,\dots,y$, w którym ostatni wiersz składa się ze słów o numerach $x,\dots,y$. Jeżeli $suma_x^y \leqslant m$, to wartość $wynik_x^y$ możemy obliczyć następująco:

$$wynik_x^y = \begin{cases} 0 & \text{dla } y \geqslant x = 1, \\ \min\{wynik_z^{x-1} + |suma_x^y - suma_z^{x-1}| \ :\ 1 \leqslant z < x\} & \text{dla } y \geqslant x \geqslant 2 \end{cases} \tag{1}$$

W przypadku, gdy $suma_x^y > m$, przyjmujemy $wynik_x^y = \infty$.
Wszystkie wartości $suma_x^y$ możemy obliczyć w czasie $\Theta(n^2)$:

```
for $x := 1$ to $n$ do
begin
  $suma_x^x := length(x)$;
  for $y := x+1$ to $n$ do
    $suma_x^y := suma_x^{y-1} + length(y)$;
end
```

Następnie korzystając ze wzoru (1) możemy zapisać prosty algorytm, pozwalający wyznaczyć końcowy wynik $\min\{wynik_x^n \ :\ 1 \leqslant x \leqslant n\}$ w czasie $\Theta(n^3)$. Jednakże przy podanym w treści zadania ograniczeniu na n taka złożoność nie jest satysfakcjonująca.

Przyspieszenie obliczeń

Okazuje się, że algorytm można przyspieszyć, jeśli zmienimy nieco sposób obliczania wartości *wynik*. W tym celu przekształcimy wzór (1) wyodrębniając przypadki, gdy ostatni wiersz jest krótszy od poprzedniego:

$$\begin{aligned} wynik_x^y = \min\{&\min\{wynik_z^{x-1} + suma_z^{x-1} - suma_x^y \ :\ suma_z^{x-1} > suma_x^y\}, \\ &\min\{wynik_z^{x-1} - suma_z^{x-1} + suma_x^y \ :\ suma_z^{x-1} \leqslant suma_x^y\}\} = \\ = \min\{&\min\{wynik_z^{x-1} + suma_z^{x-1} \ :\ suma_z^{x-1} > suma_x^y\} - suma_x^y, \\ &\min\{wynik_z^{x-1} - suma_z^{x-1} \ :\ suma_z^{x-1} \leqslant suma_x^y\} + suma_x^y\} \end{aligned}$$

Następnie rozważmy ustalone wartości x i y. Niech q będzie maksymalną liczbą w przedziale $1 \leqslant q < x$, dla której $suma_q^{x-1} > suma_x^y$. Jeżeli takie q nie istnieje, to przyjmujemy $q = 0$.

Wówczas nierówność $suma_z^{x-1} > suma_x^y$ zachodzi wtedy i tylko wtedy, gdy $z \leqslant q$, jako że wartość $suma_a^b$ maleje wraz ze wzrostem wartości a. Ponadto wprowadźmy oznaczenia:

$$\begin{aligned} mniejsze_x^y &= \min\{wynik_a^y + suma_a^y : a \leqslant x\} \\ wieksze_x^y &= \min\{wynik_a^y - suma_a^y : a \geqslant x\} \end{aligned}$$

Przyjmijmy przy tym $mniejsze_0^y = \infty$ oraz $wieksze_{y+1}^y = \infty$. Z poczynionych obserwacji wynika, że:

$$\min\{wynik_z^{x-1} + suma_z^{x-1} : suma_z^{x-1} > suma_x^y\} = \min\{wynik_z^{x-1} + suma_z^{x-1} : z \leqslant q\} = mniejsze_q^{x-1}$$
$$\min\{wynik_z^{x-1} - suma_z^{x-1} : suma_z^{x-1} \leqslant suma_x^y\} = \min\{wynik_z^{x-1} - suma_z^{x-1} : z > q\} = wieksze_{q+1}^{x-1}$$

Stąd ostatecznie wzór (1) sprowadza się do następującej postaci:

$$wynik_x^y{=}\min\{mniejsze_q^{x-1} - suma_x^y, wieksze_{q+1}^{x-1} + suma_x^y\} \tag{2}$$

Możemy więc zapisać schemat rozwiązania wzorcowego:

```
1: for $y := 1$ to $n$ do
2: begin
3:    for $x := 1$ to $y$ do oblicz $wynik_x^y$
4:    for $x := 1$ to $y$ do oblicz $mniejsze_x^y$
5:    for $x := y$ downto 1 do oblicz $wieksze_x^y$
6: end
```

Jeżeli dla danych x i y znamy wartość q, to dzięki formule (2) możemy obliczyć $wynik_x^y$ w linii 3 w czasie stałym dla każdego x. Wartości $mniejsze_x^y$ i $wieksze_x^y$ dla każdego x również łatwo obliczyć w czasie stałym — wystarczy wykorzystać $mniejsze_{x-1}^y$ do obliczenia $mniejsze_x^y$ oraz $wieksze_{x+1}^y$ do obliczenia $wieksze_x^y$ (stąd odwrócona kolejność pętli w linii 5):

$$mniejsze_x^y = \min(mniejsze_{x-1}^y, wynik_x^y + suma_x^y)$$

$$wieksze_x^y = \min(wieksze_{x+1}^y, wynik_x^y - suma_x^y)$$

Pozostaje zastanowić się, w jaki sposób wyznaczyć indeks q. Można zastosować wyszukiwanie binarne wykorzystując przy tym monotoniczność funkcji *suma*. Wówczas wyznaczenie każdej wartości q, a tym samym obliczenie wartości $wynik_x^y$, wymaga czasu $\Theta(\log n)$, więc cały algorytm działa w złożoności czasowej $\Theta(n^2 \log n)$. Można jednak postąpić sprytniej. Zauważmy, że jeżeli przy ustalonym y zwiększamy x (tak jak w linii 3), to zmniejsza się $suma_x^y$, więc wartość q może się tylko zwiększyć lub pozostać niezmieniona. Tak więc wykonując obliczenia w linii 3, możemy dla kolejnych wartości x poszukiwać wartości q, począwszy od indeksu q znalezionego w poprzedniej iteracji pętli. W ten sposób wyznaczymy wszystkie wartości q dla danego y w łącznym czasie liniowym. Zatem każdy krok głównej pętli wykonamy w czasie liniowym, czyli otrzymamy algorytm o złożoności $\Theta(n^2)$.

Jako podsumowanie opisu algorytmu zamieszczamy jego pseudokod:

```
for $y := 0$ to $n$ do
   $mniejsze^y_0 := \infty$;
for $y := 1$ to $n$ do
   $wieksze^y_{y+1} := \infty$;
for $y := 1$ to $n$ do
begin
   { tablica $wynik$ }
   $q := 0$;
   for $x := 1$ to $y$ do
   begin
      if $suma^y_x > m$ then
      begin
         $wynik^y_x := \infty$;
         continue;
      end
      if $x = 1$ then
      begin
         $wynik^y_x := 0$;
         continue;
      end
      while $(q + 1 \leqslant x - 1)$ and $(suma^{x-1}_{q+1} > suma^y_x)$ do
         $q := q + 1$;
      $wynik^y_x := \min(mniejsze^{x-1}_q - suma^y_x, wieksze^{x-1}_{q+1} + suma^y_x)$;
   end
   { tablica $mniejsze$ }
   for $x := 1$ to $y$ do
      $mniejsze^y_x := \min(mniejsze^y_{x-1}, wynik^y_x + suma^y_x)$;
   { tablica $wieksze$ }
   for $x := y$ downto 1 do
      $wieksze^y_x := \min(wieksze^y_{x+1}, wynik^y_x - suma^y_x)$;
end
```

Implementacja rozwiązania wzorcowego znajduje się w plikach `est.cpp` i `est0.pas` (rozwiązanie działające w czasie $\Theta(n^2)$) oraz w `ests1.cpp` i `ests4.pas` (rozwiązanie działające w czasie $\Theta(n^2 \log n)$).

Testy

Poniżej znajduje się opis zestawu danych testowych, na których były oceniane rozwiązania zawodników. Pary testów 8a i 8b, 9a i 9b, 10a i 10b oraz 11a i 11b zostały pogrupowane. Rozwiązania o złożoności $\Theta(n^2)$ lub $\Theta(n^2 \log n)$ przechodziły wszystkie testy. Rozwiązania o złożoności $\Theta(n^3)$ nie przechodziły grup testów 5, 6, 8, 9, 10, 11. Niepoprawne rozwiązania, które polegały na zachłannym umieszczaniu w wierszach maksymalnej liczby słów, nie przechodziły prawie żadnego z testów.

Nazwa	**m**	**n**	**Opis**
est1.in	10	4	mały, poprawnościowy
est2.in	11	8	mały, poprawnościowy
est3.in	99	3	mały, poprawnościowy
est4.in	1 000	80	losowy
est5.in	3 000	1 000	zawierający dużo jednoliterowych słów i na końcu kilka długich losowych
est6.in	650	1 303	zawierający dużo jednoliterowych słów i na końcu 3 dłuższe
est7.in	1 000	1 400	losowy; zawierający na początku na zmianę jedno- i 999-literowe słowa, których nie można połączyć w wiersze
est8a.in	100	2 000	losowy; zawierający na zmianę krótkie i długie słowa — niektóre pary można połączyć w wiersze, inne nie
est8b.in	100 000	1 754	zawierający na początku i na końcu długie słowo, a w środku dużo krótkich losowych
est9a.in	1 000 000	2 000	losowy; zawierający krótkie słowa, które się mieszczą w jednym wierszu
est9b.in	1 000	2 000	jak wyżej, ale słowa się nie mieszczą w jednym wierszu
est10a.in	1 000 000	2 000	losowy
est10b.in	999 752	2 000	losowy; zawierający w środku 500 długich słów, a na początku i na końcu po 750 krótkich
est11a.in	13	7	mały, poprawnościowy
est11b.in	999 999	2 000	zawierający wiele krótkich słów poprzedzielanych długimi, odpowiedź 0

Jakub Pawlewicz
Treść zadania, Opracowanie

Jakub Radoszewski
Program

OI, Etap III, dzień pierwszy, 29.03.2006

Kryształy

Bajtocy opracował proces tworzenia kryształów złożonych z określonej liczby atomów różnych pierwiastków. Po latach eksperymentów udało mu się znaleźć formułę, która opisuje, dla jakiej liczby atomów poszczególnych pierwiastków można wytworzyć kryształ składający się z tych atomów. Ciekawi go jak wiele różnych kryształów może otrzymać.

Niech x i y oznaczają nieujemne liczby całkowite. Przez $x \oplus y$ oznaczmy wynik wykonania operacji xor na odpowiadających sobie bitach liczb x i y. Wyniki działania operacji xor na bitach to: $1 \oplus 1 = 0 \oplus 0 = 0$, $1 \oplus 0 = 0 \oplus 1 = 1$.

Na przykład, $44 \oplus 6 = (101100)_2 \oplus (000110)_2 = (101010)_2 = 42$.

Bajtocy zna n różnych pierwiastków, ponumerowanych od 1 do n. Dla każdego pierwiastka i istnieje górne ograniczenie m_i na liczbę atomów tego pierwiastka, jaka może być użyta do stworzenia kryształu. Utworzenie kryształu, który składa się z a_i atomów i-tego pierwiastka (dla $i = 1,\dots,n$) jest możliwe tylko, gdy:

- $0 \leqslant a_i \leqslant m_i$, *dla* $i = 1,2,\dots,n$,
- $a_1 \oplus \dots \oplus a_n = 0$, *oraz*
- $a_1 + a_2 + \dots + a_n \geqslant 1$.

Ostatni z powyższych warunków oznacza, że kryształ musi składać się z przynajmniej jednego atomu.

Zadanie

Napisz program, który:

- *Wczyta ze standardowego wejścia liczbę pierwiastków oraz ograniczenia na liczbę atomów danego pierwiastka w krysztale.*
- *Wyliczy liczbę różnych kryształów jakie można otrzymać.*
- *Wypisze wynik na standardowe wyjście.*

Wejście

Pierwszy wiersz standardowego wejścia zawiera liczbę pierwiastków n, $2 \leqslant n \leqslant 50$. W drugim i ostatnim wierszu znajduje się n liczb $m_1,\dots,m_n$ pooddzielanych pojedynczymi odstępami, $1 \leqslant m_i < 2^{32} - 1$.

Wyjście

Twój program powinien wypisać na standardowe wyjście jedną liczbę — liczbę kryształów, które można otrzymać. Możesz założyć, że liczba ta jest mniejsza od $2^{64} - 1$.

Przykład

Dla danych wejściowych:

```
3
2 1 3
```

poprawnym wynikiem jest:

```
5
```

Możliwe ilości atomów poszczególnych pierwiastków to: (0,1,1), (1,0,1), (1,1,0), (2,0,2), (2,1,3).

Rozwiązanie

XOR

Podstawą rozwiązania zadania jest znajomość własności operacji *różnicy symetrycznej* `xor`, dlatego zanim przejdziemy do poszukiwania rozwiązania, warto własności te przypomnieć. Operację $\oplus$ wykonujemy na bitach, czyli liczbach 0 i 1. Możemy ją zdefiniować jako

$$x \oplus y = x + y \bmod 2,$$

co można przedstawić także w formie tabeli:

$\oplus$	0	1
0	0	1
1	1	0

Następujące własności łatwo wynikają z definicji:

$$x \oplus x = 0 \tag{1}$$

$$x \oplus 0 = x \tag{2}$$

$$x \oplus y = y \oplus x \tag{3}$$

$$x \oplus (y \oplus z) = (x \oplus y) \oplus z \tag{4}$$

Własności (3) i (4) — *przemienność* i *łączność* działania — oznaczają, że wynik operacji dla większej liczby argumentów:

$$x_1 \oplus x_2 \oplus \cdots \oplus x_k$$

nie zależy od tego, w jakiej kolejności ustawimy argumenty i jak rozstawimy nawiasy w wyrażeniu.

Operację $\oplus$ uogólnia się na dowolne nieujemne liczby całkowite — w tym celu liczby zapisujemy *binarnie*, czyli w systemie dwójkowym:

$$x = (b_{k-1} \ldots b_1 b_0)_2 = b_{k-1} \cdot 2^{k-1} + \ldots + b_1 \cdot 2^1 + b_0 \cdot 2^0,$$

gdzie $b_0, \ldots, b_{k-1} \in \{0,1\}$ są cyframi, zwanymi też *bitami*. Bit b_0 nazywamy *najmniej znaczącym*, a bit b_{k-1} — *najbardziej znaczącym*. Dla wygody ponumerujmy jeszcze bity x w następujący sposób: b_0 będzie bitem pierwszym, b_1 bitem drugim, ..., b_{k-1} bitem k-tym.

Zauważmy, że za pomocą k bitów możemy zapisać każdą liczbę x spełniającą nierówności $0 \leqslant x \leqslant 2^k - 1$ — zapis taki nazywamy *reprezentacją k-bitową* liczby. Jeżeli dodatkowo $2^{k-1} \leqslant x \leqslant 2^k - 1$, czyli $b_{k-1} = 1$, to x nazywamy *liczbą k-bitową*.

Wynik operacji `xor` dla dwóch liczb definiujemy jako liczbę złożoną z bitów, powstałych przez wykonanie operacji `xor` na odpowiadających sobie bitach tych liczb:

$$(a_{k-1} \ldots a_1 a_0)_2 \oplus (b_{k-1} \ldots b_1 b_0)_2 = \big((a_{k-1} \oplus b_{k-1}) \ldots (a_1 \oplus b_1)(a_0 \oplus b_0)\big)_2.$$

Na przykład:

$$\begin{aligned} 12 \oplus 5 &= (2^3 + 2^2) \oplus (2^2 + 2^0) = (1100)_2 \oplus (0101)_2 \\ &= \big((1 \oplus 0)(1 \oplus 1)(0 \oplus 0)(0 \oplus 1)\big)_2 = (1001)_2 = 2^3 + 2^0 = 9. \end{aligned}$$

Własności (1)-(4) pokazane dla bitów, zachodzą także dla nieujemnych liczb całkowitych.

Przypomnijmy, że naszym zadaniem jest znalezienie dla zadanego ciągu $m_1, \ldots, m_n$ liczby elementów zbioru:

$$K(m_1, \ldots, m_n) = \{(a_1, \ldots, a_n) : a_1 \oplus \cdots \oplus a_n = 0, 0 \leqslant a_i \leqslant m_i \text{ dla } i = 1, \ldots, n\}.$$

Szukana liczba kryształów będzie oczywiście o jeden mniejsza, ponieważ z rozwiązania wykluczamy element $(0, \ldots, 0)$.

Rozwiązanie wzorcowe

Przyjmijmy, że ograniczenia na liczbę atomów każdego z pierwiastków spełniają nierówności: $m_1 \geqslant m_2 \geqslant \ldots \geqslant m_n$. Jeśli tak nie jest, to oczywiście możemy na wstępie te liczby posortować.

Spróbujmy wyrazić w inny sposób równość $a_1 \oplus \cdots \oplus a_n = 0$. Zauważmy, że wynik `xor` dla ciągu bitów jest równy zero wtedy i tylko wtedy, gdy w tym ciągu występuje parzysta liczba jedynek. Można stąd wywnioskować, że:

Fakt 1 *Równość $a_1 \oplus \cdots \oplus a_n = 0$ zachodzi wtedy i tylko wtedy, gdy dla każdego j liczba jedynek w ciągu powstałym z wzięcia j-tych bitów liczb $a_1, \ldots, a_n$ jest parzysta.*

Zanotujmy też inną własność.

Fakt 2 *Równość $a_1 \oplus \cdots \oplus a_n = 0$ zachodzi wtedy i tylko wtedy, gdy $a_1 = a_2 \oplus \cdots \oplus a_n$.*

Dowód Następujące równości są równoważne:

$$\begin{aligned} a_1 \oplus a_2 \oplus \cdots \oplus a_n &= 0, \\ a_1 \oplus a_1 \oplus a_2 \oplus \cdots \oplus a_n &= a_1 \oplus 0, \\ 0 \oplus a_2 \oplus \cdots \oplus a_n &= a_1, \\ a_2 \oplus \cdots \oplus a_n &= a_1. \end{aligned}$$

■

Łatwy przypadek

Fakt 2 oznacza, że jeśli mamy ciąg n argumentów i ustalimy wartości $n-1$ spośród nich, to tylko dla jednej wartości pozostałego argumentu wynik `xor` dla całego ciągu jest równy zero. Ze spostrzeżenia tego może zrodzić się następujący pomysł. Wybierzmy dowolnie ciąg liczb $a_2, \ldots, a_n$ spełniających ograniczenia $0 \leqslant a_i \leqslant m_i$ dla $i = 2, \ldots, n$ i wyznaczmy wartość a_1 ze wzoru z faktu 2. Ponieważ m_1 — ograniczenie na a_1, jest największe (czyli jest większe lub równe od każdej z liczb $a_2, \ldots, a_n$), więc może także $a_1 \leqslant m_1$.

Przy takich optymistycznych założeniach liczba szukanych wyborów $a_1, \ldots, a_n$, takich że $a_1 \oplus \cdots \oplus a_n = 0$, wynosiłaby po prostu $(m_2+1) \cdot \ldots \cdot (m_n+1)$. Niestety, nie zawsze jest tak dobrze. Na przykład, dla $a_2 = 1$ i $a_3 = 2$ oraz $m_1 = m_2 = m_3 = 2$, mamy $a_1 = 1 \oplus 2 = 3$, czyli liczbę większą od 2.

Zastanówmy się jednak, czy potrafimy określić, jakie warunki muszą spełniać wartości $m_1, \ldots, m_n$, żeby dla dowolnie wybranych $a_2 \leqslant m_2, \ldots, a_n \leqslant m_n$, zachodziła nierówność $a_2 \oplus \cdots \oplus a_n \leqslant m_1$? Otóż, jeśli $m_2, \ldots, m_n$ są co najwyżej k-bitowe, to wtedy także $a_2, \ldots, a_n$ są co najwyżej k-bitowe i $a_2 \oplus \cdots \oplus a_n \leqslant 2^k - 1$. Wystarczy więc, że $m_k \geqslant 2^k - 1$, by a_1 mieściło się w zadanym ograniczeniu. Podsumujmy powyższy wywód następującym faktem:

Fakt 3 *Jeśli istnieje k takie, że $m_1 \geqslant 2^k - 1 \geqslant m_2 \geqslant \ldots \geqslant m_n$, to liczba elementów zbioru $K(m_1, \ldots, m_n)$ jest równa $(m_2+1) \cdot \ldots \cdot (m_n+1)$.*

Redukcja

Niestety nie zawsze mamy do czynienia z łatwym przypadkiem. Spróbujmy więc podążać w stronę rozwiązania problemu redukując dany (niełatwy) przypadek do nieco mniejszego. Sposób redukcji będzie raczej nietypowy. Zamiast zmniejszać liczbę elementów ciągu $a_1, \ldots, a_n$ postaramy się go trochę „określić”, czyli ustalić wartości wybranych bitów w liczbach $a_1, \ldots, a_n$. Aby było nam wygodniej odwoływać się do poszczególnych bitów tych liczb, wprowadźmy następujące oznaczenia. Załóżmy, że m_1 jest liczbą k-bitową, stąd wszystkie dopuszczalne wartości $a_1, \ldots, a_n$ są liczbami co najwyżej k-bitowymi i dla każdej z nich możemy przedstawić reprezentację k-bitową: $a_i = (b^i_{k-1} \ldots b^i_1 b^i_0)_2$ dla $1 \leqslant i \leqslant n$. Jeden krok redukcji będzie polegał na ustaleniu wartości wszystkich najbardziej znaczących bitów $b^1_{k-1}, b^2_{k-1}, \ldots, b^n_{k-1}$. W ten sposób rzeczywiście zredukujemy rozmiar problemu nie zmniejszając co prawda długości zadanego ciągu, ale „skracając” jego elementy.

Załóżmy, że z ograniczeń wynika, iż tylko s początkowych liczb $a_1, \ldots, a_n$ może być k-bitowych, czyli:

$$2^k > m_1, \ldots, m_s \geqslant 2^{k-1} > m_{s+1}, \ldots, m_n.$$

Oznacza to, że bity $b^{s+1}_{k-1}, b^{s+2}_{k-1}, \ldots, b^n_{k-1}$ muszą być zerami, a wartości $b^1_{k-1}, b^2_{k-1}, \ldots, b^s_{k-1}$ możemy wybrać. Oczywiście wybór nie może być całkowicie dowolny — zgodnie z faktem 1 musimy wybrać jedynkę dla parzystej liczby bitów. Niech $I \subseteq \{1, \ldots, s\}$ będzie zbiorem indeksów tych bitów, dla których wybraliśmy wartość 1.

Oznaczmy przez $a'_1, \ldots, a'_n$ pozostałe do ustalenia części elementów $a_1, \ldots, a_n$, czyli $a'_i = (b^i_{k-2} \ldots b^i_0)_2$. Jakie ograniczenia muszą spełniać ich wartości?

