MINISTERSTWO EDUKACJI NARODOWEJ I SPORTU
UNIWERSYTET WROCŁAWSKI
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

XI OLIMPIADA INFORMATYCZNA
2003/2004

Olimpiada Informatyczna jest organizowana przy współudziale

PROKOM SOFTWARE SA

WARSZAWA, 2004

MINISTERSTWO EDUKACJI NARODOWEJ I SPORTU
UNIWERSYTET WROCŁAWSKI
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

XI OLIMPIADA INFORMATYCZNA
2003/2004

WARSZAWA, 2004

Autorzy tekstów:

dr Piotr Chrząstowski–Wachtel
Karol Cwalina
dr hab. Krzysztof Diks
dr hab. Wojciech Guzicki
mgr Łukasz Kowalik
dr Marcin Kubica
Paweł Parys
mgr Jakub Pawlewicz
prof. dr hab. Wojciech Rytter
mgr Krzysztof Sikora
Piotr Stańczyk
mgr Marcin Stefaniak
Bartosz Walczak
mgr Tomasz Waleń

Autorzy programów na dysku CD-ROM:

Michał Adamaszek
Tomasz Czajka
dr hab. Krzysztof Diks
mgr Łukasz Kowalik
dr Marcin Kubica
Tomasz Malesiński
mgr Marcin Mucha
Krzysztof Onak
Arkadiusz Paterek
mgr Jakub Pawlewicz
Rafał Rusin
mgr Piotr Sankowski
mgr Krzysztof Sikora
Paweł Wolff

Opracowanie i redakcja:

dr hab. Krzysztof Diks
Krzysztof Onak
mgr Tomasz Waleń

Skład:

Krzysztof Onak
mgr Tomasz Waleń

Pozycja dotowana przez Ministerstwo Edukacji Narodowej i Sportu.

Druk książki został sfinansowany przez PROKOM SOFTWARE SA

ISBN 83–917700–5–2

Spis treści

Krzysztof Diks

Wstęp

Drogi Czytelniku!

Olimpiada Informatyczna wkroczyła w swoje drugie dziesięciolecie i niezmiernie miłe jest, że już kolejne pokolenie olimpijczyków podtrzymuje dobrą passę swych poprzedników. Już po oddaniu do druku sprawozdań z poprzedniej, X-tej Olimpiady, odbyła się w Stanach Zjednoczonych XV-ta Międzynarodowa Olimpiada Informatyczna. Wszyscy nasi reprezentanci (zdobywcy czterech pierwszych miejsc w X-tej Olimpiadzie) zdobyli medale: Bartek Walczak i Filip Wolski – medale złote, Marcin Michalski – medal srebrny, a Michał Brzozowski – medal brązowy.

Najlepsi zawodnicy XI-tej Olimpiady nie spisują się gorzej. Ich występ w X-tej Bałtyckiej Olimpiadzie Informatycznej zakończył się spektakularnym sukcesem. Cała szóstka naszych reprezentantów zdobyła medale. Złoto wywalczyli: Filip Wolski (pierwsze miejsce w całym konkursie), Szymon Acedański, Jakub Łącki i Jakub Kallas. Medale srebrne przywieźli Tomasz Kuras i Adam Radziwończyk-Syta. Warte podkreślenia jest, że Polacy zdobyli cztery pierwsze miejsca w konkursie.

W tym roku Polska była organizatorem Olimpiady Informatycznej Europy Środkowej. Olimpiada miała miejsce w Rzeszowie, w dniach 12 - 18 lipca. Gościny Olimpiadzie udzieliła Wyższa Szkoła Informatyki i Zarządzania w Rzeszowie, za co chciałbym jeszcze raz gorąco podziękować władzom szkoły i wszystkim jej pracownikom, którzy z oddaniem pomagali w organizacji tej ważnej imprezy. W Rzeszowie nasi reprezentanci okazali się niezmiernie gościnni i oddali pierwsze miejsce Luce Kalinovcicowi z Chorwacji. Wszyscy nasi reprezentanci zdobyli jednak medale: złote – Filip Wolski i Bartek Romański, srebrny – Jakub Łącki i brązowy – Tomasz Kuras. Teraz przed nimi Międzynarodowa Olimpiada Informatyczna w Atenach. Mam wielką nadzieję, że w roku olimpijskim wyjazd Polaków do Aten zakończy się wielkim sukcesem.

Żaden z wyżej opisanych sukcesów nie byłby możliwy bez całorocznej pracy wielu osób związanych z krajową Olimpiadą Informatyczną. Należą do nich zarówno uczniowie, którzy poświęcają dziesiątki godzin na przygotowanie się do zawodów, ich nauczyciele opiekujący się na codzień swoimi wychowankami, jak i organizatorzy, którzy starają się, żeby Olimpiada była z roku na rok coraz lepsza.

Prezentowana książeczka zawiera dokładny opis przebiegu XI-tej Olimpiady Informatycznej. Jednak najcenniejszą jej częścią są autorskie opisy rozwiązań zadań olimpijskich. Warte podkreślenia jest to, że wśród autorów dominują jeszcze niedawni olimpijczycy. Może to właśnie było powodem tego, że tegoroczne zadania okazały się wyjątkowo trudne.

Oprócz zadań z XI-tej Olimpiady, w przedstawianej książeczce znajdują się też zadania z XV-tej Międzynarodowej Olimpiady Informatyczne i X-tej Bałtyckiej Olimpiady Informatycznej. Jednym z autorów zadań Bałtyckiej Olimpiady Informatycznej jest prof. Wojciech Rytter. Zamieszczamy jego autorski opis rozwiązania zadania *Waga*.

Od trzech lat Olimpiada Informatyczna jest współorganizatorem cieszącego się niezwykłym powodzeniem konkursu programistycznego *Pogromcy Algorytmów*. W tym roku postanowiliśmy zawrzeć w „niebieskiej książeczce” opisy autorskich rozwiązań zadań z ostatniej edycji *Pogromców*. Zachęcamy wszystkich do udziału w tych zawodach. Jest to znakomity trening przed Olimpiadą.

Do książeczki jest dołączony dysk z programami wzorcowymi dla zadań z XI-tej Olimpiady Informatycznej oraz trudniejszych zadań z *Pogromców Algorytmów*. Na dysku znajdują się też dane testowe na podstawie których oceniano rozwiązania zawodników.

Na zakończenie chciałbym podziękować wszystkim członkom Komitetu Głównego, jurorom i innym współorganizatorom, za pełne zaangażowanie w przygotowanie i realizację XI-tej Olimpiady Informatycznej. Kolejny raz chciałbym także wyrazić wdzięczność firmie PROKOM SOFTWARE SA za modelowe wspieranie młodych polskich informatyków.

Krzysztof Diks

Sprawozdanie z przebiegu XI Olimpiady Informatycznej 2003/2004

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 2003/2004 odbyły się zawody XI 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.

7 października 2003 r. rozesłano plakaty zawierające zasady organizacji zawodów I stopnia oraz zestaw 5 zadań konkursowych do 3781 szkół i zespołów szkół młodzieżowych ponadgimnazjalnych oraz do wszystkich kuratorów i koordynatorów edukacji informatycznej. Zawody I stopnia rozpoczęły się dnia 20 października 2003 r. Ostatecznym terminem nadsyłania prac konkursowych był 17 listopada 2003 roku.

Zawody II i III stopnia były dwudniowymi sesjami stacjonarnymi, poprzedzonymi jednodniowymi sesjami próbnymi. Zawody II stopnia odbyły się w pięciu okręgach: Warszawie, Wrocławiu, Toruniu, Katowicach i Krakowie oraz w Sopocie i Rzeszowie, w dniach 10–12.02.2004 r., natomiast zawody III stopnia odbyły się w ośrodku firmy Combidata Poland S.A. w Sopocie, w dniach 30.03–3.04.2004 r.

Uroczystość zakończenia XI Olimpiady Informatycznej odbyła się w dniu 3.04.2004 r. w Sali Posiedzeń Urzędu Miasta w Sopocie z udziałem pana Tadeusza Sławeckiego, podsekretarza stanu w Ministerstwie Edukacji Narodowej i Sportu.

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:
prof. dr hab. Zbigniew Czech (Politechnika Śląska)
mgr Jerzy Dałek (Ministerstwo Edukacji Narodowej i Sportu)
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. Wojciech Rytter (Uniwersytet Warszawski)
mgr Krzysztof J. Święcicki (Ministerstwo Edukacji Narodowej i Sportu)
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. 23 stycznia 2004r. przeprowadzono seminarium przygotowujące zawody II stopnia.

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:
dr hab. Grzegorz Jarzembski, prof. UMK (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:
mgr Anna Beata Kwiatkowska (IV Liceum Ogólnokształcące w Toruniu)
dr Krzysztof Skowronek (V Liceum Ogólnokształcące w Toruniu)

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

Górnośląski Komitet Okręgowy
przewodniczący:
prof. dr hab. Zbigniew Czech (Politechnika Śląska w Gliwicach)
zastępca przewodniczącego:
mgr 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)
mgr inż. Marcin Ciura (Politechnika Śląska w Gliwicach)
mgr inż. Marcin Szołtysek (Politechnika Śląska w Gliwicach)
mgr Jacek Widuch (Politechnika Śląska w Gliwicach)
mgr Wojciech Wieczorek (Uniwersytet Śląski w Sosnowcu)

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

Komitet Okręgowy w Krakowie
przewodniczący:
prof. dr hab. Paweł Idziak (Uniwersytet Jagielloński)
zastępca przewodniczącego:
dr Maciej Ślusarek (Uniwersytet Jagielloński)
sekretarz:
mgr Edward Szczypka (Uniwersytet Jagielloński)

członkowie:
mgr Henryk Białek (Kuratorium Oświaty w Krakowie)
dr inż. Janusz Majewski (Akademia Górniczo-Hutnicza w Krakowie)

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

Jury Olimpiady Informatycznej

W pracach Jury, które nadzorował Krzysztof Diks, a którymi kierował Krzysztof Stencel, brali udział pracownicy, doktoranci i studenci Instytutu Informatyki Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego.
Michał Adamaszek
mgr Krzysztof Ciebiera
Karol Cwalina
Tomasz Czajka
Wojciech Dudek
Tomasz Idziaszek
mgr Łukasz Kowalik
Tomasz Malesiński
Marcin Michalski
mgr Marcin Mucha
Anna Niewiarowska
Krzysztof Onak
Paweł Parys
Arkadiusz Paterek
mgr Jakub Pawlewicz
Marcin Pilipczuk
Jakub Radoszewski
Rafał Rusin
mgr Piotr Sankowski
mgr Krzysztof Sikora
Piotr Stańczyk
mgr Marcin Stefaniak
mgr Tomasz Waleń
Paweł Wolff
Marek Żylak

ZAWODY I STOPNIA

W XI Olimpiadzie Informatycznej wzięło udział 805 zawodników.

Decyzją Komitetu Głównego do zawodów zostało dopuszczonych 40 uczniów gimnazjów. Byli to uczniowie z następujących gimnazjów:

- Gimnazjum nr 24, Gdynia: 15 uczniów
- Gimnazjum nr 50, Bydgoszcz: 5

- Gimnazjum nr 16, Szczecin: 3
- Gimnazjum im. Św. Jadwigi Królowej, Kielce: 2
- Gimnazjum nr 7, Chorzów: 1
- Gimnazjum nr 8, Elbląg: 1
- Gimnazjum, Goczałkowice-Zdrój: 1
- Gimnazjum nr 2, Kamienna Góra: 1
- Gimnazjum nr 15, Kielce: 1
- Kolegium Szkół Prywatnych, Kielce: 1
- Gimnazjum nr 16 im. S. Batorego, Kraków: 1
- Gimnazjum nr 52 Ojców Pijarów, Kraków: 1
- Gimnazjum nr 18, Lublin: 1
- Gimnazjum nr 11, Łódź: 1
- Gimnazjum, Łysa Góra: 1
- Gimnazjum nr 16, Szczecin: 1
- Gimnazjum Akademickie, Toruń: 1
- Gimnazjum nr 9, Tychy: 1
- Publiczne Gimnazjum, Węgrów: 1

Kolejność województw pod względem liczby uczestników była następująca:

pomorskie	114
mazowieckie	109
małopolskie	108
śląskie	85
kujawsko-pomorskie	71
dolnośląskie	50
zachodniopomorskie	48
podkarpackie	44
lubelskie	37
wielkopolskie	34
łódzkie	26
podlaskie	25
świętokrzyskie	19
opolskie	13
lubuskie	11
warmińsko-mazurskie	11

W zawodach I stopnia najliczniej reprezentowane były szkoły:

V LO im. A. Witkowskiego, Kraków	54 uczniów
III LO im. Marynarki Wojennej RP, Gdynia	45
XIV LO im St. Staszica, Warszawa	35
VI LO im. W. Sierpińskiego, Gdynia	22
XIII LO, Szczecin	18
VI LO im. J. i J. Śniadeckich, Bydgoszcz	17
Gimnazjum nr 24, Gdynia	15
VIII LO im. A. Mickiewicza, Poznań	12
XIV LO im. Polonii Belgijskiej, Wrocław	11
VIII LO im. M. Skłodowskiej-Curie, Katowice	10
IV LO im. T. Kościuszki, Toruń	10
III LO im. A. Mickiewicza, Wrocław	9
I LO im. A. Mickiewicza, Białystok	8
Katolickie LO Ojców Pijarów, Kraków	8
I LO im. C. K. Norwida, Bydgoszcz	7
IV LO im. M. Kopernika, Rzeszów	7
X LO im. prof. S. Banacha, Toruń	7
Gimnazjum nr 50, Bydgoszcz	5
I LO im. St. Staszica, Chrzanów	5
I LO im. M. Kopernika, Gdańsk	5
II LO im. M. Konopnickiej, Opole	5
Zespół Szkół Elektronicznych, Rzeszów	5
Gimnazjum i Liceum Akademickie, Toruń	5

Najliczniej reprezentowane były miasta:

Gdynia	89 uczniów
Kraków	79
Warszawa	76
Bydgoszcz	35
Szczecin	33
Toruń	26
Wrocław	24
Rzeszów	21
Katowice	17
Poznań	15
Lublin	13
Białystok	13
Częstochowa	10
Gdańsk	10
Kielce	10
Łódź	9
Radom	9

Opole	8
Bielsko-Biała	7
Dąbrowa Górnicza	6
Gliwice	6
Słupsk	6
Zielona Góra	6
Chorzów	5
Chrzanów	5
Cieszyn	5
Koszalin	5
Olsztyn	5
Rybnik	5
Sosnowiec	5
Stalowa Wola	5

Zawodnicy uczęszczali do następujących klas:

do klasy I	gimnazjum	4 zawodników
do klasy II	gimnazjum	15
do klasy III	gimnazjum	21
do klasy I	szkoły ponadgimnazjalnej	118
do klasy II	szkoły ponadgimnazjalnej	313
do klasy III	szkoły średniej	19
do klasy IV	szkoły średniej	298
do klasy V	szkoły średniej	12

5 zawodników nie podało informacji, do której klasy uczęszczali.

W zawodach I stopnia zawodnicy mieli do rozwiązania pięć zadań: „Gra", „PIN-kod", „Sznurki", „Szpiedzy" i „Zawody".

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

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

- Gra

	liczba zawodników	czyli
100 pkt.	21	2,6%
75–99 pkt.	1	0,1%
50–74 pkt.	2	0,3%
1–49 pkt.	240	29,8%
0 pkt.	290	36,0%
brak rozwiązania	251	31,2%

- PIN-kod

	liczba zawodników	czyli
100 pkt.	429	53,3%
75–99 pkt.	75	9,3%
50–74 pkt.	27	3,4%
1–49 pkt.	91	11,3%
0 pkt.	60	7,5%
brak rozwiązania	123	15,2%

- Sznurki

	liczba zawodników	czyli
100 pkt.	31	3,8%
75–99 pkt.	15	1,9%
50–74 pkt.	15	1,9%
1–49 pkt.	129	16%
0 pkt.	304	37,8%
brak rozwiązania	311	38,6%

- Szpiedzy

	liczba zawodników	czyli
100 pkt.	208	25,8%
75–99 pkt.	102	12,7%
50–74 pkt.	97	12,1%
1–49 pkt.	187	23,2%
0 pkt.	102	12,7%
brak rozwiązania	109	13,5%

- Zawody

	liczba zawodników	czyli
100 pkt.	75	9,3%
75–99 pkt.	101	12,6%
50–74 pkt.	135	16,8%
1–49 pkt.	177	22,0%
0 pkt.	109	13,5%
brak rozwiązania	208	25,8%

W sumie za wszystkie 5 zadań konkursowych:

SUMA	liczba zawodników	czyli
500 pkt.	7	0,8%
375–499 pkt.	31	3,9%
250–374 pkt.	194	24,1%
1–249 pkt.	477	59,3%
0 pkt.	96	11,9%

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 334 zawodników, którzy osiągnęli w zawodach I stopnia wynik nie mniejszy niż 200 pkt.

Pięciu zawodników nie stawiło się na zawody. W zawodach II stopnia uczestniczyło 329 zawodników.

Zawody II stopnia odbyły się w dniach 10–12 lutego 2004 r. w pięciu stałych okręgach oraz w Sopocie i Rzeszowie:

- w Toruniu — 39 zawodników z następujących województw:
  - kujawsko-pomorskie (39)
- we Wrocławiu — 57 zawodników z następujących województw:
  - dolnośląskie (14)
  - lubuskie (2)
  - opolskie (8)
  - wielkopolskie (15)
  - zachodniopomorskie (18)
- w Warszawie — 65 zawodników z następujących województw:
  - mazowieckie (46)
  - podlaskie (10)
  - świętokrzyskie (6)
  - warmińsko-mazurskie (3)
- w Krakowie — 44 zawodników z następujących województw:
  - małopolskie (44)
- w Gliwicach — 34 zawodników z następujących województw:
  - łódzkie (6)
  - małopolskie (2)
  - śląskie (26)
- w Sopocie — 69 zawodników z następujących województw:
  - pomorskie (67)
  - zachodniopomorskie (2)
- w Rzeszowie — 20 zawodników z następujących województw:
  - lubelskie (4)
  - podkarpackie (15)
  - zachodniopomorskie (1)

W zawodach II stopnia najliczniej reprezentowane były szkoły:

III LO im. Marynarki Wojennej RP, Gdynia	36 zawodników
V LO im. A. Witkowskiego, Kraków	34
XIV LO im. St. Staszica, Warszawa	22
VI Liceum Ogólnokształcące, Gdynia	17
XIII LO, Szczecin	14
VI LO im. J. i J. Śniadeckich, Bydgoszcz	11
VIII LO im. A. Mickiewicza, Poznań	9
IV LO im. T. Kościuszki, Toruń	8
I LO im. A. Mickiewicza, Białystok	6
Gimnazjum nr 24, Gdynia	6
VIII LO im. M. Skłodowskiej-Curie, Katowice	5
IV LO im. M. Kopernika, Rzeszów	5
II LO im. M. Konopnickiej, Opole	4
X LO im. prof. S. Banacha, Toruń	4
XXVII LO im. T. Czackiego, Warszawa	4
XIV LO im. Polonii Belgijskiej, Wrocław	4
I LO im. C. K. Norwida, Bydgoszcz	3
Gimnazjum nr 50, Bydgoszcz	3
I LO im. M. Kopernika, Gdańsk	3
I LO im. M. Kopernika, Łódź	3
LO im. KEN, Stalowa Wola	3
XXVIII LO im. J. Kochanowskiego, Warszawa	3

Najliczniej reprezentowane były miasta:

Gdynia	60
Kraków	42
Warszawa	42
Bydgoszcz	20
Szczecin	16
Toruń	15
Poznań	10
Rzeszów	8
Wrocław	8
Katowice	7
Opole	7
Białystok	6
Gdańsk	5
Chorzów	4
Lublin	4
Łódź	4
Kielce	3
Stalowa Wola	3

W dniu 10 lutego odbyła się sesja próbna, na której zawodnicy rozwiązywali nie liczące się do ogólnej klasyfikacji zadanie "Most". W dniach konkursowych zawodnicy rozwiązywali zadania: „Bramki”, „Jaskinia”, „Przeprawa” oraz „Turniej”, każde oceniane w skali od 0 do 100 punktów.

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

- Most (zadanie próbne)

	liczba zawodników	czyli
100 pkt.	53	16,1%
75–99 pkt.	4	1,2%
50–74 pkt.	2	0,6%
1–49 pkt.	217	66,0%
0 pkt.	46	14,0%
brak rozwiązania	7	2,1%

- Bramki

	liczba zawodników	czyli
100 pkt.	3	0,9%
75–99 pkt.	45	13,7%
50–74 pkt.	29	8,8%
1–49 pkt.	135	41,0%
0 pkt.	49	14,9%
brak rozwiązania	68	20,7%

- Jaskinia

	liczba zawodników	czyli
100 pkt.	0	0,0%
75–99 pkt.	0	0,0%
50–74 pkt.	1	0,3%
1–49 pkt.	109	33,1%
0 pkt.	187	56,9%
brak rozwiązania	32	9,7%

- Przeprawa

	liczba zawodników	czyli
100 pkt.	15	4,5%
75–99 pkt.	14	4,3%
50–74 pkt.	43	13,1%
1–49 pkt.	197	59,8%
0 pkt.	46	14,0%
brak rozwiązania	14	4,3%

- Turniej

	liczba zawodników	czyli
100 pkt.	2	0,6%
75–99 pkt.	17	5,2%
50–74 pkt.	20	6,1%
1–49 pkt.	212	64,4%
0 pkt.	51	15,5%
brak rozwiązania	27	8,2%

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

SUMA	liczba zawodników	czyli
400 pkt.	0	0,0%
300–399 pkt.	1	0,3%
200–299 pkt.	15	4,6%
1–199 pkt.	298	90,6%
0 pkt.	15	4,5%

Wszystkim zawodnikom przesłano informację o uzyskanych wynikach, a na stronie Olimpiady dostępne były testy według których sprawdzano rozwiązania.

ZAWODY III STOPNIA

Zawody III stopnia odbyły się w ośrodku firmy Combidata Poland S.A. w Sopocie w dniach od 30 marca do 3 kwietnia 2004 r.

Do zawodów III stopnia zakwalifikowano 46 najlepszych uczestników zawodów II stopnia, którzy uzyskali wynik nie mniejszy niż 152 pkt. Jeden zawodnik nie zgłosił się na zawody.

Zawodnicy uczęszczali do szkół w następujących województwach:

pomorskie	10
kujawsko-pomorskie	8
mazowieckie	7
śląskie	6
małopolskie	3
wielkopolskie	3
zachodniopomorskie	3
dolnośląskie	1
łódzkie	1
podkarpackie	1
podlaskie	1
warmińsko-mazurskie	1

Niżej wymienione szkoły miały w finale więcej niż jednego zawodnika:

III LO im. Marynarki Wojennej RP, Gdynia	8 zawodników

XIV LO im. St. Staszica, Warszawa	5
IV LO im. T. Kościuszki, Toruń	3
XIII LO, Szczecin	3
VIII LO im. A. Mickiewicza, Poznań	3
V LO im. A. Witkowskiego, Kraków	3
VI LO im. J. i J. Śniadeckich, Bydgoszcz	3
VIII LO im. M. Skłodowskiej-Curie, Katowice	2

30 marca odbyła się sesja próbna, na której zawodnicy rozwiązywali nie liczące się do ogólnej klasyfikacji zadanie: „Zgadywanka”. W dniach konkursowych zawodnicy rozwiązywali zadania: „Misie-Patysie”, „Wschód-Zachód”, „Wyspy”, „Kaglony” i „Maksymalne rzędy permutacji”, 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:

- Zgadywanka (zadanie próbne)

	liczba zawodników	czyli
100 pkt.	6	13,3%
75–99 pkt.	2	4,5%
50–74 pkt.	0	0,0%
1–49 pkt.	1	2,2%
0 pkt.	34	75,6%
brak rozwiązania	2	4,4%

- Misie-Patysie

	liczba zawodników	czyli
100 pkt.	6	13,3%
75–99 pkt.	4	8,9%
50–74 pkt.	1	2,2%
1–49 pkt.	20	44,4%
0 pkt.	7	15,6%
brak rozwiązania	7	15,6%

- Wschód-Zachód

	liczba zawodników	czyli
100 pkt.	3	6,7%
75–99 pkt.	3	6,7%
50–74 pkt.	5	11,1%
1–49 pkt.	14	31,1%
0 pkt.	16	35,5%
brak rozwiązania	4	8,9%

- Wyspy

	liczba zawodników	czyli
100 pkt.	5	11,1%
75–99 pkt.	2	4,5%
50–74 pkt.	7	15,6%
1–49 pkt.	19	42,2%
0 pkt.	5	11,1%
brak rozwiązania	7	15,5%

- Kaglony

	liczba zawodników	czyli
100 pkt.	2	4,5%
75–99 pkt.	0	0,0%
50–74 pkt.	0	0,0%
1–49 pkt.	16	35,5%
0 pkt.	22	48,9%
brak rozwiązania	5	11,1%

- Maksymalne rzędy permutacji

	liczba zawodników	czyli
100 pkt.	1	2,2%
75–99 pkt.	1	2,2%
50–74 pkt.	0	0,0%
1–49 pkt.	34	75,6%
0 pkt.	7	15,6%
brak rozwiązania	2	4,4%

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

SUMA	liczba zawodników	czyli
500 pkt.	0	0,0%
375–499 pkt.	2	4,5%
250–374 pkt.	2	4,5%
125–249 pkt.	7	15,5%
1–124 pkt.	33	73,3%
0 pkt.	1	2,2%

W dniu 3 kwietnia 2004 roku, w Sali Posiedzeń Urzędu Miasta w Sopocie, ogłoszono wyniki finału XI Olimpiady Informatycznej 2003/2004 i rozdano nagrody ufundowane przez: PROKOM Software S.A., Ogólnopolską Fundację Edukacji Komputerowej, Wydawnictwa Naukowo-Techniczne i Olimpiadę Informatyczną.

Laureaci I, II i III miejsca otrzymali, odpowiednio, złote, srebrne i brązowe medale. Poniżej zestawiono listę wszystkich laureatów i finalistów oraz nagród, które otrzymali:

(1) Bartłomiej Romański, XIV LO im. St. Staszica w Warszawie, laureat I miejsca, złoty medal, 469 pkt. (puchar — Olimpiada Informatyczna; notebook — PROKOM; roczny abonament na książki — WNT)

(2) Filip Wolski, III LO im. Marynarki Wojennej RP w Gdyni, laureat I miejsca, złoty medal, 421 pkt. (notebook — PROKOM)

(3) Jakub Łącki, III LO im. Marynarki Wojennej RP w Gdyni, laureat II miejsca, srebrny medal, 292 pkt. (notebook — PROKOM)

(4) Tomasz Kuras, V LO im. A. Witkowskiego w Krakowie, laureat II miejsca, srebrny medal, 278 pkt. (notebook — PROKOM)

(5) Jakub Kallas, III LO im. Marynarki Wojennej RP w Gdyni, laureat II miejsca, srebrny medal, 228 pkt. (aparat cyfrowy — PROKOM)

(6) Szymon Acedański, VIII LO im. Marii Skłodowskiej-Curie w Katowicach, laureat II miejsca, srebrny medal, 224 pkt. (aparat cyfrowy — PROKOM)

(7) Adam Radziwończyk-Syta, LXVII LO w Warszawie, laureat II miejsca, srebrny medal, 212 pkt. (aparat cyfrowy — PROKOM)

(8) Michał Jaszczyk, XIII LO w Szczecinie, laureat II miejsca, srebrny medal, 207 pkt. (aparat cyfrowy — PROKOM)

(9) Piotr Danilewski, V LO im. A. Witkowskiego w Krakowie, laureat III miejsca, brązowy medal, 188 pkt. (aparat cyfrowy — PROKOM)

(10) Tomasz Sadura, II LO im. A. F. Modrzewskiego w Rybniku, laureat III miejsca, brązowy medal, 165 pkt. (aparat cyfrowy — PROKOM)

(11) Andrzej Grzywocz, III LO im. Marynarki Wojennej RP w Gdyni, laureat III miejsca, brązowy medal, 146 pkt. (aparat cyfrowy — PROKOM)

(12) Robert Dyczkowski, XIV LO im. St. Staszica w Warszawie, laureat III miejsca, brązowy medal, 123 pkt. (aparat cyfrowy — PROKOM)

(13) Tomasz Kulczyński, Gimnazjum nr 50 w Bydgoszczy, laureat III miejsca, brązowy medal, 120 pkt. (odtwarzacz mp3 — PROKOM)

(14) Aleksander Piotrowski, VI LO im. J. i J. Śniadeckich w Bydgoszczy, laureat III miejsca, brązowy medal, 118 pkt. (odtwarzacz mp3 — PROKOM)

(15) Dawid Sieradzki, VIII LO im. A. Mickiewicza w Poznaniu, laureat III miejsca, brązowy medal, 116 pkt. (odtwarzacz mp3 — PROKOM)

(16) Piotr Biliński, VIII LO im. A. Mickiewicza w Poznaniu, laureat III miejsca, brązowy medal, 109 pkt. (odtwarzacz mp3 — PROKOM)

(17) Marek Materzok, LO im. H. Sienkiewicza w Nowej Rudzie, laureat III miejsca, brązowy medal, 108 pkt. (odtwarzacz mp3 — PROKOM)

Lista pozostałych finalistów w kolejności alfabetycznej:

- Łukasz Bieniasz-Krzywiec, I LO im. B. Krzywoustego w Słupsku (pamięć wymienna USB — OFEK)

- Adam Blokus, III LO im. Marynarki Wojennej RP w Gdyni (pamięć wymienna USB — OFEK)
- Łukasz Bogacki, XIII LO w Szczecinie (odtwarzacz mp3 — PROKOM)
- Rafał Bryk, IV LO im. T. Kościuszki w Toruniu (odtwarzacz mp3 — PROKOM)
- Mateusz Brzeszcz, VIII LO im. M. Skłodowskiej-Curie w Katowicach (odtwarzacz mp3 — PROKOM)
- Wojciech Budniak, XIII LO w Szczecinie (pamięć wymienna USB — OFEK)
- Paweł Chodaczek, XIV LO im. St. Staszica w Warszawie (odtwarzacz mp3 — PROKOM)
- Sebastian Chojniak, I Liceum Ogólnokształcące w Piszu (pamięć wymienna USB — OFEK)
- Daniel Czajka, LO im. Komisji Edukacji Narodowej w Stalowej Woli (odtwarzacz mp3 — PROKOM)
- Piotr Daszkiewicz, I LO im. M. Kopernika w Gdańsku (pamięć wymienna USB — OFEK)
- Przemysław Dobrowolski, I LO im. Wł. Broniewskiego w Bełchatowie (pamięć wymienna USB — OFEK)
- Adam Gaj, V LO im. A. Witkowskiego w Krakowie (pamięć wymienna USB — OFEK)
- Adam Gawarkiewicz, X LO im. prof. S. Banacha w Toruniu (pamięć wymienna USB — OFEK)
- Paweł Gosztyła, I LO im. L. Kruczkowskiego w Tychach (pamięć wymienna USB — OFEK)
- Krzysztof Jakubowski, I LO im. A. Mickiewicza w Białymstoku (pamięć wymienna USB — OFEK)
- Maciej Jaśkowski, VI LO im. J. i J. Śniadeckich w Bydgoszczy (pamięć wymienna USB — OFEK)
- Michał Miodek, Zespół Szkół Zawodowych nr 10 "Elektronik" w Sosnowcu (odtwarzacz mp3 — PROKOM)
- Mikołaj Kajetan Schmidt, IV LO im. T. Kościuszki w Toruniu (pamięć wymienna USB — OFEK)
- Tomasz Kłos, III LO im. Marynarki Wojennej RP w Gdyni (pamięć wymienna USB — OFEK)
- Maciej Kokociński, VIII LO im. A. Mickiewicza w Poznaniu (pamięć wymienna USB — OFEK)

- Juliusz Kopczewski, II Społeczne LO w Warszawie (pamięć wymienna USB — OFEK)
- Marcin Nowak-Przygodzki, III LO im. Marynarki Wojennej RP w Gdyni (pamięć wymienna USB — OFEK)
- Maciej Pawlisz, III LO im. Marynarki Wojennej RP w Gdyni (pamięć wymienna USB — OFEK)
- Norbert Potocki, XIV LO im. St. Staszica w Warszawie (pamięć wymienna USB — OFEK)
- Mateusz Wójcik, I LO im. M. Kopernika w Katowicach (pamięć wymienna USB — OFEK)
- Piotr Wygocki, IV LO im. T. Kościuszki w Toruniu (odtwarzacz mp3 — PROKOM)
- Piotr Wysocki, VI LO im. J. i J. Śniadeckich w Bydgoszczy (pamięć wymienna USB — OFEK)
- Paweł Zieliński, XIV LO im. St. Staszica w Warszawie (pamięć wymienna USB — OFEK)

Wszyscy uczestnicy finałów otrzymali książki ufundowane przez WNT.

Ogłoszono komunikat o powołaniu reprezentacji Polski na:

- Międzynarodową Olimpiadę Informatyczną w składzie:
  (1) Bartłomiej Romański
  (2) Filip Wolski
  (3) Jakub Łącki
  (4) Tomasz Kuras

  zawodnikami rezerwowymi zostali:

  (5) Jakub Kallas
  (6) Szymon Acedański
- Olimpiadę Informatyczną Europy Środkowej w składzie:
  (1) Bartłomiej Romański
  (2) Filip Wolski
  (3) Jakub Łącki
  (4) Tomasz Kuras

  II drużyna:

  (5) Jakub Kallas
  (6) Adam Radziwończyk-Syta
  (7) Andrzej Grzywocz

(8) Tomasz Kulczyński

zawodnikami rezerwowymi zostali:

(9) Szymon Acedański

(10) Michał Jaszczyk

- Bałtycką Olimpiadę Informatyczną w składzie:

  (1) Bartłomiej Romański

  (2) Filip Wolski

  (3) Jakub Łącki

  (4) Tomasz Kuras

  (5) Jakub Kallas

  (6) Szymon Acedański

  zawodnikami rezerwowymi zostali:

  (5) Adam Radziwończyk-Syta

  (6) Michał Jaszczyk

- obóz czesko-polsko-słowacki: członkowie reprezentacji oraz zawodnicy rezerwowi powołani na Międzynarodową Olimpiadę Informatyczną,
- obóz rozwojowo-treningowy im. A. Kreczmara dla finalistów Olimpiady Informatycznej: reprezentanci na Międzynarodową Olimpiadę Informatyczną oraz laureaci i finaliści Olimpiady, którzy nie byli w ostatnim roku szkolnym w programowo najwyższych klasach szkół ponadgimnazjalnych.

Sekretariat wystawił łącznie 17 zaświadczeń o uzyskaniu tytułu laureata i 28 zaświadczeń o uzyskaniu tytułu finalisty XI Olimpiady Informatycznej.

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

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

- Michał Baczyński (Uniwersytet Śląski, Instytut Matematyki, Katowice)
  - Szymon Acedański (laureat II miejsca)
- Ireneusz Bujnowski (I LO, Białystok)
  - Krzysztof Jakubowski (finalista)
- Roman Chojniak (TP SIRCOM Szkolenia i Rekreacja, Warszawa)
  - Sebastian Chojniak (finalista)
- Jadwiga Cogiel (studentka Uniwersytetu Warszawskiego)

  - Michał Miodek (finalista)
- Stanisław Czajka (Huta Stalowa Wola)
  - Daniel Czajka (finalista)
- Tomasz Czajka (student Uniwersytetu Warszawskiego)
  - Adam Radziwończyk-Syta (laureat II miejsca)
  - Juliusz Kopczewski (finalista)
- Andrzej Daszke (I LO im. M. Kopernika, Gdańsk)
  - Piotr Daszkiewicz (finalista)
- Krzysztof Dobrowolski (Elektrownia Bełchatów, Rogowiec)
  - Przemysław Dobrowolski (finalista)
- Piotr Dybicz (SP im. Przymierza Rodzin, Warszawa)
  - Bartłomiej Romański (laureat I miejsca)
  - Robert Dyczkowski (laureat III miejsca)
- Andrzej Dyrek (V LO im. A. Witkowskiego, Kraków)
  - Tomasz Kuras (laureat II miejsca)
  - Piotr Danilewski (laureat III miejsca)
  - Adam Gaj (finalista)
- Roman Kula (I LO im. M. Kopernika, Katowice)
  - Mateusz Wójcik (finalista)
- Krzysztof Lazaj (II LO, Rybnik)
  - Tomasz Sadura (laureat III miejsca)
- Zbigniew Ledóchowski (I LO im. B. Krzywoustego, Słupsk)
  - Łukasz Bieniasz-Krzywiec (finalista)
- Bartosz Nowierski (student Politechniki Poznańskiej) i Wojciech Roszczyński (VIII LO, Poznań)
  - Dawid Sieradzki (laureat III miejsca)
  - Piotr Biliński (laureat III miejsca)
  - Maciej Kokociński (finalista)
- Elżbieta Pławińska-Podkrólewicz (IV LO im. T. Kościuszki, Toruń)
  - Piotr Wygocki (finalista)

   - Rafał Bryk (finalista)
- Izabela Potocka (Główny Inspektorat Pracy, Warszawa)
   - Norbert Potocki (finalista)
- Mirosław Schmidt (Centrum Astronomiczne im. M. Kopernika, Toruń)
   - Mikołaj Kajetan Schmidt (finalista)
- Stefan Senczyna (I LO im. L. Kruczkowskiego, Tychy)
   - Paweł Gosztyła (finalista)
- Ryszard Szubartowski (III LO im. Marynarki Wojennej RP, Gdynia)
   - Filip Wolski (laureat I miejsca)
   - Jakub Łącki (laureat II miejsca)
   - Jakub Kallas (laureat II miejsca)
   - Marcin Nowak-Przygodzki (finalista)
   - Maciej Pawlisz (finalista)
   - Adam Blokus (finalista)
   - Tomasz Kłos (finalista)
- Michał Szuman (XIII LO, Szczecin)
   - Michał Jaszczyk (laureat II miejsca)
   - Łukasz Bogacki (finalista)
   - Wojciech Budniak (finalista)
- Witold Telus (ZSO im. H. Sienkiewicza, Nowa Ruda)
   - Marek Materzok (laureat III miejsca)
- Iwona Waszkiewicz (ZSO, Bydgoszcz)
   - Tomasz Kulczyński (laureat III miejsca)

Zgodnie z rozporządzeniem MENiS w sprawie olimpiad, wyróżnieni nauczyciele otrzymają nagrody pieniężne.

Warszawa, 11 czerwca 2004 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 ocenę celującą na koniec roku szkolnego oraz są zwolnieni z egzaminu maturalnego z informatyki na zasadach określonych w rozporządzeniu MEN z dnia 21 III 2001 r. z późniejszymi zmianami (2001/29/323, 128/1419, 2002/46/433, 155/1289, 214/1807, 2003/26/225) w sprawie warunków i sposobu oceniania, klasyfikowania i promowania uczniów i słuchaczy oraz przeprowadzenia egzaminów i sprawdzianów w szkołach publicznych.

(2) Laureaci zawodów III stopnia, a także finaliści, są zwolnieni w części lub w całości z egzaminów wstępnych do tych szkół wyższych, których senaty podjęły odpowiednie uchwały zgodnie z przepisami ustawy z dnia 12 września 1990 r. o szkolnictwie wyższym (Dz. U. 90.65.385).

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

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

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

(6) 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, 5 września 2003 r.

Zasady organizacji zawodów w roku szkolnym 2003/2004

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 2003/2004.

§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, C++ lub plik z danymi. Lista dopuszczalnych języków programowania w zawodach II i III stopnia może zostać rozszerzona.

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

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

(7) Do zawodów II stopnia zostanie zakwalifikowanych 280 uczestników, których rozwiązania zadań I stopnia zostaną ocenione najwyżej; do zawodów III stopnia — 40

uczestników, których rozwiązania zadań II stopnia zostaną ocenione najwyżej. Komitet Główny może zmienić podane liczby zakwalifikowanych uczestników co najwyżej o 20%.

(8) Podjęte przez Komitet Główny decyzje o zakwalifikowaniu uczestników do zawodów kolejnego stopnia, 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 — 20.10–17.11. 2003 r.

ogłoszenie wyników:

w witrynie Olimpiady — 15.12.2003 r.,

pocztą — 29.12.2003 r.

zawody II stopnia — 10–12.02.2004 r.

ogłoszenie wyników:

w witrynie Olimpiady – 23.02.2004 r.

pocztą — 27.02.2004 r.

zawody III stopnia — 30.03–03.04.2004 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 17 listopada 2003 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
02-018 Warszawa
tel. (0-22) 626-83-90

w nieprzekraczalnym terminie nadania do 17 listopada 2003 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 również 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 musi 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 v. 3.3. Programy w Pascalu będą kompilowane w systemie Linux za pomocą kompilatora FreePascala v. 1.0.10. Wybór polecenia kompilacji zależy od podanego rozszerzenia pliku w następujący sposób (np. dla zadania abc):

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

- Dyskietkę lub CD-ROM w standardzie dla komputerów PC, zawierające:
  - spis zawartości dyskietki oraz dane osobowe zawodnika w pliku nazwanym `SPIS.TXT`,
  - do każdego rozwiązanego zadania — program źródłowy.

  Dyskietka nie powinna zawierać żadnych podkatalogów.
- Wypełniony dokument zgłoszeniowy (dostępny w witrynie internetowej Olimpiady). Gorąco prosimy o podanie adresu elektronicznego. 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 wśród *Informacji dla zawodników* znajdują się *Odpowiedzi na pytania zawodników* dotyczące Olimpiady. Ponieważ *Odpowiedzi* mogą zawierać ważne informacje dotyczące toczących się zawodów prosimy wszystkich uczestników Olimpiady 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 1 grudnia 2003 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 5 grudnia 2003 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 5 grudnia 2003 r. nie będą rozpatrywane.

§4 UPRAWNIENIA I NAGRODY

(1) Laureaci i finaliści Olimpiady otrzymują z informatyki lub technologii informacyjnej ocenę celującą na koniec roku szkolnego oraz zwolnieni są z egzaminu maturalnego z informatyki na zasadach określonych w rozporządzeniu MEN z dnia 21 III 2001 r. z późniejszymi zmianami (2001/29/323, 128/1419, 2002/46/433, 155/1289, 214/1807, 2003/26/225) w sprawie warunków i sposobu oceniania, klasyfikowania i promowania uczniów i słuchaczy oraz przeprowadzenia egzaminów i sprawdzianów w szkołach publicznych.

(2) Laureaci zawodów III stopnia, a także finaliści, są zwolnieni w części lub w całości z egzaminów wstępnych do tych szkół wyższych, których senaty podjęły odpowiednie uchwały zgodnie z przepisami ustawy z dnia 12 września 1990 r. o szkolnictwie wyższym (Dz. U. 90.65.385).

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

(4) Komitet Główny ustala skład reprezentacji Polski na XVI Międzynarodową Olimpiadę Informatyczną w 2004 roku na podstawie wyników zawodów III stopnia i regulaminu tej Olimpiady Międzynarodowej. Szczegółowe zasady zostaną podane po otrzymaniu formalnego zaproszenia na XVI Międzynarodową Olimpiadę Informatyczną.

(5) Nauczyciel, który przygotował laureata lub finalistę Olimpiady Informatycznej, otrzymuje nagrodę przyznawaną przez Komitet Główny.

(6) 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 V Obozie Naukowo-Treningowym im. Antoniego Kreczmara, który odbędzie się w czasie wakacji 2004 r.

(7) 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 przeszedł do zawodów wyższego stopnia oraz dyrektor jego szkoły otrzymują informację o miejscu i terminie następnych zawodów.

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

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

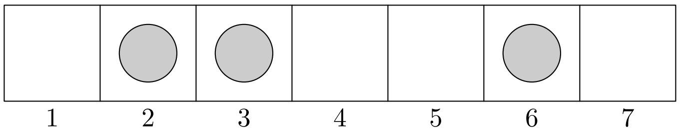
Zawody I stopnia

opracowania zadań

Bartosz Walczak — Treść zadania, Opracowanie
Jakub Pawlewicz — Programy

Gra

Rozważmy grę na prostokątnej planszy $m \times 1$ złożonej z m jednostkowych kwadratów ponumerowanych kolejno od 1 do m. Na planszy ustawionych jest n pionków, każdy na innym polu, przy czym żaden pionek nie znajduje się na polu o numerze m. Pojedynczy ruch w grze polega na przestawieniu dowolnie wybranego pionka na pierwsze wolne pole o większym numerze. Dwaj gracze wykonują na zmianę po jednym ruchu. Wygrywa ten, który postawi pionek na ostatnim polu, tzn. na polu o numerze m.



Dla przykładu z rysunku ($m = 7$), gracz może wykonać ruch z pola 2 na 4, z pola 3 na 4 lub z pola 6 na 7. Ten ostatni ruch kończy grę.

Mówimy, że ruch gracza jest wygrywający, jeżeli po jego wykonaniu gracz ten może wygrać grę niezależnie od tego, jakie ruchy będzie wykonywał jego przeciwnik.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia rozmiar planszy i początkowe rozstawienie pionków,*
- *wyznaczy liczbę różnych ruchów wygrywających, jakie w zadanej sytuacji początkowej ma do wyboru gracz rozpoczynający grę,*
- *wypisze wynik na standardowe wyjście.*

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby całkowite m i n ($2 \leqslant m \leqslant 10^9$, $1 \leqslant n \leqslant 10^6$, $n < m$) oddzielone pojedynczym odstępem. Drugi wiersz zawiera n rosnących numerów pól, na których znajdują się pionki. Liczby w wierszu są pooddzielane pojedynczymi odstępami.

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać liczbę różnych ruchów wygrywających, jakie może wykonać w zadanej sytuacji początkowej gracz rozpoczynający grę.

Przykład

Dla danych wejściowych:
```
5 2
1 3
```
prawidłową odpowiedzią jest:
```
1
```

Dla danych wejściowych:
```
5 2
2 3
```
prawidłową odpowiedzią jest:
```
0
```

Rozwiązanie

Wprowadzenie

Na początek wprowadzimy kilka pojęć, które ułatwią nam dalszą analizę problemu. I tak *pozycją początkową* będziemy nazywać pozycję, w której rozpoczyna się rozgrywka, natomiast *pozycją końcową* będziemy nazywać pozycję, w której nie można wykonać żadnego ruchu, czyli w której rozgrywka się kończy. Będziemy rozpatrywać tylko takie gry, w których każda rozgrywka jest skończona, czyli prowadzi do jakiejś pozycji końcowej.

Wszystkie możliwe rozgrywki możemy przedstawić przy pomocy tzw. *drzewa gry*. W takim drzewie będziemy utożsamiać wierzchołki z pozycjami osiągalnymi z pozycji początkowej, a krawędzie z ruchami możliwymi w danych pozycjach. Korzeń drzewa odpowiada pozycji początkowej, a liście — pozycjom końcowym.

Pojęcie *pozycji wygrywającej* i *przegrywającej* definiujemy przez indukcję w górę drzewa gry:

1. Każda pozycja końcowa jest przegrywająca.
2. Jeżeli w danej pozycji istnieje ruch prowadzący do pozycji przegrywającej, to pozycja jest wygrywająca.
3. Jeżeli w danej pozycji wszystkie ruchy prowadzą do pozycji wygrywających, to pozycja jest przegrywająca.

Warunek 1 wynika z warunku 3, ale został dodany dla zwiększenia czytelności.

Według powyższej definicji każda pozycja zostaje jednoznacznie zakwalifikowana jako wygrywająca, albo przegrywająca. Takie zakwalifikowanie pozycji będziemy określać mianem *strategii wygrywającej*. Wprowadzamy również pojęcie *ruchu wygrywającego* i *przegrywającego* jako ruchu prowadzącego, odpowiednio, do pozycji przegrywającej lub wygrywającej. To co otrzymujemy jest zgodne z intuicją: w pozycji wygrywającej istnieje ruch wygrywający, w pozycji przegrywającej wszystkie ruchy są przegrywające. Ponadto pojęcie ruchu wygrywającego z treści zadania jest równoważne tak zdefiniowanemu pojęciu ruchu wygrywającego.

W razie potrzeby możemy modyfikować drzewo gry, otrzymując drzewo innej gry, ze zmienionymi zasadami. Jak łatwo zauważyć, usunięcie z drzewa gry jakiejś krawędzi (i w efekcie całego poddrzewa pod tą krawędzią) odpowiadającej ruchowi przegrywającemu nie zmienia strategii wygrywającej. Jest to zgodne z intuicją, gdyż ruchów przegrywających nigdy nie opłaca się wykonywać.