- Dla $i > s$ wartością a'_i może być dowolna wartość ze zbioru $\{0, \ldots, m_i\}$, a więc ograniczenie na wartości a'_i nadal wynosi m_i.

- Rozważmy $i \leqslant s$ oraz $i \notin I$, czyli element a_i, dla którego przyjęliśmy $b^i_{k-1} = 0$. Ponieważ jednocześnie $m_i \geqslant 2^{k-1}$, więc pozostałe bity a_i możemy wybierać całkowicie dowolnie — na pewno nie przekroczymy zadanego ograniczenia. Możemy więc przyjąć, że pozostaje nam do wyboru wartość a'_i ze zbioru $\{0,\ldots,2^{k-1}-1\}$.
- Pozostał przypadek $i \leqslant s$ oraz $i \in I$, czyli wartość a_i, dla której przyjęliśmy $b^i_{k-1} = 1$. Aby nie przekroczyć ograniczenia m_i wartość a'_i musimy wybrać ze zbioru $\{0,\ldots,m_i - 2^{k-1}\}$.

Podsumowując, po ustaleniu k-tych bitów w elementach $a_1,\ldots,a_n$ otrzymaliśmy ciąg $a'_1,\ldots,a'_n$, dla którego ograniczenia, oznaczmy je $m'_1,\ldots,m'_n$, wynoszą odpowiednio:

- $m'_i = m_i$ dla $i > s$,
- $m'_i = 2^{k-1} - 1$ dla $i \leqslant s$ i $i \notin I$,
- $m'_i = m_i - 2^{k-1}$ dla $i \leqslant s$ i $i \in I$.

Nowe ograniczenia (a tym samym wybierane elementy) są więc liczbami co najwyżej $(k-1)$-bitowymi.

Plon redukcji

Niestety plon zaproponowanej redukcji może być dość liczny. Z ciągu k-bitowych ograniczeń $m_1,\ldots,m_n$ powstaje tyle podproblemów, na ile sposobów możemy ustawić najbardziej znaczące bity w opisanym wyżej kroku redukcji. Ponieważ liczba różnych podzbiorów zbioru $\{1,\ldots,s\}$ parzystej mocy jest rzędu $O(2^s)$, więc otrzymamy właśnie tyle mniejszych problemów do rozwiązania.

Pesymizm jest jednak przedwczesny, gdyż wygenerowane w trakcie redukcji podproblemy nie są tak zupełnie przypadkowe. Właściwie jak się dobrze przyjrzeć, to można zauważyć, że większość z nich należy do przypadków łatwych. Otóż jeśli nie ustawiliśmy wszystkich bitów równych jeden, czyli istnieje $j \in \{1,\ldots,s\}$ takie że $b^j_{k-1} = 0$, to wtedy $m'_j = 2^{k-1} - 1$. W ciągu ograniczeń $m'_1,\ldots,m'_n$ występuje więc największa liczba $(k-1)$-bitowa — czyli mamy do czynienia z przypadkiem łatwym! Stąd liczbę elementów zbioru $K(m'_1,\ldots,m'_n)$ możemy wyznaczyć na podstawie faktu 3, jako:

$$|K(m'_1,\ldots,m'_n)| = (m'_1+1)\cdot\ldots\cdot(m'_{j-1}+1)\cdot(m'_{j+1}+1)\cdot\ldots\cdot(m'_n+1). \tag{5}$$

Pozostaje tylko jeden przypadek niełatwy — gdy nie istnieje $j \notin I$, czyli przyjęliśmy $b^1_{k-1} = b^2_{k-1} = \cdots = b^s_{k-1} = 1$ (oczywiście może się to zdarzyć tylko wówczas, gdy s jest parzyste). Jest więc *co najwyżej* jeden przypadek, gdy nie będzie można użyć wzoru (5) i trzeba będzie zastosować rekursję.

Została nam ostatnia kwestia, z którą musimy się uporać. Niestety przypadków łatwych jest bardzo dużo, więc choć mamy gotowy wzór, to policzenie sumarycznej liczby kryształów, w których przynajmniej na jednej pozycji i, takiej że $1 \leqslant i \leqslant s$, k-ty bit został ustawiony na zero nie jest zadaniem łatwym. Jest to wręcz najtrudniejsza część rozwiązania i pokażemy cztery metody jej rozwiązania.

Ostatnia kwestia

Metoda 1 — przeglądanie wszystkich podzbiorów

Najprostszą metodą jest przejrzenie wszystkich podzbiorów $I \subseteq \{1,\ldots,s\}$ o parzystej liczbie elementów i różnych od zbioru $\{1,\ldots,s\}$. Dla każdego takiego podzbioru wyznaczamy liczbę kombinacji używając wzoru (5), a następnie sumujemy wyniki częściowe. Podzbiorów takich jest rzędu $O(2^s)$, więc dla dużych s rozwiązanie to jest zupełnie niepraktyczne — przypomnijmy, że s może być równe nawet 50.

Metoda 2 — grupowanie podzbiorów

Możemy uzyskać duże przyspieszenie w stosunku do metody 1, jeśli na wybieranie podzbioru $I = \{i_1, i_2, \ldots\} \subseteq \{1,\ldots,s\}$ spojrzymy raczej jak na wybieranie podciągu $m_{i_1}, m_{i_2}, \ldots$ ciągu $m_1,\ldots,m_s$. Zauważmy, że jeśli elementy $m_1,\ldots,m_s$ nie są różne, to dwa różne podzbiory indeksów $I, I' \subset \{1,\ldots,s\}$ mogą dawać takie same zbiory (z powtórzeniami) wartości $\{m_i \mid i \in I\} = \{m_i \mid i \in I'\}$. Dla takich podzbiorów formuła (5) zwraca taką samą wartość, więc obliczenia wystarczy przeprowadzić raz. Jest to bardzo opłacalne wtedy, gdy wśród liczb $m_1,\ldots,m_s$ często pojawiają się takie same, co jest bardzo prawdopodobne dla małych k, gdyż wiemy, że $2^{k-1} \leqslant m_1,\ldots,m_s < 2^k$.

Wystarczy więc wygenerować wszystkie możliwe podzbiory wartości $\{m_1,\ldots,m_s\}$ i dla każdego z nich wyznaczyć liczbę podzbiorów indeksów $\{1,\ldots,s\}$, z których ten podciąg możemy otrzymać. Oznaczmy przez L_j, $0 \leqslant j < 2^{k-1}$, liczbę wystąpień wartości $2^{k-1}+j$ w ciągu $m_1,\ldots,m_s$. Wtedy jeden podciąg wartości możemy otrzymać przez wybranie dla każdego $0 \leqslant j < 2^{k-1}$ liczby l_j, oznaczającej liczbę wystąpień wartości $2^{k-1}+j$ w tym podciągu. Muszą być przy tym spełnione warunki:

- $0 \leqslant l_j \leqslant L_j$,
- długość podciągu, czyli $\sum\limits_{0\leqslant j<2^{k-1}} l_j$, jest liczbą parzystą.

Dla danych l_j liczba podzbiorów indeksów, które dają w efekcie dany podciąg wynosi

$$\prod_{0\leqslant j<2^{k-1}} \binom{L_j}{l_j}.$$

Technikę grupowania możemy stosować w dowolnym momencie, nawet dla większych k, gdyż nie musimy rozpatrywać w ogóle tych j, dla których $L_j = 0$.

Metoda ta okazuje się być szybka ze względu na nałożone w zadaniu ograniczenie na liczbę kryształów, co wykazujemy w poniższym lemacie.

Fakt 4 *Liczba różnych podciągów ciągu* $\{m_1,\ldots,m_s\}$ *nie przekracza* $2^{16} = 65536$.

Dowód Oszacujmy od dołu wartość $K(m_1,\ldots,m_n)$. W tym celu policzmy tylko niektóre z możliwych kryształów — rozważmy przypadki, gdy $a_{s+1} = a_{s+2} = \cdots = a_n = 0$ oraz $a_i < 2^{k-1}$ dla $i \leqslant s$. Wtedy możemy dowolnie wybrać wartości a_i dla $1 < i \leqslant s$ oraz wyznaczyć a_1 ze wzoru $a_2 \oplus \cdots \oplus a_s$ (na pewno będzie spełniać ograniczenie $a_1 \leqslant m_1$, bo $m_1 \geqslant 2^{k-1}$). W ten sposób, biorąc pod uwagę tylko niektóre z dopuszczalnych kryształów, już otrzymaliśmy ich $\left(2^{k-1}\right)^{s-1}$. Z warunków podanych w zadaniu wiemy, że liczba kryształów jest mniejsza niż 2^{64}, więc muszą zachodzić nierówności

$$2^{(k-1)(s-1)} \leqslant K(m_1,\ldots,m_n) < 2^{64},$$

czyli $(k-1)(s-1) < 64$ i dalej $s-1 < \frac{64}{k-1}$, skąd ostatecznie mamy oszacowanie:

$$s \leqslant \left\lceil \frac{64}{k-1} \right\rceil. \tag{6}$$

Wynika stąd, że dla $k \geqslant 5$ ograniczenie dla s wynosi $s \leqslant \frac{64}{4} = 16$, więc liczba wszystkich podzbiorów $\{1,\ldots,s\}$, a więc i liczba różnych podciągów jest mniejsza niż 2^{16}. To wykazuje prawdziwość faktu dla $k \geqslant 5$.

Przejdźmy teraz do przypadku, gdy $k \leqslant 4$. Liczba wszystkich różnych podciągów ciągu $m_1,\ldots,m_s$ (niekoniecznie parzystej wielkości i rozmiaru mniejszego od s) wynosi

$$P = \prod_{0 \leqslant j < 2^{k-1}} (L_j + 1),$$

gdyż dla każdego j liczbę l_j możemy wybierać spośród liczb $0,\ldots,L_j$. Ponadto wiemy, że $\sum\limits_{0 \leqslant j < 2^{k-1}} L_j = s$, skąd $\sum\limits_{0 \leqslant j < 2^{k-1}} (L_j + 1) = s + 2^{k-1}$. Stosując nierówność Cauchy'ego dla średniej arytmetycznej i geometrycznej mamy:

$$\sqrt[2^{k-1}]{\prod_{0 \leqslant j < 2^{k-1}} (L_j + 1)} \leqslant \frac{\sum\limits_{0 \leqslant j < 2^{k-1}} (L_j + 1)}{2^{k-1}} = \frac{s + 2^{k-1}}{2^{k-1}},$$

skąd dostajemy ograniczenie

$$P \leqslant \left(\frac{s + 2^{k-1}}{2^{k-1}} \right)^{2^{k-1}}. \tag{7}$$

Stosując ograniczenia (6) i (7) dla $k \leqslant 4$ otrzymamy następujące ograniczenia na liczbę różnych podciągów P. Dla $k = 4$ mamy $s \leqslant \left\lceil \frac{64}{3} \right\rceil = 22$, więc $P \leqslant \left(\frac{22+8}{8}\right)^8 \approx 39107$. Dla $k = 3$ mamy $s \leqslant \left\lceil \frac{64}{2} \right\rceil = 32$, więc $P \leqslant \left(\frac{32+4}{4}\right)^4 = 9^4 = 6561$. Dla $k = 2$ wiemy, że mamy $s \leqslant 50$, więc $P \leqslant \left(\frac{50+2}{2}\right)^2 = 26^2 = 676$. Podobnie dla $k = 1$, $P \leqslant (\frac{50+1}{1})^1 = 51$. ■

Chociaż przedstawione rozwiązanie nie jest optymalne pod względem złożoności (na przykład w porównaniu z przedstawioną dalej metodą 4), jest ono wystarczająco szybkie dla ograniczeń z zadania i zostało użyte w rozwiązaniu wzorcowym.

Metoda 3 — programowanie dynamiczne dla małych k

Problem z dużą liczbą podzbiorów możemy też rozwiązać inaczej. Przy dowodzie faktu 4 uzyskaliśmy ograniczenie (6) na wartość s. Oznacza to, że dla większych k liczba podzbiorów $\{1,\ldots,s\}$ jest na tyle mała, że możemy pozwolić sobie na przejrzenie ich wszystkich. Natomiast dla małych k, dajmy na to dla $k \leqslant 6$, możemy zastosować technikę programowania dynamicznego do wyznaczenia liczności $K(m_1,\ldots,m_n)$.

Przypomnijmy, że liczby $m_1,\ldots,m_n$ są co najwyżej k-bitowe, co między innymi oznacza, że $m_1,\ldots,m_n < 2^k$. Niech $A[t,x]$ dla $0 \leqslant t \leqslant n$ i $0 \leqslant x < 2^k$ oznacza liczbę takich krotek $(a_1,\ldots,a_t)$, że $a_1 \oplus \cdots \oplus a_t = x$ oraz dla $i = 1,\ldots,t$ spełnione są ograniczenia $0 \leqslant a_i \leqslant m_i$. Przy tych oznaczeniach szukaną liczbą jest $A[n,0]$.

Na początku inicjujemy $A[0,0] = 1$ i $A[0,x] = 0$ dla wszystkich $1 \leqslant x < 2^k$. Następnie dla kolejnych $t = 1,\ldots,n$ i każdego $0 \leqslant x < 2^k$ wartość $A[t,x]$ wyliczamy na podstawie wartości $A[t-1,\cdot]$ używając wzoru:

$$A[t,x] = \sum_{a_t=0}^{m_t} A[t-1, x \oplus a_t].$$

Poprawność wzoru wynika stąd, że równość $a_1 \oplus \cdots \oplus a_{t-1} \oplus a_t = x$ jest równoznaczna równości $a_1 \oplus \cdots \oplus a_{t-1} = x \oplus a_t$, a więc liczba wszystkich takich krotek $(a_1,\ldots,a_t)$, że $a_1 \oplus \cdots \oplus a_t = x$ jest równa sumie, po wszystkich możliwych wyborach a_t, liczby takich krotek $(a_1,\ldots,a_{t-1})$, że $a_1 \oplus \cdots \oplus a_{t-1} = x \oplus a_t$.

Czas liczenia $A[t,x]$ dla danych t i x wynosi $O(m_t) \leqslant O(2^k)$, więc całkowita złożoność czasowa tego rozwiązania jest równa $O(n \cdot 2^k \cdot 2^k) = O(n \cdot 2^{2k})$. Zatem dla $k \leqslant 6$ jest ono wystarczająco szybkie.

Metoda 4 — pełne programowanie dynamiczne

Przypomnijmy pierwotne sformułowanie problemu, z którym teraz staramy się uporać. Mamy dane ograniczenia — co najwyżej k-bitowe liczby $m_1,\ldots,m_n$. Przez s oznaczyliśmy największy indeks ograniczenia dokładnie k-bitowego, czyli $m_1 \geqslant \cdots \geqslant m_s > 2^{k-1} - 1 \geqslant m_{s+1} \geqslant \cdots \geqslant m_n$. Poszukujemy krotek $(a_1,a_2,\ldots,a_n)$ spełniających ograniczenia $0 \leqslant a_i \leqslant m_i$ dla $1 \leqslant i \leqslant n$, dla których $a_1 \oplus \cdots \oplus a_n = 0$ oraz nie wszystkie $a_1,\ldots,a_s$ są jednocześnie większe niż $2^k - 1$. W tym celu ustalamy wartości najbardziej znaczących bitów $a_1,\ldots,a_n$. Dla $a_{s+1},a_{s+2},\ldots,a_n$ z racji ograniczeń musimy przyjąć $b_{k-1}^{s+1} = b_{k-1}^{s+2} = \cdots = b_{k-1}^{n} = 0$ i pozostaje nam znaleźć liczbę układów pozostałych bitów. Wiemy przy tym, że suma bitów $b_{k-1}^1,\ldots,b_{k-1}^s$ musi być parzysta i wszystkie te bity nie mogą być równocześnie jedynkami.

Oznaczmy przez $P[t]$ dla $t = 1,\ldots,s+1$ liczby takich ciągów $(a_t,\ldots,a_n)$ spełniających ograniczenia $(m_t,\ldots,m_n)$, dla których zachodzą warunki:

(i) istnieje $t \leqslant i \leqslant s$, taki że $b_{k-1}^i = 0$ oraz

(ii) liczba jedynek wśród bitów $b_{k-1}^t, b_{k-1}^{t+1},\ldots,b_{k-1}^s$ jest parzysta i

(iii) $a_t \oplus \cdots \oplus a_n = 0$.

Analogicznie zdefiniujemy wartości $N[t]$ dla $t = 1,\ldots,s+1$, jako liczbę ciągów $(a_t,\ldots,a_n)$ spełniających ograniczenia $(m_t,\ldots,m_n)$, dla których zachodzą warunki:

(i) istnieje $t \leqslant i \leqslant s$, takie że $b_{k-1}^i = 0$ oraz

(ii') liczba jedynek wśród bitów $b_{k-1}^t, b_{k-1}^{t+1},\ldots,b_{k-1}^s$ jest nieparzysta i

(iii') $a_t \oplus \cdots \oplus a_n = 2^{k-1}$.

Dla tak określonych tablic P i N szukaną przez nas wartością jest $P[1]$. Aby sprawnie policzyć wartości P i N udowodnimy pomocniczy

Lemat 5 *Niech* $1 \leqslant t \leqslant s+1$ *oraz* $0 \leqslant x < 2^{k-1}$*. Liczba takich ciągów* $(a_t, \ldots, a_n)$*, dla których zachodzą ograniczenia* $(m_t, \ldots, m_n)$ *oraz warunki (i), (ii) i* $a_t \oplus \cdots \oplus a_n = x$ *wynosi* $P[t]$*. Z kolei liczba krotek* $(a_t, \ldots, a_n)$*, dla których zachodzą ograniczenia* $(m_t, \ldots, m_n)$ *oraz warunki (i), (ii') i* $a_t \oplus \cdots \oplus a_n = 2^{k-1} + x$ *wynosi* $N[t]$*.*

Dowód Pokażemy dowód pierwszej części lematu dotyczącej $P[t]$ — dowód dla $N[t]$ jest całkowicie analogiczny. Układy, które mamy zliczyć różnią się od tych z definicji $P[t]$ tylko warunkiem $a_t \oplus \cdots \oplus a_n = x$. Pokażemy, że każdą krotkę spełniającą warunki definicji $P[t]$ można przekształcić w układ spełniający warunki pierwszej części lematu. Ponadto przekształcając różne krotki, dostaniemy różne układy wynikowe.

Niech $(a_t, \ldots, a_n)$ spełnia ograniczenia $(m_t, \ldots, m_n)$ oraz warunki (i), (ii) i (iii). Niech i będzie indeksem, dla którego $b^i_{k-1} = 0$ (jeśli jest takich kilka, to wybieramy najmniejszy z nich). Wiadomo, że znajdziemy $i \leqslant s$, dla którego zachodzi ten warunek. Rozważmy ciąg $(a_t, \ldots, a_{i-1}, a_i \oplus x, a_{i+1}, \ldots, a_n)$. Spełnia on wszystkie warunki z definicji $P[t]$:

(i) bit $b^i_{k-1} = 0$,

(ii) suma bitów $b^t_{k-1}, \ldots, b^s_{k-1}$ nie uległa zmianie, gdyż x jest liczbą $(k-1)$-bitową,

(iii) $a_t \oplus \cdots \oplus a_{i-1} \oplus (a_i \oplus x) \oplus a_{i+1} \oplus \cdots \oplus a_n = (a_t \oplus \cdots \oplus a_n) \oplus x = x$.

Co równie ważne, wartość $a_i \oplus x$ nie przekracza ograniczenia m_i, ponieważ wybraliśmy indeks i, dla którego $b^i_{k-1} = 0$, więc $a_i < 2^{k-1}$ (wtedy także $a_i \oplus x < 2^{k-1}$), a ograniczenie m_i jest równe co najmniej 2^{k-1}.

Pozostaje już tylko zauważyć, że dla dwóch różnych ciągów $A = (a_t, \ldots, a_n)$ i $C = (c_t, \ldots, c_n)$ transformacja daje dwa różne ciągi.

- Jeśli ciągi różnią się którymś z k-tych bitów, to po transformacji nie mogą stać się równe, bo operacja nie wpływa na k-te bity elementów ciągów.
- Jeśli A i C mają identyczne k-te bity, to w trakcie transformacji w obu ciągach wybraliśmy ten sam i-ty element, który poddaliśmy operacji `xor` z x. Niech j będzie indeksem, dla którego $a_j \neq c_j$ na $k-1$ mniej znaczących bitach.
  - Jeśli $i = j$, czyli $a_i \neq c_i$, to także $a_i \oplus x \neq c_i \oplus x$.
  - W przeciwnym razie elementy a_j oraz c_j nie są zmieniane w trakcie transformacji, więc pozostają różne.

Z powyższego rozumowania wynika, że układów spełniających warunki z pierwszej części lematu jest nie mniej, niż układów spełniających warunki z definicji $P[t]$. Zdefiniowanie analogicznej transformacji w drugą stronę pozwala z kolei wykazać, że $P[t]$ jest nie mniejsze niż liczba układów spełniających lemat, co dowodzi tę część twierdzenia.

■

Możemy teraz pokazać, jak wyznaczyć wartości $P[t]$ i $N[t]$ kolejno dla $t = s+1, s, \ldots, 1$. Dla $t = s+1$ mamy $P[t] = N[t] = 0$, gdyż nie możemy spełnić warunku (i). Rozważmy $P[t]$ dla $t \leqslant s$:

1. Załóżmy, że ustawiamy $b^t_{k-1} = 1$. Ponieważ żądamy, aby liczba bitów ustawionych na jeden wśród $b^t_{k-1}, b^{t+1}_{k-1}, \ldots, b^s_{k-1}$ była parzysta, więc tym samym liczba bitów ustawionych na jeden wśród $b^{t+1}_{k-1}, \ldots, b^s_{k-1}$ musi być nieparzysta. Dla każdego a_t spełniającego ograniczenia $2^{k-1} \leqslant a_t \leqslant m_t$ chcemy wyznaczyć liczbę takich krotek $(a_{t+1}, \ldots, a_n)$, że $a_t \oplus a_{t+1} \oplus \cdots \oplus a_n = 0$, czyli że $a_{t+1} \oplus \cdots \oplus a_n = a_t$. Z lematu, przyjmując $x = a_t$, wiemy że jest ich $N[t+1]$. Ponieważ wartość a_t możemy wybrać na $m_t - 2^{k-1} + 1$ sposobów, więc w tym przypadku otrzymujemy $(m_t - 2^{k-1} + 1) \cdot N[t+1]$ krotek spełniających warunki definicji $P[t]$

2. Załóżmy, że ustawiamy $b^t_{k-1} = 0$. Tym razem chcemy, by liczba bitów ustawionych na jeden wśród $b^{t+1}_{k-1}, \ldots, b^s_{k-1}$ była parzysta. Podobnie jak w poprzednim przypadku zauważamy, że dla każdego $0 \leqslant a_t < 2^{k-1}$ liczba takich krotek spełniających równość $a_t \oplus \cdots \oplus a_n = 0$, wynosi $P[t+1]$. Tym razem a_t możemy wybrać na 2^{k-1} sposobów, więc otrzymujemy $2^{k-1} \cdot P[t+1]$ krotek.