Większość zadań o grach sprowadza się do znalezienia strategii wygrywającej. Tak jest również w tym przypadku. Oczywiście brutalne rozwiązanie polegające na przeszukiwaniu całego drzewa gry nie ma szans powodzenia już dla średniej wielkości danych, dlatego potrzebujemy łatwiej obliczalnego kryterium pozwalającego określić strategię wygrywającą. W jego znalezieniu pomoże nam analiza gier: Nim i „Staircase Nim".

Gra Nim

Planszą do gry Nim jest pewna liczba stosów. W pozycji początkowej każdy stos zawiera pewną liczbę pionków. Ruch w grze Nim polega na zdjęciu pewnej (dowolnej), niezerowej liczby pionków z dokładnie jednego stosu. Zdjęte pionki nie biorą udziału w dalszej rozgrywce. Przegrywa ten, kto nie może wykonać ruchu. Zatem w pozycji końcowej wszystkie stosy są puste, bo tylko wtedy nie można wykonać żadnego ruchu. Każda rozgrywka prowadzi do jakiejś pozycji końcowej, gdyż każdy ruch zmniejsza liczbę pionków biorących udział w grze.

W dalszej analizie będziemy korzystać z prostych własności operacji xor na liczbach całkowitych, czyli sumy modulo 2 na bitach (cyfrach w zapisie dwójkowym) tych liczb. Oto najważniejsze z nich:

- $x \operatorname{xor} 0 = x$,
- $x \operatorname{xor} x = 0$,
- $(x \operatorname{xor} y) \operatorname{xor} z = x \operatorname{xor} (y \operatorname{xor} z)$.

Ostatnia własność (łączność) pozwala nam pisać po prostu $x \operatorname{xor} y \operatorname{xor} z$.

Nim-sumą danej pozycji w grze Nim będziemy nazywać xor wielkości wszystkich stosów (wielkość stosu to liczba pionków, które się na nim znajdują). Następujące twierdzenie określa strategię wygrywającą dla gry Nim:

Twierdzenie 1 *Pozycja w grze Nim jest przegrywająca wtedy i tylko wtedy, gdy jej nim-suma jest równa* 0.

Dowód Udowodnimy następujące własności nim-sumy:

1. Nim-suma dowolnej pozycji końcowej jest równa 0.
2. Jeżeli nim-suma danej pozycji jest różna od 0, to istnieje ruch prowadzący do pozycji, której nim-suma jest równa 0.
3. Jeżeli nim-suma danej pozycji, różnej od końcowej, jest równa 0, to każdy ruch prowadzi do pozycji, której nim-suma jest różna od 0.

Punkt 1 jest oczywisty, pozostaje więc udowodnić punkty 2 i 3. Nim-sumę danej pozycji oznaczmy przez x.

Załóżmy najpierw, że $x \neq 0$. Niech i będzie numerem najbardziej znaczącego bitu x, który jest równy 1. Istnieje taki stos, że i-ty bit jego wielkości jest równy 1 — wielkość tego stosu oznaczmy przez s. Wówczas xor wielkości pozostałych stosów jest równy $x \operatorname{xor} s$. W liczbie tej i-ty bit jest równy 0, a wszystkie bardziej znaczące bity są równe tym w liczbie s. Stąd $x \operatorname{xor} s < s$. Zatem istnieje ruch, który usuwając pionki ze stosu o wielkości s, pozostawia w nim $s' = x \operatorname{xor} s$ pionków. Nim-suma po takim ruchu wynosi

$$x \operatorname{xor} s \operatorname{xor} s' = x \operatorname{xor} s \operatorname{xor} x \operatorname{xor} s = 0.$$

Załóżmy teraz, że $x = 0$. Po wykonaniu ruchu w obrębie dowolnego stosu o wielkości s, nim-suma wynosi $x \operatorname{xor} s \operatorname{xor} s' = s \operatorname{xor} s'$, gdzie s' jest nową wielkością stosu. Jeżeli $s \operatorname{xor} s' = 0$, to $s = s'$, co daje sprzeczność. Zatem każdy ruch prowadzi do pozycji, której nim-suma jest różna od 0.

Wystarczy teraz porównać punkty 1–3 z warunkami 1–3 ze strony 42, żeby się przekonać, że przegrywające są dokładnie te pozycje, których nim-suma wynosi 0. Formalnie, prosta indukcja w górę drzewa gry dowodzi tezy. ■

Gra „Staircase Nim”

Planszą do gry „Staircase Nim” jest również pewna liczba stosów, które są ponumerowane kolejnymi liczbami, począwszy od 0. W pozycji początkowej każdy stos zawiera pewną liczbę pionków. Ruch polega na przestawieniu pewnej (dowolnej) niezerowej liczby pionków z jakiegoś stosu na stos o numerze o jeden mniejszym. W końcowej pozycji wszystkie pionki znajdują się na stosie 0.

Nim-sumą pozycji w grze „Staircase Nim” będziemy nazywać xor wielkości wszystkich stosów o numerach nieparzystych. Widać tutaj wyraźną analogię do gry Nim. Każdy ruch w grze „Staircase Nim” powoduje bowiem zmianę wielkości dokładnie jednego stosu o numerze nieparzystym, a w pozycji końcowej wszystkie stosy o numerach nieparzystych są puste. Różnica jest taka, że ruch w grze „Staircase Nim” może spowodować powiększenie wielkości stosu o nieparzystym numerze.

Ale dlaczego w grze Nim nie wolno zwiększać stosów? Odpowiedź jest prosta: w przeciwnym przypadku istniałaby nieskończona rozgrywka. Jednak w grze „Staircase Nim” skończoność jest zapewniona inaczej — każdy ruch przybliża co najmniej jeden pionek do stosu 0. Zatem dla gry „Staircase Nim” jest prawdziwe twierdzenie analogiczne do twierdzenia 1:

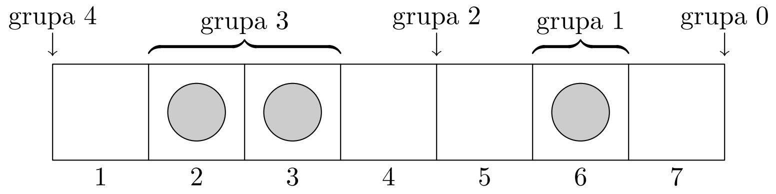
Twierdzenie 2 *Pozycja w grze „Staircase Nim” jest przegrywająca wtedy i tylko wtedy, gdy jej nim-suma jest równa* 0*.*

Dowód Analogiczny do dowodu twierdzenia 1. ■

Analiza gry z zadania

Zajmiemy się teraz analizą gry, której poświęcone jest zadanie.

Grupą pionków będziemy nazywać dowolny maksymalny zbiór pionków, które zajmują na planszy pola o kolejnych numerach. Maksymalność oznacza, że każda grupa pionków jest ograniczona z lewej i z prawej przez wolne pola lub końce planszy. Grupa pionków może być pusta — taka grupa znajduje się np. pomiędzy dwoma kolejnymi polami wolnymi. Grupy numerujemy od 0, począwszy od skrajnie prawej grupy.



Rysunek 1: Podział na grupy dla przykładu z treści zadania

Każdy ruch polega na przeniesieniu pewnej niezerowej liczby pionków z jakiejś grupy do grupy o numerze o jeden mniejszym. Z każdej grupy można przenieść dowolną liczbę pionków, która nie przekracza jej wielkości. W pozycji końcowej grupa 0 jest niepusta (zakładamy, że wtedy już żaden ruch nie jest możliwy). Natomiast jeżeli grupa 0 jest pusta, a grupa 1 jest niepusta, to pozycja jest wygrywająca. Wtedy każdy ruch z grupy 1 jest wygrywający, a każdy inny jest przegrywający.

Wobec tego interesować nas będą tylko takie sytuacje początkowe, w których grupy 0 i 1 są puste. Modyfikujemy drzewo gry poprzez usunięcie wszystkich ruchów prowadzących do pozycji, w których grupa 1 jest niepusta. Taka modyfikacja nie zmienia strategii wygrywającej. Natomiast otrzymujemy nową, jedyną pozycję końcową — pozycję, w której wszystkie pionki znajdują się w grupie 2.

Jeżeli w tak zmodyfikowanej grze grupy pionków potraktujemy jako stosy, a ich numery zmniejszymy o 2, otrzymamy dokładnie grę „Staircase Nim". Zatem twierdzenie 2 określa strategię wygrywającą dla zmodyfikowanej gry, a ta, jak wiemy, nie różni się od strategii wygrywającej dla gry z treści zadania. Teraz, znając strategię wygrywającą, możemy łatwo sprawdzić, które ruchy są wygrywające, a które są przegrywające.

Rozwiązanie wzorcowe

Rozwiązanie wzorcowe dokonuje podziału na grupy i oblicza nim-sumę już podczas wczytywania danych. Pamięta tylko niepuste grupy, gdyż wszystkich grup może być zbyt wiele. Najpierw rozważane są dwa proste przypadki szczególne: jeśli grupa 1 nie jest pusta, wynikiem jest liczba pionków w grupie 1, w przeciwnym przypadku, jeśli nim-suma jest równa 0, to wynikiem jest 0.

Zajmijmy się przypadkiem, w którym grupa 1 jest pusta i nim-suma jest różna od 0 — oznaczmy ją przez x. Rozwiązanie wzorcowe sprawdza dla każdej niepustej grupy, czy istnieje z niej jakiś ruch wygrywający, czyli ruch, który powoduje wyzerowanie nim-sumy. Niech s będzie wielkością aktualnie rozważanej grupy. Jeśli grupa ma numer nieparzysty, to ruch powodujący wyzerowanie nim-sumy powinien zostawić w tej grupie $x \operatorname{xor} s$ pionków. Taki ruch jest możliwy, gdy $x \operatorname{xor} s < s$. Natomiast jeśli grupa ma numer parzysty różny od 2, to szukany ruch powinien przenieść tyle pionków do grupy o numerze o jeden mniejszym (oznaczmy jej wielkość przez t), żeby po jego wykonaniu w grupie tej znajdowało się dokładnie $x \operatorname{xor} t$ pionków. Jest to możliwe, gdy $0 < (x \operatorname{xor} t) - t \leqslant s$. W obu przypadkach z jednej grupy istnieje co najwyżej jeden ruch wygrywający. Wynikiem jest liczba tak znalezionych ruchów wygrywających.

Czas działania rozwiązania wzorcowego wynosi $\Theta(n)$. Zapotrzebowanie na pamięć jest liniowe ze względu na liczbę niepustych grup, czyli wynosi $O(n)$. Rozwiązanie wzorcowe zostało zaimplementowane w `gra.c` oraz `gra.pas`.

Testy

Poniżej znajdują się opisy testów.

- `gra1a.in` — $m = 10$, $n = 8$, przegrana, wszystkie pionki w grupie 2.
- `gra1b.in` — $m = 8$, $n = 3$.
- `gra2a.in` — $m = 8$, $n = 5$, grupa 1 zawiera 3 pionki.
- `gra2b.in` — $m = 10$, $n = 5$.
- `gra3a.in` — $m = 13$, $n = 6$, nieoczywista przegrana.
- `gra3b.in` — $m = 15$, $n = 9$.
- `gra4.in` — $m = 17$, $n = 10$.
- `gra5.in` — $m = 24$, $n = 14$.
- `gra6.in` — $m = 85$, $n = 44$, losowy.
- `gra7.in` — $m = 585$, $n = 393$, losowy.
- `gra8.in` — $m = 7166$, $n = 4027$, losowy.
- `gra9.in` — $m = 376213$, $n = 193363$, losowy.
- `gra10.in` — $m = 1906731$, $n = 801226$, losowy.
- `gra11.in` — $m = 1543149$, $n = 999183$, losowy.
- `gra12.in` — $m = 932995895$, $n = 998566$, losowy.
- `gra13.in` — $m = 940238451$, $n = 997643$, losowy.
- `gra14.in` — $m = 935204727$, $n = 999989$, losowy.
- `gra15.in` — $m = 983091535$, $n = 999240$, losowy.

Pary testów 1a i 1b, 2a i 2b oraz 3a i 3b były zgrupowane.

Marcin Stefaniak — Treść zadania, Opracowanie
Tomasz Czajka — Programy

PIN-kod

Każda karta bankomatowa ma swój 4-cyfrowy numer identyfikacyjny, tzw. PIN (ang. personal identification number*). To, czy transakcja z użyciem karty zostanie wykonana, zależy od tego, czy numer karty i jej PIN są zgodne. Zgodność PIN-u i numeru karty sprawdza moduł HSM (ang.* hardware security module*). Moduł ten otrzymuje trzy parametry: numer karty x, PIN p oraz tablicę konwersji a (16-elementową tablicę liczb od 0 do 9, indeksowaną od 0 do 15). Moduł HSM działa w następujący sposób:*

1. *szyfruje numer karty x, otrzymując liczbę y zapisaną szesnastkowo,*
2. *z otrzymanej liczby y pozostawia tylko 4 pierwsze cyfry szesnastkowe,*
3. *pozostawione cyfry szesnastkowe zamienia na dziesiętne za pomocą tablicy a (tzn. cyfra h zamieniana jest na $a[h]$, gdzie $h = A$ jest utożsamiane z 10, $h = B$ z 11, $h = C$ z 12, $h = D$ z 13, $h = E$ z 14 i $h = F$ z 15),*
4. *tak otrzymany 4-cyfrowy numer dziesiętny musi być identyczny z podanym PIN-em.*

Standardową tablicą konwersji jest $a = (0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5)$.

Załóżmy na przykład, że numerem karty jest $x = \mathrm{4556\ 2385\ 7753\ 2239}$, a po zaszyfrowaniu uzyskujemy numer szesnastkowy $y = \mathrm{3F7C\ 2201\ 00CA\ 8AB3}$. Żeby uzyskać 4-cyfrowy PIN: bierzemy pierwsze 4 cyfry szesnastkowe (3F7C) i kodujemy je za pomocą standardowej tablicy konwersji. Wynikiem jest PIN $p = 3572$.

Niestety, nieuczciwy pracownik banku lub komputerowy włamywacz może uzyskać dostęp do modułu HSM i próbować odgadnąć PIN manipulując tablicą konwersji.

Zadanie

Napisz program, który będzie starał się odgadnąć PIN za pomocą zapytań do modułu HSM, przy czym może on zadać co najwyżej 30 zapytań.

Opis interfejsu

Twój program powinien komunikować się ze „światem zewnętrznym" jedynie poprzez funkcje udostępniane przez moduł `hsm` *(*`hsm.pas` *w Pascalu i* `hsm.h` *w C/C++). Oznacza to, że nie może on otwierać żadnych plików ani korzystać ze standardowego wejścia/wyjścia.*

Przy każdym uruchomieniu Twój program powinien odgadnąć jeden PIN, zgodny z numerem karty znanej modułowi `hsm` *przy* **standardowej** *tablicy konwersji.*

Moduł `hsm` *udostępnia funkcje:* `sprawdz` *oraz* `wynik`*. Ich deklaracje w Pascalu wyglądają następująco:*

```
function sprawdz(pin: array of Longint, a: array of Longint): Boolean;
procedure wynik (pin: array of Longint);
```

a w C/C++ następująco:

```
int sprawdz(int pin[], int a[]);
void wynik(char pin[]);
```

Parametrami funkcji `sprawdz` *są: badany PIN (w postaci 4-elementowej tablicy cyfr) oraz tablica konwersji (16-elementowa). Jej wynikiem jest wartość logiczna określająca, czy podany PIN jest zgodny z numerem karty, przy podanej tablicy konwersji. Twój program powinien co najwyżej 30 razy wywoływać funkcję* `sprawdz` *i dokładnie raz funkcję* `wynik`*. Wywołanie procedury* `wynik` *kończy działanie programu. Jej parametrem powinien być PIN zgodny z numerem karty przy standardowej tablicy konwersji.*

Żeby przetestować swoje rozwiązanie powinieneś napisać własny moduł `hsm`*. Nie jest on jednak elementem rozwiązania i nie należy go przesyłać wraz z programem.*

Przykład

Przykładowa interakcja programu z modułem `hsm` *może wyglądać następująco:*

```
sprawdz('1111', (0,0,1,1,0,1,0,1,0,0,0,0,1,1,0,1)) = true
sprawdz('1100', (0,0,0,1,0,1,0,0,0,0,0,0,0,1,0,1)) = true
sprawdz('1100', (0,0,0,1,0,0,0,0,0,0,0,0,0,1,0,0)) = false
sprawdz('1000', (0,0,0,1,0,0,0,0,0,0,0,0,0,1,0,0)) = true
sprawdz('0010', (0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,0)) = false
sprawdz('0001', (0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,0)) = true
wynik('3572')
```

Rozwiązanie

Geneza zadania

Historia przedstawiona w treści zadania jest, niestety, prawdziwa. Za godne ubolewania należy uznać, że taki prosty system, jakim jest moduł sprawdzania zgodności PIN-u i numeru karty, zaprojektowano w sposób wprowadzający lukę w ochronie danych. Istotnie, gdyby nie lekkomyślne wzbogacenie interfejsu tego modułu o tablicę konwersji, tylko brutalna siła zdołałaby wydrzeć zeń tajemnicę PIN-u.

Sprawa ta ujrzała światło dzienne całkiem niedawno, kiedy to naukowcy z Uniwersytetu Cambridge (wśród nich były laureat Olimpiady Informatycznej, Piotr Zieliński) zbadali podstawy działania systemów bankomatowych. Odbiło się to szerokim echem w popularnych mediach („Polak złamał PIN-y", „Niebezpieczne bankomaty", itp.), nierzadko wyolbrzymiających skalę zagrożenia.

To jest pierwszy przykład na to, że bezpieczeństwo i ochrona danych jest niezwykle trudną do osiągnięcia — bo bardziej subtelną i delikatną — cechą systemu informatycznego. Drugi przykład pochodzi z własnego podwórka. W treści zadania umieszczony został przykład obliczania PIN-u, identyczny z tym zamieszczonym w oryginalnym artykule Piotra Zielińskiego. W konsekwencji uczestnik olimpiady mógł, wpisując w wyszukiwarce internetowej podany w zadaniu ciąg 3F7C 2201 00CA 8AB3, z łatwością odnaleźć ów artykuł i poznać cenne (być może) uwagi, wskazówki i inspiracje.

Rozwiązanie

Ograniczenie liczby pytań do 30 jasno wskazuje, że metoda brutalnego próbowania wszystkich 10 000 kombinacji PIN-ów prowadzi donikąd. Ale jasno widać po przykładowej interakcji programu z modułem HSM, że kluczem do szybkiego wyznaczenia PIN-u jest podawanie specjalnych tablic konwersji. Istotnie, jeśli użyjemy np. tablicy konwersji o tylko dwóch wartościach 0 i 1, to każda z odpowiedzi TAK i NIE da nam całkiem dużo informacji.

Odgadywanie kodu PIN przypomina bardzo grę Mastermind albo grę „w dwadzieścia pytań”. Za każdym razem zadajemy pytanie typu TAK/NIE, którego odpowiedź zawęża nam zbiór możliwych wyników. Najlepiej, by zadane pytanie dzieliło zbiór potencjalnych wyników na części jak najbardziej zbliżonej wielkości. W pesymistycznym przypadku bowiem (gdy mamy pecha lub gdy moduł grający stosuje tzw. diabelską strategię) odpowiedź na pytanie pozostawi nas z większym zbiorem możliwości.

Rozwiązanie wzorcowe

W rozwiązaniu wzorcowym szukamy pytania, które dzieli możliwe rozwiązania jak „najbardziej po połówce”. Ograniczamy przy tym się do tablic konwersji zawierających same 0 i 1. Dodatkowo w tablicy `a` zawsze bierzemy `a[10+i] = a[i]`. W ten sposób utożsamiamy `A-F` z `0-5` i ograniczamy liczbę możliwych tablic do 1024, dzięki czemu możemy przeglądnąć wszystkie pytania i wybrać z nich najlepsze.

Najprościej byłoby sprawdzić każde pytanie osobno. Mamy 1024 możliwe tablice konwersji i 16 „fałszywych” PIN-ów złożonych z samych 0 i 1, co razem daje 16 * 1024 pytań, dla każdego z nich przeglądamy bieżący zbiór możliwych PIN-ów (na początku 10 000).

Ale można szybciej: dla każdej tablicy konwersji i każdego prawdziwego PIN-u istnieje dokładnie jeden PIN złożony z 0 i 1, dla którego dostaniemy odpowiedz TAK. Zliczając wystąpienia każdego z tych 16 „fałszywych” PIN-ów, możemy wybrać najlepszy fałszywy PIN do zapytania dla danej tablicy konwersji — taki, dla którego liczba wystąpień jest jak najbliższa połowy. Całe pytanie wybieramy zatem, sprawdzając każdą tablicę konwersji z każdym pozostałym możliwym prawdziwym PIN-em, co upraszcza nam obliczenia o czynnik 16.

Rozwiązanie wzorcowe nie potrzebuje więcej niż dwadzieścia parę pytań dla odgadnięcia któregokolwiek PIN-u. Najwięcej czasu potrzebuje na zadanie pierwszego pytania, każde następne oblicza coraz szybciej.

Dalsze optymalizacje

Przede wszystkim zauważmy, że najwięcej obliczeń rozwiązanie wzorcowe wykonuje na samym początku gry, kiedy to zbiór potencjalnych PIN-ów jest największy. Ale te początkowe pytania możemy obliczyć uprzednio i stablicować w kodzie programu. Jeśli ustalimy drzewo gry aż do głębokości d, to na jego zapamiętanie potrzebujemy $O(2^d)$ pamięci, ale za to zmniejszamy czas działania o czynnik podobnego rzędu — coś za coś.

Ponadto zachłanna metoda wybierania najlepszego w danej sytuacji pytania nie gwarantuje optymalnej gry. Niemniej jest to wystarczająca technika, by zgadnąć PIN zadając mniej niż 30 pytań.

Testy

Rozwiązania były testowane przy użyciu 25 osobnych testów. Każdy test składał się z jednego PIN-u do zgadnięcia. Moduł HSM odpowiadał na pytania zgodnie z tym PIN-em — czyli nie używał „diabelskiej” strategii.

Marcin Kubica
Treść zadania, Opracowanie

Marcin Kubica, Marcin Mucha
Programy

Sznurki

Przedsiębiorstwo String-Toys S.A. zwróciło się do Ciebie o pomoc w rozwiązaniu następującego problemu. String-Toys S.A. chce produkować ze sznurka modele spójnych grafów acyklicznych.

Każdy graf składa się z wierzchołków i pewnej liczby krawędzi łączących różne wierzchołki. Ten sam wierzchołek może być połączony z wieloma innymi wierzchołkami. Graf jest spójny i acykliczny, gdy od każdego wierzchołka można przejść do każdego innego wierzchołka, wędrując po krawędziach i co więcej, bez zawracania można to zrobić dokładnie na jeden sposób.

Wierzchołki grafów mają być modelowane przez węzły zawiązane na kawałkach sznurka, a krawędzie przez odcinki sznurka pomiędzy węzłami. Każdy węzeł może być wynikiem zasupłania kawałka sznurka lub związania w tym samym miejscu wielu kawałków. Ze względów technicznych koszty produkcji zależą od liczby kawałków sznurka użytych do wykonania modelu oraz od długości najdłuższego z kawałków. (Każda krawędź ma długość 1. Długość sznurka użytego do zrobienia węzłów jest pomijalna.)

Zadanie

Zadanie polega na napisaniu programu, który:

- *wczyta ze standardowego wejścia opis spójnego grafu acyklicznego, który ma być modelowany,*
- *obliczy:*
  - *minimalną liczbę kawałków sznurka potrzebnych do wykonania modelu,*
  - *zakładając, że liczba kawałków sznurka jest minimalna, minimalną długość najdłuższego kawałka sznurka potrzebnego do wykonania modelu,*
- *wypisze dwie obliczone liczby na standardowe wyjście.*

Wejście

Pierwszy wiersz zawiera jedną dodatnią liczbę całkowitą n — liczbę wierzchołków w grafie, $2 \leqslant n \leqslant 10\,000$. Wierzchołki są ponumerowane od 1 do n. Kolejne $n-1$ wierszy zawiera opisy krawędzi, po jednej w wierszu. Każdy z tych wierszy zawiera parę liczb całkowitych a i b oddzielonych pojedynczym odstępem, $1 \leqslant a,b \leqslant n$, $a \neq b$. Para taka reprezentuje krawędź łączącą wierzchołki a i b.