   W tym przypadku istnieje jeszcze inna możliwość utworzenia krotek spełniających warunki definicji $P[t]$. Ponieważ mamy już jeden bit ustawiony na zero ($b^t_{k-1} = 0$), możemy dopuścić przypadek, gdy wszystkie pozostałe bity $b^{t+1}_{k-1}, \ldots, b^s_{k-1}$ są równe jeden. Takie układy niestety nie są uwzględnione w definicji $P[t+1]$. Wyznaczmy liczbę takich krotek.

   - Jeśli $s - t$ jest nieparzyste, to nie można spełnić warunku (i) definicji $P[t]$ i liczba szukanych krotek wynosi zero.
   - Niech $s - t$ będzie liczbą parzystą. Ustawiliśmy $b^t_{k-1} = 0$ oraz $b^{t+1}_{k-1} = \ldots b^s_{k-1} = 1$ (bity $b^{s+1}_{k-1}, \ldots, b^n_{k-1}$ są oczywiście zerami z racji ograniczeń $(m_{s+1}, \ldots, m_n)$). Wybierzmy pozostałe bity $a_{t+1}, \ldots, a_n$ i wyznaczmy wartość $a_t = a_{t+1} \oplus \cdots \oplus a_n$ — na pewno spełnia ona ograniczenie $m_t \geqslant 2^k$, gdyż liczba jedynek wśród k-tych bitów $a_{t+1}, \ldots, a_n$ jest parzysta. Pozostaje nam obliczyć, na ile sposobów możemy wybrać wartości $a_{t+1}, \ldots, a_n$ po poczynionych ustaleniach. Dla $i > s$ wartość a_i wybieramy na $m_i + 1$ sposobów. Natomiast dla $t + 1 \leqslant i \leqslant s$ — na $m_i - 2^{k-1} + 1$ sposobów. Zatem łączna liczba krotek rozważanego typu wynosi $(m_{t+1} - 2^{k-1} + 1) \cdot \ldots \cdot (m_s - 2^{k-1} + 1) \cdot (m_{s+1} + 1) \cdot \ldots \cdot (m_n + 1)$.

Analogiczne rozważania można przeprowadzić dla $N[t]$. W rezultacie otrzymujemy wzory:

$$\begin{aligned}
P[t] &= (m_t - 2^{k-1} + 1) \cdot N[t+1] + 2^{k-1} \cdot P[t+1] + \begin{cases} J[t+1] & s-t \text{ jest parzyste} \\ 0 & s-t \text{ jest nieparzyste} \end{cases} \\
N[t] &= (m_t - 2^{k-1} + 1) \cdot P[t+1] + 2^{k-1} \cdot N[t+1] + \begin{cases} 0 & s-t \text{ jest parzyste} \\ J[t+1] & s-t \text{ jest nieparzyste} \end{cases}
\end{aligned}$$

gdzie

$$J[t+1] = (m_{t+1} - 2^{k-1} + 1) \cdot \ldots \cdot (m_s - 2^{k-1} + 1) \cdot (m_{s+1} + 1) \cdot \ldots \cdot (m_n + 1).$$

Tablicę J wyliczamy także stosując metodę programowania dynamicznego. Inicjujemy $J[s+1] = (m_{s+1} + 1) \cdot \ldots \cdot (m_n + 1)$, a następnie dla $1 \leqslant t \leqslant s$ przyjmujemy $J[t] = (m_t - 2^{k-1} + 1) \cdot J[t+1]$.

Przedstawiona metoda jest bardzo szybka — ma złożoność czasową $O(n)$.

Testy

Na zawody zostało przygotowanych 27 testów, za pomocą których zostało utworzonych 20 zestawów testów.

Nazwa	n	Opis
kry1a.in	6	test poprawnościowy
kry1b.in	2	najmniejszy test — dwie jedynki
kry2a.in	7	test poprawnościowy
kry2b.in	4	test poprawnościowy
kry3a.in	7	nieduże liczby (do 20)
kry3b.in	4	test poprawnościowy
kry4.in	10	nieduże liczby (do 20)
kry5.in	13	nieduże liczby (do 20)
kry6.in	20	kilka średnich liczb (ok. 1 000)
kry7.in	25	nieduże liczby (do 25)
kry8.in	10	3 duże liczby (ok. 1 000 000)
kry9.in	3	3 bardzo duże liczby (ok. 2^{32})
kry10.in	25	3 średnie liczby (ok. 20 000) i reszta dwójek
kry11a.in	11	3 bardzo duże liczby (ok. 10^8), reszta dwójki i jedynki
kry11b.in	28	same czwórki, piątki i szóstki
kry12a.in	9	4 duże liczby (od 10^5 do 10^6), 10 i reszta dwójki i jedynki
kry12b.in	34	kilka niedużych liczb (do 20) i reszta 1, 2 i 3
kry13.in	43	małe liczby (do 5)
kry14.in	28	dwie liczby ok. 100 000, dwie ok. 1 000, reszta jedynki
kry15a.in	9	kilka dużych, kilka małych liczb
kry15b.in	30	cztery liczby ok. 4 000, reszta jedynki i kilka dwójek
kry16a.in	5	5 liczb, w tym 3 bardzo duże (ok. 10^8–10^9)
kry16b.in	50	liczby nie większe od 4
kry17.in	28	nieznacznie zmodyfikowany test 14
kry18.in	28	dwie liczby ok. 10^6, dwie ok. 10^3 i reszta jedynki
kry19.in	28	dwie liczby ok. 2^{20}, dwie ok. 2^9 i reszta jedynki
kry20.in	28	dwa razy po 10^6, dwa razy po $2^9 - 1$ i reszta jedynki

Jakub Radoszewski	**Jakub Radoszewski, Adam Iwanicki**	**Adam Iwanicki**
Treść zadania	**Opracowanie**	**Program**

OI, Etap III, dzień drugi, 30.03.2006

Palindromy

Mały Jaś lubi bawić się słowami. Wybrał on sobie n palindromów (palindromem nazywamy słowo, które czytane od przodu i od tyłu jest dokładnie takie samo, jak np. `ala`, `anna` *czy* `kajak`*), a następnie utworzył wszystkie możliwe n^2 par spośród nich i posklejał palindromy występujące w tych parach w pojedyncze słowa. Na koniec Jaś policzył, ile spośród tak otrzymanych słów stanowią palindromy. Nie jest on jednak pewien, czy się nie pomylił, dlatego poprosił Ciebie o wykonanie tych samych czynności i podanie mu wyniku. Napisz program, który wykona to za Ciebie.*

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia palindromy, które podał Ci Jaś,*
- *wyznaczy liczbę słów, utworzonych z par wczytanych palindromów, które są palindromami,*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu wejścia znajduje się jedna liczba całkowita n ($n \geqslant 2$), oznaczająca liczbę palindromów podanych przez Jasia. Następnych n wierszy zawiera opisy palindromów. Wiersz $i+1$ składa się z jednej dodatniej liczby całkowitej a_i, oznaczającej długość i-tego palindromu, oraz palindromu złożonego z a_i małych liter alfabetu angielskiego. Liczba a_i i palindrom oddzielone są pojedynczym odstępem. Palindromy podane w różnych wierszach są różne. Łączna długość wszystkich palindromów nie przekracza $2\,000\,000$.

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać jedną liczbę całkowitą: liczbę różnych uporządkowanych par palindromów, które po sklejeniu ze sobą dają w wyniku palindromy.

Przykład

Dla danych wejściowych:

```
6
2 aa
3 aba
3 aaa
6 abaaba
5 aaaaa
4 abba
```

poprawnym wynikiem jest:

```
14
```

Rozwiązanie

Problem postawiony w zadaniu można rozwiązać na wiele różnych sposobów. W niniejszym opracowaniu ograniczymy się do analizy trzech najlepszych rozwiązań, a o pozostałych wspomnimy pokrótce na końcu opisu. W kolejnych z prezentowanych metod w coraz większym stopniu będziemy wykorzystywać specyficzne właściwości palindromów, co pozwoli nam konstruować coraz prostsze algorytmy i w końcu doprowadzi nas do rozwiązania wzorcowego.

Na wstępie wprowadźmy kilka podstawowych definicji związanych ze słowami.

- *Prefiksem* słowa nazywamy dowolny spójny początkowy fragment tego słowa, czyli słowo złożone z pewnej liczby jego początkowych liter. Na przykład prefiksami słowa `aba` są: `a`, `ab` i `aba`, a także słowo puste, które oznaczamy zazwyczaj symbolem ε.
- *Sufiksem* słowa nazywamy dowolny spójny końcowy fragment tego słowa. Na przykład sufiksami słowa `aba` są: ε, `a`, `ba` i `aba`.
- *Podsłowem* słowa jest jego dowolny spójny fragment.
- *Prefikso-sufiksem* słowa nazywamy podsłowo, które jest jednocześnie prefiksem i sufiksem danego słowa. Prefikso-sufiksami słowa `aba` są: ε, `a` i `aba`.
- *Odwróceniem* słowa u (oznaczanym przez u^R) nazywamy słowo, które składa się z tych samych liter co u, ale zapisanych w odwrotnej kolejności. Oczywistą i ważną własnością odwrócenia słów jest równość $(uv)^R = v^R u^R$, gdzie uv oznacza sklejenie słów u i v.
- *Długością* słowa u (oznaczaną przez $|u|$) jest liczba liter, z których składa się to słowo.
- *k-tą potęgą* słowa u nazywamy słowo (oznaczane u^k) równe k-krotnemu sklejeniu słowa u ze sobą.
- *Okresem* słowa u nazywamy takie słowo v, że v jest prefiksem u i istnieje taka liczba naturalna k, że u jest prefiksem v^k. Łatwo zauważyć, że ta trochę odmienna od intuicyjnej definicja jest jej równoważna: okres słowa to taki początkowy fragment słowa, którego kolejnymi powtórzeniami można pokryć całe słowo, przy czym ostatnie powtórzenie może być niepełne.

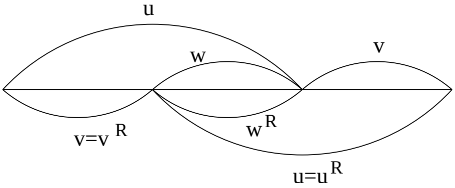
Rozwiązanie pierwsze

Na wstępie zauważmy, że zachodzi następująca równoważność:

Obserwacja 1 *Sklejenie dwóch palindromów jest palindromem wtedy i tylko wtedy, gdy krótszy z nich jest prefiksem dłuższego i słowo powstałe przez odcięcie krótszego palindromu od dłuższego jest także palindromem.*

Dowód Niech u i v będą palindromami. Zauważmy, że jeśli słowo uv jest palindromem, to $uv = (uv)^R = v^R u^R = vu$, więc uv można przedstawić zarówno jako sklejenie dłuższego palindromu z krótszym (w tej kolejności), jak i odwrotnie. Stąd bez straty ogólności możemy przyjąć, że $|u| \geqslant |v|$.

Równość $uv = vu$ możemy zilustrować następująco:



Rys. 1: Ilustracja równości $uv = vu$

Pozwala to nam dostrzec, że v rzeczywiście musi być prefiksem u, a ewentualna pozostała część u (oznaczona na rysunku przez w) musi być palindromem, gdyż musi spełniać $w = w^R$. Z rysunku widać także, że te dwie własności wystarczają do tego, aby sklejenie uv było palindromem. ■

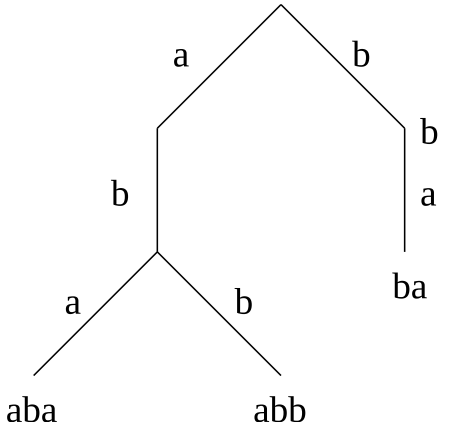
Przy okazji powyższego dowodu wykazaliśmy również, że w naszym zadaniu słowo uv jest palindromem wtedy i tylko wtedy, gdy vu jest palindromem, a zatem wystarczy wykonać co najwyżej jedno sprawdzenie dla każdej nieuporządkowanej pary słów.

Teraz pozostało nam jeszcze wymyślić, w jaki sposób zastosować poczynioną obserwację w konstrukcji efektywnego rozwiązania. W tym celu wykorzystamy kilka technik i struktur, często występujących w algorytmach tekstowych.

Drzewo TRIE

Definicja 1 *Drzewo typu TRIE* to drzewo z wyróżnionym korzeniem służące do zapisywania słów. Krawędzie tego drzewa są etykietowane literami alfabetu i powiemy, że słowo w jest zapisane w drzewie, jeśli idąc ścieżką od korzenia w dół drzewa możemy słowo w „odczytać” z etykiet krawędzi na tej ścieżce. Sposób zapisu słowa w drzewie musi być jednoznaczny. To znaczy, że kierując się kolejnymi literami słowa mamy zawsze tylko jedną możliwość zejścia w dół drzewa — z żadnego wierzchołka nie wychodzą dwie krawędzie etykietowane takimi samymi literami. Dodatkowo, jeśli zakładamy, że w drzewie są zapisane (tylko) słowa w_1, w_2, …, w_k, to ścieżki wyznaczone przez te słowa muszą pokryć całe drzewo TRIE. Warto zauważyć, że każde słowo prowadzi nas w drzewie do pewnego wierzchołka — może to być liść, może to być także wierzchołek wewnętrzny (w takim przypadku słowo na pewno jest prefiksem innego słowa zapisanego w drzewie).

Przykład drzewa TRIE dla słów: `aba`, `ba`, `b` i `abb` jest przedstawiony na rysunku 2.



Rys. 2: Przykładowe drzewo TRIE

Jeżeli łączna długość wszystkich słów znajdujących się w drzewie TRIE wynosi d, to wielkość tego drzewa możemy ograniczyć przez $O(d)$ — wynika to stąd, iż słowo długości d_i pokrywa co najwyżej $d_i + 1$ wierzchołków oraz d_i krawędzi.

Więcej o drzewach TRIE można przeczytać na przykład w książce [15].

Wstawienie nowego słowa do drzewa TRIE jest koncepcyjnie bardzo proste — trzeba, począwszy od korzenia, wędrować w dół drzewa po ścieżce odpowiadającej kolejnym literom słowa, dopóki istnieją odpowiednie krawędzie. Gdy ich zabraknie, trzeba utworzyć nowe tak, by słowo znalazło się w drzewie. Zauważmy, że podczas tego procesu napotykamy w węzłach końce wszystkich wstawionych wcześniej słów, które są prefiksami danego słowa. Stąd pomysł na algorytm. Posortujmy wszystkie zadane palindromy niemalejąco względem długości i w takim porządku wstawiajmy je do początkowo pustego drzewa TRIE. Jeżeli przy wstawianiu kolejnego słowa napotkamy wierzchołek oznaczający koniec wcześniej wstawionego palindromu i będziemy potrafili szybko (powiedzmy w czasie $O(1)$) sprawdzić, czy pozostała część wstawianego słowa jest palindromem, to w czasie $O(d)$ policzymy żądany wynik.

Algorytm Manachera

Zastanówmy się teraz, jak rozpoznać, czy sufiks zadanego słowa jest palindromem. Możemy w tym celu dla każdego wejściowego palindromu zastosować algorytm Manachera, który pozwala wyznaczyć dla każdej pozycji w słowie maksymalny promień podsłowa-palindromu o środku na tejże pozycji. Dokładniej, algorytm ten dla słowa $u = u_1u_2\dots u_k$ wyznacza:

- dla każdego $i \in \{1,\dots,k\}$ największą taką wartość $r_i \in N$, że słowo $u_{i-r_i}u_{i-(r_i-1)}\dots u_{i-1}u_iu_{i+1}\dots u_{i+r_i}$ jest palindromem nieparzystej długości (w tym wypadku środek palindromu wypada na pewnej literze słowa) oraz
- dla każdego $j \in \{1,\dots,k-1\}$ największy taki promień $R_j \in N$, że słowo $u_{j-(R_j-1)}\dots u_{j-1}u_ju_{j+1}\dots u_{j+R_j}$ jest palindromem parzystej długości (w tym wypadku środek palindromu wypada między dwiema kolejnymi literami słowa).

Dokładniejszy opis tego algorytmu można znaleźć m.in. w książce [15]. W tym miejscu wystarczy nam informacja, że jego złożoność jest liniowa względem liczby liter słowa, dla którego jest wykonywany (i jego kod jest stosunkowo krótki) — a zatem możemy go wykonać na wstępie dla wszystkich wejściowych słów i nie zwiększy to złożoności algorytmu. Posiadając wyniki działania algorytmu Manachera, sprawdzenie czy dany sufiks słowa jest palindromem możemy wykonać w czasie stałym, badając maksymalny promień palindromu w środku tego sufiksu.

Złożoność czasowa i pamięciowa

Zaprezentowany algorytm ma złożoność czasową $O(n\log n + d)$. Wydaje się, że (przynajmniej teoretycznie) złożoność pamięciowa również jest akceptowalna — główny wpływ ma na nią rozmiar drzewa TRIE, które, jak już ustaliliśmy, ma wielkość $O(d)$. Jednak w asymptotycznym oszacowaniu pomijamy pewien znaczący szczegół implementacyjny — dla każdego węzła drzewa musimy mieć możliwość zapisania krawędzi etykietowanej dowolną literą alfabetu wychodzącej z tego węzła i/lub sprawdzenia, czy taka krawędź już istnieje. Możemy w tym celu w każdym węźle utrzymywać tablicę indeksowaną literami, jednak wówczas faktyczny rozmiar drzewa mógłby osiągnąć wielkość około $26 \cdot d$. Zakładając, że rozmiar każdego wskaźnika wynosi 4 bajty i d może osiągnąć wartość $2\,000\,000$, otrzymujemy strukturę zajmującą $4 \cdot 26 \cdot 2\,000\,000B \approx 200MB$, co nie mieści się w limicie pamięciowym zadania ($128MB$).

Można spróbować inaczej zaimplementować wierzchołki drzewa:

- krawędzie wychodzące z wierzchołka można zapisać w postaci listy — w ten sposób mamy szansę zmieścić się w limicie pamięciowym, ale łatwo możemy przekroczyć limit czasowy, gdyż w poszukiwaniu odpowiedniej krawędzi musimy za każdym razem tę listę przeszukać,
- w każdym węźle można utworzyć zrównoważone drzewo poszukiwań binarnych (AVL, czerwono-czarne itp.) — taka struktura pozwala znaleźć odpowiednią krawędź w czasie logarytmicznym i nie powoduje znacznego zwiększenia pamięci — niestety stopień skomplikowania struktury i procedur wyszukiwania oraz aktualizacji utrudnia zmieszczenie się w limicie czasowym.

Z powyższych rozważań wynika, że przedstawiona metoda mogła sprawić wiele problemów implementacyjnych i trudno stosując ją zmieścić się jednocześnie w limicie czasowym i pamięciowym. Zatem takie rozwiązanie, mimo asymptotycznie takiej samej złożoności czasowej jak dalej omówione rozwiązanie wzorcowe, mogło nie uzyskać pełnej punktacji.

Uniwersalność rozwiązania pierwszego

Mimo istotnych wad, rozwiązanie pierwsze ma pewną przewagę nad pozostałymi zaprezentowanymi w tym opracowaniu. Jest ono zdecydowanie bardziej ogólne — można je zastosować także do słów wejściowych nie będących palindromami! W tym celu wystarczy sformułować spostrzeżenie podobne do obserwacji 1: „Sklejenie dwóch słów u i v, gdzie $|u| \geqslant |v|$, jest palindromem wtedy i tylko wtedy, gdy słowo v^R jest prefiksem u i słowo powstałe przez obcięcie v^R od u jest także palindromem”. Korzystając z niego możemy zmodyfikować przedstawione rozwiązanie dostosowując je do nowych warunków wejściowych (m.in. wstawiając do drzewa TRIE zarówno słowa, jak i ich odwrócenia) — dokładny zapis nowego algorytmu pozostawiamy Czytelnikowi jako ćwiczenie.

Rozwiązanie drugie

Stwierdziliśmy już, że w powyższym rozwiązaniu nie wykorzystujemy w istotny sposób faktu, że słowa wejściowe są palindromami; w niniejszym rozwiązaniu weźmiemy ten fakt pod uwagę i sformułujemy łatwiejsze (wymagające mniej wyrafinowanego algorytmu) kryterium sprawdzania, czy określony sufiks palindromu jest palindromem.

Obserwacja 2 *Sufiks palindromu jest palindromem wtedy i tylko wtedy, gdy jest także jego prefiksem.*

Dowód Niech u będzie palindromem, a v jego sufiksem. Pokażemy najpierw, że jeśli v jest palindromem, to jest także prefiksem u. Zapiszmy dany palindrom w następującej postaci: $u = wv$, gdzie w jest pewnym słowem. Stąd, ponieważ $u^R = u$, więc $(wv)^R = u$, czyli $v^R w^R = u$, a zatem $vw^R = u$. Sufiks v rzeczywiście jest także prefiksem u.

Teraz pokażemy, że prefikso-sufiks v palindromu u musi być palindromem. Jeżeli v jest prefikso-sufiksem u, to $u = wv = vw'$ dla pewnych słów w i w'. Stąd $u^R = (wv)^R = v^R w^R$, ale $u^R = u$, więc $v^R w^R = vw'$. Ponieważ słowa v i v^R mają tę samą długość, to z powyższego zapisu wnioskujemy, że $v = v^R$, czyli v jest palindromem. ■

Funkcja prefiksowa

Dzięki powyższej obserwacji sprawdzenie, czy sufiks palindromu jest palindromem redukuje się do sprawdzenia, czy jest on prefikso-sufiksem. Listę wszystkich prefikso-sufiksów słowa (a dokładniej ich długości) możemy uzyskać za pomocą funkcji prefiksowej p, stosowanej między innymi w algorytmie Knutha-Morrisa-Pratta (KMP). Przypomnijmy, że dla k-literowego słowa $u = u_1 \ldots u_k$ i indeksu $i \in \{1,\ldots,k\}$ definiujemy $p(i)$ jako długość najdłuższego właściwego (czyli krótszego od całego słowa) prefikso-sufiksu słowa $u_1 \ldots u_i$.

Najważniejsze z naszego punktu widzenia własności funkcji prefiksowej są następujące:

- Wszystkie wartości $p(i)$ dla $i \in \{1,\ldots,k\}$ można policzyć w złożoności czasowej $O(k)$.
- Długości wszystkich prefikso-sufiksów słowa u można uzyskać, wielokrotnie stosując funkcję prefiksową: k, $p(k)$, $p(p(k))$, ... (proces kontynuujemy, dopóki nie dojdziemy do zerowej długości prefikso-sufiksu).

Dowody własności funkcji prefiksowej i sposób jej obliczania można znaleźć m.in. w książkach: [18], [15] oraz w opisie rozwiązania zadania *Szablon* w [12].

Sformułowanie rozwiązania drugiego

Większa część rozwiązania drugiego jest identyczna jak rozwiązanie pierwsze — stosujemy w nim drzewo TRIE, do którego wstawiamy palindromy w kolejności niemalejących długości. W trakcie wstawiania, gdy napotykamy węzeł oznaczający koniec innego słowa, sprawdzamy, czy pozostała część słowa jest palindromem. I w tym miejscu pojawia się jedyna różnica pomiędzy rozwiązaniami — zamiast sprawdzać, czy pozostała po odcięciu krótszego palindromu część słowa wejściowego jest także palindromem, sprawdzamy za pomocą funkcji prefiksowej, czy jest to prefikso-sufiks. Wszystkie wartości funkcji prefiksowej dla wszystkich słów wejściowych obliczamy przed rozpoczęciem budowy drzewa, więc sprawdzenie wykonywane przy wstawianiu słowa do drzewa TRIE możemy wykonać w czasie stałym $O(1)$.

Złożoności czasowa i pamięciowa tego algorytmu są takie same jak w przypadku poprzedniego i, co ważniejsze, stosujemy w nim także trudne do efektywnego zaimplementowania drzewo TRIE. Zatem rozwiązanie to również może nie uzyskiwać pełnej punktacji. Ze względu na zastosowanie szerzej znanego algorytmu KMP zamiast algorytmu Manachera rozwiązanie to było łatwiejsze do wymyślenia i część uczestników III etapu olimpiady właśnie to rozwiązanie zaimplementowała.

Rozwiązanie wzorcowe

Rozpocznijmy od kolejnych obserwacji:

Obserwacja 3 *Niech u, v będą palindromami. Słowo uv jest palindromem wtedy i tylko wtedy, gdy u i v są potęgami tego samego słowa.*

Dowód Wiemy już, że jeśli u, v oraz uv są palindromami, to muszą zachodzić równości: $uv = (uv)^R = v^R u^R = vu$. Pokażemy, że równość $uv = vu$ może zachodzić tylko dla słów u i v, które są potęgą tego samego słowa. Dowód tego faktu przeprowadzimy przez indukcję względem sumy długości słów u i v.

- Baza indukcji: dla $|u| = |v| = 1$ obserwacja w sposób oczywisty zachodzi.
- Jeżeli $|u| = |v|$, to teza jest prawdziwa, gdyż $u = u^1$ oraz $v = u^1$. W przeciwnym przypadku, nie tracąc ogólności możemy założyć, że $|u| > |v|$. Wówczas z tego, że v jest prefiksem uv, mamy, że $u = vw$ dla pewnego słowa w. Zatem

  $$uv = vu \Leftrightarrow vwv = vvw \Leftrightarrow wv = vw.$$

  W ten sposób otrzymaliśmy równość analogiczną do wyjściowej, ale dla słów v i w, dla których $|v| + |w| = |v| + (|u| - |v|) < |u| + |v|$. Z założenia indukcyjnego wiemy więc, że $w = x^k$ oraz $v = x^l$ dla pewnego słowa x oraz liczb naturalnych k i l. Stąd $u = vw = x^{k+l}$ i $v = x^l$, co kończy dowód tezy indukcyjnej.

■

Jak sprawdzić ten warunek?

Otrzymaliśmy pewną charakteryzację poszukiwanych par palindromów. Zastanówmy się, jak ją weryfikować w praktyce. Zanim otrzymamy właściwą metodę, pokażemy najpierw pomocniczy

Lemat 4 (o okresowości) *Jeżeli p i q są długościami dwóch okresów słowa u, spełniającymi nierówność $p + q \leqslant |u|$, to $NWD(p,q)$ jest również długością okresu słowa u.*

Jest to tak zwana „słaba wersja” lematu o okresowości; istnieje wersja silna, w której zakładamy tylko, że $p+q-NWD(p,q) \leqslant |u|$ (dowód można znaleźć m.in. w książce [35]).

Dowód Dowód lematu przeprowadzimy przez indukcję względem sumy $p+q$.

- Baza indukcji: jeżeli $p=q=1$, to lemat oczywiście zachodzi.
- Jeżeli $p=q$, to teza jest oczywista. W przeciwnym przypadku, nie tracąc ogólności możemy założyć, że $p>q$. Pokażemy, że wówczas $p-q$ jest również długością okresu słowa u. Faktycznie, dla dowolnej pozycji $i \in \{1,\ldots,|u|\}$ w słowie u, takiej że $i+(p-q) \leqslant |u|$, zachodzi $i-q \geqslant 1$ lub $i+p \leqslant |u|$ (w przeciwnym przypadku byłoby $p+q>|u|$). W pierwszym z tych przypadków mamy $u_i = u_{i-q}$ (gdyż q jest długością okresu u), czyli $u_i = u_{i-q} = u_{i-q+p} = u_{i+(p-q)}$. W drugim podobnie zachodzi $u_i = u_{i+p} = u_{i+p-q} = u_{i+(p-q)}$, co pokazuje, że niezależnie od przypadku $p-q$ jest długością okresu słowa u.

  Korzystając teraz z założenia indukcyjnego mamy, że słowo u ma okres długości $NWD(p-q,q)$, a z własności największego wspólnego dzielnika wynika, że dla $p>q$ zachodzi równość $NWD(p-q,q) = NWD(p,q)$, czyli słowo ma okres długości $NWD(p,q)$, co należało pokazać.

■

Z lematu o okresowości wynika w szczególności, że jeżeli dane słowo u można przedstawić jako nietrywialną potęgę na dwa sposoby: $u = v^k = w^l$, to słowa v i w są potęgami tego samego słowa — oznaczmy je x. Wynika to stąd, że w takim przypadku zarówno v, jak i w są okresami słowa u, a zatem u ma także okres o długości $NWD(|v|,|w|)$. Ponieważ $NWD(|v|,|w|)$ dzieli $|v|$ i dzieli $|w|$, to prefiks u o długości $NWD(|v|,|w|)$ jest właśnie szukanym słowem x. Z przeprowadzonego rozumowania wynika także, iż dla słowa u istnieje najkrótsze słowo $\rho(u)$, takie że:

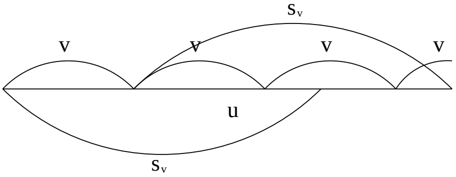
- u jest potęgą $\rho(u)$ oraz
- wszystkie inne słowa y, dla których $u = y^i$, są także potęgami $\rho(u)$.

Słowo $\rho(u)$ nazywamy *pierwiastkiem pierwotnym u*. Jeśli $u = \rho(u)$, to u nazywamy *słowem pierwotnym*.

Wreszcie wnioskiem z powyższej analizy jest pomysł na praktyczne zweryfikowanie, czy dwa słowa są potęgami tego samego słowa — wystarczy sprawdzić, czy mają te same pierwiastki pierwotne. Ostatnia (z pozoru dość zaskakująca) obserwacja wskaże nam metodę, dzięki której będziemy w stanie szybko znajdować pierwiastki pierwotne słów:

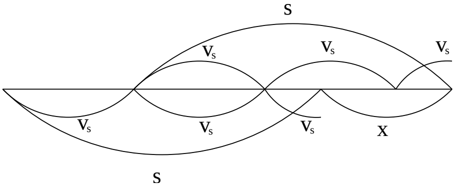
Obserwacja 5 *Niech p oznacza funkcję prefiksową słowa u, które ma długość k. Jeżeli $(k-p(k))|k$, to pierwiastek pierwotny słowa u ma długość $k-p(k)$. W przeciwnym przypadku słowo u jest pierwotne.*

Dowód Na początek pokażemy, że $k-p(k)$ jest długością najkrótszego okresu słowa u. W tym celu zauważmy, że istnieje wzajemnie jednoznaczna odpowiedniość między okresami słowa a jego prefikso-sufiksami. Faktycznie, jeżeli v jest okresem słowa u, to można dla niego zdefiniować prefikso-sufiks s_v, jako słowo pozostałe po odcięciu od słowa u początkowego wystąpienia v (rys. 3).



Rys. 3: Prefikso-sufiks odpowiadający okresowi słowa

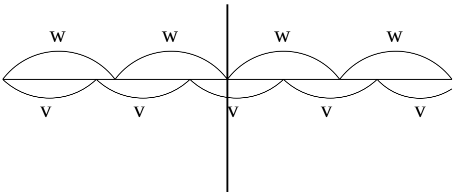
Z kolei niech teraz s będzie właściwym prefikso-sufiksem słowa u ($s \neq u$). Pokażemy, że prefiks v_s słowa u długości $|u| - |s|$ jest okresem u. Zauważmy, że wystarczy pokazać, że v_s jest okresem s (gdyż $v_s s = u$). Pokrycie słowa s wystąpieniami v_s generujemy iteracyjnie. Widzimy, że zachodzi równość $v_s s = sx$, gdzie x jest pewnym sufiksem słowa u o długości $|v_s|$. Wnosimy z niej, że v_s jest prefiksem słowa s, a zatem możemy zapisać: $v_s v_s s' = sx$, gdzie s' to słowo powstałe po obcięciu od s prefiksu v_s. Z tej równości z kolei widzimy, że $v_s v_s$ jest prefiksem słowa s, co oznacza, że zachodzi równość: $v_s v_s v_s s'' = sx$. Kontynuując to postępowanie pokrywamy w całości s powtórzeniami v_s, co pozwala wykazać, że v_s jest okresem prefikso-sufiksu s (rys. 4).



Rys. 4: Okres odpowiadający prefikso-sufiksowi słowa

Na podstawie właśnie pokazanej wzajemnie jednoznacznej odpowiedniości między okresami a prefikso-sufiksami słowa wnioskujemy, że najkrótszy okres słowa odpowiada najdłuższemu właściwemu prefikso-sufiksowi całego słowa. Ponieważ $p(k)$ to długość najdłuższego prefikso-sufiksu u, stąd $k - p(k)$ jest długością najkrótszego okresu u. Jeżeli teraz $(k - p(k))|k$, to u jest pewną potęgą tego okresu. W tym przypadku okres ten będzie oczywiście pierwiastkiem pierwotnym słowa u (gdyby istniał krótszy pierwiastek pierwotny słowa u, to byłby on zarazem krótszym okresem u). Jeżeli zaś ta podzielność nie zachodzi, to wykażemy, że słowo u jest pierwotne.

Dowód przeprowadzimy przez sprowadzenie do sprzeczności. Załóżmy nie wprost, że istnieje słowo w takie, że $u = w^j$ dla $j \geqslant 2$ — dodatkowo wybierzmy najkrótsze w o tej własności, czyli $w = \rho(u)$. Z kolei niech v_0 oznacza najkrótszy okres u. Ponieważ w także jest okresem u, więc musi zachodzić $|w| > |v_0|$, stąd słowo v_0 musi być również okresem słowa $w^2 = ww$ (rys. 5).



Rys. 5: Najkrótszy okres i pierwiastek pierwotny słowa

Wreszcie z tego, że $|w| + |v_0| \leqslant |w| + |w| = |w^2| \leqslant |u|$, na mocy lematu o okresowości widzimy, że słowo u musi mieć jeszcze inny okres — o długości $NWD(|w|, |v_0|)$ — a więc krótszy niż w i dodatkowo taki, że u jest jego potęgą. Jest to sprzeczne z tym, jak zdefiniowaliśmy w. ■

Sformułowanie rozwiązania wzorcowego

Wykorzystując wykazane własności słów możemy skonstruować następujący algorytm. Wyznaczmy pierwiastki pierwotne wszystkich danych palindromów — z obserwacji 5 wiemy, że stosując funkcję prefiksową możemy to zrobić w czasie proporcjonalnym do sumarycznej długości danych — i posortujmy je leksykograficznie: $\rho_1 \leqslant \rho_2 \leqslant \ldots \leqslant \rho_n$. Jeśli pierwiastek ρ_i występuje w tym ciągu m_i razy, to oznacza, że palindromy, z których powstał, pozwalają utworzyć $\frac{m_i(m_i-1)}{2}$ palindromów-par. Co więcej pary skomponowane z palindromów o różnych pierwiastkach pierwotnych nie mogą być palindromami. To pozwala nam prosto wyznaczyć wynik końcowy.

Łączna złożoność czasowa tego algorytmu zależy od algorytmu sortowania, jaki zastosujemy (resztę obliczeń możemy wykonać w czasie $O(d)$). Można wykorzystać sortowanie przez scalanie — jego pesymistyczna złożoność wynosi $O(n \cdot \max(a_i) \cdot \log n)$, gdzie $\max(a_i)$ oznacza maksimum z długości słów wejściowych. Stosując sortowanie szybkie uporządkujemy słowa w takim samym czasie oczekiwanym. Można też sortować słowa za pomocą sortowania pozycyjnego; co prawda jest ono oryginalnie sformułowane dla słów równej długości, lecz można je zmodyfikować i zastosować także w przypadku dowolnych słów przy zachowaniu złożoności równej sumie długości wszystkich słów ($O(d)$). W praktyce dla danych testowych wymienione metody sortowania były czasowo nieodróżnialne i zastosowanie dowolnej z nich wystarczało do przejścia wszystkich testów.

Inne rozwiązania

Poza trzema opisanymi sposobami, istnieje bardzo dużo innych rozwiązań zadania *Palindromy*. Zawodnicy prezentowali zazwyczaj albo rozwiązania siłowe, które polegały na tworzeniu wszystkich par słów i sprawdzaniu wprost, czy otrzymane słowo jest palindromem, albo różne wersje powyższych rozwiązań, w których pewne kroki były wykonywane wolniej (na przykład sprawdzanie, czy sufiks słowa jest palindromem lub wyznaczanie pierwiastka pierwotnego słowa).

Testy

Zadanie testowane było na 10 zestawach danych testowych, zawierających łącznie 23 pojedyncze testy.

Nazwa	**n**	**d**	**Opis**
pal1.in	15	78	mały test poprawnościowy
pal2a.in	36566	253168	palindromy zaczynające się na litery `a` i `b`
pal2b.in	2	2000000	dwa palindromy: jeden jednoliterowy, drugi o maksymalnej długości

Nazwa	**n**	**d**	**Opis**
pal3a.in	893	1 995 855	potęgi krótkiego słowa; test ten pozwala wyeliminować sprytne rozwiązania siłowe
pal3b.in	33 290	1 999 959	test losowy, składający się z dużej liczby krótkich palindromów, słów postaci a^kba^k i słowa o długim pierwiastku pierwotnym
pal4a.in	999	1 998 000	potęgi krótkiego słowa; test ten pozwala wyeliminować sprytne rozwiązania siłowe
pal4b.in	98 391	1 999 981	test losowy, składający się z dużej liczby krótkich palindromów, słów postaci a^kba^k i słowa o długim pierwiastku pierwotnym
pal5a.in	1 154	1 999 305	potęgi krótkiego słowa; test ten pozwala wyeliminować sprytne rozwiązania siłowe
pal5b.in	127 351	1 999 997	test losowy, składający się z dużej liczby krótkich palindromów, słów postaci a^kba^k i słowa o długim pierwiastku pierwotnym
pal6a.in	1 413	1 997 982	potęgi krótkiego słowa; test ten pozwala wyeliminować sprytne rozwiązania siłowe
pal6b.in	38 827	1 999 999	test losowy, składający się z dużej liczby krótkich palindromów, słów postaci a^kba^k i słowa o długim pierwiastku pierwotnym
pal7a.in	1 413	1 997 982	potęgi krótkiego słowa; test ten pozwala wyeliminować sprytne rozwiązania siłowe
pal7b.in	131 775	1 999 998	test losowy, składający się z dużej liczby krótkich palindromów, słów postaci a^kba^k i słowa o długim pierwiastku pierwotnym
pal8a.in	1 413	1 997 982	potęgi krótkiego słowa; test ten pozwala wyeliminować sprytne rozwiązania siłowe
pal8b.in	37 969	1 999 999	test losowy, składający się z dużej liczby krótkich palindromów, słów postaci a^kba^k i słowa o długim pierwiastku pierwotnym
pal8c.in	105 264	2 000 000	długie słowa powodujące maksymalne zużycie pamięci dla drzewa TRIE
pal9a.in	1 999	1 999 000	potęgi krótkiego słowa; test ten pozwala wyeliminować sprytne rozwiązania siłowe
pal9b.in	80 714	2 000 000	krótkie palindromy
pal9c.in	222 222	1 999 998	długie słowa powodujące maksymalne zużycie pamięci dla drzewa TRIE

Nazwa	**n**	**d**	**Opis**
pal9d.in	126	1 996 220	test pozwalający wyeliminować rozwiązania, w których pierwiastki pierwotne są obliczane nieefektywnie
pal10a.in	2 047	2 000 000	największy test, składający się wyłącznie z potęg słów jednoliterowych
pal10b.in	224 200	2 000 000	długie słowa powodujące maksymalne zużycie pamięci dla drzewa TRIE
pal10c.in	108	1 880 780	test pozwalający wyeliminować rozwiązania, w których pierwiastki pierwotne są obliczane nieefektywnie

Łukasz Kowalik
Treść zadania, Opracowanie

Krzysztof Dulęba
Program

OI, Etap III, dzień drugi, 30.03.2006

Zosia

Mała Zosia urządza przyjęcie urodzinowe. Sporządziła wstępną listę n swoich znajomych z przedszkola, których chciałaby zaprosić. Dzieci są jednak bardzo wymagające. Maja powiedziała, że przyjdzie, ale tylko jeśli na przyjęciu nie będzie Kamilki i Emilki, które w zeszłym tygodniu zabrały jej lalkę. Mały Krzyś bawi się tylko z Zosią oraz Kamilką, i nie chce widzieć na przyjęciu innych dzieci. I tak dalej …

Zosia mówi, że przyjęcie jest **udane**, *jeżeli żaden spośród gości nie ma nic przeciwko obecności pozostałych gości. Zosia postanowiła, że nie zaprosi niektórych dzieci, aby przyjęcie było udane. Z drugiej strony, Zosia chciałaby zaprosić jak najwięcej dzieci. Uznała, że jeśli nie będzie mogła zaprosić przynajmniej k dzieci, to w ogóle nie urządzi przyjęcia.*

Zadanie

Pomóż małej Zosi! Napisz program, który:

- *wczyta ze standardowego wejścia liczbę n wszystkich znajomych Zosi, liczbę k oraz opis wymagań dzieci,*
- *sprawdzi, czy można zaprosić co najmniej k dzieci tak, aby przyjęcie było udane,*
- *jeżeli nie jest to możliwe, to wypisze na standardowe wyjście słowo* `NIE`*; jeżeli jest to możliwe, to znajdzie i wypisze na standardowe wyjście najliczniejszą grupę dzieci, które można zaprosić na przyjęcie, tak aby było ono udane.*

Wejście

Pierwszy wiersz standardowego wejścia zawiera dwie liczby całkowite nieujemne, oddzielone pojedynczym odstępem: n — liczbę wszystkich znajomych Zosi ($2 \leqslant n \leqslant 1\,000\,000$), k — minimalną liczbę dzieci, które Zosia chce zaprosić na przyjęcie ($n-10 \leqslant k < n$). Dzieci są ponumerowane liczbami od 1 do n.

Kolejne wiersze zawierają opis wymagań dzieci. W drugim wierszu znajduje się jedna liczba całkowita m, $1 \leqslant m \leqslant 3\,000\,000$. Każdy z kolejnych m wierszy zawiera parę liczb całkowitych a,b, oddzielonych pojedynczym odstępem ($1 \leqslant a,b \leqslant n$, $a \neq b$). Możesz założyć, że każda para (uporządkowana) pojawia się na wejściu co najwyżej raz. Para a,b oznacza, że dziecko o numerze a nie chce spotkać na przyjęciu dziecka o numerze b.

Wyjście

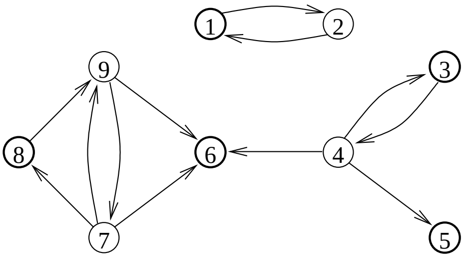
Jeżeli nie jest możliwe zaproszenie na przyjęcie k dzieci, tak aby było ono udane, to pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać tylko jedno słowo: `NIE`.

Jeżeli jest to możliwe, to pierwszy wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą — maksymalną liczbę dzieci, które można zaprosić na przyjęcie, tak aby było *ono udane. Drugi wiersz standardowego wyjścia powinien wówczas zawierać numery dzieci, które należy zaprosić, podane w rosnącej kolejności i pooddzielane pojedynczymi odstępami. Jeżeli istnieje wiele poprawnych wyników, Twój program powinien wypisać dowolny z nich.*

Przykład

Dla danych wejściowych:

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



poprawnym wynikiem jest:

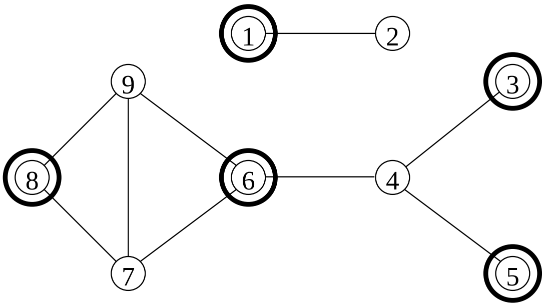
```
5
1 3 5 6 8
```

Rozwiązanie

Zosia i grafy

Przyjrzyjmy się rysunkowi z treści zadania. Antypatie wśród przyjaciół Zosi przedstawiono na nim za pomocą grafu skierowanego. *Graf skierowany* to zbiór obiektów (w tym przypadku numerów przyjaciół Zosi) połączonych strzałkami. Strzałka od a do b oznacza, że dziecko o numerze a nie chce spotkać na przyjęciu dziecka o numerze b. Zauważmy od razu, że z punktu widzenia postawionego zadania kierunek strzałki nie ma tu żadnego znaczenia – spośród dwóch osób połączonych strzałką możemy zaprosić tylko jedną, niezależnie od tego, kto kogo nie lubi. Przypomnijmy jeszcze, że w teorii grafów połączenia między obiektami nazywamy *krawędziami*, a same obiekty *wierzchołkami*. Gdy krawędzie nie mają kierunku (możemy je przedstawiać graficznie za pomocą zwykłych linii), mamy do czynienia z grafem nieskierowanym.

Zdefiniujmy *graf antypatii* jako graf nieskierowany, którego wierzchołkami są numery dzieci oraz wierzchołek a jest połączony krawędzią z wierzchołkiem b, jeżeli dziecko o numerze a nie chce widzieć na przyjęciu dziecka o numerze b lub odwrotnie, dziecko o numerze b nie chce widzieć na przyjęciu dziecka o numerze a. Zauważmy, że graf antypatii może różnić się od grafu z treści zadania, w którym po prostu zamieniono strzałki na linie — po takiej prostej modyfikacji niektóre wierzchołki mogą być połączone podwójnie, a takiej sytuacji nie dopuszczamy w grafie antypatii. Na rysunku 1 przedstawiono graf antypatii dla przykładu z treści zadania.