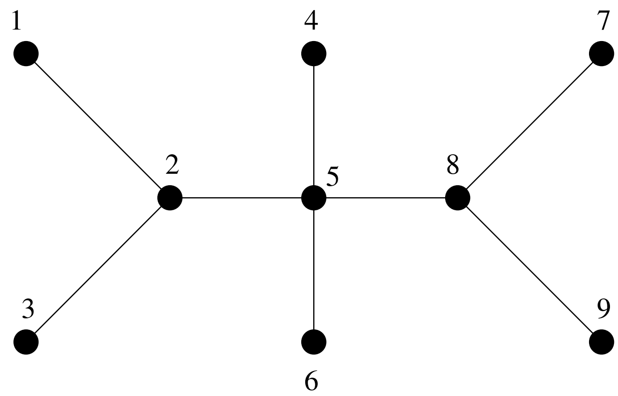
Wyjście

Twój program powinien wypisać w pierwszym wierszu wyjścia dwie liczby całkowite k i l oddzielone pojedynczym odstępem: k powinno być minimalną liczbą kawałków sznurka potrzebnych do wykonania modelu grafu, a l powinno być minimalną długością najdłuższego kawałka sznurka (zakładając, że zużyliśmy k kawałków sznurka).

Przykład

Dla danych wejściowych:

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



poprawnym wynikiem jest:

```
4 2
```

Rozwiązanie

W zadaniu należy obliczyć dwie liczby: minimalną liczbę sznurków oraz najmniejsze możliwe ograniczenie na długości sznurków, przy założeniu, że ich liczba jest minimalna. Zajmiemy się najpierw pierwszą z tych liczb, a potem drugą.

Minimalna liczba sznurków

Minimalną liczbę sznurków można łatwo obliczyć uogólniając twierdzenie Eulera ([14] p. 7.4, [23] p. 2.7, [32] par. 6). W zadaniu zajmujemy się spójnymi grafami acyklicznymi, czyli drzewami. Twierdzenie Eulera możemy zapisać następująco, używając terminologii z zadania:

Twierdzenie 1 *Jeżeli mamy dany spójny graf, w którym z każdego wierzchołka, z wyjątkiem co najwyżej dwóch, wychodzi parzysta liczba krawędzi, to model takiego grafu można wykonać z jednego kawałka sznurka. Jeżeli mamy dwa wierzchołki, z których wychodzi nieparzysta liczba krawędzi, to w wierzchołkach tych są końce sznurka. Jeżeli z każdego wierzchołka wychodzi parzysta liczba krawędzi, to sznurek tworzy pętlę (cykl).*

Zauważmy prosty fakt:

Fakt 1 Liczba wierzchołków, z których wychodzi nieparzysta liczba krawędzi jest parzysta.

Dowód Każda krawędź ma dwa końce, a łączna liczba krawędzi jest całkowita. ■

Fakt ten jest znany jako lemat o podawaniu rąk ([17] zad. 5.4-1, [32] par. 2). W szczególności nie jest możliwe, abyśmy mieli tylko jeden wierzchołek, z którego wychodzi nieparzysta liczba krawędzi.

Twierdzenie Eulera można uogólnić:

Fakt 2 Jeżeli w spójnym grafie mamy $2k$ wierzchołków, z których wychodzą nieparzyste liczby krawędzi, to model takiego grafu możemy wykonać z k sznurków.

Dowód Dodajmy do grafu na chwilę k dodatkowych krawędzi, w taki sposób, że z wszystkich wierzchołków wychodzą parzyste liczby krawędzi. Model takiego grafu możemy wykonać z jednego kawałka sznurka i to w formie pętli. Usuńmy teraz k dodanych krawędzi, rozcinając pętlę na k kawałków sznurka. Otrzymujemy model oryginalnego grafu. ■

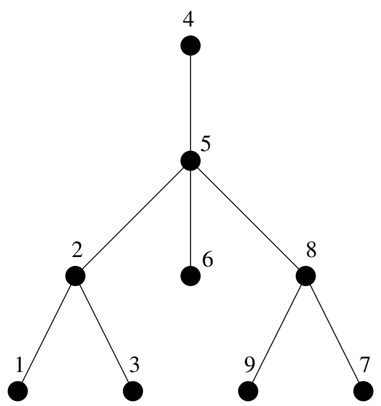
Tak więc, aby obliczyć minimalną liczbę sznurków, wystarczy policzyć, ile jest takich wierzchołków, z których wychodzi nieparzysta liczba krawędzi i podzielić tę liczbę przez 2.

W zadaniu zajmujemy się drzewami, czyli spójnymi grafami acyklicznymi. W przypadku drzew można prosto opisać, jak może wyglądać model drzewa. W każdym wierzchołku, z którego wychodzi nieparzysta liczba krawędzi umieszczamy jeden koniec sznurka i wyprowadzamy go wzdłuż dowolnej krawędzi. Dla każdego wierzchołka pozostaje parzysta liczba krawędzi, które modelujemy sznurkiem przechodzącym przez wierzchołek. Dla każdego wierzchołka, krawędzie takie łączymy w dowolny sposób w pary i przeprowadzamy sznurki zgodnie z podziałem na pary. Każda taka konstrukcja daje model drzewa wykonany z minimalnej liczby kawałków sznurka.

Ograniczenie na długość kawałków sznurka

Wiemy już, jak mogą wyglądać modele drzew wykonane z minimalnej liczby kawałków sznurka. Zastanówmy się, jak poprowadzić sznurki tak, aby długość najdłuższego sznurka była jak najmniejsza.

Dla uproszczenia rozważań ukorzenimy nasze drzewo. Na korzeń wybieramy dowolny liść (wierzchołek, z którego wychodzi tylko jedna krawędź). Ukorzenienie możemy sobie wyobrazić jako chwycenie modelu grafu za korzeń i podniesienie do góry tak, że cały model swobodnie zwisa. Wierzchołek znajdujący się bezpośrednio powyżej danego wierzchołka nazywamy jego przodkiem, a wierzchołki znajdujące się bezpośrednio poniżej nazywamy potomkami. Poniższy rysunek przedstawia ukorzenione drzewo z przykładu z treści zadania, gdzie jako korzeń wybrano wierzchołek nr 4.



Ukorzenione drzewo.

Poniżej przedstawiamy procedurę, która wczytuje dane wejściowe, oblicza minimalną liczbę kawałków sznurka potrzebnych do wykonania modelu oraz konstruuje ukorzenione drzewo[1]. Wykorzystuje ona algorytm przeszukiwania grafów w głąb (DFS).

```
(* Typ reprezentujący dowolne drzewo ukorzenione. *)
type drzewo = Wezel of drzewo list;;
(* Wczytanie danych. Wynikiem jest drzewo i liczba sznurków. *)
let wczytaj_dane () =
  (* Liczba węzłów. *)
  let n = scanf "%d " id
  in
    let sasiedzi = make (n+1) []
    and odwiedzone = make (n+1) false
    and konce = ref 0
    in
      (* Wczytanie krawędzi. *)
      (* Wynikiem jest korzeń - jeden z liści. *)
      let rec wczytaj () =
        let i = ref 1
        in
          begin
            (* Wczytanie krawędzi. *)
            for j = 1 to n-1 do
              let a = scanf "%d " id
              and b = scanf "%d " id
              in begin
                sasiedzi.(a) <- b::sasiedzi.(a);
                sasiedzi.(b) <- a::sasiedzi.(b)
              end
            done;
            (* Końce sznurków. *)
            for j = 1 to n do
              if length sasiedzi.(j) mod 2 = 1 then
                konce := !konce + 1
            done;
            (* Wybór korzenia - jeden z liści. *)
            while length sasiedzi.(!i) > 1 do i:= !i + 1 done;
            !i
          end
      (* Konstrukcja drzewa - DFS. *)
      and dfs v =
        begin
          odwiedzone.(v) <- true;
```

[1]Procedura ta pochodzi z pliku `szn.ml`, który można znaleźć na załączonym dysku CD-ROM. Rozwiązanie to jest napisane w języku Ocaml. (Opis języka Ocaml, dystrybucję kompilatora oraz inne informacje na ten temat można znaleźć na stronie `http://caml.inria.fr/ocaml/`.) Na załączonym dysku można również znaleźć rozwiązanie napisane w języku C++: `szn.cpp`.

```
          Wezel
            (fold_left
              (fun a w -> if odwiedzone.(w) then a else (dfs w)::a)
              [] sasiedzi.(v))
        end
      in
        let korzen = wczytaj ()
        in (dfs korzen, !konce / 2);;
```

Modele drzewa mogą mieć następującą postać:

1. Jeżeli dany wierzchołek ma nieparzystą liczbę potomków i nie jest to korzeń, to sznurek wychodzący z jednego z jego potomków prowadzi do jego przodka. Sznurki wychodzące z pozostałych potomków są połączone ze sobą parami.
2. Jeżeli dany wierzchołek ma parzystą liczbę potomków, to jeden ze sznurków ma w nim koniec. Możliwe są dwa przypadki:
   (a) Sznurki wychodzące z potomków są połączone ze sobą parami. Do przodka prowadzi sznurek o końcu w danym wierzchołku.
   (b) Do przodka prowadzi sznurek wychodzący od jednego z potomków. Sznurek od innego z potomków kończy się w danym wierzchołku. Sznurki wychodzące z pozostałych potomków są połączone ze sobą parami.
3. Korzeń wybraliśmy tak, że ma tylko jednego potomka. Sznurek wychodzący z niego kończy się w korzeniu.

Problem wyznaczenia minimalnej długości najdłuższego kawałka sznurka można rozwiązać stosując programowanie dynamiczne. Dla każdego wierzchołka v możemy rozważyć poddrzewo o korzeniu w tym wierzchołku. Z każdego takiego poddrzewa (z wyjątkiem całego drzewa) jeden sznurek wychodzi do góry, a dodatkowo pewne sznurki mogą być w nim zawarte w całości. Niech d będzie ograniczeniem na długość sznurków całkowicie zawartych w rozważanym poddrzewie. Przez $S(v,d)$ oznaczymy minimalną długość sznurka wychodzącego z poddrzewa o korzeniu w v do góry (a dokładniej tej jego części, która jest zawarta w rozważanym poddrzewie). Jeżeli nie jest możliwe, żeby sznurki zawarte w całości w poddrzewie nie były dłuższe niż d, to przyjmujemy $S(v,d) = \infty$. Zauważmy, ze funkcja S jest niemalejąca, tzn. im ostrzejsze ograniczenie nakładamy na sznurki zawarte w całości w poddrzewie, tym dłuższy może być sznurek wychodzący z poddrzewa.

Niech $\varphi(v)$ będzie długością najdłuższej ścieżki w poddrzewie o korzeniu v (tzw. średnica poddrzewa). Interesować nas będą wartości funkcji $S(v,d)$ dla $d = 0,1,\ldots,\varphi(v)$. Niech r będzie korzeniem drzewa. Wówczas szukane ograniczenie na długość kawałków sznurka to:

$$\min_{d=0,1,\ldots,\varphi(r)} (\max(d, S(r,d)))$$

Niech v będzie wierzchołkiem w grafie, dla którego wiemy już jak przebiega sznurek wychodzący z v do góry. Oznaczmy przez $p(v)$ liczbę potomków v. Pozostaje nam parzysta liczba sznurków wychodzących z potomków v: $w_1, w_2, \ldots, w_{p(v)}$, które należy połączyć w pary. Zastanówmy się, jak sprawdzić, czy da się tak połączyć te sznurki, aby długość żadnego sznurka zawartego w poddrzewie nie przekraczała zadanego ograniczenia d? Pokażemy, że można je łączyć na zasadzie największy z najmniejszym.

Fakt 3 Niech $p(v)$ będzie parzyste i niech $l_1 \leqslant l_2 \leqslant \ldots \leqslant l_{p(v)}$ będą minimalnymi długościami sznurków wychodzących z $w_1, w_2, \ldots, w_{p(v)}$, przy założeniu, że żaden sznurek całkowicie zawarty w poddrzewach o korzeniach $w_1, w_2, \ldots, w_{p(v)}$ nie jest dłuższy niż d. Wówczas sznurki wychodzące z $w_1, w_2, \ldots, w_{p(v)}$ można połączyć w pary tak, aby żaden z nich nie był dłuższy niż d wtedy i tylko wtedy, gdy $l_1 + l_{p(v)} \leqslant d$, $l_2 + l_{p(v)-1} \leqslant d, \ldots$ oraz $l_{\frac{p(v)}{2}} + l_{\frac{p(v)}{2}+1} \leqslant d$.

Dowód Załóżmy, że można sznurki połączyć w pary tak, aby ich długości nie przekraczały d. Jeżeli sznurek wychodzący z w_1 nie jest połączony ze sznurkiem wychodzącym z $w_{p(v)}$, to niech i i j będą takie, że sznurek wychodzący z w_1 jest połączony ze sznurkiem wychodzącym z w_i, a sznurek wychodzący z $w_{p(v)}$ będzie połączony ze sznurkiem wychodzącym z w_j, $i \neq j$.

Zauważmy, że zachowując ograniczenie d na długość sznurków możemy zamienić połączenia i połączyć sznurek wychodzący z w_1 ze sznurkiem wychodzącym z $w_{p(v)}$, oraz sznurek wychodzący z w_i ze sznurkiem wychodzącym z w_j. Skoro $l_1 + l_i \leqslant d$, $l_{p(v)} + l_j \leqslant d$ oraz $l_1 \leqslant l_i, l_j \leqslant l_{p(v)}$, to $l_i + l_j \leqslant d$ oraz $l_1 + l_{p(v)} \leqslant d$. Analogicznie zamieniamy połączenia pozostałych sznurków. ■

Fakt ten pozwala nam zaimplementować zachłanny algorytm sprawdzający, czy sznurki o danych długościach można połączyć w pary tak, aby ich długości nie przekroczyły danego ograniczenia. Algorytm ten jest ujęty w postaci procedury `pary`. Procedura ta ma dwa parametry `l` i `d` — `l` to lista długości sznurków uporządkowana nierosnąco, `d` to ograniczenie na długości sznurków. Procedura ta korzysta z dwóch procedur pomocniczych: `ile_par` i `cut`. Procedura `ile_par` ma trzy parametry: dwie listy liczb oraz ograniczenie `d`. Bada ona, ile kolejnych elementów z danych list można połączyć w pary o sumach nie przekraczających `d`. Procedura `cut` ma charakter pomocniczy i wybiera z podanej listy `l` elementy od `i`-tego do `j`-tego. Procedury `ile_par` i `cut` zostały wydzielone, gdyż będziemy z nich jeszcze korzystać dalej.

```
(* Wybiera z listy fragment od i-tego do j-tego elementu. *)
let cut i j l =
  let rec cut_iter i j l a =
    if i > 1 then
      cut_iter (i-1) (j-1) (tl l) a
    else if j > 0 then
      cut_iter i (j-1) (tl l) (hd l :: a)
    else
      rev a
  in
    cut_iter i j l [];;
(* Sprawdza ile pierwszych elementów z obu list *)
(* można połączyć w pary o sumach <= d.        *)
let ile_par l1 l2 d =
  let rec ile_par_iter l1 l2 a =
    match (l1, l2) with
      ([], []) -> a |
      (h1::t1, h2::t2) ->
        if h1 + h2 + 2 <= d then
```

```
          ile_par_iter t1 t2 (a+1)
        else a |
      _ -> a
  in
    ile_par_iter l1 l2 0;;
(* Sprawdź czy elementy listy posortowanej *)
(* nierosnąco można połączyć w pary <= d   *)
let pary l d =
  let dl = length l
  in
    if dl mod 2 = 0 then
      ile_par (cut 1 (dl / 2) l)
              (rev (cut (dl / 2 + 1) dl l)) d = dl / 2
    else false;;
```

Zastanówmy się, jak można obliczyć wartości funkcji $S(v,d)$. Rozważmy najpierw przypadek, gdy v ma nieparzystą liczbę potomków. Niech $l_1 \leqslant l_2 \leqslant \ldots \leqslant l_{p(v)}$ będą minimalnymi długościami sznurków wychodzących z potomków v. Zauważmy, że jeżeli nie jesteśmy w stanie połączyć w pary $l_1, l_2, \ldots, l_{p(v)}$ tak, aby ich sumy nie przekraczały d (przypadek, gdy do góry przechodzi najdłuższy ze sznurków), to $S(v,d) = \infty$. Jeżeli można to zrobić, to możemy dalej sprawdzać, jaki jest najkrótszy wychodzący z poddrzew sznurek, który może przechodzić do góry, tzn. taki, że pozostałe sznurki możemy połączyć w pary o długościach nie większych niż d. Sprawdzamy kolejno coraz krótsze sznurki. Powiedzmy, że sprawdziliśmy, iż do góry mogą wychodzić sznurki o długościach $l_{i+1}, \ldots, l_{p(v)}$ i chcemy sprawdzić, czy może wychodzić do góry sznurek o długości l_i. Korzystając z faktu 3, wystarczy sprawdzić jeden z dwóch warunków:

- $l_{i+1} + l_{p(v)-i} + 2 \leqslant d$, dla $i \geqslant \frac{p(v)+1}{2}$,
- $l_{i+1} + l_{p(v)+1-i} + 2 \leqslant d$, dla $i < \frac{p(v)+1}{2}$.

Możemy więc napisać następującą procedurę obliczającą $S(v,d)$ w przypadku, gdy v ma nieparzystą liczbę potomków.

```
(* S(v,d) dla nieparzystej liczby potomków v,   *)
(* na podstawie d oraz [S(wp,d); ...; S(w1,d)]. *)
let nieparzyste ls d =
  let dl = length ls
  in
    if pary (tl ls) d then
      (* Najdłuższy sznurek może wychodzić do góry. *)
      (* Szukamy krótszego.                         *)
      let i = ile_par (cut 1 (dl/2) ls)
                      (rev (cut (dl/2 + 2) dl ls)) d
      in
        if i = dl/2 then
          (* Środkowy (co do długości) sznurek może *)
          (* wychodzić do góry. Szukamy krótszego.  *)
```

```
     let j = ile_par (cut (dl/2 + 1) (dl-1) ls)
             (rev (cut 1 (dl/2) ls)) d
     in
      nth ls (j + dl/2) + 1
    else nth ls i + 1
  else
   (* Nie da się. *)
   inf
```

Załóżmy teraz, że wierzchołek v ma parzystą liczbę potomków. Jeżeli sznurki wychodzące z tych potomków można połączyć w pary tak, aby ich długości nie przekraczały d, to sznurek wychodzący z v do góry może mieć początek w v. Jeżeli tak nie jest, to jeden ze sznurków wychodzących z potomków v musi się kończyć w v, a jeden musi iść dalej do góry. Pozostałe sznurki są połączone w pary ze sobą.

Jeżeli $l_{p(v)} + 1 \leqslant d$, to najkorzystniej jest dokonać wyboru tak, aby w v kończył się najdłuższy sznurek, wychodzący z $w_{p(v)}$. Wówczas problem redukuje się do problemu, gdy v ma nieparzystą liczbę potomków.

Jeżeli $l_{p(v)} + 1 > d$, to sznurek wychodzący z $w_{p(v)}$ musi przechodzić przez v do góry. Podobnie, problem redukuje się wówczas do przypadku, gdy v ma nieparzystą liczbę potomków — musimy sprawdzić, iż pozostałe sznurki można połączyć w pary (z wyjątkiem jednego, który ma koniec w v) tak, że ich długości nie przekraczają d. W przeciwnym przypadku $S(v,d) = \infty$.

Powyższe obserwacje przekładają się na następującą procedurę:

```
(* S(v,d) dla parzystej liczby potomków v,      *)
(* na podstawie d oraz [S(wp,d); ...; S(w1,d)]. *)
let parzyste ls d =
 if pary ls d then
  (* Z v wychodzi nowy sznurek. *)
  0
 else
  if hd ls + 1 <= d then
   (* Najdłuższy z wychodzących sznurków kończymy, *)
   (* a pozostałych mamy nieparzystą liczbę.       *)
   nieparzyste (tl ls) d
  else
   if nieparzyste (tl ls) d <= d then
    (* Najdłuższy z wychodzących sznurków przechodzi  *)
    (* do góry, o ile pozostałe nie są dłuższe niż d. *)
    hd ls + 1
   else
    (* Nie da się. *)
    inf;;
```

Poniższa procedura łączy oba przypadki: dla $p(v)$ parzystego i nieparzystego.

```
(* Obliczenie S(v,d) na podstawie d oraz [S(w1,d); ...; S(wp,d)]. *)
let s d l =
```

```
(* Posortowana nierosnąco lista  [S(w1,d); ...; S(wp,d)]. *)
let l_sort =
  let por x y = if x < y then 1 else if x > y then -1 else 0
  in sort por l
in
  if length l mod 2 = 1 then
    (* Nieparzysta liczba potomków. *)
    nieparzyste l_sort d
  else
    (* Parzysta liczba potomków. *)
    parzyste l_sort d;;
```

Ze względu na konieczność zastosowania sortowania, procedura ta ma złożoność czasową rzędu $O(p(v) \cdot \log p(v))$. Jej złożoność pamięciowa jest rzędu $O(p(v))$. Obliczenie wszystkich wartości $S(v,0), S(v,1), \ldots, S(v,\varphi(v))$ wymaga czasu rzędu $O(p(v) \cdot \log p(v) \cdot \varphi(v))$. Jak jednak obliczać $\varphi(v)$?

Zastanówmy się gdzie może znajdować się najdłuższa ścieżka w poddrzewie o korzeniu v? Pierwsza możliwość jest taka, że ścieżka ta nie przechodzi przez v — mieści się ona całkowicie w jednym z poddrzew, których korzeniami są potomkowie v. Druga możliwość jest taka, że ścieżka ta przechodzi przez v. Wówczas, o ile v ma przynajmniej dwóch potomków, to ścieżka ta łączy dwa najgłębiej położone liście w dwóch najwyższych poddrzewach v. Tak więc, żeby obliczyć $\varphi(v)$ musimy znać wysokości potomków v oraz średnice poddrzew, których korzeniami są potomkowie v: $\varphi(w_1), \varphi(w_2), \ldots, \varphi(w_{p(v)})$.

Oto procedura, która rekurencyjnie przegląda wczytane drzewo, dla każdego wierzchołka v oblicza wysokość poddrzewa o korzeniu v, jego średnicę, oraz wartości $S(v,0), S(v,1), \ldots, S(v,\varphi(v))$.

```
(* Analiza danego drzewa.           *)
(* Obliczane są:                    *)
(* - wysokość drzewa h,             *)
(* - średnica drzewa sr,            *)
(* - sl = [S(v,0); ... ; S(v, sr)]. *)
let rec analiza (Wezel l) =
  (* Poddaj analizie poddrzewa. *)
  let w = map analiza l
  in
    (* Wysokość i średnica drzewa. *)
    let (h, sr)  =
      (* Dwie największe wysokości poddrzew. *)
      let (h1, h2) =
        fold_left (fun (a1, a2) (h, _, _) ->
          if h >= a1 then (h, a1) else
          if h >= a2 then (a1, h) else (a1, a2))
          (-1, -1) w
      in
        (h1 + 1,
         fold_left (fun a (_,s,_) -> max a s) (h1 + h2 + 2) w)
```

```
    in
      (* [S(v,0); ... ; S(v, sr)] *)
      let sl =
        (* Iteracja obliczająca S(v, d) dla kolejnych d. *)
        let rec s_iter d ll ak =
          (* Ogon listy, przy czym jeśli pusty, to *)
          (* dublowana jest głowa.                 *)
          let tail l =
            match l with
              [x]  -> [x] |
              h::t -> t |
              _ -> failwith "tail: l = []"
          in
            if d > sr then rev ak else
              s_iter (d+1)
                (map tail ll)
                ((s d (map hd ll)) :: ak)
          in
            s_iter 0 (map (fun (_, _, l) -> l) w)[]
      in (h, sr, sl);;
```

Jaka jest złożoność tej procedury? Obliczenie wysokości i średnicy poddrzewa o korzeniu v na podstawie wysokości i średnic poddrzew o korzeniach w potomkach v wymaga czasu liniowego ze względu na $p(v)$. Tak więc dominujący jest czas potrzebny na obliczenie wartości $S(v,d)$, czyli:

$$\sum_{v=1,2,\ldots,n} (p(v)\cdot \log p(v)\cdot \varphi(v)) \leqslant n\cdot \sum_{v=1,2,\ldots,n} (p(v)\cdot \log p(v)) = O(n^2 \log n).$$

Złożoność pamięciowa tej procedury jest rzędu $\Theta(n)$. Z jednej strony konstrukcja drzewa wymaga takiej pamięci. Z drugiej strony, łączna liczba wartości funkcji S, jakie musimy równocześnie pamiętać, nie jest większa niż liczba przeanalizowanych już wierzchołków drzewa.