Rys. 1: Przykładowy graf antypatii i zbiór niezależny

Zosia buduje graf w pamięci komputera

Zajmiemy się teraz zagadnieniem budowy grafu antypatii — zastanowimy się, w jaki sposób można go zbudować w pamięci komputera na podstawie danych wejściowych dla zadania. Czytelnicy znający dobrze algorytmy grafowe mogą pominąć ten rozdział i przeskoczyć do następnego punktu.

Najprostszym sposobem przechowywania informacji o grafie w pamięci jest wykorzystanie kwadratowej tablicy `T` zawierającej zera i jedynki. Jedynka w komórce `T[a,b]` oznacza, że wierzchołki o numerach `a` i `b` są połączone krawędzią, zero — że takiej krawędzi nie ma. Tablicę `T` można bardzo łatwo wypełnić na podstawie danych wejściowych, ma ona jednak wielką wadę: zajmuje zbyt dużo miejsca! Zwróćmy uwagę, że Zosia z zadania jest bardzo towarzyską dziewczynką i może mieć nawet 10^6 przyjaciół z przedszkola, a więc cała tablica miałaby 10^{12} (bilion!) komórek.

Alternatywną metodą przechowywania grafów w pamięci są listy sąsiedztwa. W tej reprezentacji dla każdego wierzchołka tworzymy listę numerów jego sąsiadów. List jest tyle co wierzchołków, a każdej krawędzi grafu odpowiadają dokładnie 2 elementy list sąsiedztwa, a więc wszystkie listy zajmują w pamięci komputera $O(m+n)$ komórek.

Teraz zastanówmy się, jak utworzyć listy sąsiedztwa na podstawie danych wejściowych. Sprawa wydaje się prosta: po przeczytaniu wiersza z parą liczb a i b, dodajemy a do listy wierzchołka b oraz b do listy wierzchołka a. Jest tylko jeden problem: niektóre krawędzie grafu antypatii zapiszemy podwójnie (np. krawędź łączącą 1 i 2 w przykładzie z zadania). Aby tego uniknąć, moglibyśmy, wstawiając a do listy wierzchołka b, sprawdzać, czy a już na tej liście jest. Niestety zajmuje to czas proporcjonalny do długości listy. Gdyby jakiś wierzchołek miał wielu sąsiadów — na przykład był połączony ze wszystkimi pozostałymi $n-1$ wierzchołkami — to takie sprawdzenia zajęłyby czas proporcjonalny do $1+2+\ldots+(n-3)+(n-2)$, czyli $\Theta(n^2)$.

Posłużymy się innym fortelem. Utworzymy listy sąsiedztwa w zwykły sposób, a następnie usuniemy z nich podwójne krawędzie. Aby łatwo takie podwójne krawędzie znaleźć, wystarczy posortować każdą z list sąsiedztwa — wtedy podwójne krawędzie będą występować obok siebie. W językach C/C++ wystarczy użyć jednej z funkcji sortujących ze standardowych bibliotek. Funkcje te są skonstruowane najczęściej w oparciu o algorytm szybkiego sortowania (*Quicksort*), który działa w oczekiwanym czasie $O(k \log k)$ dla ciągu k-elementowego. Sortowanie wszystkich ciągów, których całkowita długość wynosi $2m$, a maksymalna długość jest ograniczona przez $n-1$, zajmie czas $O(\sum_{i=1}^{n} \deg(i) \log \deg(i))$, gdzie $\deg(i)$ oznacza stopień wierzchołka i, czyli liczbę jego sąsiadów i równocześnie długość jego listy sąsiedztwa. Stąd otrzymujemy: $\sum_{i=1}^{n} \deg(i) \log \deg(i) \leqslant \sum_{i=1}^{n} \deg(i) \log(n-1) = \log(n-1) \sum_{i=1}^{n} \deg(i) = 2m \log(n-1)$ i dowiadujemy się, że całkowity (oczekiwany) czas sortowania wynosi $O(m \log n)$.

Powyższa metoda jest dostatecznie dobra dla ograniczeń z rozwiązywanego zadania, można jednak w prosty sposób posortować listy sąsiedztwa w czasie liniowym. Z tego sposobu mogą także skorzystać osoby programujące w Pascalu, które nie mają dostępu do gotowych funkcji sortujących. Użyjemy sortowania kubełkowego — w tym celu potrzebna będzie dodatkowa tablica list `kubełki[1..n]`. Sam algorytm działa następująco:

1. Utwórz tablicę `kubełki[1..n]` zawierającą puste listy.
2. Dla każdego wierzchołka i, od 1 do n, wykonuj, dopóki lista sąsiedztwa i jest niepusta:
    (a) Usuń wierzchołek z listy sąsiedztwa i. Niech j będzie numerem tego wierzchołka.
    (b) Wstaw i do listy `kubełki[`j`]`.
3. Dla każdego j, od 1 do n, wykonuj, dopóki lista `kubełki[`j`]` jest niepusta:
    (a) Usuń element z listy `kubełki[`j`]`. Niech i będzie tym elementem.
    (b) Wstaw j do listy sąsiedztwa wierzchołka i.

Fakt, że listy sąsiedztwa po wykonaniu algorytmu są posortowane, wynika z kolejności wstawiania elementów do tych list w punkcie 3. Zwróćmy uwagę, że w powyższym algorytmie sortujemy wszystkie listy równocześnie! Sortowanie kubełkowe każdej listy z osobna zajęłoby znacznie więcej czasu.

Zosia i trudne problemy

Dowolny podzbiór wierzchołków grafu, z których żadne dwa nie są połączone krawędzią, nazywamy *zbiorem niezależnym*. Przykładowy zbiór niezależny przedstawiono na rysunku 1 (wierzchołki zbioru niezależnego wyróżniono dodatkową obwódką). Zauważmy, że w naszym zadaniu należy znaleźć zbiór niezależny rozmiaru co najmniej k.

Wprowadźmy jeszcze jedno pojęcie. *Pokryciem wierzchołkowym* grafu nazywamy dowolny podzbiór C wierzchołków grafu taki, że każda krawędź grafu ma co najmniej jeden z końców w zbiorze C (mówimy, że krawędź jest pokryta przez ten wierzchołek). Spójrzmy ponownie na rysunek 1. Wierzchołki niezaznaczone tworzą pokrycie wierzchołkowe. Czy to tylko przypadek? Rozważmy dowolny graf G o zbiorze wierzchołków V i dowolny zbiór niezależny N w grafie G. Wybierzmy dowolną krawędź w grafie G. Z definicji zbioru niezależnego wiemy, że co najmniej jeden z jej końców jest poza N — innymi słowy co najmniej jeden z jej końców należy do zbioru $V \setminus N$. Stąd wynika, że $V \setminus N$ jest pokryciem wierzchołkowym. To samo rozumowanie można przeprowadzić w drugą stronę. Otrzymamy wtedy następujący fakt:

Fakt 1 *W dowolnym grafie G o zbiorze wierzchołków V podzbiór wierzchołków N jest zbiorem niezależnym wtedy i tylko wtedy, gdy zbiór* $V \setminus N$ *jest pokryciem wierzchołkowym.*

Z powyższego faktu płynie wniosek, że nasze zadanie można sformułować także następująco: „znajdź pokrycie wierzchołkowe rozmiaru co najwyżej $n-k$”. Widzimy jasno, że problem znajdowania zbioru niezależnego (pierwsze sformułowanie zadania) jest równoważny problemowi znajdowania pokrycia wierzchołkowego (drugie sformułowanie), tzn. jeśli mamy efektywny sposób rozwiązywania pierwszego problemu, to umiemy też rozwiązywać problem drugi, i odwrotnie.

Problemy zbioru niezależnego i pokrycia wierzchołkowego wydają się tak naturalne, że byłoby bardzo dziwne, gdyby zostały postawione po raz pierwszy na tegorocznej Olimpiadzie Informatycznej. Istotnie, znajdziemy je łatwo w podręcznikach algorytmiki (np. problem pokrycia wierzchołkowego jest opisany w książce Cormena i innych [18]; zamiast problemu zbioru niezależnego jest tam opisany mocno z nim związany problem kliki). Dowiemy się stamtąd, że oba te problemy należą do grupy tzw. problemów *NP-zupełnych*. Nie będziemy tutaj dokładnie opisywać, co to pojęcie oznacza. Wspomnijmy jedynie, że dla żadnego z tych problemów nie znaleziono dotąd algorytmu działającego w czasie wielomianowym (tzn. w czasie $O(n)$, $O(n^3)$, $O(n^{15})$ ani w czasie $O(n^c)$, gdzie c jest dowolną stałą). Co więcej, gdyby ktoś rozwiązał dowolny problem NP-zupełny w czasie wielomianowym, to takiego algorytmu można użyć do rozwiązania w czasie wielomianowym dowolnego innego problemu NP-zupełnego. Ponieważ dotąd znaleziono mnóstwo problemów NP-zupełnych w wielu różnych działach informatyki, a pomimo intensywnych badań ciągle dla nich nie wymyślono algorytmów wielomianowych, więc wydaje się mało prawdopodobne, żeby w końcu taki algorytm został odkryty.

Zosia upraszcza zadanie

Na szczęście w treści zadania znajduje się informacja, która znacznie upraszcza nasz problem. Mianowicie, Zosia godzi się na nie zaproszenie jedynie kilku spośród swoich przyjaciół, bowiem $k \geqslant n-10$ — oznaczmy liczbę tych pechowców przez $l = n-k$. Innymi słowy, zadanie polega na sprawdzeniu, czy graf antypatii zawiera pokrycie wierzchołkowe rozmiaru l, gdzie $l \leqslant 10$. Naiwny algorytm wymagający przejrzenia i przetestowania wszystkich podzbiorów zbioru wierzchołków, które mają nie więcej niż 10 elementów, działa w czasie $O(\sum_{i \leqslant 10} \binom{n}{i}) = O(n^{10})$, a więc wielomianowym! Taka złożoność jest dla nas jednakże zupełnie nieakceptowalna. Aby uzyskać lepsze rozwiązanie, w inny sposób wykorzystamy fakt, że poszukiwane pokrycie wierzchołkowe jest bardzo małe.

Algorytm I: Zosia szybko się zniechęca

Przypomnijmy, że poszukujemy pokrycia wierzchołkowego rozmiaru co najwyżej l. Rozważmy dowolną krawędź uv grafu antypatii. Jeden z jej końców musi znaleźć się w pokryciu. Możemy więc usunąć z grafu wierzchołek u wraz z wszystkimi jego krawędziami, a następnie rekurencyjnie rozwiązać zadanie dla tak otrzymanego mniejszego grafu, poszukując pokrycia wierzchołkowego rozmiaru co najwyżej $l-1$. Jeśli okaże się, że takie pokrycie istnieje, wystarczy dodać do niego wierzchołek u, aby otrzymać pokrycie całego grafu. Jeśli natomiast nie istnieje, ponownie wstawiamy do grafu wierzchołek u i wszystkie uprzednio usunięte krawędzie. Następnie usuwamy wierzchołek v wraz z przyległymi krawędziami i postępujemy analogicznie, jak dla wierzchołka u. Jeśli i w tym przypadku nie udało się znaleźć pokrycia, oznacza to, że w oryginalnym grafie nie ma pokrycia rozmiaru co najwyżej l — odpowiedź brzmi więc „NIE”.

Opisany powyżej algorytm wydaje się mało odkrywczy: jest to tylko pewien sposób sprawdzania wszystkich możliwości. Zauważmy jednak, że przy każdym wywołaniu rekurencyjnym ograniczenie na rozmiar pokrycia zmniejsza się o 1. W chwili, gdy to ograniczenie osiągnie wartość 0, wystarczy sprawdzić, czy graf zawiera *puste* pokrycie wierzchołkowe, a to ma miejsce jedynie wówczas, gdy nie ma w nim już żadnych krawędzi. Taki test można oczywiście łatwo wykonać w czasie liniowym. Całe drzewo wywołań rekurencyjnych jest więc stosunkowo płytkie — ma co najwyżej $l+1$ poziomów, a więc zawiera nie więcej niż 2^l wywołań procedury rekurencyjnej. Ponieważ usuwanie wierzchołka, podobnie jak sprawdzenie, czy zbiór pusty jest pokryciem, zajmuje czas $O(n)$, a więc całkowity czas działania algorytmu to $O(n2^l)$ — pomijając wstępną fazę tworzenia grafu antypatii. Trzeba przyznać, że to znaczna poprawa w stosunku do wcześniejszych rozwiązań. Rozmiar grafu (liczbę wierzchołków n) udało się bowiem usunąć w złożoności z funkcji wykładniczej — nie występuje ona już ani w podstawie, ani w wykładniku.

Algorytm II: Zosia nie zaprasza największych urwisów

Zosia dobrze wie, że na przyjęcie nie należy zapraszać największych urwisów, bo pobiją inne dzieci lub w najlepszym razie zabiorą im zabawki. Podobnie, wierzchołek stopnia większego niż l (czyli taki, który jest połączony z więcej niż l wierzchołkami) musi należeć do pokrycia wierzchołkowego, jeśli rozmiar tego pokrycia nie może przekraczać l. Gdyby bowiem nie należał, to aby pokryć wszystkie wychodzące z niego krawędzie, należałoby wybrać wszystkich jego sąsiadów, a tych jest więcej niż l. Dlatego możemy rozpocząć algorytm od usunięcia z grafu wszystkich wierzchołków stopnia większego niż l (powiedzmy, że jest ich m), zapisując na boku ich numery, by potem dodać je do pokrycia oryginalnego grafu. Jeśli wierzchołków do usunięcia jest więcej niż l, to oczywiście musimy odpowiedzieć „NIE” i możemy zakończyć pracę. W przeciwnym przypadku w okrojonym grafie szukamy pokrycia rozmiaru l', gdzie $l' = l - m$. Poczynimy teraz kluczową, choć bardzo prostą, obserwację:

Fakt 2 *Jeśli graf G ma wierzchołki stopni co najwyżej l oraz zawiera pokrycie wierzchołkowe rozmiaru co najwyżej l', to liczba krawędzi w G nie przekracza $l \cdot l'$.*

Ponieważ każda krawędź, podobnie jak każdy kij, ma dwa końce, otrzymujemy następujący wniosek.

Wniosek 3 *Liczba wierzchołków o niezerowym stopniu w grafie G nie przekracza $2ll'$.*

Możemy wykorzystać powyższy wniosek w zaproponowanym poprzednio algorytmie. Jeśli po usunięciu wierzchołków stopnia większego niż l zobaczymy, że liczba wierzchołków o niezerowym stopniu przekracza $2ll'$, to możemy śmiało odpowiedzieć „NIE” i zakończyć działanie algorytmu. Jeśli nie przekracza, zadanie nie jest jeszcze rozwiązane, ale rozmiar badanego grafu drastycznie się zmniejszył – jest ograniczony przez $2l^2$. Zastanówmy się jeszcze, jakim kosztem została osiągnięta taka redukcja, czyli jaka jest złożoność czasowa procesu wykrywania „urwisów”. Aby wyznaczyć stopnie wszystkich wierzchołków, dla każdego wierzchołka zliczamy przylegające do niego krawędzie. Czas tej operacji można podzielić na czas zerowania liczników, czyli $O(n)$, i czas powiększania liczników, czyli $O(m)$ – razem $O(n+m)$. Choć usuwanie jednego wierzchołka stopnia większego niż l może zająć nawet czas $\Theta(n)$, to zauważmy, że usuwanie wszystkich takich wierzchołków zajmuje czas $O(n+m)$, gdyż każdy wierzchołek i każda krawędź jest usuwana co najwyżej raz. Wreszcie, zliczanie wierzchołków o niezerowym stopniu trwa jedynie $O(n)$. A więc cała pierwsza faza algorytmu zajmuje jedynie czas liniowy — $O(n+m)$. Co więcej, czas drugiej fazy będzie zależał już tylko od l.

Jeśli w drugiej fazie algorytmu użyjemy naiwnego podejścia, czyli sprawdzania wszystkich podzbiorów zbioru wierzchołków rozmiaru co najwyżej l, czas drugiej fazy będzie ograniczony przez $O(\binom{2l^2}{l})$. Najlepszą złożoność czasową można oczywiście uzyskać, łącząc pomysły z algorytmów I i II. Dostajemy w ten sposób rozwiązanie o złożoności czasowej $O(n+m+l^2 2^l)$.

Szukajcie jądra problemu!

Podejście zaprezentowane w tym opracowaniu jest ogólną metodą „radzenia” sobie z problemami NP-zupełnymi. Polega ono na zmodyfikowaniu problemu poprzez wprowadzenie dodatkowego parametru — w naszym przypadku jest to rozmiar poszukiwanego pokrycia wierzchołkowego. Następnie próbujemy skonstruować taki algorytm, którego złożoność zależy w sposób wykładniczy jedynie od wprowadzonego parametru, a nie od rozmiaru danych wejściowych. W przypadku algorytmów grafowych, z reguły cel ten osiągamy szybko redukując rozwiązywanie problemu w oryginalnym grafie do badania mniejszego grafu, którego rozmiar zależy już tylko od wartości parametru. Metoda ta nazywana jest „redukcją do jądra problemu”.

Testy

Rozwiązania były testowane na zestawie 27 testów połączonych w 10 grup. Zawodnik otrzymywał punkty za testy w danej grupie jedynie wtedy, gdy jego program zwrócił dostatecznie szybko poprawne rozwiązania dla wszystkich testów z grupy. W każdej grupie znajdował się przynajmniej jeden test, w którym poszukiwane pokrycie wierzchołkowe istnieje, oraz przynajmniej jeden trudny przypadek, pozwalający wykryć niepoprawne algorytmy, polegające na wyborze do pokrycia wierzchołkowego l kolejnych wierzchołków o największych stopniach. Duże testy były tworzone w sposób losowy, poprzez wybranie kilku specjalnych wierzchołków „centralnych” połączonych z prawie wszystkimi pozostałymi wierzchołkami.

Oto charakterystyka plików testowych:

Nazwa	n	k	m	Opis
zos1.in	7	1	9	test wygenerowany ręcznie

Nazwa	**n**	**k**	**m**	**Opis**
zos2.in	14	5	36	test wygenerowany ręcznie
zos3.in	7	2	21	test wygenerowany ręcznie
zos4.in	51	42	9	test wygenerowany ręcznie
zos5a.in	102	93	323	test losowy
zos5b.in	3	1	6	test losowy
zos5c.in	11	3	47	test losowy
zos6a.in	70 000	69 990	349 884	test wygenerowany ręcznie z dodanym centrum
zos6b.in	769 341	769 331	2 461 243	test losowy
zos6c.in	775 231	775 223	2 481 305	test losowy
zos7a.in	70 000	69 990	279 942	test wygenerowany ręcznie z dodanym centrum
zos7b.in	1 000 000	999 995	3 000 000	maksymalny test z odpowiedzią NIE
zos7c.in	679 109	679 101	2 173 981	test losowy
zos7d.in	1 000 000	999 990	986 713	graf z pokryciem wielkości 9 oraz jeden wierzchołek połączony z prawie wszystkimi innymi
zos8a.in	70 000	69 990	279 925	test wygenerowany ręcznie z dodanym centrum
zos8b.in	277 114	277 105	1 773 719	test losowy
zos8c.in	824 849	824 842	2 638 722	test losowy
zos8d.in	900 000	899 990	70	cykl długości 70; odpowiedź NIE
zos9a.in	70 000	69 990	139 957	test wygenerowany ręcznie z dodanym centrum
zos9b.in	618 922	618 913	2 971 106	test losowy
zos9c.in	950 357	950 347	3 000 000	test losowy
zos9d.in	120	110	35	każda spójna składowa jest pojedynczą krawędzią (odpowiedź NIE)
zos10a.in	70 000	69 990	209 941	test wygenerowany ręcznie z dodanym centrum
zos10b.in	678 692	678 688	2 172 173	test losowy
zos10c.in	423166	423 156	2 709 748	test losowy
zos10d.in	900 000	899 990	70	cykl długości 70; odpowiedź NIE

Maciej Jaśkowski
Treść zadania, Opracowanie

Szymon Acedański
Program

OI, Etap III, dzień drugi, 30.03.2006

Misie

Bajtocka firma 0101010 produkuje zabawki dla dzieci. 0101010 jest bardzo znaną firmą, a ich zabawki mają opinię bardzo solidnych. Pracownicy firmy z przerażeniem stwierdzili, że ostatnie cztery modele misiów: A1, A2, B1 i B2 mają ukrytą wadę: jeśli weźmiemy trzy misie, które wszystkie mają tę samą literę w oznaczeniu modelu, lub wszystkie mają tę samą cyfrę w oznaczeniu modelu i ustawimy je obok siebie w rzędzie, to misie ulegną nieodwracalnemu uszkodzeniu.

Ustawienie misiów w rzędzie nazwiemy bezpiecznym, jeśli w jego wyniku żaden miś nie ulegnie uszkodzeniu, tzn. żadne trzy kolejne misie nie będą wszystkie miały tej samej litery w oznaczeniu modelu, ani tej samej cyfry.

Bajtazar ma kolekcję misiów, w której znajdują się tylko feralne modele. Bajtazar bawi się misiami ustawiając je w rzędzie. Zastanawia się, ile jest możliwych bezpiecznych ustawień misiów. Napisz program, który pomoże mu to ustalić.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia liczbę misiów każdego modelu,*
- *obliczy liczbę bezpiecznych ustawień misiów w rzędzie, modulo 1 000 000,*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym i jedynym wierszu wejścia znajdują się cztery nieujemne liczby całkowite: n_{A1}, n_{A2}, n_{B1}, n_{B2}, *oddzielone pojedynczymi odstępami* ($0 \leqslant n_{A1}, n_{A2}, n_{B1}, n_{B2} \leqslant 38$). *Oznaczają one liczbę misiów, odpowiednio modelu A1, A2, B1 i B2. Możesz założyć, że sumaryczna liczba misiów jest dodatnia.*

Wyjście

W pierwszym i jedynym wierszu wyjścia Twój program powinien wypisać liczbę dobrych ustawień misiów w rzędzie modulo 1 000 000.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`0 1 2 1`	`6`

Istnieje 6 poprawnych ustawień misiów: B1 A2 B1 B2, B1 A2 B2 B1, B2 A2 B1 B1, B2 B1 A2 B1, B1 B2 A2 B1 oraz B1 B1 A2 B2.

Rozwiązanie

Wprowadzenie

Pytanie przedstawione w zadaniu jest modyfikacją problemu, który spotyka się w ... badaniach psychologicznych. Na przykład człowiek mający reagować w doświadczeniu na bodziec L naciskając przycisk lewą ręką, zaś na bodziec P — prawą, ma tendencję do wykorzystywania pozornych regularności do upraszczania sobie zadania — gdy trzy razy z rzędu powtórzy się bodziec L, to za czwartym razem najpewniej również zareaguje lewą ręką, nawet jeśli pojawi się bodziec P.

Psychologowie zwykle chcą unikać takich reakcji i stąd wzięło się zadanie. Oryginalny problem polegał na wygenerowaniu losowego ciągu spełniającego założenia takie, jak w zadaniu. Czytelnik bez trudu zmodyfikuje rozwiązanie wzorcowe tak, aby generowało żądany ciąg.

Oznaczenia i spostrzeżenia

Niech $T = \{A1, A2, B1, B1\}$.

Definicja 1 Uporządkowaną trójkę misiów $(p,q,r) \in T^3$ nazwiemy *trójką bezpieczną*, jeśli ustawienie tych misiów w rzędzie jest bezpieczne.

1. Niech U będzie zbiorem trójek bezpiecznych.
2. Niech $M(a_1,a_2,b_1,b_2)$ oznacza liczbę bezpiecznych ustawień a_1 misiów modelu $A1$, a_2 misiów modelu $A2$ itd.
3. Niech $N(a_1,a_2,b_1,b_2,p,q)$ oznacza liczbę bezpiecznych ustawień misiów takich, że przedostatni i ostatni miś w rzędzie to odpowiednio modele p oraz q $(p,q \in T)$.

Odnotujmy jeszcze następujący fakt, prawdziwy dla układów, w których $a_1 + a_2 + b_1 + b_2 \geqslant 2$.

Fakt 1 $M(a_1,a_2,b_1,b_2) = \sum_{(p,q)\in T\times T} N(a_1,a_2,b_1,b_2,p,q)$

Rozwiązanie wzorcowe

Skorzystamy z faktu 1. Zamiast liczyć od razu wartości funkcji M, wyznaczymy najpierw, korzystając z metody programowania dynamicznego, wartości funkcji N.
Uczyńmy kluczowe spostrzeżenie.

Spostrzeżenie 2 *Dla* $a_1 + a_2 + b_1 + b_2 > 2$ *zachodzą równości:*

$$\begin{aligned}
N(a_1,a_2,b_1,b_2,q,A1) &= \sum_{\{p:(p,q,A1)\in U\}} N(a_1-1,a_2,b_1,b_2,p,q),\\
N(a_1,a_2,b_1,b_2,q,A2) &= \sum_{\{p:(p,q,A2)\in U\}} N(a_1,a_2-1,b_1,b_2,p,q),
\end{aligned}$$

$$N(a_1,a_2,b_1,b_2,q,B1) = \sum_{\{p:(p,q,B1)\in U\}} N(a_1,a_2,b_1-1,b_2,p,q),$$
$$N(a_1,a_2,b_1,b_2,q,B2) = \sum_{\{p:(p,q,B2)\in U\}} N(a_1,a_2,b_1,b_2-1,p,q),$$

Pisząc w skrócie:
$N(a_1,a_2,b_1,b_2,q,r) =$
$= \sum_{\{p:(p,q,r)\in U\}} N(a_1-[r=A1],a_2-[r=A2],b_1-[r=B1],b_2-[r=B2],p,q),$
gdzie $[r = x]$ *dla* $x \in T$*, to odpowiedź na pytanie, czy* $r = x$*, czyli jedynka, gdy* $r = x$*, a zero w przeciwnym przypadku. W ogólności zapis [wyrażenie logiczne], nazywany notacją Iversona, jest zdefiniowany jako* 1*, jeżeli wyrażenie jest prawdziwe, a* 0 *w przeciwnym przypadku.*

Zauważmy, że aby policzyć $N(a_1,a_2,b_1,b_2,q,r)$, dla dowolnego r wystarczy znać wartości $N(a'_1,a'_2,b'_1,b'_2,s,t)$, gdzie s i t są dowolnymi modelami misiów, zaś $a'_1+a'_2+b'_1+b'_2 = a_1+a_2+b_1+b_2-1$. Wartości N, które musimy wyznaczyć bezpośrednio, to przypadki, gdy mamy tylko dwa misie: $a_1+a_2+b_1+b_2 = 2$ — można je policzyć „na palcach" (na przykład $N(1,1,0,0,A1,A2) = 1$). Dla wartości $a_1+a_2+b_1+b_2 < 2$ funkcję $N(a_1,a_2,b_1,b_2,*,*)$ definiujemy jako równą zeru.

Implementacja

Powyższe spostrzeżenia pozwalają nam zapisać następującą funkcję.

```
function licz_ustawienia($a_1$, $a_2$, $b_1$, $b_2$)
  { $a_1$, $a_2$, $b_1$, $b_2$ to odpowiednio liczby misiów modelu A1, A2, B1, B2 }
  if $a_1+a_2+b_1+b_2 \leqslant 1$ then
    return 1;
  else begin
    { zainicjuj wartości $N$ dla $a_1+a_2+b_1+b_2 = 2$ }
    { pozostałe pola $N$ wyzeruj }
    for $i := 0$ to $a_1$ do
      for $j := 0$ to $a_2$ do
        for $k := 0$ to $b_1$ do
          for $l := 0$ to $b_2$ do
            for $q \in T$ do
              for $r \in T$ do
                for $p \in T$ do
                  if $i+j+k+l > 2$ and $(p,q,r) \in U$ then
                    { Dodawanie wykonywane modulo 1000000 }
                    $N(i,j,k,l,q,r) := N(i,j,k,l,q,r)+$
                    $+N(i-[r=A1],j-[r=A2],k-[r=B1],l-[r=B2],p,q)$;
  end;
  { M — wynik }
  $M := 0$;
  for $p \in T$ do
```

```
23:       for $q \in T$ do
24:         { Dodawanie wykonywane modulo 1 000 000 }
25:         $M := M + N(a_1, a_2, b_1, b_2, p, q)$;
26:     return $M$;
```

Rzut oka na wiersze 8-11 pozwala przekonać się, że algorytm działa w czasie $O(a_1 a_2 b_1 b_2)$. Jest jednak jeszcze jeden problem: program zużywa za dużo pamięci! W tak zaimplementowanej procedurze tablica N zajmuje niemal $160MB$ pamięci, a mamy tylko $32MB$.

Można temu jednak zaradzić. Zauważmy, że po policzeniu $N(a_1 + 1, a_2, b_1, b_2, p, q)$ już nigdy nie skorzystamy z wartości $N(a_1, a_2, b_1, b_2, p, q)$. W szczególności więc możemy tak zaimplementować procedurę `licz_ustawienia`, żeby tablica N miała rozmiar $2 \times 39 \times 39 \times 39 \times 4 \times 4$. Szczegóły implementacyjne pozostawiamy Czytelnikowi jako nietrudne ćwiczenie.

Testy

Rozwiązania zawodników sprawdzane były na zestawie 13 testów, podzielonych na 10 grup. Większość testów została wygenerowana w sposób losowy.

Nazwa	a_1	a_2	b_1	b_2	Opis
mis1a.in	2	5	3	7	
mis1b.in	0	0	1	0	przypadek brzegowy
mis2a.in	7	1	8	9	
mis2b.in	5	2	1	1	wynikiem jest 0
mis3.in	6	6	4	3	
mis4.in	9	8	7	6	
mis5.in	15	17	8	21	
mis6.in	28	17	4	33	
mis7.in	38	0	1	37	
mis8.in	20	35	20	37	
mis9a.in	38	32	28	30	
mis9b.in	35	20	10	11	wynikiem jest 0
mis10.in	37	37	30	38	

XII Bałtycka Olimpiada Informatyczna,

Heinola, Finlandia 2006

BOI 2006, dzień próbny, 18.05.2006

Squint

Zadanie

Napisz program do wyznaczania pierwiastków z liczb całkowitych.

Wejście

Wejście znajduje się w pliku `squint.in`*. Jedyny wiersz wejścia zawiera jedną liczbę całkowitą* $0 \leqslant n < 2^{63}$.

Wyjście

Wyjście powinno znaleźć się w pliku `squint.out`*. Jedyny wiersz wyjścia powinien zawierać najmniejszą nieujemną liczbę całkowitą* q*, taką że* $q^2 \geqslant n$.

Przykład

Dla pliku wejściowego `squint.in`*:*

```
122333444455555
```

poprawnym wynikiem jest plik wyjściowy `squint.out`*:*

```
11060446
```

jako że $\sqrt{122333444455555} \approx 11060445.038765619$.

Jakub Pawlewicz, Jakub Radoszewski
Tłumaczenie

BOI 2006, dzień pierwszy, 19.05.2006

Wyrażenia bitowe

W dziedzinie przetwarzania sygnałów często występuje potrzeba wyznaczenia maksymalnej możliwej wartości pewnego wyrażenia, zawierającego działania bitowe AND i OR, przy założeniu, że zmienne wykorzystane w wyrażeniu są liczbami całkowitymi o pewnych zakresach. Twoim zadaniem jest napisanie programu, który dla danego wyrażenia i zakresów zmiennych w nim występujących wyznaczy maksymalną wartość, jaką to wyrażenie może osiągnąć.

Analizowane wyrażenie będzie miało uproszczoną postać, mianowicie będzie złożone z pewnej liczby podwyrażeń, poumieszczanych w nawiasach i połączonych działaniem bitowym AND (oznaczanym znakiem &). Każde podwyrażenie składać się będzie z jednej lub z większej liczby zmiennych, połączonych działaniem bitowym OR (oznaczanym znakiem |). Przy użyciu tej konwencji można jednoznacznie opisać dane wyrażenie, podając liczbę podwyrażeń, z których się składa i liczby zmiennych zawartych w poszczególnych podwyrażeniach. Zmienne są ponumerowane w kolejności pojawiania się w wyrażeniu.

Dla rozjaśnienia powyższego opisu przeanalizujmy przykład. Jeżeli liczba podwyrażeń wyrażenia wynosi 4, a liczbami zmiennych w kolejnych podwyrażeniach są: 3, 1, 2 i 2, to takie wyrażenie ma postać:

$$E = (v_1|v_2|v_3)\&(v_4)\&(v_5|v_6)\&(v_7|v_8)$$

Działania bitowe są zdefiniowane w standardowy sposób. Na przykład, w celu wykonania działania $21\&6$*, na początku znajdujemy zapisy binarne obu liczb (argumentów): 10101 oraz 110 (jako że* $21 = 2^4 + 2^2 + 2^0$*, a* $6 = 2^2 + 2^1$*). Każda cyfra binarna wyniku zależy od odpowiadających cyfr argumentów: jeżeli obie są równe 1, to cyfra wynikowa będzie równa 1, a w przeciwnym przypadku 0. Na poniższym rysunku z lewej strony znajduje się ilustracja działania* $21\&6 = 4$*. Gdybyśmy natomiast chcieli policzyć* $21|6$*, to cała procedura wyglądałaby podobnie, z tym wyjątkiem, że wynikowa cyfra byłaby równa 1, jeżeli co najmniej jedna z odpowiadających cyfr w argumentach byłaby równa 1, natomiast jeżeli obie odpowiadające cyfry byłyby zerami, to wynikowa cyfra byłaby równa 0. Na środku poniższego rysunku znajduje się ilustracja działania* $21|6 = 4$*. Definicje działań bitowych łatwo się uogólniają na operacje wieloargumentowe. Z prawej strony rysunku jest wykonane działanie* $30\&11\&7 = 2$*.*

$$\begin{array}{rr} & \\ & 10101 \\ \& & \underline{00110} \\ & 00100 \end{array} \qquad \begin{array}{rr} & \\ & 10101 \\ | & \underline{00110} \\ & 10111 \end{array} \qquad \begin{array}{rr} & 11110 \\ & 01011 \\ \& & \underline{00111} \\ & 00010 \end{array}$$

Wejście

Wejście znajduje się w pliku `bitwise.in`*. W pierwszym wierszu wejścia znajdują się dwie liczby całkowite* N *oraz* P*, gdzie* N *oznacza łączną liczbę zmiennych* ($1 \leqslant N \leqslant 100$)*, a* P *oznacza liczbę podwyrażeń* ($1 \leqslant P \leqslant N$)*. Kolejny wiersz zawiera* P *liczb całkowitych* ($K_1, K_2, \ldots, K_P$)*, gdzie* K_i *oznacza liczbę zmiennych, występujących w i-tym podwyrażeniu.*

Wszystkie liczby K_i są niemniejsze od 1 i ich suma jest równa N. Kolejnych N wierszy zawiera po dwie liczby całkowite A_j i B_j ($0 \leqslant A_j \leqslant B_j \leqslant 2\,000\,000\,000$), definiujące zakres wartości j-tej zmiennej: $A_j \leqslant v_j \leqslant B_j$.

Wyjście

Wyjście powinno znaleźć się w pliku `bitwise.out`*. Wyjście powinno się składać z jednego wiersza, zawierającego jedną liczbę całkowitą — maksymalną możliwą wartość, jaką może osiągnąć wczytane wyrażenie.*

Przykład

Dla pliku wejściowego `bitwise.in`*:*

```
8 4
3 1 2 2
2 4
1 4
0 0
1 7
1 4
1 2
3 4
2 3
```

poprawnym wynikiem jest plik wyjściowy `bitwise.out`*:*

```
6
```

Powyższej zawartości pliku `bitwise.in` *odpowiadają następujące ograniczenia na osiem zmiennych z przykładu z treści zadania: $2 \leqslant v_1 \leqslant 4$, $1 \leqslant v_2 \leqslant 4$, $v_3 = 0$, $1 \leqslant v_4 \leqslant 7$, $1 \leqslant v_5 \leqslant 4$, $1 \leqslant v_6 \leqslant 2$, $3 \leqslant v_7 \leqslant 4$ i $2 \leqslant v_8 \leqslant 3$. W zapisie binarnym, jednym z najlepszych wartościowań dla tego przykładu zmiennych jest: (100|011|000)&(111)&(100|010)&(100|011), w którym wszystkie podwyrażenia z wyjątkiem trzeciego mają wartość 7.*

Ocenianie

W 30% testów liczba wszystkich możliwych wartościowań zmiennych będzie mniejsza od jednego miliona.

Jakub Pawlewicz, Jakub Radoszewski
Tłumaczenie

BOI 2006, dzień pierwszy, 19.05.2006

Kolekcjoner monet

W Bajtocji w obiegu jest N nominałów monet, a wśród nich jest moneta 1-centowa. Dodatkowo jest jeszcze K-centowy banknot, którego wartość przekracza wartości wszystkich monet. Kolekcjoner monet chciałby zebrać co najmniej po jednym egzemplarzu monety każdego nominału. Kolekcjoner ma już trochę monet w domu i jeden K-centowy banknot w portfelu. Wszedł on do sklepu, w którym sprzedawane są przedmioty, których ceny są mniejsze niż K centów (1 cent, 2 centy, 3 centy, …, $K-1$ centów). W tym sklepie algorytm wydawania reszty jest następujący:

1. *Niech reszta do wydania będzie równa A centów.*
2. *Znajdź najwyższy nominał monety, który nie przekracza A (niech to będzie moneta B-centowa).*
3. *Daj klientowi monetę B-centową i zmniejsz wartość A o B.*
4. *Jeżeli $A = 0$, to zakończ; w przeciwnym przypadku wróć do kroku 2.*

Kolekcjoner monet kupuje jeden przedmiot, płacąc swoim K-centowym banknotem.

Zadanie

Napisz program, który wyznaczy:

- *Ile jest różnych monet, których kolekcjoner jeszcze nie ma w swojej kolekcji i które może pozyskać w wyniku takiej transakcji?*
- *Jaka jest cena najdroższego przedmiotu, który sklep może mu sprzedać w tej transakcji?*

Wejście

Wejście znajduje się w pliku `coins.in`. *Pierwszy wiersz wejścia zawiera dwie liczby całkowite N ($1 \leqslant N \leqslant 500\,000$) oraz K ($2 \leqslant K \leqslant 1\,000\,000\,000$). Kolejnych N wierszy opisuje nominały monet znajdujących się w obiegu. Wiersz $(i+1)$-wszy zawiera liczby całkowite c_i ($1 \leqslant c_i < K$) i d_i, gdzie c_i jest wartością monety (w centach), natomiast d_i jest równe 1, jeżeli kolekcjoner posiada już tę monetę lub 0, jeżeli jej nie posiada. Monety są podane w kolejności rosnących wartości, to znaczy $c_1 < c_2 < \ldots < c_n$. Pierwsza moneta jest 1-centowa, to znaczy $c_1 = 1$.*

Wyjście

Wyjście powinno znaleźć się w pliku `coins.out`. *Pierwszy wiersz wyjścia powinien zawierać jedną liczbę całkowitą — maksymalną liczbę nominałów, których kolekcjoner jeszcze nie* *posiada, i które może uzyskać za pomocą jednego zakupu. Drugi wiersz wyjścia powinien zawierać jedną liczbę całkowitą — maksymalną cenę przedmiotu, jaki kolekcjoner może nabyć, tak żeby wydana zgodnie z algorytmem reszta zawierała maksymalną liczbę nowych nominałów, zadeklarowaną w pierwszym wierszu wyjścia.*

Przykład

Dla pliku wejściowego `coins.in`*:*

```
7 25
1 0
2 0
3 1
5 0
10 0
13 0
20 0
```

poprawnym wynikiem jest plik wyjściowy `coins.out`*:*

```
3
6
```

BOI 2006, dzień pierwszy, 19.05.2006

Kraje

Niniejsze zadanie pokazuje, jak skromne są początki powstawania potężnych krajów. Rozważmy dwuwymiarową mapę terenu. Na mapie znajduje się n miast. i-te miasto położone jest na mapie w punkcie (x_i, y_i)*. Położenia miast są parami różne. W i-tym mieście stacjonuje* s_i *żołnierzy pod dowództwem generała.*

Wpływ jaki wywiera i-te miasto na inny punkt na mapie (x, y) *liczymy poprzez podzielenie* s_i *przez kwadrat odległości tego miasta od* (x, y)*. Jest to tak, jakby oddział żołnierzy w i-tym mieście działał siłą grawitacyjną na wszystkie punkty na mapie. Mówimy, że i-te miasto jest zagrożone przez inne j-te miasto, jeżeli wpływ wywierany przez j-te miasto na położenie* (x_i, y_i) *i-tego miasta przekracza liczbę stacjonujących w nim żołnierzy* s_i *— wówczas miasto j-te może wysłać wystarczająco wielu żołnierzy, aby pokonać żołnierzy broniących i-te miasto. Jeżeli pewne i-te miasto nie jest zagrożone przez żadne inne, to wdzięczni mieszkańcy ustanawiają niezwyciężonego generała swoim królem i przemieniają swoje miasto w stolicę królestwa.*

Z drugiej strony, jeżeli pewne miasto j-te, zagrażające i-temu miastu, wywiera ściśle większy wpływ na jego położenie (x_i, y_i) *niż jakiekolwiek inne miasto k-te, to mieszkańcy i-tego miasta nie mają wyboru — i-te miasto musi się poddać j-temu miastu. Dlatego też i-te miasto musi uznać tę samą stolicę co j-te miasto. Jednakże* s_i *żołnierzy stacjonujących w i-tym mieście nie dołącza do armii ani j-tego miasta, ani swojej nowej stolicy. Może też zajść przypadek przeciwny, w którym i-te miasto zostaje ocalone przez wzajemnie nieufne, jednakowo mu grożące miasta j-te i k-te: jeżeli pierwsze z nich zaatakowałoby i pokonało i-te miasto, to drugie z nich mogłoby także zaatakować i pokonać zmęczonych bojem żołnierzy j-tego miasta. W takiej sytuacji mieszkańcy i-tego miasta nie mogą jednak ustanowić swojego generała królem, ponieważ nie jest on w stanie obronić miasta przed zagrożeniem. Dlatego też przemieniają oni wówczas swoje miasto w stolicę demokracji.*

Zadanie

Twoim zadaniem jest napisanie programu, który wczyta informację o położeniach miast na mapie i wypisze dla każdego miasta i dokładnie jedno z trzech stwierdzeń:

- *Jest ono stolicą królestwa.*
- *Jest ono stolicą demokracji.*
- *Uznaje j-te miasto za swoją stolicę.*

Wejście

Wejście znajduje się w pliku `countries.in`*. Pierwszy wiersz wejścia zawiera jedną liczbę całkowitą* $1 \leqslant n \leqslant 1\,000$*, oznaczającą liczbę miast. Kolejnych n wierszy podaje informacje o n miastach. Każda informacja znajduje się w osobnym wierszu. Wiersz* $(i + 1)$*-wszy* *podaje informację o i-tym mieście w postaci trzech liczb całkowitych x_i, y_i, s_i, pooddzielanych pojedynczymi odstępami. Wszystkie te liczby spełniają nierówności $0 \leqslant x_i, y_i, s_i \leqslant 1\,000$.*

Wyjście

Wyjście powinno znaleźć się w pliku `countries.out`*. Wyjście powinno się składać z n wierszy, z których i-ty powinien zawierać odpowiedź dla i-tego miasta:*

- *Literę* `K`*, jeżeli i-te miasto jest stolicą królestwa.*
- *Literę* `D`*, jeżeli i-te miasto jest stolicą demokracji.*
- *Numer miasta $1 \leqslant j \leqslant n$, które i-te miasto uznaje za swoją stolicę, w przypadku, gdy i-te miasto musiało się poddać.*

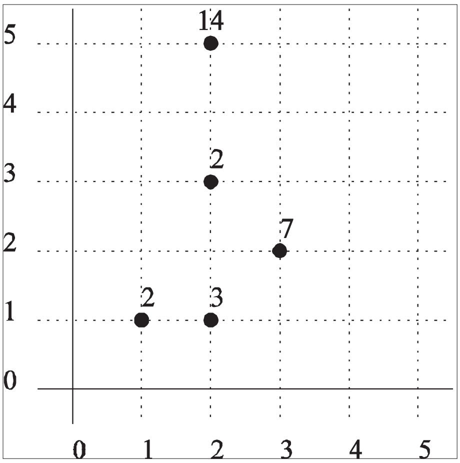
Przykład

Dla pliku wejściowego `countries.in`*:*

```
5
2 5 14
2 3 2
3 2 7
1 1 2
2 1 3
```

poprawnym wynikiem jest plik wyjściowy `countries.out`*:*

```
K
D
K
3
3
```



Na powyższej mapie każda kropka reprezentuje miasto. Nad każdą kropką znajduje się liczba żołnierzy stacjonujących w odpowiadającym mieście. Powyższe 5 miast jest ponumerowanych z góry na dół i od lewej do prawej.

3-cie miasto, położone w punkcie (3,2) jest stolicą królestwa, które zawiera także miasta: 4 w położeniu (1,1) i 5 w położeniu (2,1). Z drugiej strony, miasto 1 w położeniu (2,5) samo stanowi królestwo, a miasto 2 w położeniu (2,3) samo stanowi demokrację.