Procedura obliczająca końcowy wynik ma postać:

```
(* Minimalne ograniczenie na długość sznurków. *)
let ograniczenie t =
  let (_, sr, sl) = analiza t
  in
    snd
      (fold_left
        (fun (d, m) s -> (d+1, min (max d s) m))
        (0, inf) sl);;
```

Złożoność całego algorytmu pozostaje taka sama, jak złożoność procedury `analiza`, czyli złożoność czasowa jest rzędu $O(n^2 \log n)$, a pamięciowa rzędu $O(n)$.

Testy

Rozwiązania zawodników były testowane na dziesięciu testach: jednym małym teście poprawnościowym oraz dziewięciu, coraz większych testach losowych. Poniższa tabela przedstawia wielkości tych testów.

nr testu	n
1	19
2	143
3	397
4	1 431
5	2 101
6	2 837
7	4 041
8	7 230
9	9 115
10	10 000

Paweł Parys **Tomasz Malesiński**
Treść zadania, Opracowanie Program

Szpiedzy

Bajtocka Agencja Wywiadowcza ma w swoich szeregach szpiegów. Każdy szpieg w ramach obowiązków służbowych śledzi dokładnie jednego innego szpiega.

Król Bajtazar chce powierzyć tajną misję jak największej liczbie szpiegów. Misja jest jednak na tyle ważna, że każdy szpieg biorący w niej udział musi być śledzony przez przynajmniej jednego szpiega niebiorącego udziału w misji (przydział obowiązków związanych ze śledzeniem innych szpiegów nie ulega zmianie).

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis tego, którzy szpiedzy śledzą których,*
- *obliczy, ilu maksymalnie szpiegów można wysłać z tajną misją tak, aby każdy z nich był śledzony przez przynajmniej jednego szpiega nie biorącego udziału w misji,*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu wejścia zapisano jedną dodatnią liczbę całkowitą n — liczbę szpiegów, $2 \leqslant n \leqslant 1\,000\,000$. Szpiedzy są ponumerowani od 1 do n. W kolejnych n wierszach opisano kogo śledzi każdy ze szpiegów. W każdym z tych wierszy znajduje się po jednej dodatniej liczbie całkowitej. Liczba a_k znajdująca się w wierszu o numerze $k+1$ oznacza, że szpieg numer k śledzi szpiega numer a_k, $1 \leqslant k \leqslant n$, $1 \leqslant a_k \leqslant n$, $a_k \neq k$.

Wyjście

Twój program powinien wypisać w pierwszym wierszu wyjścia jedną liczbę całkowitą — maksymalną liczbę szpiegów, których można wysłać z tajną misją.

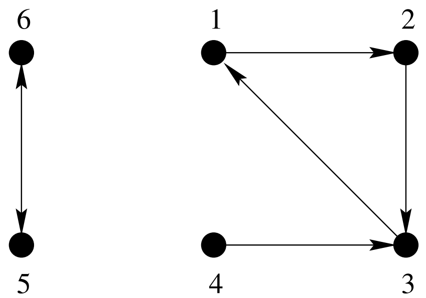
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
3
```



Rozwiązanie

Rozmiar danych sugeruje, że rozwiązanie powinno działać w czasie liniowym. Będzie to rozwiązanie zachłanne. Należy znajdować pary: szpieg pilnujący i szpieg uczestniczący w akcji, a następnie rozpatrywać pozostałych szpiegów. Ogólna idea jest taka, żeby znajdować szpiega, który nie jest przez nikogo śledzony. Może on albo kogoś pilnować, albo nic nie robić, więc lepiej żeby kogoś pilnował.

Grafem śledzenia będziemy nazywać graf skierowany, którego wierzchołkami są szpiedzy, a krawędź od szpiega s do szpiega t istnieje wtedy i tylko wtedy, gdy szpieg s śledzi szpiega t. Zadanie sprowadza się do znalezienia liczności maksymalnego skojarzenia w grafie śledzenia. Nie będziemy tu używać jakiegoś ogólnego algorytmu rozwiązywania tego problemu, gdyż byłby on wolniejszy, a ponadto bardziej skomplikowany. Trzeba tutaj skorzystać ze specjalnej postaci grafu, czyli z tego, że każdy szpieg śledzi tylko jednego innego szpiega.

Rozwiązanie wzorcowe

Rozpatrzmy ogólniejszy problem, w którym niektórzy szpiedzy mogą nie śledzić żadnego innego szpiega (takie sytuacje będą powstawały w czasie działania algorytmu). Krok naszego algorytmu wygląda następująco:

- Jeśli istnieje szpieg s, którego nikt nie śledzi i śledzi on szpiega t, to wyślij szpiega t na misję i niech s go śledzi. Usuń s i t z grafu śledzenia (jeśli ktoś śledził t, to teraz nie śledzi nikogo).
- Jeśli istnieje szpieg s, którego nikt nie śledzi i który nikogo nie śledzi, to usuń go z grafu.
- W przeciwnym przypadku w grafie zostały już tylko cykle, bo każdy jest śledzony przez co najmniej jednego innego szpiega, ale każdy może śledzić co najwyżej jednego innego, więc każdy śledzi dokładnie jednego i jest śledzony przez dokładnie jednego. Wybierz z każdego cyklu $\lfloor l/2 \rfloor$, gdzie l jest długością cyklu, szpiegów do udziału w misji (biorąc co drugiego).

Dowód poprawności

Poprawność algorytmu jest oczywista w przypadku, gdy w grafie zostały tylko cykle oraz szpiedzy nie śledzący i nie śledzeni przez nikogo. Wtedy jasne jest, że więcej szpiegów nie może brać udziału w misji. W przeciwnym przypadku niech G będzie grafem śledzenia, s szpiegiem, którego nikt nie śledzi, a t szpiegiem śledzonym przez s. Załóżmy, że istnieje rozwiązanie optymalne, w którym t nie bierze udziału w misji lub s go nie śledzi. Chcemy udowodnić, że w tym rozwiązaniu optymalnym wynik jest taki sam jak w naszym, czyli w pewnym takim, że t bierze udział w misji śledzony przez s. Mamy kilka przypadków:

- W danym rozwiązaniu optymalnym t śledzi szpiega v biorącego udział w misji. Wtedy rozwiązanie, w którym s śledzi wysłanego z misją t, a v nie uczestniczy w misji jest również optymalne.

- W danym rozwiązaniu optymalnym t bierze udział w misji, ale jest śledzony przez $v \neq s$. Wtedy rozwiązanie, w którym s śledzi t, a v nie śledzi nikogo jest również optymalne.

- W danym rozwiązaniu optymalnym t nie bierze udziału w misji i nie śledzi innego szpiega wysłanego z misją. Tak nie może się zdarzyć w rozwiązaniu optymalnym, bo można w takim przypadku jeszcze wysłać t z misją i s, żeby go śledził.

Zatem istnieje rozwiązanie optymalne, w którym t bierze udział w misji, a s go śledzi. Niech S będzie zbiorem szpiegów wysłanych z misją w tym rozwiązaniu. Niech S' będzie zbiorem szpiegów biorących udział w misji w optymalnym rozwiązaniu dla grafu G bez szpiegów s i t. Załóżmy, że $|S| > |S' \cup \{t\}|$. Tak jednak być nie może, bo wtedy $S - \{t\}$ byłoby poprawnym rozwiązaniem, lepszym od S', dla grafu G bez szpiegów s i t. Zatem wybranie szpiega t do udziału w misji i rozwiązanie problemu dla grafu G bez szpiegów s i t prowadzi do optymalnego rozwiązania.

Implementacja rozwiązania

Pozostaje jeszcze problem, w jaki sposób znajdować szpiegów opisanych w rozwiązaniu. Jest to jednak stosunkowo proste. Wystarczy dla każdego szpiega pamiętać ilu szpiegów aktualnie go śledzi. W programie wzorcowym `szp.c` służy do tego tablica `Deg`. Ponadto mamy kolejkę szpiegów, którzy nie są przez nikogo śledzeni (kolejka `Q`). Obie te struktury można łatwo uaktualniać przy usuwaniu wierzchołków z grafu. Przetwarzamy więc kolejnych szpiegów z kolejki. Gdy kolejka jest pusta, to trzeba już tylko posprawdzać długości cykli. Rozwiązanie działa więc w czasie i pamięci $O(n)$.

Testy

Zadanie testowane było na zestawie 12 danych testowych.

nr testu	typ testu	n	wynik
1	mały test	27	12
2	drzewa	1 237	609
3	cykl z ogonami	21 252	9 413
4	test losowy	10 000	4 330
5	test losowy	200 000	18 703
6	test rozgałęziony	701 011	1 005
7	długie ciągi	818 315	409 157
8	test rozgałęziony	1 000 000	4
9	długi ciąg	1 000 000	500 000
10	losowe cykle	1 000 000	499 997
11	test losowy	700 000	19 215
12	test losowy	1 000 000	19 358

Paru słów komentarza wymaga zastosowana tu terminologia:

Drzewa Cykl z podoczepianymi drzewami dwumianowymi.

Cykl z ogonami Jest to jeden duży cykl, do którego dochodzą różnej długości ścieżki.

Test rozgałęziony W teście 8 prawie wszyscy szpiedzy śledzą jednego. W teście 6 jest tysiąc szpiegów głównych śledzących jednego, a każdy z nich jest śledzony przez siedmiuset innych.

Długi ciąg Większość testu to jedna długa ścieżka szpiegów śledzących każdy następnego.

Długie ciągi Test składa się z wielu długich ścieżek zakończonych pojedynczymi pętelkami.

Losowe cykle Test zawiera wiele cykli różnej długości i nic poza tym.

Test losowy Każdy szpieg śledzi losowo wybranego innego szpiega.

Łukasz Kowalik — Treść zadania, Opracowanie
Tomasz Czajka — Program

Zawody

U stóp Bajtogóry znajduje się wejście do jaskini. Jaskinia to system komnat połączonych korytarzami. Wejście do jaskini prowadzi bezpośrednio do komnaty nazywanej **wejściową**. *Korytarze nie przecinają się (spotykają się jedynie w komnatach). Dwie komnaty mogą być albo połączone jednym korytarzem, albo mogą nie być połączone wcale (być może można jednak wtedy przejść z jednej do drugiej przechodząc po drodze przez inne komnaty). Korytarz łączy zawsze dwie różne komnaty.*

Postanowiono zorganizować zawody o puchar króla Bajtocji. Celem każdego z zawodników jest przebycie dowolnie wybranej trasy w jaskini i wyjście na zewnątrz w jak najkrótszym czasie. Trasa musi przechodzić przez co najmniej jedną komnatę inną niż wejściowa. Obowiązują tylko dwie zasady: podczas wędrówki po jaskini, każdą komnatę można odwiedzić co najwyżej raz (z wyjątkiem komnaty wejściowej), podobnie tylko raz można przejść każdym z korytarzy.

Do zawodów przygotowuje się słynny grotołaz Bajtała. Bajtała długo trenował w jaskini i dokładnie poznał sieć korytarzy. Dla każdego z korytarzy wyznaczył czasy potrzebne do jego przejścia w każdą stronę. Czas potrzebny do poruszania się po komnatach można zaniedbać. Bajtała chciałby teraz wyznaczyć taką trasę spełniającą wymogi zawodów, którą można przebyć w jak najkrótszym czasie (czas potrzebny do przebycia trasy jest sumą czasów potrzebnych do przejścia korytarzy składających się na trasę).

Zadanie

Pomóż Bajtale! Napisz program, który:

- *wczyta ze standardowego wejścia opis jaskini oraz czasy potrzebne do przejścia poszczególnych korytarzy,*
- *obliczy trasę zgodną z zasadami zawodów, dla której suma czasów przejść korytarzy składających się na trasę jest najmniejsza,*
- *wypisze na standardowe wyjście czas potrzebny do przejścia wyznaczonej trasy.*

Wejście

W pierwszym wierszu wejścia znajdują się dwie liczby całkowite n i m oddzielone pojedynczym odstępem, oznaczające odpowiednio liczbę komnat w jaskini, oraz liczbę łączących je korytarzy, $3 \leqslant n \leqslant 5\,000$, $3 \leqslant m \leqslant 10\,000$. Komnaty są ponumerowane od 1 do n. Komnata wejściowa ma numer 1. W kolejnych m wierszach opisane są poszczególne korytarze. W każdym z tych wierszy znajduje się czwórka liczb naturalnych oddzielonych pojedynczymi odstępami. Czwórka a,b,c,d oznacza, że jaskinie a i b są połączone korytarzem, czas przejścia z jaskini a do b wynosi c, natomiast z b do a wynosi d, $1 \leqslant a,b \leqslant n$, $a \neq b$, $1 \leqslant c,d \leqslant 10\,000$. Możesz założyć, że zawsze istnieje przynajmniej jedna trasa spełniająca wymogi zawodów.

Wyjście

Twój program powinien wypisać w pierwszym i jedynym wierszu wyjścia jedną liczbę całkowitą — minimalny czas potrzebny do przebycia trasy spełniającej warunki zawodów.

Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
6
```

Rozwiązanie

Wyjdźmy z jaskini

W treści zadania jest mowa o jaskini, ale i tak każdy wie, że chodzi o graf. Wierzchołkami naszego grafu są komnaty jaskini, to jasne. Ale czym są krawędzie? Wiemy, że dwie komnaty mogą być połączone korytarzem. Każdy z korytarzy Bajtała przemierzył w obie strony, zapisując czasy przejścia. Umówmy się, że każdemu korytarzowi w jaskini odpowiadają w grafie *dwie* krawędzie. Jeśli korytarz łączy komnaty odpowiadające wierzchołkom u i v, jedna z tych krawędzi prowadzi od u do v, natomiast druga odwrotnie, od v do u. Dodatkowo, każdej z krawędzi e przypisano *wagę* $w(e)$ — tzn. liczbę naturalną równą czasowi przejścia korytarza w odpowiednią stronę. Tak opisany graf będziemy oznaczać przez G. Zbiory wierzchołków i krawędzi G będziemy tradycyjnie oznaczać, odpowiednio, przez V i E. Przyjmiemy też, że n i m oznaczają, odpowiednio, liczby wierzchołków i krawędzi grafu G. Zauważmy, że G jest grafem skierowanym, choć bardzo specyficznym: jeśli w G istnieje krawędź od u do v (oznaczamy ją (u,v)), to jest tam również krawędź (v,u).

Spróbujmy teraz sformułować nasze zadanie w języku teorii grafów. W tym celu potrzebujemy kilku definicji.

Dowolny ciąg krawędzi (v_0,v_1), (v_1,v_2), …, (v_{k-1},v_k) nazywamy *ścieżką*. Jeśli $v_0 = v_k$ to taką ścieżkę nazywamy *cyklem*. Gdy wierzchołki v_0, v_1, …, v_k są różne mamy do czynienia ze *ścieżką prostą*. Podobnie, gdy wierzchołki v_0, v_1, …, v_{k-1} są różne i $v_0 = v_k$, to mamy *cykl prosty*. *Wagą* albo *długością* ścieżki nazywamy sumę wag jej krawędzi. Często zamiast pisać „ścieżka o najmniejszej wadze” będziemy używać określenia *najkrótsza ścieżka*.

Teraz, gdy przyswoiliśmy sobie tych kilka prostych pojęć, możemy wreszcie nazwać rzecz po imieniu. Niech v_0 będzie wierzchołkiem odpowiadającym komnacie wejściowej. Zadanie polega na opracowaniu algorytmu, który znajdzie w grafie G najkrótszy (o najmniejszej wadze) cykl prosty przechodzący przez v_0 i składający się z więcej niż dwóch krawędzi (zauważmy, że przechodząc w jaskini trasę odpowiadającą takiemu cyklowi Bajtała co najwyżej raz odwiedza każdą komnatę, z wyjątkiem wejściowej, podobnie co najwyżej raz przechodzi przez każdy korytarz).

Obserwacja

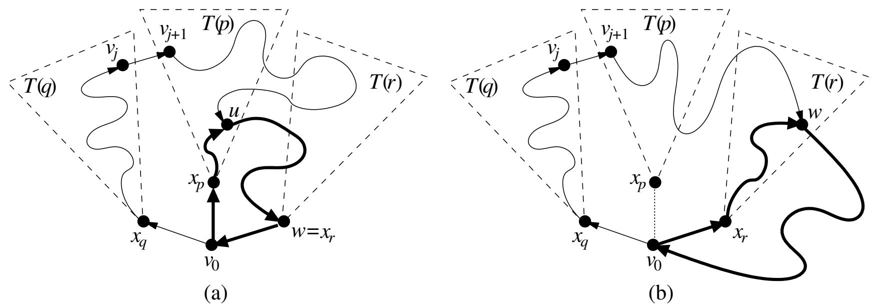
Niech C będzie szukanym cyklem i niech v_0, v_1, ..., v_k będą jego kolejnymi wierzchołkami oraz $v_0 = v_k$. Wybierzmy największe $j \in \{0,1,\ldots,k-1\}$ takie, że ścieżka P_1 przechodząca kolejno przez wierzchołki v_0, ..., v_j jest najkrótszą ścieżką od v_0 do v_j. Oczywiście takie j zawsze istnieje. W najgorszym razie $j = 0$ i jedyna taka ścieżka ma długość 0. Bez trudu zauważamy, że ścieżka $P_2 = v_j, v_{j+1}, \ldots, v_k$ (pamiętamy, że $v_k = v_0$) jest najkrótszą ścieżką od v_j do v_0 omijającą wierzchołki v_1, ..., v_{j-1}.

Wnioski

Bez straty ogólności możemy założyć, że do każdego wierzchołka w grafie prowadzi ścieżka o początku w v_0 (choć treść zadania nie wyklucza istnienia wierzchołków, do których nie prowadzi żaden korytarz, nie będą one miały wpływu na rozwiązanie). W takim razie każdemu wierzchołkowi w grafie, powiedzmy v, możemy przypisać najkrótszą ścieżkę od v_0 do v i oznaczyć ją przez $s(v)$ (gdy takich najkrótszych ścieżek jest wiele, wybieramy dowolną z nich). Niech x_1, ..., x_d będą sąsiadami v_0. Dla $i = 1,\ldots,d$ oznaczmy

$$T(i) = \{v \in V : \text{ścieżka } s(v) \text{ zaczyna się od krawędzi } (v_0, x_i)\}.$$

Oczywiście może się zdarzyć, że $T(i) = \emptyset$ dla niektórych wartości i. Dodatkowo przyjmujemy $T(0) = \{v_0\}$. Nietrudno zauważyć, że zbiory $T(i)$ tworzą podział zbioru V, tzn. każdy wierzchołek jest w dokładnie jednym z tych zbiorów. Przypomnijmy sobie teraz naszą obserwację i zastanówmy się, przez ile zbiorów $T(i)$ może przechodzić szukany przez nas cykl C. Najpierw przyjrzyjmy się ścieżce $P_1 = v_0,\ldots,v_j$. Widzimy, że wszystkie jej wierzchołki, z wyjątkiem v_0, znajdują się w jednym zbiorze $T(q)$, dla $v_1 = x_q$. Teraz zastanówmy się nad drugą ścieżką, tzn. ścieżką $P_2 = v_j,\ldots,v_k$. Pamiętamy, że $v_j \in T(q)$, ale $v_{j+1} \in T(p)$, dla $p \neq q$. Co się dzieje dalej? Załóżmy, że jeden z kolejnych wierzchołków ścieżki należy do zbioru $T(r)$, dla $r \neq p$. Pokażemy, że istnieje wtedy cykl prosty C', składający się z co najmniej 3 krawędzi i zawierający v_0, ale krótszy od C.



Rysunek 1: Zastępowanie cyklu C przez krótszy cykl C' (pogrubiony)

Niech w będzie ostatnim wierzchołkiem na ścieżce P_2, który należy do $T(r)$ oraz niech u będzie ostatnim wierzchołkiem na P_2 przed w, który należy do $T(p)$. Jeśli $w = x_r$ oraz krawędź wychodząca z w na ścieżce P_2 prowadzi do v_0, możemy otrzymać cykl C' poprzez sklejenie ścieżki $s(u)$ z tą częścią ścieżki P_2, która zaczyna się w u (patrz rys. 1a). W przeciwnym przypadku cykl C' powstaje przez sklejenie ścieżki $s(w)$ z tą częścią P_2, która zaczyna się w w (patrz rys. 1b). W obu przypadkach cykl C' jest krótszy niż C. To jest niemożliwe, a więc wierzchołki $v_{j+1}, \ldots, v_{k-1}$ leżą wszystkie w jednym zbiorze $T(p)$. Więcej, widzimy że ścieżka $v_{j+1}, \ldots, v_k$ jest najkrótszą ścieżką od v_{j+1} do $v_k = v_0$ spośród ścieżek, które przechodzą wyłącznie po wierzchołkach z $T(p)$. Podsumujmy:

Fakt 1 *Najkrótszy cykl prosty o więcej niż dwóch krawędziach i zawierający wierzchołek v_0, składa się z*

- *ścieżki $v_0, \ldots, v_j$ równej ścieżce $s(v_j)$,*
- *krawędzi (v_j, v_{j+1}) takiej, że jeśli $v_1 = x_q$, to $v_{j+1} \in T(p)$, $p \neq q$,*
- *najkrótszej ścieżki od v_{j+1} do v_0 przechodzącej wyłącznie po wierzchołkach $T(p)$.*

Algorytm

Krok pierwszy: tam...

Odkrycie powyższego faktu prowadzi nas już bez przeszkód do algorytmu. Na początek dla każdego wierzchołka w grafie obliczamy jego odległość od v_0, tzn. długość najkrótszej ścieżki od v_0 do tego wierzchołka. W tym celu stosujemy klasyczny algorytm Dijkstry. Jeśli jeszcze go nie poznałeś, zajrzyj koniecznie do jednej z wielu książek, w których jest on dokładnie opisany (książka „Wprowadzenie do algorytmów” Cormena, Leisersona i Rivesta ([17]) jest tu szczególnie godna polecenia). Algorytm Dijkstry wyznacza nie tylko długości najkrótszych ścieżek. Dla każdego wierzchołka v wyznacza także wierzchołek poprzedzający v na pewnej najkrótszej ścieżce od v_0 do v. Nam ta informacja będzie niepotrzebna, ale z jej pomocą możemy łatwo (w czasie liniowym) obliczyć dla każdego wierzchołka $v \neq v_0$ wierzchołek $x_{p(v)}$ taki, że najkrótsza ścieżka od v_0 do v znaleziona przez algorytm Dijkstry zaczyna się od krawędzi $(v_0, x_{p(v)})$. Dodatkowo przyjmujemy $p(v_0) = 0$. Niech ponadto $d(v)$ oznacza obliczoną odległość v od v_0.

Krok drugi: ...z powrotem

Następnie dla każdego wierzchołka $v \neq v_0$ wyznaczamy długość najkrótszej ścieżki do v_0 przechodzącej wyłącznie po wierzchołkach ze zbioru $T(p(v))$. Obojętnie, jaką metodą znajdujemy ścieżki, możemy wymusić to ograniczenie po prostu poprzez ignorowanie krawędzi łączących wierzchołki o różnych wartościach $p(\cdot)$ (choć zostawiamy wszystkie krawędzie wchodzące do i wychodzące z v_0). Aby obliczyć długości ścieżek moglibyśmy dla każdego wierzchołka uruchomić algorytm Dijkstry. Byłoby to jednak nadmiernie czasochłonne, gdyż wystarczy nam jedno uruchomienie tego algorytmu! Jedyne, co musimy zrobić, to odwrócić kierunki krawędzi w grafie i uruchomić algorytm Dijkstry, który w takim *odwróconym* grafie znajdzie długości najkrótszych ścieżek od v_0 do wszystkich innych wierzchołków. Ścieżka od v_0 do v w grafie odwróconym to oczywiście ścieżka od v do v_0 w grafie oryginalnym, a więc dostajemy dokładnie to, co chcieliśmy. Niech $h(v)$ oznacza obliczoną odległość. Przyjmujemy $h(v_0) = 0$.