Jakub Pawlewicz, Jakub Radoszewski
Tłumaczenie

BOI 2006, dzień drugi, 20.05.2006

Plan budowy miasta

Na planecie budowana jest stacja kosmiczna. Stacja ma zatrudniać N osób, które muszą gdzieś mieszkać, więc wokół stacji należy wybudować miasto. Ziemia dokoła stacji jest podzielona na równe kwadraty. Na każdym kwadracie można wybudować jeden budynek, mający co najwyżej K pięter. Każdy budynek musi mieć dokładnie jedno mieszkanie na każdym ze swoich pięter. Wszystkie osoby muszą być zakwaterowane w oddzielnych mieszkaniach. Pozycję kwadratu określamy przez parę liczb całkowitych (x,y), przy czym $(0,0)$ oznacza pozycję stacji, a pozostałe kwadraty są rozmieszczone tak jak na rysunku poniżej:

…	…	…	…	…
…	$(-1,1)$	$(0,1)$	$(1,1)$	…
…	$(-1,0)$	$(0,0)$	$(1,0)$	…
…	$(-1,-1)$	$(0,-1)$	$(1,-1)$	…
…	…	…	…	…

Ponieważ ruch może odbywać się tylko na ulicach znajdujących się pomiędzy budynkami, odległość pomiędzy kwadratem (x,y) a stacją wynosi $|x|+|y|-1$ jednostek.

Koszt wybudowania jednego budynku jest równy sumie kosztów budowy każdego z pięter. Wiadomo, że koszt budowy piętra zależy od wysokości na jakiej ma się ono znaleźć i nie zależy od miejsca postawienia budynku.

Czas życia każdego budynku wynosi 30 lat. Mieszkańcy tych budynków będą dojeżdżać do pracy do stacji kosmicznej w tą i z powrotem te 30 lat. Przez ten czas, koszt dojazdów wyniesie pojedynczą osobę $T \cdot d$, gdzie d jest odległością z budynku danej osoby do stacji.

Można przyjąć, że planeta jest na tyle duża, a miasto na tyle małe, że teren, na którym ma być wybudowane miasto, jest płaski.

Zadanie

Twoim zadaniem jest napisanie programu, znajdującego minimalny sumaryczny koszt postawienia budynków i dojazdów w czasie 30 lat.

Wejście

Wejście znajduje się w pliku tekstowym `city.in`*. Pierwszy wiersz wejścia zawiera liczby całkowite N $(1 \leqslant N \leqslant 1\,000\,000\,000\,000)$, T $(1 \leqslant T \leqslant 500\,000)$ oraz K $(1 \leqslant K \leqslant 20\,000)$. Następne K wierszy opisują koszt wybudowania mieszkania na każdym z pięter. $(i+1)$-wszy wiersz zawiera liczbę całkowitą c_i $(1 \leqslant c_i \leqslant 2\,000\,000\,000)$ — koszt zbudowania mieszkania* *na i-tym piętrze (przy założeniu, że każde z $i-1$ niższych pięter zostało już zbudowane). Wiadomo, że koszt budowy wyższego piętra jest większy niż koszt budowy niższego piętra:* $c_1 < c_2 < \ldots < c_K$.

Wyjście

Wyjście powinno znaleźć się w pliku tekstowym `city.out`. *Pierwszy i jedyny wiersz wyjścia powinien zawierać jedną liczbę całkowitą — całkowity koszt budowy miasta i dojazdów w czasie 30 lat. Można założyć, że wynik nie przekracza* $8 \cdot 10^{18}$ *(czyli mieści się w 64-bitowej liczbie całkowitej ze znakiem).*

Przykład

Dla pliku wejściowego `city.in`*:*

```
17 5 4
100
107
114
121
```

poprawnym wynikiem jest plik wyjściowy `city.out`*:*

```
1778
```

Jakub Pawlewicz, Jakub Radoszewski
Tłumaczenie

BOI 2006, dzień drugi, 20.05.2006

Kompresja RLE

RLE jest prostym algorytmem, używanym do kompresowania ciągów zawierających ciągłe wystąpienia jednego znaku. W wyniku kompresji danego ciągu otrzymujemy jego **kod***. Idea jest taka, aby zastępować ciąg składający się z powtórzeń jednego znaku (np.* `aaaaa`*) licznikiem określającym liczbę wystąpień. Mianowicie taki ciąg zastępujemy trójką, składającą się ze znaku specjalnego, powtarzającego się znaku i liczby całkowitej, określającej liczbę jego wystąpień. Na przykład* `aaaaa` *możemy zakodować jako* `#a5` *(gdzie* `#` *jest znakiem specjalnym).*

Ponieważ musimy jakoś reprezentować alfabet, znak specjalny i licznik, zdefiniujemy kodowanie bardziej formalnie. Niech alfabet składa się z n *znaków reprezentowanych przez liczby całkowite ze zbioru* $\Sigma = \{0, 1, \ldots, n-1\}$*. Kodem ciągu składającego się ze znaków ze zbioru* Σ *jest również ciąg znaków z* Σ*. W danym momencie znak specjalny jest reprezentowany przez znak z* Σ *i oznaczamy go przez* e*. Z początku* e *wynosi* 0*, ale może się to zmienić podczas kodowania.*

Kod interpretujemy w następujący sposób:

- *dowolny znak* a *występujący w kodzie i różny od specjalnego reprezentuje po prostu ten znak,*
- *jeśli w kodzie pojawia się znak specjalny* e*, to następne dwa znaki mają specjalne znaczenie:*
  - *jeśli po* e *występuje* ek*, to taka sekwencja reprezentuje* $k+1$ *wystąpień* e*,*
  - *w przeciwnym przypadku, jeśli po* e *występuje* $b0$ *(gdzie* $b \neq e$*), to od tego momentu* b *staje się znakiem specjalnym,*
  - *w p.p., jeśli po* e *występuje* bk *(gdzie* $b \neq e$ *i* $k > 0$*), to reprezentuje to* $k+3$ *wystąpienia* b*.*

Używając powyższego schematu kodowania możemy zakodować dowolny ciąg znaków z Σ*. Na przykład, dla* $n = 4$*, ciąg* `1002222223333303020000` *możemy zakodować jako* `10010230320100302101`*. Pierwszy znak kodu* `1` *oznacza po prostu* `1`*. Następnie* `001` *koduje* `00`*. Dalej* `023` *reprezentuje* `222222`*,* `032` *reprezentuje* `33333` *i* `010` *zmienia znak specjalny na* `1`*. Następnie* `0302` *reprezentuje siebie i na końcu* `101` *koduje* `0000`*.*

Dany ciąg możemy zakodować na wiele sposobów, przy czym długości otrzymywanych kodów mogą się różnić. Twoim zadaniem jest dla zakodowanego już ciągu znaleźć inny kod, który ma najmniej znaków.

Zadanie

Napisz program, który:

- *Wczyta rozmiar alfabetu i kod pewnego ciągu.*

- *Znajdzie najkrótszy kod dla tego ciągu.*
- *Zapisze wynik.*

Wejście

Wejście znajduje się w pliku tekstowym `rle.in`*. Pierwszy wiersz wejścia zawiera liczbę całkowitą* n *(*$2 \leqslant n \leqslant 100\,000$*) — rozmiar alfabetu. Drugi wiersz zawiera liczbę całkowitą* m *(*$1 \leqslant m \leqslant 2\,000\,000$*) — długość kodu. Ostatni wiersz zawiera* m *liczb całkowitych ze zbioru* $\{0, 1, \ldots, n-1\}$*, pooddzielanych pojedynczymi odstępami i reprezentujących kod ciągu.*

Wyjście

Wyjście powinno się znaleźć w pliku tekstowym `rle.out`*. Pierwszy wiersz wyjścia powinien zawierać jedną liczbę całkowitą* m' *— najmniejszą możliwą liczbę znaków w kodzie reprezentującym dany ciąg. Drugi i ostatni wiersz wyjścia powinien zawierać* m' *liczb całkowitych ze zbioru* $\{0, 1, \ldots, n-1\}$*, pooddzielanych pojedynczymi odstępami i reprezentujących kod ciągu. Jeśli istnieje więcej niż jeden najkrótszy kod danego ciągu, twój program powinien wypisać dowolny z nich.*

Przykład

Dla pliku wejściowego `rle.in`*:*

```
4
20
1 0 0 1 0 2 3 0 3 2 0 1 0 0 3 0 2 1 0 1
```

poprawnym wynikiem jest plik wyjściowy `rle.out`*:*

```
19
1 0 1 0 0 0 1 2 3 1 3 2 0 3 0 2 1 0 1
```

Natomiast dla pliku wejściowego `rle.in`*:*

```
14
15
10 10 10 0 10 0 10 10 13 10 10 13 10 10 13
```

poprawnym wynikiem jest plik wyjściowy `rle.out`*:*

```
9
0 10 13 0 10 13 0 10 10
```

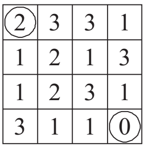
Jakub Pawlewicz, Jakub Radoszewski
Tłumaczenie

BOI 2006, dzień drugi, 20.05.2006

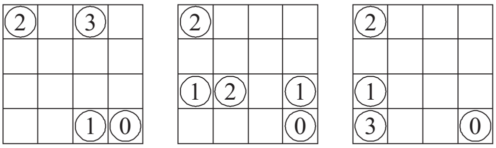
Przeskocz szachownicę!

Na szachownicy o wymiarach $n \times n$ rozmieszczono liczby całkowite, po jednej nieujemnej liczbie całkowitej na pole. Celem jest przeskoczenie z lewego górnego pola do prawego dolnego pola planszy poprzez wykonywanie tylko dozwolonych skoków. Liczba na polu mówi o ile dokładnie pól trzeba się przesunąć z tego pola. Skok można wykonywać tylko w linii prostej w prawo bądź w dół. Zabronione są skoki, które powodują wyjście poza szachownicę. Zauważ, że 0 oznacza ślepy zaułek, z którego nie można się już dalej ruszyć.

Rozważmy szachownicę o wymiarach 4×4, taką jak na Rysunku 1. Na tym rysunku kółko narysowane linią ciągłą oznacza pozycję startową, a kółko narysowane linią przerywaną oznacza pozycję docelową.



Rysunek 1



Rysunek 2

Na rysunku 2 pokazane są wszystkie trzy dozwolone drogi ze startu do celu. Na tym rysunku usunięto liczby, które nie mają wpływu na wykonywane skoki.

Zadanie

Twoim zadaniem jest napisanie programu, który znajdzie liczbę wszystkich dozwolonych dróg z lewego górnego pola do prawego dolnego pola.

Wejście

Pierwszy wiersz pliku wejściowego `jump.in` *zawiera jedną liczbę całkowitą n, $4 \leqslant n \leqslant 100$ — rozmiar szachownicy. Następnie na wejściu jest n wierszy, opisujących liczby znajdujące się na szachownicy. Każdy z tych wierszy zawiera n liczb całkowitych z przedziału $0 \dots 9$, pooddzielanych pojedynczymi odstępami.*

Wyjście

W jedynym wierszu pliku wyjściowego `jump.out` *powinna znaleźć się jedna liczba całkowita — liczba dozwolonych dróg z lewego górnego pola do prawego dolnego pola.*

Przykład

Dla pliku wejściowego `jump.in`*:*

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

poprawnym wynikiem jest plik wyjściowy `jump.out`*:*

```
3
```

Ocenianie

Liczba dozwolonych dróg może być dosyć spora. Używając zmiennych 64-bitowych (`long long int` *w C,* `Int64` *w Pascalu) można osiągnąć jedynie 70% maksymalnej punktacji. Możesz założyć, że dla wszystkich testów wynik będzie liczbą całkowitą, mającą nie więcej niż 100 cyfr.*

XIII Olimpiada Informatyczna Krajów Europy Środkowej,

Vrsar, Chorwacja 2006

Marcin Kubica, Paweł Gawrychowski
Tłumaczenie

CEOI 2006, dzień pierwszy, 03.07.2006

Nadajnik

Nowa rozgłośnia radiowa chce zacząć nadawać w Vrsarze. Umowa jaką zawarli z władzami miasta określa minimalną liczbę domostw, jaka ma być w zasięgu nadajnika. Ze względu na ograniczony budżet mogą postawić tylko jeden nadajnik o określonym zasięgu. Koszt nadajnika jest proporcjonalny do jego zasięgu. Dlatego też stacja radiowa chciałaby tak umiejscowić nadajnik, żeby spełnić warunki umowy, a jego zasięg był jak najmniejszy.

W Vrsarze jest N domostw — każde z nich jest reprezentowane przez parę całkowitych współrzędnych. Nadajnik może być umieszczony w **dowolnym punkcie na płaszczyźnie** *(niekoniecznie o współrzędnych całkowitych) a zasięg nadajnika może być* **dowolną dodatnią liczbą rzeczywistą**. *Jeśli nadajnik ma zasięg R, to domostwo jest w zasięgu nadajnika jeżeli jego odległość od nadajnika wynosi* **co najwyżej R**.

Zadanie

Napisz program, który na podstawie położenia domostw oraz liczby całkowitej K wyznaczy minimalny wymagany zasięg oraz jedno z możliwych położeń nadajnika, dla których przynajmniej K domostw jest w zasięgu nadajnika.

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby całkowite N i K ($2 \leqslant K \leqslant N \leqslant 500$) — łączną liczbę domostw oraz minimalną liczbę domostw, które muszą być w zasięgu nadajnika.
Każdy z kolejnych N wierszy zawiera po dwie liczby całkowite X i Y ($0 \leqslant X, Y \leqslant 10000$) — współrzędne jednego z domostw. Żadne dwa domostwa nie mają tych samych (obydwu) współrzędnych.

Wyjście

Pierwszy wiersz wyjścia powinien zawierać liczbę rzeczywistą — minimalny wymagany zasięg nadajnika R. Drugi wiersz wyjścia powinien zawierać współrzędne nadajnika — dwie liczby rzeczywiste X i Y.
Uwaga: *Jeżeli istnieje wiele możliwych poprawnych wyników, Twój program powinien wypisać dowolny z nich. Wszystkie trzy liczby powinny być wypisane albo w standardowej postaci dziesiętnej, albo w notacji inżynierskiej.*

Ocenianie

Podany przez Twój program wynik zostanie uznany za poprawny wtedy i tylko wtedy, gdy będą spełnione następujące dwa warunki:

- *Wartość bezwzględna różnicy między* R *i minimalnym wymaganym zasięgiem (wyznaczonym przez jury) jest mniejsza lub równa* 0.0001*.*
- *Nadajnik o zasięgu* $R+0.0002$*, umieszczony w punkcie o współrzędnych* (X, Y) *pokrywa swym zasięgiem przynajmniej* K *domostw.*

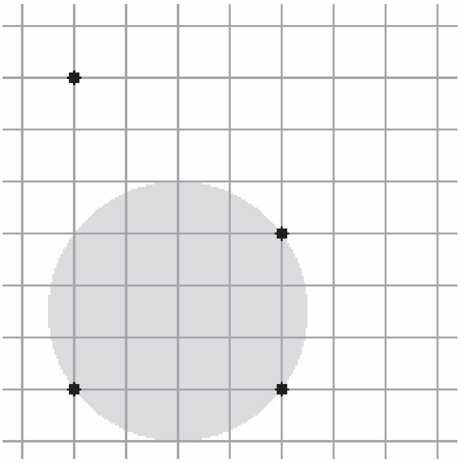
Przykład

Dla danych wejściowych:

```
4 3
2 2
6 2
6 5
2 8
```

poprawnym wynikiem jest:

```
2.5
4 3.5
```

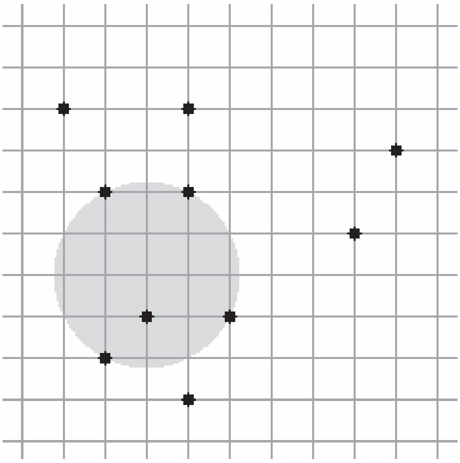


Natomiast dla danych:

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

poprawnym wynikiem jest:

```
2.236068
3 4
```



Uwaga: *każdy z rysunków przedstawia odpowiednie wejście wraz z przedstawionym wynikiem.*

CEOI 2006, dzień pierwszy, 03.07.2006

Kolejka kibiców

Jeśli jesteś choćby umiarkowanym kibicem piłki nożnej, to zapewne wiesz, że zdobycie biletów na tegoroczne mistrzostwa Świata w Niemczech było prawie niemożliwe. Zachłanni organizatorzy i skorumpowane federacje piłkarskie przejęły większość dostępnych biletów i rozprowadziły je wśród sponsorów, działaczy, a także ich rodzin oraz znajomych. W rezultacie, trybuny były pełne osób z koneksjami, a prawdziwi kibice zostali w domach oglądając mecze przerywane reklamami kiepskiego piwa i bezcukrowej gumy do żucia.

Zostało jednak jeszcze kilka biletów na mecz finałowy. Przed kasą utworzyła się gigantyczna kolejka. W miarę jak kibice stawali w kolejce, zostali oznaczeni kolejnymi liczbami całkowitymi — pierwsza osoba w kolejce została oznaczona liczbą 1, druga liczbą 2 itd.

Kolejka zaczęła tworzyć się już poprzedniej nocy. Kibice czekając tak długo wypili dużo kiepskiego piwa i wyżuli mnóstwo bezcukrowej gumy. Nic więc dziwnego, że wielu z nich musiało skorzystać z toalety. Za każdym razem, gdy któryś z kibiców idzie za potrzebą, opuszcza kolejkę i wraca do niej po załatwieniu potrzeby – jednak wracając do kolejki może stanąć w innym miejscu niż poprzednio. Ponieważ dostępna jest tylko jedna kabina WC, żaden z kibiców nie opuszcza kolejki, dopóki poprzedni kibic nie wróci z toalety (tak więc, w każdej chwili co najwyżej jednego kibica nie ma w kolejce).

Przez noc toaleta został odwiedzony przez kibiców N razy. Każda taka wizyta jest opisana dwiema dodatnimi liczbami całkowitymi A i B — kibic oznaczony liczbą A **wyszedł z kolejki***, a następnie wrócił do niej i stanął* **bezpośrednio przed kibicem oznaczonym liczbą** *B. Po zakończeniu wszystkich wyjść do toalety organizatorzy muszą znaleźć odpowiedź na szereg pytań. Każde pytanie ma postać albo 'PX', co oznacza pytanie o* **pozycję kibica oznaczonego numerem** *X, albo 'LX', co oznacza pytanie o to* **jaką liczbą jest oznaczony kibic na pozycji numer** *X.*

Zadanie

Pierwsza osoba w kolejce stoi na pozycji 1, druga na pozycji 2 itd.
Napisz program, który mając dane opisy wizyt kibiców w kabinie WC oraz ciąg pytań **odpowie na wszystkie dane pytania**.

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę całkowitą N $(2 \leqslant N \leqslant 50000)$ — łączną liczbę wizyt w toalecie. Każdy z kolejnych N wierszy zawiera po dwie różne liczby całkowite A i B $(1 \leqslant A, B \leqslant 10^9)$, opisujące jedną wizytę w toalecie.

Kolejny wiersz zawiera jedną liczbę całkowitą Q $(1 \leqslant Q \leqslant 50000)$ — łączną liczbę pytań. Każdy z kolejnych Q wierszy zawiera pojedynczą literę (jest to albo duża litera 'P', albo duża litera 'L') oraz liczbę całkowitą X $(1 \leqslant X \leqslant 10^9)$, opisujące jedno pytanie.

Wyjście

Wyjście powinno łącznie składać się z Q *wierszy.*
Wiersz numer i *powinien zawierać liczbę całkowitą* R *— odpowiedź na pytanie numer* i*. Jeśli odpowiednie pytanie jest postaci ’*PX_i*’, to* R *powinno być końcową pozycją kibica oznaczonego liczbą* X_i*. Jeśli odpowiednie pytanie jest postaci ’*LX_i*’, to* R *powinno być liczbą, którą oznaczono kibica stojącego na pozycji* X_i*.*

Ocena

W tym zadaniu można dostać częściowe punkty za niepoprawne rozwiązania, które jednak poprawnie odpowiadają na jeden rodzaj pytań. Jeśli Twój program poprawnie odpowie na wszystkie pytania postaci ’PX*’, lub na wszystkie pytania postaci ’*LX*’, otrzyma 50% punktów za dany test.*
Jednak żeby otrzymać częściowe punkty, wyjście musi mieć postać zgodną ze specyfikacją. Tak więc nawet jeśli zdecydujesz się odpowiadać tylko na jeden typ pytań, to musisz również wypisywać odpowiedzi na wszystkie pytania drugiego typu.

Przykład

Dla danych wejściowych:

```
2
6 3
9 6
8
L 1
L 2
L 3
L 4
P 1
P 2
P 3
P 4
```

poprawnym wynikiem jest:

```
1
2
9
6
1
2
5
6
```

Natomiast dla danych:

```
5
7 2
2 7
9 7
10 1
100005 99995
9
L 1
P 2
L 2
P 7
L 7
P 9
P 10
P 99999
L 100000
```

poprawnym wynikiem jest:

```
10
3
1
5
4
4
1
100000
99999
```

Marcin Kubica, Paweł Gawrychowski
Tłumaczenie

CEOI 2006, dzień pierwszy, 03.07.2006

Spacer

Znalezienie drogi w dużym i nieznanym mieście może być całkiem skomplikowanym zadaniem, szczególnie gdy jest się — tak jak Krzyś — informatykiem, który zawsze stara się wybierać najkrótszą możliwą drogę. Dysponując mapą miasta Krzyś chce znaleźć najkrótszą ścieżkę pomiędzy miejscem, w którym aktualnie się znajduje a miejscem, do którego chce się dostać

Mapę miasta możemy wyobrażać sobie jako nieskończoną siatkę złożoną z kwadratów jednostkowych.

Krzyś znajduje się w kwadracie o współrzędnych $(0,0)$ *i chce się dostać do kwadratu o współrzędnych* (X,Y).

W mieście jest N *budynków. Każdy z nich ma na mapie postać prostokąta zajmującego pewną liczbę kwadratów jednostkowych.* **Żadne dwa budynki nie stykają się ani nie mają części wspólnej** *(co oznacza, że Krzyś może wokół nich swobodnie chodzić). Każdy budynek jest zdefiniowany przez podanie współrzędnych dwóch skrajnych kwadratów znajdujących się po przekątnej.*

W jednym kroku Krzyś może przejść do jednego z czterech sąsiednich kwadratów, pod warunkiem, że nie wejdzie na obszar zajmowany przez budynek. Wszystkie kwadraty zajęte przez budynki mają współrzędne X **ostro większe od zera**.