Krok trzeci: minimum

Krawędź $e = (v, w) \in E$ nazwiemy *pośrednią*, gdy $p(v) \neq p(w)$. Fakt 1 mówi nam, że aby znaleźć długość szukanego najkrótszego cyklu, wystarczy teraz znaleźć taką krawędź pośrednią $e = (v, w)$, że suma $d(v) + w(e) + h(w)$ jest najmniejsza. Wartość tej sumy jest szukaną liczbą.

Uwagi o złożoności

Krok trzeci algorytmu zajmuje oczywiście czas liniowy, a więc całkowita złożoność czasowa jest zdominowana przez złożoność czasową użytej implementacji algorytmu Dijkstry. Najprostsza implementacja tego algorytmu ma złożoność $O(n^2)$, jednakże testy zostały tak dobrane, że takie rozwiązanie nie zdobywało maksymalnej liczby punktów. Taki wynik gwarantowało dopiero użycie w algorytmie Dijkstry kopców binarnych, które zmniejszają pesymistyczny czas obliczeń do $O((n+m)\log n)$.

Program wzorcowy

Rozwiązanie zaprogramowane w programie wzorcowym jest nieco inne od opisanego powyżej, bazuje jednak na tych samych obserwacjach. Zaczynamy tak samo, od wykonania kroku pierwszego, tzn. obliczamy $d(v)$ i $p(v)$ dla wszystkich wierzchołków. Następnie budujemy taki graf G', żeby najkrótsza ścieżka pomiędzy dwoma wyróżnionymi jego wierzchołkami, S i M, miała długość taką samą, jak poszukiwany cykl w G. Oprócz S i M do grafu G' dodajemy jeszcze po jednym wierzchołku v' dla każdego wierzchołka $v \neq v_0$ grafu G. Graf G' zawiera następujące krawędzie (u, v oznaczają dowolne wierzchołki w grafie G takie, że $u, v \neq v_0$ i $d(u), d(v) \neq \infty$):

(e1) krawędź (S, M) z wagą $d(u) + w$ dla każdej $(u, v_0) \in E$ z wagą w, $u \neq x_p(u)$,

(e2) krawędź (S, v') z wagą w dla każdej $(v_0, v) \in E$ z wagą w, $v \neq x_p(v)$,

(e3) krawędź (S, v') z wagą $d(u) + w$ dla każdej $(u, v) \in E$ z wagą w, $x_p(u) \neq x_p(v)$,

(e4) krawędź (u', M) z wagą w dla każdej $(u, v_0) \in E$ z wagą w, $u = x_p(u)$,

(e5) krawędź (u', v') z wagą w dla każdej $(u, v) \in E$ z wagą w, $x_p(u) = x_p(v)$.

Widzimy, że G' jest liniowego rozmiaru względem oryginalnego grafu G, a więc jego skonstruowanie zajmie czas $O(n+m)$, a algorytm Dijkstry zaimplementowany za pomocą kopców binarnych znajdzie długość najkrótszej ścieżki od S do M w czasie $O((n+m)\log n)$.

Pozostaje jedynie uzasadnić, że wyniki otrzymywane przez oba algorytmy są takie same. W tym celu należy pokazać, że najkrótsza ścieżka od S do M w G' ma długość $d(v) + w(e) + h(w)$ dla pewnej krawędzi pośredniej $e = (v, w) \in E$. I odwrotnie, dla każdej krawędzi pośredniej $e = (v, w) \in E$ w grafie G' znajdzie się ścieżka od S do M długości co najwyżej $d(v) + w(e) + h(w)$. Proste dowody tych dwóch faktów pozostawiamy Tobie, Szanowny Czytelniku, jako ćwiczenie.

Rozwiązanie poprawne, ale nieoptymalne

Rozważmy następujący algorytm: dla każdej krawędzi wychodzącej z v_0 i prowadzącej do pewnego wierzchołka x obliczamy za pomocą algorytmu Dijkstry długość najkrótszej ścieżki od x do v_0 w grafie, w którym usunięto krawędź (x, v_0). Ten bardzo prosty algorytm jest oczywiście poprawny, ale w pesymistycznym przypadku, gdy z komnaty wejściowej istnieją korytarze do wszystkich innych komnat, jego złożoność czasowa wynosi $O(n(n+m)\log n)$.

Testy

Wszystkie testy, oprócz dwóch pierwszych, zostały wygenerowane automatycznie. Każdy z testów `zaw2.in`, `zaw3.in`, ..., `zaw10.in` zawierał dwie komnaty połączone korytarzem, do których nie można było dojść z komnaty wejściowej.

- `zaw1.in` — mały test poprawnościowy, $n = 13$, $m = 15$.
- `zaw2.in` — prawie cykl, $n = 32$, $m = 58$.
- `zaw3.in` — prawie graf pełny (prawie wszystkie wierzchołki połączone ze sobą), $n = 42$, $m = 781$.
- `zaw4.in` — dosyć losowy, duże rozgałęzienie z komnaty wejściowej, $n = 102$, $m = 2001$.
- `zaw5.in` — prawie drzewo binarne, $n = 4997$, $m = 4996$.
- `zaw6.in` — krata, dużo możliwości, $n = 2502$, $m = 4997$.
- `zaw7.in` — losowy, duże rozgałęzienie, $n = 3002$, $m = 9996$.
- `zaw8.in` — krata, $n = 5000$, $m = 9993$.
- `zaw9.in` — długi cykl z dołożonymi krawędziami od v_0 do wszystkich pozostałych wierzchołków, $n = 4998$, $m = 9990$.
- `zaw10.in` — prawie cykl, $n = 5000$, $m = 9994$.

Zawody II stopnia

opracowania zadań

Jakub Pawlewicz Treść zadania, Opracowanie

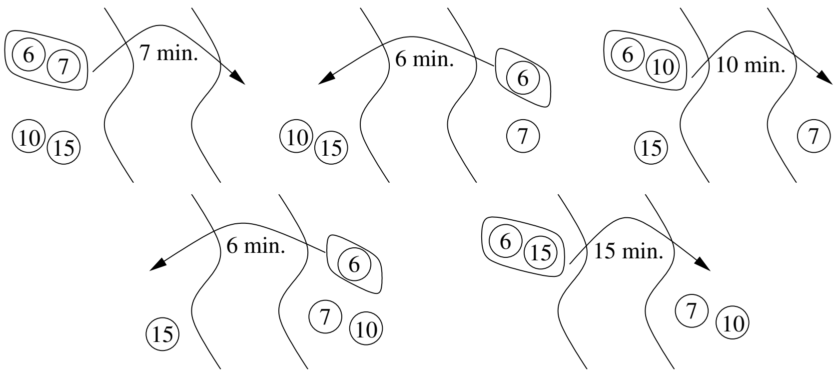
Michał Adamaszek Program

Most

W środku nocy grupa turystów chce przejść przez stary, zniszczony, dziurawy most. Turyści mają jedną latarkę. Światło latarki umożliwia przejście przez most maksymalnie dwóm turystom na raz. Turyści nie mogą przechodzić przez most bez latarki ani w większych niż dwuosobowe grupach, gdyż grozi to wpadnięciem do rzeki. Każdemu turyście przejście przez most zajmuje określony czas. Dwóch turystów idących razem potrzebuje na przejście przez most tyle czasu, co wolniejszy z nich. Jaki jest najkrótszy czas, w którym wszyscy turyści mogą przejść przez most?

Przykład

Przypuśćmy, że grupa liczy czterech turystów. Pierwszy z nich potrzebuje na przejście przez most 6 minut, drugi 7 minut, trzeci 10 minut, a czwarty 15 minut. Poniższy rysunek przedstawia, w jaki sposób turyści mogą przejść przez most w 44 minuty. Mogą to jednak zrobić szybciej. Jak?



Przykładowy sposób przejścia mostu w 44 minuty. Liczby w kółeczkach oznaczają liczby minut, jaką turysta potrzebuje na przejście mostu.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis grupy turystów,*
- *znajdzie najkrótszy czas wystarczający do przejścia przez most,*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu standardowego wejścia zapisana jest jedna dodatnia liczba całkowita n — liczba turystów, $1 \leqslant n \leqslant 100\,000$. W kolejnych n wierszach zapisany jest niemalejący ciąg n dodatnich liczb całkowitych nie większych niż $1\,000\,000\,000$, po jednej w każdym wierszu. Liczba w $i+1$-szym wierszu ($1 \leqslant i \leqslant n$) określa czas potrzebny i-temu turyście na przejście przez most. Suma tych czasów nie przekracza $1\,000\,000\,000$.

Wyjście

Twój program powinien wypisać na standardowe wyjście, w pierwszym wierszu, jedną liczbę całkowitą — najkrótszy czas wystarczający, aby wszyscy turyści przeszli na drugą stronę mostu.

Przykład

Dla danych wejściowych:

```
4
6
7
10
15
```

poprawnym wynikiem jest:

```
42
```

Rozwiązanie

Załóżmy, że $n \geqslant 2$. Przypadek z tylko jednym turystą, jest trywialny. Rozwiązanie zadania jest bardzo proste i można opisać je jednym wzorem przedstawionym w twierdzeniu 1. O wiele trudniej jest udowodnić, że wzór (1) opisuje rzeczywiście najkrótszy czas przejścia turystów przez most.

Oznaczmy turystów etykietami od 1 do n. Niech czasy przejścia turystów wynoszą $t_1 \leqslant t_2 \leqslant \ldots \leqslant t_n$.

Twierdzenie 1 *Minimalny czas przejścia przez most wynosi*

$$\min_{0 \leqslant k \leqslant \lfloor \frac{n}{2} \rfloor - 1} S_k, \tag{1}$$

gdzie

$$S_k = (n-k-2)t_1 + (2k+1)t_2 + \sum_{i=3}^{n-2k} t_i + \sum_{i=0}^{k-1} t_{n-2i}. \tag{2}$$

Przykład 1 Dla $n = 6$ wzór (1) ma postać:

$$\min\{4t_1 + t_2 + t_3 + t_4 + t_5 + t_6, 3t_1 + 3t_2 + t_3 + t_4 + t_6, 2t_1 + 5t_2 + t_4 + t_6\}.$$

Liczenie (1) można wykonać w czasie $O(n)$, gdyż $S_k - S_{k-1} = -t_1 + 2t_2 - t_{n-2k+1}$, skąd mamy równoważność:

$$S_{k-1} > S_k \iff 2t_2 - t_1 < t_{n-2k+1}. \tag{3}$$

Z (3) wynika, że wyznaczenie optymalnej wartości k polega na znalezieniu miejsca dla wartości $2t_2 - t_1$ w posortowanej liście wartości t_i:

$$k = \max(\{i : 1 \leqslant 2i \leqslant n, 2t_2 - t_1 < t_{n-2i+1}\} \cup \{0\}). \tag{4}$$

Dzięki temu, że czasy przejścia turystów są posortowane od najmniejszej wartości do największej, rozwiązanie można przy odrobinie wysiłku zapisać w pamięci o rozmiarze $O(1)$, bez zapamiętywania wszystkich czasów w tablicy.

Dowód twierdzenia 1

Poniższy dowód zaczerpnięty jest z [27]. Przejście zbioru turystów A z lewego brzegu na prawy oznaczamy $\overrightarrow{A}$, natomiast z prawego na lewy oznaczamy $\overleftarrow{A}$. Będziemy rozważać ciągi takich przejść, a dokładniej przemienne ciągi takich przejść. Są to ciągi postaci $\overrightarrow{A_1}, \overleftarrow{A_2}, \overrightarrow{A_3}, \ldots, \overleftarrow{A_{m-1}}, \overrightarrow{A_m}$. Taki ciąg tworzy możliwe przejście turystów przez most tylko wtedy, jeśli zachodzą warunki:

- Każdy zbiór A_i jest jedno lub dwuelementowym podzbiorem $\{1, \ldots, n\}$.
- Dla każdego turysty $a \in \{1, \ldots, n\}$ podciąg, utworzony z przejść ciągu $\overrightarrow{A_1}, \overleftarrow{A_2}, \ldots, \overrightarrow{A_m}$, w których uczestniczył a, składa się z przejść na przemian w prawo i w lewo, zaczyna się i kończy na przejściu w prawo.

Innymi słowy pierwszy warunek oznacza, że przez most mogą przechodzić tylko 1 lub 2 osoby, a drugi warunek oznacza, że turysta a na początku musi się znajdować na lewym brzegu, przy przejściu zmieniać brzeg na przeciwny, a na końcu ma się znaleźć na prawym brzegu.

Lemat 2 *W rozwiązaniu optymalnym w prawo przechodzą zawsze dwie osoby, zaś w lewo przechodzi zawsze jedna osoba.*

Dowód Chcemy udowodnić, że w rozwiązaniu występują tylko przejścia postaci $\overrightarrow{\{x,y\}}$ oraz $\overleftarrow{\{x\}}$. Rozważmy pierwsze wystąpienie w ciągu przemiennym, które nie jest danej postaci. Rozważmy dwa przypadki.

1. Rozważane przejście jest postaci $\overrightarrow{\{a\}}$. Jeśli jest to pierwsze przejście w ciągu, to następnym musi być $\overleftarrow{\{a\}}$, wtedy dwa kolejne przejścia $\overrightarrow{\{a\}}, \overleftarrow{\{a\}}$ mogą być usunięte.

   Zatem, niech przejście $\overrightarrow{\{a\}}$ nie będzie pierwszym przejściem. Weźmy przejście bezpośrednio przed $\overrightarrow{\{a\}}$. Będzie to $\overleftarrow{\{b\}}$. W sytuacji, gdy $a = b$, oba przejścia można usunąć. Niech więc $a \neq b$. Weźmy ostatnie przejście przed $\overleftarrow{\{b\}}$, w którym brał udział turysta a lub b. Będzie to przejście $\overleftarrow{\{a\}}$ lub $\overrightarrow{\{b,c\}}$. W obu przypadkach możemy zmodyfikować ciąg tak, aby otrzymać rozwiązanie o krótszym sumarycznym czasie przejścia:

$$\begin{array}{rcl} P_1, \overleftarrow{\{a\}}, P_2, \overleftarrow{\{b\}}, \overrightarrow{\{a\}}, P_3 & \implies & P_1, \overleftarrow{\{b\}}, P_2, P_3 \\ P_1, \overrightarrow{\{b,c\}}, P_2, \overleftarrow{\{b\}}, \overrightarrow{\{a\}}, P_3 & \implies & P_1, \overrightarrow{\{a,c\}}, P_2, P_3, \end{array}$$

gdzie P_1, P_2, P_3 oznaczają pewne ciągi przejść, przy czym P_2 jest ciągiem przejść nie zawierających a i b.

2. Rozważane przejście jest postaci $\overleftarrow{\{a,b\}}$. Weźmy ostatnie przejście przed $\overleftarrow{\{a,b\}}$, w którym brał udział turysta a. Niech to będzie przejście $\overrightarrow{\{a,x\}}$ (możliwe jest $x = b$). Możemy z obu ruchów usunąć a, nie zwiększając sumarycznego czasu przejścia:

$$P_1, \overrightarrow{\{a,x\}}, P_2, \overleftarrow{\{a,b\}}, P_3 \quad \Longrightarrow \quad P_1, \overrightarrow{\{x\}}, P_2, \overleftarrow{\{b\}}, P_3.$$

Otrzymany ciąg nie może być optymalny, co wynika z analizy przypadku 1. ■

W dalszej części rozważań najpierw uogólnimy nasz problem. Dzięki temu łatwo będzie wskazać rozwiązanie optymalne dla uogólnionego problemu. Na końcu pokażemy, że dla uzyskanego rozwiązania optymalnego daje się skonstruować ciąg przejść turystów spełniający warunki zadania.

Opiszemy teraz nasz problem na grafie składającym się z wierzchołków $\{1, \ldots, n\}$. Dla każdej pary $\overrightarrow{\{x,y\}}$ w ciągu przejść tworzymy krawędź $\{x,y\}$ z wagą $\max\{t_x, t_y\}$, równą czasowi przejścia turystów x, y na prawą stronę. Tak opisane rozwiązanie jest multigrafem (graf z dozwolonymi krawędziami równoległymi) $G = (V, E)$ spełniającym warunki:

$$\deg(i) \geqslant 1 \qquad \text{dla każdego } i = 1, \ldots, n, \tag{5}$$
$$|E| = n - 1. \tag{6}$$

Wynikają one z lematu 2. Warunek (5) zachodzi dlatego, że każdy turysta musi co najmniej raz przejść most w prawo. Ponadto ciąg przejść będzie się składać z $n-1$ przejść dwóch osób w prawo i $n-2$ przejść jednej osoby w lewo, skąd mamy (6).

Wartość $\deg(i)$, stopień wierzchołka i, oznacza, ile razy turysta i idzie w prawo. A zatem musi on iść $\deg(i) - 1$ razy w lewo, co daje dodatkowy czas przejścia $(\deg(i) - 1)t_i$. Zatem całkowity czas przejścia turystów przez most w sposób opisany przez graf G wynosi

$$\sum_{i=1}^{n} (\deg(i) - 1)t_i + \sum_{\{x,y\} \in E} \max\{t_x, t_y\}. \tag{7}$$

Zwróćmy uwagę, że w sumie $\sum_{\{x,y\} \in E}$ krawędź wielokrotna występuje tyle razy, ile wynosi jej krotność. Zamiast minimalizować (7) możemy dodać stałą $\sum_{i=1}^{n} t_i$ i minimalizować wyrażenie

$$\sum_{i=1}^{n} \deg(i)t_i + \sum_{\{x,y\} \in E} \max\{t_x, t_y\}, \tag{8}$$

co przepisujemy jako

$$\sum_{\{x,y\} \in E} \left(t_x + t_y + \max\{t_x, t_y\}\right). \tag{9}$$

Możemy tak zrobić, gdyż wierzchołek i jest końcem $\deg(i)$ krawędzi, a więc składnik t_i wystąpi w sumie (9) $\deg(i)$ razy, skąd mamy (8)$\Leftrightarrow$(9). Teraz przyjmując $c_{xy} = t_x + t_y + \max\{t_x, t_y\}$, dostajemy zadanie:

$$\text{Znaleźć taki graf } G\text{, który minimalizuje } \sum_{\{x,y\} \in E} c_{xy} \text{ przy warunkach (5) i (6).} \tag{10}$$

Tak postawiony problem wydaje się być ogólniejszy niż szukanie optymalnego ciągu przejść turystów. Oczywiste jest, że dla każdego ciągu przejść możemy skonstruować graf G spełniający (5) i (6). Nieoczywistym, ale prawdziwym faktem jest, że z grafu G możemy skonstruować odpowiedni ciąg przejść turystów. Nie będziemy jednak dowodzić tego w ogólnym przypadku. Wystarczy pokazać, że dla optymalnego grafu G ciąg przejść jest konstruowalny. Wpierw jednak zobaczmy, jakie są właściwości rozwiązania optymalnego.

Lemat 3 *Istnieje optymalne rozwiązanie G spełniające warunki:*

(a) Jeśli dwie krawędzie mają końce w czterech różnych wierzchołkach $x < y < z < u$, to tymi krawędziami są $\{x,y\}$ i $\{z,u\}$.

(b) Jeśli dwie krawędzie mają tylko jeden wspólny koniec, to tym końcem jest wierzchołek 1.

(c) Jeśli dwie krawędzie mają oba końce wspólne, to są to 1 i 2.

(d) Jeśli w grafie jest krawędź $\{1,i\}$, to w G są też krawędzie $\{1,j\}$ dla $j = 2,\ldots,i-1$.

(e) Jeśli krawędź $\{x,y\}$ nie ma końca w 1, czyli jest $1 < x < y$, to $y = x+1$.

Dowód

(a) Mamy trzy możliwości łączenia x, y, z, u dwoma rozłącznymi krawędziami. Najmniejszy koszt otrzymujemy dla krawędzi $\{x,y\}$ i $\{z,u\}$. Wynosi on $c_{xy}+c_{zu} = t_x+2t_y+t_z+2t_u$. Dla pozostałych dwóch par koszt ten wynosi $c_{xz}+c_{yu} = c_{xu}+c_{yz} = t_x+t_y+2t_z+2t_u$.

(b) Mamy dwie krawędzie $\{x,y\}$, $\{x,z\}$ i y lub z jest różne od 1. Niech $y \neq 1$. Wtedy, jeśli $x \neq 1$, to w grafie G moglibyśmy zamiast krawędzi $\{x,y\}$ wziąć krawędź $\{1,y\}$. Otrzymany graf miałby nie większą sumę (10) i spełniałby warunki (5) i (6).

(c) Mamy dwie krawędzie $\{x,y\}$. Jeśli byłoby $\{x,y\} \neq \{1,2\}$, to w grafie G moglibyśmy zamiast jednej krawędzi $\{x,y\}$ wziąć krawędź $\{1,2\}$. Ponownie otrzymany graf miałby nie większą sumę (10) i spełniałby warunki (5) i (6).

(d) Niech $1 < j < i$. Zastanówmy się, czy krawędź $\{x, j\}$, dla $x \neq 1$, może być w G. Z (b) mamy $x \neq i$, więc liczby 1, i, j, x są parami różne. Z (a) wynika, że j nie może być w parze z x, a ponieważ j musi być końcem jakiejś krawędzi, więc w G jest $\{1, j\}$.

(e) Załóżmy, że istnieje z takie, że $x < z < y$. Ponieważ $\deg(z) \geqslant 1$, więc w grafie jest krawędź $\{z,u\}$. Z (b) wynika, że $u \neq x$ i $u \neq y$. Zatem z (a) wynika, że w takim układzie x i y nie będą połączone krawędzią. Sprzeczność dowodzi, że $x+1 = y$. ■

Na podstawie lematu 3 możemy ustalić możliwą strukturę optymalnego rozwiązania G. Z (c) dostajemy, że jedyną krawędzią wielokrotną jest $\{1,2\}$. Wszystkie krawędzie o końcu w 1 to $\{1, j\}$, gdzie $j = 2,\ldots,i$, dla pewnego i — patrz (d). Każdy z pozostałych wierzchołków $x \in \{i+1,\ldots,n\}$ jest końcem dokładnie jednej krawędzi (b) i jego sąsiadem jest $x-1$ lub $x+1$ (e). Podsumowując otrzymujemy:

Twierdzenie 4 *Istnieje optymalny graf będący rozwiązaniem (10), który dla pewnego k,* $0 \leqslant k \leqslant n/2-1$*, składa się z następujących krawędzi:*

- *k „parujących krawędzi”* $\{n,n-1\}$, $\{n-2,n-3\}$, …, $\{n-2k+2,n-2k+1\}$,
- $k+1$ *kopii krawędzi* $\{1,2\}$,
- *oraz* $n-2k-2$ *krawędzi* $\{1,3\}$, $\{1,4\}$, …, $\{1,n-2k\}$.

Lemat 5 *Dla grafu przedstawionego w twierdzeniu 4 istnieje odpowiedni ciąg przejść turystów.*

Dowód Konstruujemy odpowiedni ciąg przejść indukcyjnie po n. Dla $n=2$ ciąg przejść to $\overrightarrow{\{1,2\}}$. Dla $n=3$, to $\overrightarrow{\{1,3\}}$, $\overleftarrow{\{1\}}$, $\overrightarrow{\{1,2\}}$. Dla $n \geqslant 4$ mamy dwa przypadki:

1. $k \geqslant 1$. W grafie jest krawędź $\{n,n-1\}$. Ciąg przejść zaczynamy od $\overrightarrow{\{1,2\}}$, $\overleftarrow{\{1\}}$, $\overrightarrow{\{n,n-1\}}$, $\overleftarrow{\{2\}}$. Po usunięciu z grafu krawędzi $\{1,2\}$ i $\{n,n-1\}$ otrzymujemy graf optymalny dla $n-2$ osób i z $k-1$ parującymi krawędziami.
2. $k=0$. W grafie jest krawędź $\{1,n\}$. Ciąg przejść zaczynamy od $\overrightarrow{\{1,n\}}$, $\overleftarrow{\{1\}}$. Po usunięciu z grafu krawędzi $\{1,n\}$ dostajemy optymalny graf dla $n-1$ osób i $k=0$. ■

W celu dokończenia dowodu twierdzenia 1 wystarczy sprawdzić, że całkowity czas przejścia turystów dany wzorem (7) dla grafu optymalnego z twierdzenia 4 wynosi S_k (2).

Rysunek 1: Graf optymalny dla $n=8$ i $k=2$

Przykład 2 Przykładowy graf będący optymalnym rozwiązaniem dla $n=8$ i $k=2$ przedstawiony jest na rysunku 1. Ciąg przejść skonstruowany zgodnie z dowodem lematu 5 jest następujący: $\overrightarrow{\{1,2\}}$, $\overleftarrow{\{1\}}$, $\overrightarrow{\{8,7\}}$, $\overleftarrow{\{2\}}$, $\overrightarrow{\{1,2\}}$, $\overleftarrow{\{1\}}$, $\overrightarrow{\{6,5\}}$, $\overleftarrow{\{2\}}$, $\overrightarrow{\{1,4\}}$, $\overleftarrow{\{1\}}$, $\overrightarrow{\{1,3\}}$, $\overleftarrow{\{1\}}$, $\overrightarrow{\{1,2\}}$.

Inne rozwiązania

Jeśli zauważymy tylko, że dowolny turysta może przechodzić tylko z 1 lub z sąsiadem otrzymamy wzór rekurencyjny. Niech c_i oznacza najkrótszy czas przejścia turystów 1, 2, …, i, wtedy zachodzi:

$$\begin{aligned} c_2 &= t_2, \\ c_3 &= t_1+t_2+t_3, \\ c_i &= \min\{c_{i-2}+t_1+2t_2+t_i, c_{i-1}+t_1+t_i\}, \quad \text{dla } i \geqslant 4. \end{aligned}$$

Wartości c_i wyliczamy dynamicznie. Taki algorytm działa w czasie $O(n)$ i pamięci $O(1)$.

Błędne rozwiązania

Jedno z błędnych rozwiązań polegało na tym, że każdego turystę parowało się z turystą 1. Odpowiada to $k = 0$ we wzorze (1).

W innych błędnych rozwiązaniach każdego turystę, oprócz 1, 2 i może 3, próbowano wysłać na drugi brzeg wraz z sąsiadem. Odpowiada to rozwiązaniu optymalnemu dla $k = \lfloor n/2 \rfloor - 1$.

Testy

Każdy z 10 testów, poza pierwszym, zawiera czasy zarówno mniejsze jak i większe od wartości granicznej $2t_2 - t_1$. Powodowało to, że takie rozwiązania jak opisane wyżej, dawały zbyt duży wynik.

nr testu	n	opis
1	4	prosty test poprawnościowy
2	10	czterej ostatni przechodzą podwójnie
3	19	któryś przechodzi pojedynczo, chociaż jego czas jest większy od wartości granicznej
4	32	po kilka osób z tym samym czasem, w szczególności z czasem granicznym
5	2001	dwaj z granicznym czasem, potem nieparzyście wielu
6	31415	losowy, mało osób poniżej granicznego czasu, dużo z równym i większym
7	50002	duże t_2
8	69999	małe t_2, tylko kilka osób z czasem mniejszym od granicznego
9	83001	nie ma w ogóle osób z czasem granicznym, wszystkie czasy mniejsze są takie same
10	100000	maksymalny test, siłą rzeczy ma dużo powtórzeń

Bartosz Walczak — Treść zadania, Opracowanie
Rafał Rusin — Program

Bramki

Dany jest układ złożony z n bramek. Bramki są ponumerowane od 0 do $n-1$. Każda bramka posiada pewną liczbę wejść i jedno wyjście. Wejścia i wyjścia mogą przyjmować stany 0, 1 lub $\frac{1}{2}$. Każde wejście jest połączone z dokładnie jednym wyjściem pewnej bramki. Stan wejścia jest równy stanowi wyjścia, z którym jest ono połączone. Każde wyjście może być połączone z dowolną liczbą wejść. Bramki o numerach 0 i 1 są specjalne — nie posiadają wejść i zawsze przyjmują określone stany na wyjściu: 0 dla bramki o numerze 0, 1 dla bramki o numerze 1.

Mówimy, że stan wyjścia bramki (krótko: stan bramki) jest **poprawny**, *jeżeli:*

a) jest równy 0 i bramka ma więcej wejść w stanie 0 niż w stanie 1;

b) jest równy $\frac{1}{2}$ i bramka ma tyle samo wejść w stanie 0 co w stanie 1;

c) jest równy 1 i bramka ma więcej wejść w stanie 1 niż w stanie 0;

d) dana bramka jest specjalna, tzn. ma numer 0 lub 1, i jej stan jest równy odpowiednio 0 lub 1.

Mówimy, że stan układu jest **poprawny**, *jeżeli stan każdej z bramek jest poprawny. Mówimy, że stan bramki jest* **zdeterminowany**, *jeżeli w każdym poprawnym stanie układu bramka ta przyjmuje ten sam stan.*

Zadanie

Napisz program, który:

- *Wczyta z wejścia opis układu bramek.*
- *Dla każdej bramki sprawdzi, czy jej stan jest zdeterminowany i jeżeli tak, to wyznaczy go.*
- *Wypisze wyznaczone stany bramek na wyjście.*

Wejście

Pierwszy wiersz standardowego wejścia zawiera liczbę bramek n, $2 \leqslant n \leqslant 10\,000$. Kolejne $n-2$ wierszy zawiera opisy połączeń bramek. Wiersz nr i opisuje połączenia łączące wyjścia bramek z wejściami bramki nr i. W wierszu tym znajduje się liczba k_i wejść bramki nr i, po której następuje k_i numerów bramek, $k_i \geqslant 1$. Są to numery bramek, których wyjścia są połączone z kolejnymi wejściami bramki nr i. Liczby w wierszach są pooddzielane pojedynczymi odstępami. Łączna liczba wszystkich wejść bramek nie przekracza $200\,000$.

Wyjście

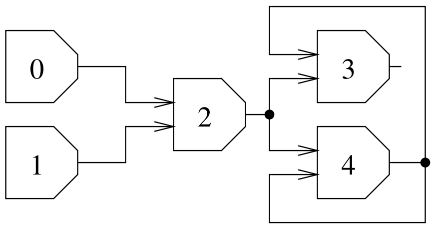
Twój program powinien wypisać na standardowe wyjście n wierszy. W zależności od stanu bramki numer $i-1$, i-ty wiersz powinien zawierać:

- `0` — *jeżeli stan bramki jest zdeterminowany i wynosi 0,*
- `1/2` — *jeżeli stan bramki jest zdeterminowany i wynosi $\frac{1}{2}$,*
- `1` — *jeżeli stan bramki jest zdeterminowany i wynosi 1,*
- `?` *(znak zapytania) — jeżeli stan bramki nie jest zdeterminowany.*

Przykład

Dla danych wejściowych:

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



poprawnym wynikiem jest:

```
0
1
1/2
?
?
```

Rozwiązanie

Aby uprościć opis, przyjmijmy, że w układzie nie ma bramek specjalnych, a niektóre wejścia mają po prostu ustalone stany (0 lub 1) i nie są połączone z wyjściami żadnych bramek.

Na bramki możemy patrzeć jak na przetworniki, które na podstawie stanów wejść obliczają stan wyjścia. Stan układu, który nie jest poprawny, możemy traktować jak stan, w którym jakaś bramka nie zdążyła jeszcze obliczyć poprawnego stanu wyjścia. W tym sensie układy w stanach niepoprawnych są niestabilne, a bramki dążą do ich stabilizacji.

Ogólny schemat rozwiązania można przedstawić następująco:

1. Ustaw stany wszystkich bramek na 0, po czym ustabilizuj układ.
2. Ustaw stany wszystkich bramek na 1, po czym ustabilizuj układ.
3. Stan bramki jest zdeterminowany wtedy i tylko wtedy, gdy jest taki sam po ustabilizowaniu w krokach 1 i 2.

Skąd wiadomo, że układ da się ustabilizować, czyli że istnieje stan poprawny? Wynika to z monotoniczności bramek. Ale na razie to tylko intuicja. W dalszej części postaramy się nieco sformalizować powyższe rozważania.

Stany układu będziemy oznaczać wielkimi, a bramki małymi literami. Stan bramki a, gdy układ jest w stanie S, będziemy oznaczać przez $s(a,S)$. *Wartością* bramki a w stanie układu S (oznaczenie $v(a,S)$) nazywamy liczbę:

- 0, gdy a ma więcej wejść w stanie 0 niż w stanie 1;
- $\frac{1}{2}$, gdy a ma tyle samo wejść w stanie 0, co w stanie 1;
- 1, gdy a ma więcej wejść w stanie 1 niż w stanie 0.

Stan bramki a jest poprawny wtedy i tylko wtedy, gdy $s(a,S) = v(a,S)$. Monotoniczność bramki oznacza, że jeżeli stan wejścia się zwiększa, to wartość bramki się nie zmniejsza, a jeżeli stan wejścia się zmniejsza, to wartość bramki się nie zwiększa.

Mówimy, że stan układu S jest *niski*, gdy dla każdej bramki a zachodzi $s(a,S) \leqslant v(a,S)$. Analogicznie, mówimy, że stan układu S jest *wysoki*, gdy dla każdej bramki a zachodzi $s(a,S) \geqslant v(a,S)$. Wprowadzamy również porównywanie stanów układu: jeżeli S_1 i S_2 są dwoma stanami układu, to piszemy $S_1 \leqslant S_2$, gdy dla każdej bramki a zachodzi $s(a,S_1) \leqslant s(a,S_2)$. Oczywiście nie każde dwa stany układu można ze sobą porównać.

Zdefiniujemy teraz operację poprawiania stanu bramki. Jeżeli stan bramki a w stanie układu S jest niepoprawny, to *poprawieniem* jej stanu nazywamy:

- zwiększenie go o $\frac{1}{2}$, gdy $s(a,S) < v(a,S)$;
- zmniejszenie go o $\frac{1}{2}$, gdy $s(a,S) > v(a,S)$.

Twierdzenie 1 *Dla każdego niskiego stanu układu S istnieje stan poprawny T taki, że jeżeli $S \leqslant P$, to $T \leqslant P$ dla dowolnego stanu poprawnego P.*

Dowód Niech S będzie dowolnym niskim stanem układu. Jeżeli S jest stanem poprawnym, to wystarczy przyjąć $T = S$ i teza jest oczywista. W przeciwnym przypadku istnieje bramka a, taka że $s(a,S) < v(a,S)$. Oznaczmy przez S' stan układu powstały z S przez poprawienie bramki a (stany pozostałych bramek się nie zmieniają). W wyniku operacji poprawiania wartości bramek mogą co najwyżej wzrosnąć. Zatem stan S' również jest stanem niskim.

Niech P będzie dowolnym takim stanem poprawnym układu, że $S \leqslant P$. Z monotoniczności bramek wynika, że dla dowolnej bramki b zachodzi $v(b,S) \leqslant v(b,P)$. W szczególności, dla poprawianej bramki a mamy $s(a,S) < v(a,S) \leqslant v(a,P) = s(a,P)$, skąd $s(a,S') \leqslant s(a,P)$. Stany pozostałych bramek się nie zmieniają, więc własność $s(b,S') \leqslant s(b,P)$ zostaje zachowana dla wszystkich bramek b, co oznacza, że $S' \leqslant P$.

Powyższą operację poprawiania bramek możemy powtarzać, otrzymując kolejno stany S', $(S')'$ itd., dopóki nie dostaniemy stanu poprawnego. Ponieważ w każdym kroku zwiększa się stan co najmniej jednej bramki oraz stan żadnej bramki się nie zmniejsza, takie postępowanie musi się skończyć, czyli otrzymamy jakiś stan poprawny T. Z zasady indukcji wynika, że otrzymany stan T spełnia tezę. ■

Twierdzenie 2 *Dla każdego wysokiego stanu układu S istnieje stan poprawny T, taki że jeżeli $P \leqslant S$, to $P \leqslant T$ dla dowolnego stanu poprawnego P.*

Dowód Analogiczny do dowodu twierdzenia 1. ■

Oznaczmy przez $\bar{0}$ stan układu, w którym wszystkie bramki są w stanie 0, a przez $\bar{1}$ stan układu, w którym wszystkie bramki są w stanie 1. Oczywiście $\bar{0}$ jest stanem niskim, a $\bar{1}$ jest stanem wysokim. Ponadto dla dowolnego stanu P (w szczególności, dla dowolnego stanu poprawnego) zachodzi $\bar{0} \leqslant P \leqslant \bar{1}$. Zatem na mocy twierdzeń 1 i 2 istnieją stany poprawne L i H, takie że $L \leqslant P \leqslant H$, dla dowolnego stanu poprawnego P. Jeżeli $s(a,L) < s(a,H)$, to stan bramki a nie jest zdeterminowany, gdyż jest różny w stanach poprawnych L i H. Natomiast jeżeli $s(a,L) = s(a,H)$, to dla dowolnego stanu poprawnego P zachodzi $s(a,L) = s(a,P) = s(a,H)$, więc stan bramki a jest zdeterminowany.

Rozwiązanie wzorcowe

Rozwiązanie wzorcowe znajduje stan L, implementując metodę przedstawioną w dowodzie twierdzenia 1 przy pomocy przeszukiwania wszerz:

1. Ustaw stany wszystkich bramek na 0.
2. Utwórz kolejkę Q bramek, których stany są niepoprawne (za niskie).
3. Dopóki kolejka Q nie jest pusta, powtarzaj następujące operacje:
   (a) Pobierz z kolejki Q dowolną bramkę a.
   (b) Popraw stan bramki a.
   (c) Wstaw do kolejki Q wszystkie bramki, których stany były poprawne, a w wyniku operacji poprawienia stanu a stały się niepoprawne.
   (d) Usuń z kolejki Q bramkę a, jeżeli po poprawieniu jej stan stał się poprawny (tylko stan bramki a mógł się stać poprawny).

Dla każdej bramki a pamiętana jest lista wszystkich bramek, których wejścia są połączone z wyjściem bramki a. Tylko te bramki mogą zmienić swoją wartość podczas poprawiania bramki a, a więc tylko je wystarczy rozpatrywać w punkcie (c) pętli. Cały układ jest więc reprezentowany przez graf, którego wierzchołki odpowiadają bramkom, a krawędzie połączeniom skierowanym od wyjścia do wejścia. Ponadto dla każdej bramki jest pamiętana liczba wejść w każdym z trzech dopuszczalnych stanów oraz bieżący stan bramki, co pozwala określić relację pomiędzy stanem bramki, a jej wartością w czasie jednostkowym. Stan każdej bramki jest poprawiany co najwyżej dwa razy. Zatem czas wykonania tej fazy rozwiązania wynosi $O(n+m)$, gdzie m jest łączną liczbą wszystkich połączeń między bramkami.

Analogicznie, w czasie $O(n+m)$ znajdowany jest stan H. Sprawdzenie dla każdej bramki a, czy $s(a,L) = s(a,H)$ oczywiście również można wykonać w czasie liniowym.

Powyższe rozwiązanie zostało zaimplementowane w `bra.cpp`.

Testy

Rozwiązania były oceniane na następującym zestawie testów:

- testy poprawnościowe: 1a, 1b, 2, 3a, 3b, 4;

- testy wydajnościowe: 5–11.

Testy 1a i 1b oraz 3a i 3b były zgrupowane w pary.

nr testu	n	m	opis
1a	2	0	przypadek szczególny $n = 2$
1b	7	9	
2	9	9	
3a	9	16	tylko stany niezdeterminowane[a]
3b	6	8	
4	27	32	
5	10000	10021	tylko stany zdeterminowane
6	1965	184365	prawie tylko stany zdeterminowane
7	1965	184365	prawie tylko stany zdeterminowane
8	9625	199769	prawie tylko stany zdeterminowane
9	9994	199904	tylko stany niezdeterminowane[a]
10	10000	199990	prawie tylko stany niezdeterminowane
11	10000	199990	prawie tylko stany niezdeterminowane

[a] Nie dotyczy bramek specjalnych

Paweł Parys **Arkadiusz Paterek**
Treść zadania, Opracowanie Program

Jaskinia

W Bajtocji znajduje się jaskinia, która składa się z n komór oraz z łączących je korytarzy. Korytarze są tak ułożone, że między każdymi dwiema komorami można przejść tylko w jeden sposób. W jednej z komór Jaś ukrył skarb, ale nie chce powiedzieć w której. Małgosia koniecznie chce się tego dowiedzieć. Pyta więc Jasia kolejno o różne komory. Jeśli Małgosia trafi, to Jaś mówi jej o tym, a jeśli nie trafi, to mówi jej, w którą stronę trzeba iść z danej komory w kierunku skarbu.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis jaskini,*
- *obliczy minimalną liczbę pytań, które w pesymistycznym przypadku musi zadać Małgosia, żeby wiedzieć, w której komorze znajduje się skarb,*
- *wypisze obliczony wynik na standardowe wyjście.*

Wejście

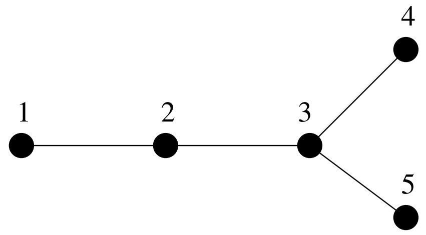
W pierwszym wierszu standardowego wejścia znajduje się jedna dodatnia liczba całkowita n, $1 \leqslant n \leqslant 50\,000$. Jest to liczba komór w jaskini. Komory jaskini są ponumerowane od 1 do n. W kolejnych $n-1$ wierszach opisane są korytarze łączące komory, po jednym w wierszu. W każdym z tych wierszy znajduje się para różnych dodatnich liczb całkowitych a i b ($1 \leqslant a,b \leqslant n$), oddzielonych pojedynczym odstępem. Para taka oznacza, że komory a i b są połączone korytarzem.

Wyjście

Na standardowe wyjście powinna być wypisana jedna liczba całkowita, minimalna liczba pytań, jakie musi zadać Małgosia w pesymistycznym przypadku (tzn. zakładamy, że Małgosia zadaje pytania najlepiej jak można, ale skarb jest umieszczony w komorze, która wymaga zadania największej liczby pytań).

Przykład

Dla danych wejściowych:
```
5
1 2
2 3
4 3
5 3
```



poprawnym wynikiem jest:
```
2
```

Rozwiązanie

Plan jaskini jest drzewem. Pytając o dany wierzchołek, dzielimy to drzewo na pewną liczbę części, które powstają przez usunięcie tego wierzchołka. W odpowiedzi dostajemy jedną z tych części i teraz w niej musimy znaleźć skarb. Możliwe są jednak różne odpowiedzi, więc trzeba rozważyć każdą z nich. Można więc spojrzeć na to tak: W drzewie usuwamy wybrany jeden wierzchołek. W następnym kroku z każdej z powstałych części wybieramy znowu po jednym wierzchołku i je usuwamy. Tak postępujemy, aż uzyskane części będą jednowierzchołkowe.

W zadaniu chodzi o zminimalizowanie liczby tych kroków. Chcemy więc, aby powstające w każdym kroku części (największa z nich) były jak najmniejsze.

W przypadku, gdy w jaskini nie ma rozgałęzień, możemy po prostu zastosować wyszukiwanie binarne — pytamy zawsze o wierzchołek znajdujący się w samym środku jaskini. W ogólnej sytuacji też widać, że trzeba pytać mniej więcej w środku jaskini. Nie wiadomo jednak, w jakim sensie ma to być środek. Widać też, że bardziej opłaca się wybierać wierzchołki, które mają więcej sąsiadów, wtedy podzielimy drzewa na więcej mniejszych części. Można próbować w wierzchołku, z którego odległość do pozostałych wierzchołków jest minimalna. Albo tak, aby powstałe części miały jak najmniej wierzchołków. Można też strzelać, że wynik jest logarytmem z liczby wierzchołków lub ze średnicy drzewa (tzn. odległości między najdalszymi wierzchołkami). Łatwo jednak sprawdzić, że żadna z tych heurystyk nie jest poprawna.

Idea rozwiązania

Najpierw w dowolny sposób wybierzmy sobie korzeń naszego drzewa. Chodzi tylko o to, aby ustalić kolejność, w której będziemy przeglądać wierzchołki przy poszukiwaniu optymalnego rozwiązania.

Rozwiązanie opiera się na następującym pomyśle: Będziemy rozważać nasze drzewo, poczynając od liści i idąc w stronę korzenia. Najpierw wywołamy obliczenia dla poddrzew danego wierzchołka. Następnie na podstawie wyników dla poddrzew obliczymy wynik dla drzewa mającego korzeń w tym wierzchołku. A więc na podstawie wyników dla mniejszych drzew liczymy wynik dla większego.

Znamy więc minimalną liczbę pytań w poddrzewie. Chcemy jednak, aby (w ramach ustalonej liczby pytań) pytania zadawać jak najwyżej. Na przykład im wyżej zadamy pierwsze pytanie, tym mniejsza będzie część znajdująca się nad wierzchołkiem o który pytamy, więc tym więcej wierzchołków można jeszcze dołożyć z góry do drzewa, aby liczba potrzebnych pytań nie zmieniła się.

Można się przekonać, że znając tylko liczbę pytań potrzebną w poddrzewach, nie jesteśmy w stanie obliczyć liczby pytań w większym drzewie. Trzeba również wiedzieć coś o tym, jak duża jest część nad wierzchołkiem, w którym zadajemy pierwsze pytanie. W szczególności chcielibyśmy wiedzieć, ile pytań potrzeba na zbadanie tej górnej części, ale też coś na temat części ponad pierwszym pytaniem w tej części, itd.

Rozwiązanie wzorcowe

Sposób na zadawanie pytań będziemy zapisywać za pomocą funkcji f. Każdemu wierzchołkowi w będziemy chcieli przypisać nieujemną liczbę $f(w)$. Chcemy przy tym zachować następujący warunek $(*)$, że jeśli dla dwóch wierzchołków u i v jest $f(u) = f(v)$, to na drodze z u do v znajduje się wierzchołek w taki, że $f(w) > f(u)$. Intuicyjnie $f(w)$ oznacza numer pytania, licząc od końca, które będziemy zadawać w wierzchołku w.

Dla zadanej funkcji f pytania będziemy zadawać w następujący sposób: Pytamy o wierzchołek w, dla którego $f(w)$ jest największe. W odpowiedzi otrzymujemy pewną część drzewa. Znowu pytamy o wierzchołek w, dla którego $f(w)$ jest największa w tej części drzewa, itd. Zawsze w otrzymanej części drzewa mamy dokładnie jeden wierzchołek o największym $f(w)$ (wynika to z warunku $(*)$). Albo w którymś momencie trafimy, albo na koniec otrzymamy drzewo składające się z jednego wierzchołka i będziemy wiedzieli, że tam jest skarb. W ten sposób zadajemy co najwyżej $K = \max_w f(w)$ pytań. Chcemy znaleźć taką funkcję f, dla której K jest najmniejsze.

Jak już było powiedziane, najlepszej funkcji $f(w)$ będziemy szukać w drzewie ukorzenionym. A więc wybierzmy w dowolny sposób korzeń. Niech $T(w)$ oznacza poddrzewo o korzeniu w w. Oprócz funkcji $f(w)$ będziemy obliczać zbiory $S(w)$. Zbiór $S(w)$ jest zbiorem wartości funkcji, które „widać" w drzewie $T(w)$, patrząc od ojca w. Oznacza to, że liczba a należy do $S(w)$ wtedy i tylko wtedy, gdy w $T(w)$ jest wierzchołek u taki, że $f(u) = a$ i na drodze z u do ojca w nie występuje liczba większa niż $f(u)$. Inaczej mówiąc: Liczba pytań, które trzeba zadać w drzewie $T(w)$ należy do $S(w)$. Liczba pytań, które trzeba zadać w części ponad wierzchołkiem, w którym zadamy pierwsze pytanie, również należy do $S(w)$, itd.

Postępujemy w następujący sposób: Przed policzeniem f i S dla w obliczamy f i S dla wszystkich synów w. Niech R oznacza $\bigcup_u S(u)$, gdzie u to wszyscy synowie w. Niech m będzie największą liczbą taką, że $m \in S(u)$ dla więcej niż jednego u. (Jeśli takich liczb nie ma, to niech $m = -1$.) Jako $f(w)$ bierzemy najmniejszą liczbę większą od m i nie należącą do R. Zauważmy, że $S(w) = R \cup \{f(w)\} \setminus \{0, 1, \ldots, f(w)-1\}$. Łatwo sprawdzić, że powstała funkcja f spełnia warunek $(*)$. Przyglądając się dokładniej temu warunkowi widzimy, że tak zdefiniowane $f(w)$ jest najmniejszą wartością, przy której $(*)$ jest spełniony (dla już ustalonych wartości f w poddrzewach).