Zadanie

Twoim zadaniem jest napisanie programu, który na podstawie opisu budynków znajdzie **jedną najkrótszą ścieżkę** *z miejsca, w którym aktualnie znajduje się Krzyś do miejsca, do którego chce się dostać. Ścieżka powinna zostać wypisana jako ciąg poziomych i pionowych odcinków, z których żadne dwa kolejne nie są do siebie równoległe. Długością ścieżki nazywamy liczbę kwadratów jednostkowych, z których jest ona złożona, nie licząc kwadratu, z którego zaczynamy wędrówkę.*

Wejście

Pierwsza linia wejścia zawiera dwie liczby całkowite X,Y $(1 \leqslant X \leqslant 10^6, -10^6 \leqslant Y \leqslant 10^6)$ *— współrzędne kwadratu do którego chcemy się dostać. Druga linia wejścia zawiera jedną liczbę całkowita* N $(0 \leqslant N \leqslant 100\,000)$ *— liczbę budynków w mieście. Każdy z następnych* N *wierszy zawiera cztery liczby całkowite* $X1, Y1, X2, Y2$ $(1 \leqslant X1, X2 \leqslant 10^6, -10^6 \leqslant Y1, Y2 \leqslant 10^6)$ *— współrzędne dwóch skrajnych kwadratów znajdujących się na przekątnej prostokąta zajmowanego przez budynek.*

Wyjście

Pierwsza linia wyjścia powinna zawierać liczbę całkowitą L *— długość najkrótszej ścieżki prowadzącej do kwadratu docelowego. Druga linia wyjścia powinna zawierać liczbę całkowitą* M *— ilość odcinków, z których składa się najkrótsza ścieżka.* M *nie może przekroczyć*

1 000 000. Każdy z następnych M wierszy powinien zawierać dwie liczby całkowite, DX i DY, opisujące względne przemieszczenie Krzysia wzdłuż jednego odcinka ścieżki. Dla każdego odcinka, **dokładnie jedna** *z liczb DX, DY powinna być zerem, a żadne dwa kolejne odcinki nie powinny być do siebie równoległe.*

Uwaga: *jeżeli istnieje wiele prawidłowych rozwiązań, możesz podać dowolne z nich.*

Ocena

Niepełne lub nieprawidłowe rozwiązania, które poprawnie znajdują długość najkrótszej możliwej ścieżki, otrzymają punkty częściowe.

Jeżeli podana przez Ciebie minimalna długość jest prawidłowa, otrzymasz 80% punktów za test.

W przypadku, gdy znajdujesz tylko minimalna długość, nie musisz wypisywać niczego więcej.

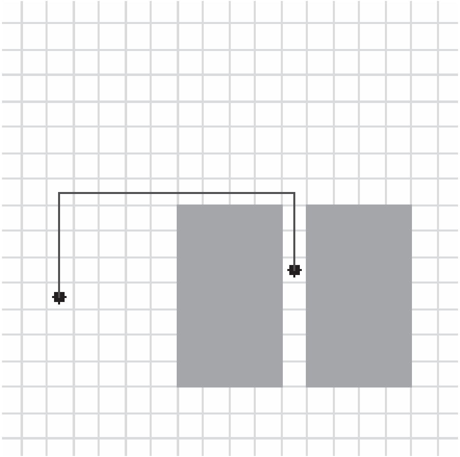
Przykład

Dla danych wejściowych:

```
9 1
2
5 -3 8 3
10 -3 13 3
```

poprawnym wynikiem jest:

```
16
3
0 4
9 0
0 -3
```

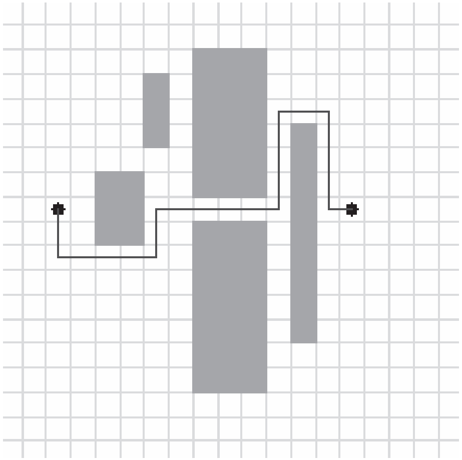


Natomiast dla danych:

```
12 0
5
2 -1 3 1
6 -7 8 -1
6 1 8 6
4 3 4 5
10 -5 10 3
```

poprawnym wynikiem jest:

```
24
8
0 -2
4 0
0 2
5 0
0 4
2 0
0 -4
1 0
```



Uwaga: *każdy obrazek przedstawia dane wejściowe oraz odpowiadające im przykładowe rozwiązanie.*

Marcin Kubica, Paweł Gawrychowski
Tłumaczenie

CEOI 2006, dzień drugi, 05.07.2006

Połącz

Dawno, dawno temu, ulubioną grą każdego, kto używał komputera pracującego w rozdzielczości 25x80 znaków, był „Nibbles", ale to nie ma najmniejszego znaczenia.

A teraz coś zupełnie innego. Do gry w Połącz potrzebna jest plansza złożona z pól tworzących R wierszy i C kolumn, gdzie zarówno R jak i C są **nieparzyste**. *Wiersze i kolumny są ponumerowane liczbami — odpowiednio — od 1 do R oraz od 1 do C. Każde pole na planszy jest wolne lub zablokowane przez mur. Co więcej, każda plansza spełnia następujące warunki:*

- *Kwadraty, których* **obydwie współrzędne są parzyste**, *nazywamy pokojami. Zawsze są one* **wolne**.
- *Kwadraty, których* **obydwie współrzędne są nieparzyste**, *nazywamy barierami. Zawsze są one* **zablokowane**.
- *Wszystkie pozostałe kwadraty nazywamy* **korytarzami**. *Mogą one być zarówno wolne, jak i zablokowane.*
- *Korytarze leżące* **na brzegach** *planszy są zawsze* **zablokowane**.

Bariery są przedstawiane jako znaki '`+`*', zablokowane pionowe korytarze jako '*`|`*', a zablokowane poziome korytarze jako '*`-`*'. Pokoje i niezablokowane korytarze są reprezentowane przez spacje.*

Na samym początku gry na planszy umieszcza się **parzystą liczbę** *pionków (reprezentowanych przez duże litery '*`X`*'), każdy z nich w innym* **pokoju**. *Ścieżką pomiędzy pionkami A i B nazywamy ciąg* **wolnych** *kwadratów, który zaczyna się w A, kończy w B, a każde dwa kolejne z nich sąsiadują w* **pionie lub w poziomie**. *Długość ścieżki definiujemy jako liczbę kroków potrzebnych do przejścia z A do B (czyli jest to pomniejszona o jeden liczba kwadratów, z których składa się ścieżka).*

Celem gracza jest najpierw **podzielenie pionków na pary**, *a następnie* **połączenie** *każdej pary ścieżką. Ścieżki muszą zostać dobrane w taki sposób, aby* **żadne dwie z nich nie przechodziły przez ten sam kwadrat**. *Wynikiem gry nazywamy* **sumę długości** *wszystkich ścieżek.*

```
+-+-+-+-+-+-+-+          +-+-+-+-+-+-+-+
|             |          |  .......    |
+ + + + + + + +          + +.+ + +.+ + +
|X  |   |     |          |X..|   |.    |
+ + + + + + + +          + + + + +.+ + +
|   |   |  X  |          |   |   |..X  |
+-+ + + + + + +          +-+ + + + + + +
|       |     |          |       |     |
+ + + +-+-+-+-+          + + + +-+-+-+-+
|            X|          |            X|
+ + +-+-+-+-+ +          + + +-+-+-+-+.+
|             |          |  ...........|
+ + + + + + + +          + +.+ + + + + +
|  X|         |          |  X|         |
+ + + + + + + +          + + + + + + + +
|   |         |          |   |         |
+-+-+-+-+-+-+-+          +-+-+-+-+-+-+-+
```

Zadanie

Twoim zadaniem jest napisanie programu, który na podstawie opisu planszy rozegra grę w taki sposób, aby osiągnąć najmniejszy możliwy wynik.
Dane wejściowe będą dobrane w taki sposób, aby zawsze istniało przynajmniej jedno (ale niekoniecznie dokładnie jedno) rozwiązanie.

Wejście

Pierwsza linia wejścia zawiera dwie nieparzyste liczby R *i* C, $(5 \leqslant R \leqslant 25,\ 5 \leqslant C < 80)$ *— ilość wierszy i kolumn. W każdej z następnych* R *linii znajduje się* C *znaków opisujących jeden wiersz planszy. Każdy z tych znaków będzie albo jednym ze znaków ’+’, ’|’, ’−’ (reprezentujący barierę, zablokowany korytarz), albo spacją reprezentującą niezablokowany korytarz lub pokój, albo znakiem ’*`X`*’ reprezentującym pionek. Na planszy będą znajdować się przynajmniej dwa pionki. Pojedynczy wiersz wejścia możesz wczytać na przykład w następujący sposób (pamiętaj, aby pierwszą linię wejścia zawierającą liczby* R *i* C *wczytać w całości, łącznie ze znakiem nowego wiersza):*

PASCAL	*C*	*C++*
var s : string; *readln(s);*	*char s[81];* *gets(s);*	*string s;* *getline(cin,s);*

Wyjście

Pierwsza linia wyjścia powinna zawierać jedną liczbę całkowitą, najmniejszy możliwy wynik. Następne R *wierszy powinno zawierać opis planszy po skończonej grze. Format opisu powinien być dokładnie taki sam jak format wejścia, za wyjątkiem niezajętych przez pionki kwadratów należących do ścieżek, które powinny zostać oznaczone przy pomocy znaków ’*$\mathbf{.}$*’.*
Uwaga: *W przypadku, gdy istnieje wiele prawidłowych rozwiązań, możesz podać dowolne z nich.*

Ocena

Niepełne lub nieprawidłowe rozwiązania, które poprawnie znajdą najmniejszy możliwy wynik gry, otrzymają punkty częściowe. Jeżeli podany przez Twój program najmniejszy możliwy wynik jest poprawny, otrzymasz 80% punktów za test. Jeżeli nie znajdujesz opisu planszy po zakończonej grze, nie musisz wypisywać niczego więcej.

Przykład

Dla danych wejściowych:

```
17 15
+-+-+-+-+-+-+-+
|             |
+ + + + + + + +
|X  |   |     |
+ + + + + + + +
|   |   |  X  |
+-+ + + + + + +
|       |     |
+ + + +-+-+-+-+
|            X|
+ + +-+-+-+-+ +
|             |
+ + + + + + + +
|  X|         |
+ + + + + + + +
|   |         |
+-+-+-+-+-+-+-+
```

poprawnym wynikiem jest:

```
30
+-+-+-+-+-+-+-+
|  .........  |
+ +.+ + + +.+ +
|X..|   |  .  |
+ + + + + +.+ +
|   |   |  X  |
+-+ + + + + + +
|       |     |
+ + + +-+-+-+-+
|            X|
+ + +-+-+-+-+.+
|  ...........|
+ +.+ + + + + +
|  X|         |
+ + + + + + + +
|   |         |
+-+-+-+-+-+-+-+
```

Natomiast dla danych:

```
15 15
+-+-+-+-+-+-+-+
|X|           |
+ + + +-+ + + +
| |   |X|  X  |
+ + + + + + +-+
|     | |   |X|
+ +-+-+ + + + +
|       |   | |
+-+-+ + + +-+ +
|     |     | |
+ + + +-+-+ + +
|     |X  | | |
+ +-+-+ + + + +
|    X|       |
+-+-+-+-+-+-+-+
```

poprawnym wynikiem jest:

```
56
+-+-+-+-+-+-+-+
|X|           |
+.+ + +-+ + + +
|.|   |X|  X  |
+.+ + +.+ +.+-+
|.    |.|  .|X|
+.+-+-+.+ +.+.+
|..... .|...|.|
+-+-+.+.+.+-+.+
|    .|...  |.|
+ + +.+-+-+ +.+
|.....|X..| |.|
+.+-+-+ +.+ +.+
|....X|  .....|
+-+-+-+-+-+-+-+
```

Marcin Kubica, Paweł Gawrychowski
Tłumaczenie

CEOI 2006, dzień drugi, 05.07.2006

Link

Webmaster Kirk porządkuje witrynę internetową swojej szkoły. Treści tworzących ją stron są dobre, ale nie są one właściwie połączone odsyłaczami. W rzeczywistości, każda strona zawiera dokładnie jeden odsyłacz prowadzący do innej ze stron. To nie jest właściwy sposób organizacji — użytkownik zaczynając od strony domowej musi zwykle przejść przez wiele odsyłaczy zanim dotrze do interesującej go strony, a niektóre strony w ogóle nie są dostępne. Na początek Kirk chce dodać kilka odsyłaczy tak, aby do każdej ze stron można było szybko dotrzeć ze strony domowej. Nowe odsyłacze mogą być dodane na dowolnych stronach szkoły.

Na witrynę szkoły składa się N stron oznaczonych liczbami całkowitymi od 1 do N, przy czym strona domowa ma numer 1. Strony łączy N odsyłaczy; na **każdej stronie znajduje się dokładnie jeden odsyłacz** *prowadzący do pewnej* **innej** *strony. Dla danej liczby całkowitej K powiemy, że strony są K***-dostępne***, jeżeli do każdej strony można dojść ze strony domowej przechodząc przez co najwyżej K* **odsyłaczy***.*

Zadanie

Napisz program, który na podstawie opisu stron i liczby całkowitej K wyznaczy **minimalną liczbę odsyłaczy, jakie trzeba dodać** *na stronach, tak aby stały się K-dostępne.*

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby całkowite N i K ($2 \leqslant N \leqslant 500\,000$, $1 \leqslant K \leqslant 20\,000$) — liczbę stron oraz pożądaną maksymalną liczbę odsyłaczy jakie trzeba przejść. Każdy z kolejnych N wierszy zawiera po dwie różne liczby całkowite A i B ($1 \leqslant A, B \leqslant N$) oznaczające, że na stronie A znajduje się odsyłacz prowadzący do strony B.

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać jedną liczbę całkowitą, minimalną liczbę odsyłaczy, jakie trzeba dodać, aby strony były K-dostępne.

Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
2
```

Natomiast dla danych:

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

poprawnym wynikiem jest:

```
3
```

Marcin Kubica, Paweł Gawrychowski
Tłumaczenie

CEOI 2006, dzień drugi, 05.07.2006

Środnia

Niektóre firmy wolą nie ujawniać zarobków swoich pracowników, a zwłaszcza zarobków pracowników na szczeblu kierowniczym. Wolą ukryć przed działaczami związkowymi jak źle opłacani są zwykli robotnicy, oraz uniknąć nudnych negocjacji dotyczących podwyżek i przywilejów (zarówno dla robotników, jak i dla samych działaczy związkowych). Z drugiej jednak strony, czasami firmy chętnie udostępniają takie dane do celów statystycznych i marketingowych.

Firma **Krzak S.A.** *chętnie odpowiada na pytania postaci: „Jaka jest* **środnia** *zarobków pracowników A, B, C i D?”.* **Środnia** *(***ang. meandian***) czterech liczb jest zdefiniowana jako średnia dwóch środkowych z nich. Bardziej formalnie, środnią ciągu a,b,c,d uzyskujemy sortując najpierw ten ciąg, a następnie obliczając $\frac{x+y}{2}$, gdzie x i y to drugi i trzeci element posortowanego ciągu.*

Twoim zadaniem jest odtworzenie dokładnych zarobków pracowników, zadając pytania podanej postaci. Zauważ, że zarobki niektórych pracowników nigdy nie mogą być odtworzone (np. zarobki najlepiej opłacanego pracownika, czyli prezesa firmy) nawet na podstawie odpowiedzi na wszystkie możliwe pytania.

Firma **Krzak S.A.** *zatrudnia N ($4 \leqslant N \leqslant 100$) pracowników, ponumerowanych od 1 do N. Zarobki każdego pracownika wyrażają się* **parzystą dodatnią liczbą całkowitą** *mniejszą lub równą 100 000 i* **żadnych dwóch pracowników nie zarabia tyle samo**.

Możesz korzystać z biblioteki implementującej funkcję `Meandian`*. Dla danych czterech różnych liczb całkowitych A,B,C,D ($1 \leqslant A,B,C,D \leqslant N$), funkcja ta oblicza środnią zarobków pracowników A, B, C i D.*

Zadanie

Napisz program, który mając dostęp do biblioteki, odtworzy **dokładne zarobki wszystkich pracowników***, z wyjątkiem tych, których zarobki nigdy nie mogą być odtworzone. Twój program może zadać* **co najwyżej** *1 000* **pytań**.

Biblioteka

Masz dostęp do biblioteki implementującej następujące trzy funkcje:

- `Init` — *funkcja bezargumentowa. Twój program powinien wywoływać tę funkcję* **dokładnie raz***, na początku. Nie ma ona argumentów, a jej wynikiem jest liczba całkowita N — liczba pracowników firmy* **Krzak S.A.**

  ```
  function Init : longint;
  int Init(void);
  ```

- `Meandian` — *tę funkcję należy wywoływać z czterema argumentami A,B,C,D —* **różnymi** *liczbami całkowitymi od 1 do N włącznie. Jej wynikiem jest liczba całkowita, środnia zarobków pracowników A, B, C i D.*

```
function Meandian(a,b,c,d : longint) : longint;
int Meandian(int,int,int,int);
```

- `Solution` — *ta funkcja powinna zostać wywołana na końcu Twojego programu. Jej argumentem jest tablica liczb całkowitych reprezentująca zarobki odpowiednich pracowników. Jeśli zarobki danego pracownika nie mogą być odtworzone, odpowiedni element tablicy powinien być równy* -1.
  Zwróć uwagę, że w każdym z dostępnych języków programowania przekazywana tablica jest indeksowana od 0. *To znaczy, że zarobki pracownika nr 1 powinny być w tablicy na pozycji* 0, *zarobki pracownika nr 2 na pozycji 1 itd.*

```
procedure Solution(var sol : array of longint);
void Solution(const int *);
```

Korzystanie z biblioteki w Pascalu

Kod źródłowy Twojego programu musi zawierać polecenie '`uses libmean`*'. Żeby skompilować i przetestować swój program możesz pobrać ze strony „Tasks” na serwerze zawodów przykładową bibliotekę. Umieść ją w katalogu, w którym będziesz kompilował program.*

Korzystanie z biblioteki w C/C++

Kod źródłowy Twojego programu musi zawierać dyrektywę '`#include "libmean.h"`*'. Żeby skompilować i przetestować swój program możesz pobrać ze strony „Tasks” na serwerze zawodów przykładową bibliotekę.*
Aby skompilować swój program, musisz go zlinkować z dostarczoną biblioteką, używając polecenia postaci:

```
gcc -o meandian meandian.c libmean.o
```

lub

```
g++ -o meandian meandian.cpp libmean.o
```

Testowanie

Testując swój program przy pomocy dostarczonej przykładowej biblioteki powinieneś podać na standardowym wejściu liczbę N, *a następnie* N *parzystych liczb całkowitych. Biblioteka wypisze komunikat informujący, czy Twoje rozwiązanie było poprawne. Utworzy ona także plik tekstowy* `meandian.log` *zawierający szczegóły działania Twojego programu.*
Poniżej przedstawiono kawałki kodu, które w żadnej mierze nie rozwiązują tego zadania, ale ilustrują użycie biblioteki w dostępnych językach programowania.

PASCAL

```
uses libmean;

var i, n : integer;
    arr : array[0..99] of longint;
    foo, bar, quux : integer;

begin
   n := Init;
   foo := Meandian(1, 2, 3, 4);
   bar := Meandian(4, 2, 3, 1);
   quux := Meandian(n, n-1, n-2, n-3);
   for i := 1 to n do
      arr[i-1] := 2*i;
   arr[3] := -1;
   Solution(arr);
end.
```

C/C++

```
#include "libmean.h"

int main(void)
{
   int i, n;
   int arr[100];
   int foo, bar, quux;

   n = Init();
   foo = Meandian(1, 2, 3, 4);
   bar = Meandian(4, 2, 3, 1);
   quux = Meandian(n, n-1, n-2, n-3);
   for (i=1; i<=n; ++i)
      arr[i-1] = 2*i;
   arr[3] = -1;
   Solution(arr);

   return 0;
}
```

Przykład (wejścia)

Przykładowe dane wejściowe:

```
10
100 500 200 400 250 300 350 600 550 410
```

Bibliografia

[1] *I Olimpiada Informatyczna 1993/1994*. Warszawa–Wrocław, 1994.

[2] *II Olimpiada Informatyczna 1994/1995*. Warszawa–Wrocław, 1995.

[3] *III Olimpiada Informatyczna 1995/1996*. Warszawa–Wrocław, 1996.

[4] *IV Olimpiada Informatyczna 1996/1997*. Warszawa, 1997.

[5] *V Olimpiada Informatyczna 1997/1998*. Warszawa, 1998.

[6] *VI Olimpiada Informatyczna 1998/1999*. Warszawa, 1999.

[7] *VII Olimpiada Informatyczna 1999/2000*. Warszawa, 2000.

[8] *VIII Olimpiada Informatyczna 2000/2001*. Warszawa, 2001.

[9] *IX Olimpiada Informatyczna 2001/2002*. Warszawa, 2002.

[10] *X Olimpiada Informatyczna 2002/2003*. Warszawa, 2003.

[11] *XI Olimpiada Informatyczna 2003/2004*. Warszawa, 2004.

[12] *XII Olimpiada Informatyczna 2004/2005*. Warszawa, 2005.

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

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

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

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

[17] I. N. Bronsztejn, K. A. Siemiendiajew. *Matematyka. Poradnik encyklopedyczny*. PWN, Warszawa, wydanie XIV, 1997.

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

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

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

[21] W. Lipski. *Kombinatoryka dla programistów*. WNT, Warszawa, 2004.

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

[23] R. Sedgewick. *Algorytmy w C++. Grafy*. RM, 2003.

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

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

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

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

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

[29] Ronald L. Graham, Donald E. Knuth, Oren Patashnik. *Matematyka konkretna*. PWN, Warszawa, 1996.

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

[31] Donald E. Knuth. *Sztuka programowania*. WNT, Warszawa, 2002.

[32] Steven S. Skiena, Miguel A. Revilla. *Wyzwania programistyczne*. WSiP, Warszawa, 2004.

[33] Franco P. Preparata, Michael Ian Shamos. *Geometria obliczeniowa. Wprowadzenie*. Helion, Warszawa, 2003.

[34] Joseph O'Rourke. *Computational geometry in C*. Cambridge University Press, 2005.

[35] M. Crochemore, W. Rytter. *Jewels of stringology*. World Scientific, 2002.

INFORMATYKA SPECJALNOŚCIĄ
MŁODYCH POLAKÓW

Niniejsza publikacja stanowi wyczerpujące źródło informacji o zawodach XIII Olimpiady Informatycznej, przeprowadzonych w roku szkolnym 2005/2006. Książka zawiera informacje dotyczące organizacji, regulaminu oraz wyników zawodów. Zawarto w niej także opis rozwiązań wszystkich zadań konkursowych. Do książki dołączony jest dysk CD-ROM zawierający wzorcowe rozwiązania i testy do wszystkich zadań XIII Olimpiady Informatycznej.

Książka zawiera też zadania z XII Bałtyckiej Olimpiady Informatycznej oraz XIII Olimpiady Informatycznej Krajów Europy Środkowej.

XIII 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. Książka może być wykorzystywana także jako pomoc do zajęć z algorytmiki na studiach informatycznych.

Olimpiada Informatyczna
jest organizowana przy współudziale
PROKOM
SOFTWARE SA

ISBN 83–922946–1–0