Trzeba teraz udowodnić, że otrzymana w ten sposób funkcja jest najlepsza. Będziemy to robić indukcyjnie ze względu na minimalną liczbę pytań K. Teza indukcyjna jest taka, że jeśli powyższy algorytm dla pewnego drzewa zwrócił K, to rzeczywiście w tym drzewie potrzeba co najmniej K pytań. Dla $K = 0$ jest to oczywiste. Dla $K = 1$ nasze drzewo składa się z co najmniej dwóch wierzchołków, czyli nie można znaleźć skarbu w 0 ruchach. Teraz chcemy udowodnić poprawność powyższego rozwiązania dla pewnego K, wiedząc, że dla wszystkich mniejszych K jest ono prawdziwe. Przyjmijmy, że dla pewnego drzewa T można znaleźć skarb w $K-1$ ruchach, zadając pierwsze pytanie w wierzchołku u. Niech v_1 będzie wierzchołkiem, dla którego $f(v_1) = K$. W drzewie istnieje również pewien potomek v_2 (co najmniej jeden) wierzchołka v_1, dla którego $f(v_2) = K-1$. Niech T_1 oznacza drzewo $T(v_1) - T(v_2)$ (czyli drzewo zawierające wierzchołki $T(v_1)$ za wyjątkiem wierzchołków $T(v_2)$). Rozważmy dwa przypadki:

- u należy do $T(v_2)$ — Po zadaniu pytania w u skarb może znajdować się w tej części drzewa T, która zawiera v_1. Ma więc istnieć sposób na znalezienie tam skarbu w $K-2$ pytaniach. Ta część zawiera drzewo T_1, czyli w drzewie T_1 również można znaleźć skarb w $K-2$ pytaniach. Popatrzmy jednak, co się stanie z obliczoną przez program funkcją f, gdy z drzewa T usuniemy drzewo $T(v_2)$. Ponieważ $S(v_2) = \{K-1\}$, to wartość $f(v_1)$ zmniejszy się najwyżej o 1, do wartości $K-1$. Zatem na mocy założenia indukcyjnego na drzewo T_1 potrzeba $K-1$ pytań. Sprzeczność.

- u nie należy do $T(v_2)$ — Po zadaniu pytania w u skarb może znajdować się w tej części drzewa T, która zawiera v_2. Ma więc istnieć sposób na znalezienie tam skarbu w $K-2$ pytaniach. Ta część zawiera drzewo $T(v_2)$, czyli w drzewie $T(v_2)$ też można znaleźć skarb w $K-2$ pytaniach. Ale ponieważ $f(v_2) = K-1$, to na mocy założenia indukcyjnego na drzewo $T(v_2)$ potrzeba co najmniej $K-1$ pytań. Sprzeczność.

W obu przypadkach dochodzimy do sprzeczności, nie można znaleźć skarbu w drzewie T w mniej niż K pytaniach. Krok indukcyjny jest prawdziwy. To kończy dowód.

Podany algorytm działa w czasie $O(n \log n)$.

Rozwiązanie w czasie liniowym

Powyższe rozwiązanie można łatwo zmodyfikować tak, aby działało w czasie liniowym. Wystarczy zbiory $S(w)$ reprezentować za pomocą kolejnych bitów liczby. Największą liczbą znajdującą się w $S(w)$ może być wynik K, który jest nie większy niż logarytm z liczby wierzchołków drzewa. Zatem do reprezentacji tych zbiorów wystarczą liczby długości takiej samej, jak do reprezentacji numerów wierzchołków, czyli uwzględniając ograniczenia zadania, od 0 do $2^{17}-1$. Pozostaje jeszcze sprawa wykonywania operacji na tych zbiorach. Chcielibyśmy to robić w czasie stałym.

- Sumę zbiorów można policzyć wykonując OR na reprezentacjach.

- Chcemy też znajdować najmniejszą lub największą wartość należącą do danego zbioru. Nie ma instrukcji procesora obliczającej tę wartość w czasie stałym. Możemy jednak obliczyć odpowiedzi na początku — dla każdej reprezentacji zbioru będziemy pamiętać wynik. Korzystamy z tego, że reprezentacja zbioru jest niewielką liczbą — różnych reprezentacji jest mniej więcej tyle, co wierzchołków drzewa.

- Wstawianie danej wartości do zbioru i usuwanie ze zbioru wartości mniejszych niż dana można zrealizować za pomocą AND, OR i przesunięć bitowych.

- Znajdowanie zbioru zawierającego liczby należące do co najmniej dwóch spośród danych zbiorów S_1, S_2, ..., S_n również można zrealizować za pomocą AND i OR. Liczymy najpierw $A_0 = \phi$, $A_k = S_k \cup A_{k-1}$, dla $k = 1, \ldots, n$. Zbiór A_k zawiera liczby należące do S_1, S_2, ..., S_k. Następnie liczymy $B_0 = \phi$, $B_k = (S_k \cap A_{k-1}) \cup B_{k-1}$, dla $k = 1, \ldots, n$. Zbiór B_k zawiera liczby należące do co najmniej dwóch zbiorów spośród $S_1, S_2, \ldots, S_k$, a zatem B_n jest szukanym zbiorem.

Testy

Zadanie testowane było na zestawie 13 danych testowych. Testy były generowane w dużym stopniu losowo, przy zadanych różnych prawdopodobieństwach otrzymania wierzchołków danego stopnia. Rozmiary testów zawiera poniższa tabela.

nr testu	n	wynik
1	17	3
2	16	3
3	32	3
4	100	5
5	1000	8
6	5000	11
7	10000	10
8	22000	12
9	24000	12
10	30000	11
11	40000	13
12	40063	7
13	50000	11

Piotr Stańczyk — Treść zadania, Opracowanie
Paweł Wolff — Program

Przeprawa

Drużyna Bajtołazów wybrała się na wycieczkę w Bajtogóry. Niestety Bajtołazi spowodowali lawinę i muszą przed nią uciekać. Na ich drodze znajduje się przepaść i stary most linowy. Muszą jak najszybciej przeprawić się przez most na drugą stronę. Bajtołazi są bardzo zgrani i postanowili, że albo wszyscy się uratują, albo nikt.

Most jest stary i zniszczony, więc nie wytrzyma zbyt dużego obciążenia. Łączna waga Bajtołazów znajdujących się równocześnie na moście nie może być większa od wytrzymałości mostu. Ponadto jest to most linowy. Bajtołazi muszą więc przechodzić przez niego grupami. Kolejna grupa może wejść na most dopiero wtedy, gdy poprzednia go opuści.

Dla każdego Bajtołaza wiadomo, ile czasu zajmie mu przeprawa przez most. Czas przeprawy przez most grupy Bajtołazów jest równy czasowi potrzebnemu na przeprawę przez most najwolniejszego członka grupy. Łączny czas przeprawy wszystkich Bajtołazów to suma czasów przeprawy wszystkich grup. Oczywiście zależy on od tego, jak się podzielą na grupy, przechodząc przez most.

Pomóż Bajtołazom uratować się! Oblicz, jaki jest minimalny czas przeprawy przez most wszystkich Bajtołazów.

Zadanie

Napisz program, który:

- *wczyta z wejścia opis mostu oraz Bajtołazów,*
- *wyznaczy najkrótszy czas przeprawy wszystkich Bajtołazów przez most,*
- *wypisze wyznaczony czas na wyjście.*

Wejście

Pierwsza linia standardowego wejścia zawiera dwie liczby całkowite oddzielone pojedynczym odstępem: w — określającą maksymalne dopuszczalne obciążenie mostu ($100 \leqslant w \leqslant 400$) oraz n — równą liczbie Bajtołazów ($1 \leqslant n \leqslant 16$). W kolejnych n wierszach znajdują się po dwie liczby całkowite oddzielone pojedynczym odstępem, opisujące kolejnych Bajtołazów: t — czas potrzebny na pokonanie mostu przez Bajtołaza ($1 \leqslant t \leqslant 50$) oraz w — waga Bajtołaza ($10 \leqslant w \leqslant 100$).

Wyjście

W pierwszym i jedynym wierszu standardowego wyjścia Twój program powinien wypisać jedną liczbę naturalną oznaczającą minimalny czas przeprawy wszystkich Bajtołazów przez most.

Przykład

Dla danych wejściowych:
```
100 3
24 60
10 40
18 50
```

poprawnym wynikiem jest:
```
42
```

Rozwiązanie

Definicja 1 *Podział zbioru* — rodzina niepustych, parami rozłącznych zbiorów, których suma teoriomnogościowa daje dzielony zbiór.

Definicja 2 *Rozbicie zbioru* — podział zbioru składający się z dwóch podzbiorów.

Definicja 3 *Podzbiór maksymalny* — maksymalny podzbiór (w sensie zawierania) zbioru Bajtołazów nieprzeciążający mostu.

W zadaniu Przeprawa mamy do czynienia z problemem optymalizacyjnym. Prawdopodobnie nie istnieje algorytm rozwiązujący podane zadanie w czasie wielomianowym. W związku z tym należy skoncentrować się na konstruowaniu algorytmu przeszukującego zbiór możliwych rozwiązań w poszukiwaniu rozwiązania optymalnego. Z uwagi na fakt, iż możliwych podziałów zbioru n Bajtołazów jest B_n (B_n to n-ta liczb Bella — więcej informacji można znaleźć np. w „Kombinatoryce dla programistów" Witolda Lipskiego [23]), to rozpatrywanie wszystkich możliwych przypadków nie jest najlepszym rozwiązaniem. Pierwszych 15 liczb Bella zawarto w tabeli 1.

n	B_n
0	1
1	1
2	2
3	5
4	15
5	52
6	203
7	877
8	4140
9	21147
10	115975
11	678570
12	4213597
13	27644437
14	190899322
15	1382958545

Tabela 1: Kilka pierwszych liczb Bella

Rozwiązanie wzorcowe

Rozwiązanie wzorcowe nie generuje wszystkich możliwych podziałów zbioru Bajtołazów, lecz realizuje następującą strategię: dla danego zbioru Bajtołazów, program sprawdza, czy mogą oni wszyscy zostać przeprawieni przez most za jednym razem. Jeśli tak, to czas przeprawienia takiej grupy jest równy czasowi przeprawy najwolniejszego członka grupy. W przeciwnym przypadku dokonujemy serii rozbić na dwa rozłączne zbiory, a następnie rozwiązujemy podproblemy niezależnie. Po obliczeniu wyników częściowych musimy je scalić (czyli dodać do siebie czasy przeprawy obu podgrup), a następnie wybrać najlepszy czas spośród wszystkich rozpatrywanych rozbić.

W takim algorytmie wiele zbiorów Bajtołazów jest analizowanych wielokrotnie, więc dobrym pomysłem jest zastosowanie spamiętywania obliczonych już wyników. W ten sposób mamy do przeanalizowania 2^n różnych podzbiorów, co przy optymalnej implementacji generowania rozbić zbioru przekłada się na algorytm działający w czasie $O(3^n)$.

Rozwiązanie wzorcowe jest dodatkowo przyspieszone poprzez zastosowanie pewnych obserwacji, które pozwalają na zmniejszenie ilości rozpatrywanych podproblemów:

- Generowanie rozbicia zbioru Bajtołazów na dwa rozłączne podzbiory można dokonywać w taki sposób, że pierwszy generowany podzbiór składa się z maksymalnej grupy Bajtołazów nie przeciążającej mostu (grupa taka od razu może zostać przepuszczona przez most). Druga grupa składa się z reszty Bajtołazów, dla której wykonujemy kolejne rozbicia.

- Liczba generowanych rozbić może zostać dodatkowo zmniejszona poprzez dołączanie najwolniejszego Bajtołaza na stałe do grupy maksymalnej. Poprawność takiej operacji wynika z faktu, że dla dowolnego zbioru Bajtołazów, zawsze istnieje podział prowadzący do rozwiązania optymalnego, w którym najwolniejszy Bajtołaz znajduje się w grupie maksymalnej. Chciałbym w tym miejscu zauważyć, że dołączanie dowolnego Bajtołaza do grupy maksymalnej nie prowadzi do algorytmu poprawnego.

Lemat 1 (O najwolniejszym Bajtołazie) *Dla każdego zbioru Bajtołazów, istnieje podział prowadzący do rozwiązania optymalnego, w którym najwolniejszy Bajtołaz znajduje się w grupie maksymalnej.*

Dowód Załóżmy odwrotnie, że istnieje taki zbiór Bajtołazów, dla którego nie istnieje opisany w lemacie podział. Wybierzmy dla niego dowolny podział, prowadzący do rozwiązania optymalnego i oznaczmy przez x podzbiór zawierający najwolniejszego Bajtołaza. Z założenia wiemy, że x nie jest maksymalny, więc istnieje pewna grupa Bajtołazów, których przeniesienie do x robi z niego podzbiór maksymalny. Najwolniejszy Bajtołaz znajduje się w x, zatem czas przeprawy rozszerzonego zbioru x nie pogorszy się, więc nowo uzyskane rozwiązanie jest optymalne. Doszliśmy do sprzeczności z założeniem, zatem dowodzony lemat jest prawdziwy. ∎

Pesymistyczna złożoność rozwiązania wzorcowego wynosi $O(3^n)$. Zastosowane optymalizacje nie zmieniają złożoności asymptotycznej, jednak w znacznym stopniu zmniejszają pracę wykonywaną przez program. Zaimplementowane rozwiązanie alternatywne o tej samej złożoności co rozwiązanie wzorcowe, lecz nie uwzględniające powyższych obserwacji, na dużych testach działało ponad 120 razy wolniej.

Rozwiązania nieoptymalne

Jednym z nieoptymalnych rozwiązań jest algorytm, który nie uwzględnia usprawnienia polegającego na rozpatrywaniu podziałów grupy Bajtołazów na grupy maksymalne. Podczas pomiarów na dużych testach takie rozwiązanie było około 10 razy wolniejsze od wzorcowego.

Kolejnym rozwiązaniem nieoptymalnym jest program, który nie dokonuje spamiętywania wyników podproblemów już obliczonych. Podczas testów jego osiągi czasowe były 5–10 razy gorsze od rozwiązania wzorcowego.

Kolejne rozwiązanie nie uwzględnia żadnych optymalizacji — nie tylko nie rozpatruje tylko grup maksymalnych, ale również nie spamiętuje wyników dla podproblemów. Taki algorytm okazuje się niezmiernie wolny w porównaniu z poprzednimi wersjami.

Inne możliwe nieoptymalne rozwiązania to stosowanie jedynie metody spamiętywania lub generowanie wszystkich możliwych podziałów zbioru Bajtołazów, które nie przeciążają mostu.

Rozwiązania niepoprawne

Podczas rozwiązywania tego zadania mogą nasuwać się pewne naturalne, lecz niepoprawne pomysły jego rozwiązania. Programy tego typu można podzielić na dwa rodzaje: algorytmy zachłanne oraz różnego rodzaju heurystyki. Oto przykłady:

- Algorytm zachłanny, który wydziela ze zbioru Bajtołazów kolejno przeprawiane grupy w następujący sposób — dla każdego nieprzeprawionego Bajtołaza (w kolejności od najwolniejszego do najszybszego) wykonuj: jeśli nie przeciąży on tworzonej grupy, to dołącz go do grupy.
- Jeszcze bardziej „naiwna" strategia zachłanna — dla wszystkich Bajtołazów (w kolejności od najwolniejszego do najszybszego) wykonuj: jeśli nie przeciąży on aktualnie tworzonej grupy, to dołącz go do tej grupy; w przeciwnym przypadku przepraw aktualną grupę i zacznij tworzyć nową, dołączając do niej przetwarzanego Bajtołaza.
- Heurystyka probabilistyczna, która generuje pewną liczbę podziałów i wybiera najlepszy z nich.

Testy

Zadanie było sprawdzane na 15 testach. Podczas ich układania duży nacisk położono na to, aby rozwiązania niepoprawne nie uzyskały zbyt wielu punktów. Testy 1–7 to testy poprawnościowe, a 8–15 wydajnościowe. Poniżej przedstawiono rozmiary poszczególnych testów i odpowiadające im wyniki.

nr testu	n	w
1	4	109
2	4	100
3	5	140
4	5	150
5	6	200
6	7	250
7	8	394
8	12	129
9	13	200
10	12	297
11	12	150
12	12	129
13	15	297
14	16	140
15	16	140

Bartosz Walczak
Treść zadania, Opracowanie

Jakub Pawlewicz
Program

Turniej

Światowa Federacja Gry X organizuje turniej programów grających w tę grę. Startuje w nim n programów ponumerowanych od 1 do n. Zasady rozgrywania turnieju są następujące: dopóki w turnieju pozostaje więcej niż jeden program, losowane są dwa różne spośród nich, które rozgrywają partię gry X. Przegrany (w grze X nie ma remisów) odpada z turnieju, po czym cała procedura jest powtarzana. Program, który pozostanie w turnieju jako jedyny po rozegraniu wszystkich $n-1$ gier, zostaje zwycięzcą.

Federacja dysponuje tabelą wyników poprzednich turniejów. Wiadomo, że programy grają deterministycznie (tzn. w powtarzalny sposób) i tak samo jak w poprzednich turniejach. Zatem jeżeli pewne dwa programy już kiedyś ze sobą grały, to na pewno ich kolejna gra zakończy się tak samo. Jeśli jednak dwa programy jeszcze nigdy nie walczyły ze sobą, rezultatu rozgrywki nie da się przewidzieć — oba mają szansę na wygraną. Federacja chciałaby poznać listę wszystkich tych programów, które mają szansę na wygranie turnieju.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia liczbę uczestniczących programów oraz tabelę wyników ich wcześniejszych gier,*
- *wyznaczy wszystkie programy, które mają szansę wygrać turniej,*
- *wypisze wynik na standardowe wyjście.*

Wejście

Pierwszy wiersz wejścia zawiera liczbę całkowitą n, $1 \leqslant n \leqslant 100\,000$. Kolejnych n wierszy zawiera tabelę wyników wcześniejszych gier: $i+1$-szy wiersz zawiera liczbę całkowitą k_i, $0 \leqslant k_i < n$, a następnie k_i numerów programów, różnych od i, podanych w kolejności rosnącej — są to numery programów, z którymi program nr i już kiedyś wygrał. Liczby w wierszach są pooddzielane pojedynczymi odstępami. Liczba wszystkich znanych wyników wcześniejszych gier nie przekracza $1\,000\,000$.

Wyjście

Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać liczbę w wszystkich programów, które mają szansę wygrać turniej, a następnie w liczb będących numerami tych programów, podanymi w kolejności rosnącej. Liczby w wierszu powinny być pooddzielane pojedynczymi odstępami.

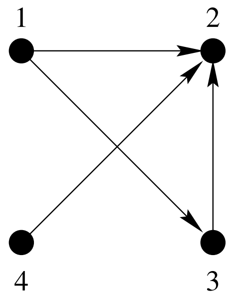
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
3 1 3 4
```



Rozwiązanie

Niech $V = \{1,2,\ldots,n\}$ oznacza zbiór wszystkich programów (każdy program jest reprezentowany przez swój numer). Przez $Win(v)$ oznaczamy zbiór wszystkich programów, z którymi program v na pewno wygra, czyli z którymi już wygrywał we wcześniejszych grach. Piszemy $u \rightsquigarrow v$, gdy $u \neq v$ i $u \notin Win(v)$, czyli gdy u może wygrać partię z v. Zatem jeśli tylko $u \neq v$, to $u \rightsquigarrow v$ lub $v \rightsquigarrow u$.

Każdy program, który może wygrać turniej, nazywamy *programem wygrywającym*. Pozostałe programy to *programy przegrywające*. Zadanie polega na znalezieniu zbioru wszystkich programów wygrywających.

Rozwiązanie zadania opiera się na następującej obserwacji: jeżeli v jest programem wygrywającym oraz $u \rightsquigarrow v$, to u także jest programem wygrywającym. Rozważmy bowiem jakiś przebieg turnieju, w którym v wygrał. Wtedy u musiał przegrać jakąś partię. Rozważmy teraz inny możliwy przebieg turnieju, który różni się od poprzedniego tylko tym, że nie jest rozgrywana partia, w której u przegrał. Wówczas po rozegraniu wszystkich pozostałych partii w turnieju pozostają v i u. Wtedy u może wygrać z v, wygrywając tym samym cały turniej.

Załóżmy, że W jest podzbiorem zbioru wszystkich programów wygrywających. Niech L będzie częścią wspólną zbiorów $Win(v)$ po wszystkich $v \in W$. Jeżeli $u \notin L$, to istnieje program $v \in W$, taki że $u \notin Win(v)$. Wtedy $u \rightsquigarrow v$ lub $u = v$, więc u jest programem wygrywającym. Zatem L jest nadzbiorem zbioru wszystkich programów przegrywających. Ponieważ zawsze $v \notin Win(v)$, zbiory W i L są rozłączne. Jeżeli $W \cup L = V$, to każdy program z W wygrywa z każdym programem spoza W. Wtedy już żaden program spoza W nie może wygrać turnieju, czyli W jest całym zbiorem programów wygrywających. Natomiast jeżeli istnieje $u \notin W \cup L$, to w szczególności $u \notin L$, więc u jest również programem wygrywającym. Wtedy możemy powiększyć zbiór W, dodając do niego u. Ta obserwacja prowadzi nas do następującego schematu algorytmu:

1. Znajdź dowolny program v, który może wygrać turniej. Przypisz $W := \{v\}$, $L := Win(v)$.

2. Dopóki $W \cup L \neq V$, powtarzaj następującą operację: weź dowolny $u \notin W \cup L$, a następnie przypisz $W := W \cup \{u\}$, $L := L \cap Win(u)$.

Po jego zakończeniu W jest szukanym zbiorem programów wygrywających.

Rozwiązanie wzorcowe

Rozwiązanie wzorcowe składa się z dwóch faz. Pierwsza faza wstępnie przetwarza zbiór programów V, dzięki czemu łatwiejsza staje się realizacja drugiej fazy. Druga faza znajduje zbiór programów wygrywających, mniej więcej implementując powyższy schemat.

Pierwsza faza polega na uporządkowaniu zbioru V w ciąg $(v_1, v_2, \dots, v_n)$, taki że $v_1 \rightsquigarrow v_2 \rightsquigarrow \dots \rightsquigarrow v_n$. Takie uporządkowanie zawsze istnieje, a można je wygenerować, stosując odpowiednio zmodyfikowany algorytm sortowania przez wstawianie:

1. Wybierz dowolny $v_1 \in V$.

2. Dla $i = 1, 2, \dots, n-1$ powtarzaj następujące operacje:

   (a) Wybierz dowolny $v \in V$, który jeszcze nie występuje w ciągu $(v_1, v_2, \dots, v_i)$.

   (b) Znajdź największą pozycję j w ciągu $(v_1, v_2, \dots, v_i)$, taką że $v_j \notin Win(v)$, czyli $v_j \rightsquigarrow v$ (jeśli takie j nie istnieje, to można przyjąć $j = 0$).

   (c) Wstaw v do ciągu $(v_1, v_2, \dots, v_i)$ bezpośrednio za v_j, czyli przypisz $(v_{j+1}, v_{j+2}, \dots, v_{i+1}) := (v, v_{j+1}, \dots, v_i)$.

Łatwo się przekonać, że powyższy algorytm rzeczywiście znajduje uporządkowanie $v_1 \rightsquigarrow v_2 \rightsquigarrow \dots \rightsquigarrow v_n$. Załóżmy bowiem, że przed wykonaniem kroku pętli zachodzi $v_1 \rightsquigarrow v_2 \rightsquigarrow \dots \rightsquigarrow v_i$. Wybieramy największe takie $j \leqslant i$, że $v_j \rightsquigarrow v$. Dlatego jeżeli $j < i$, to $v_{j+1} \not\rightsquigarrow v$, skąd $v \rightsquigarrow v_{j+1}$. Tym samym zachodzi $v_1 \rightsquigarrow v_2 \rightsquigarrow \dots \rightsquigarrow v_j \rightsquigarrow v \rightsquigarrow v_{j+1} \rightsquigarrow \dots \rightsquigarrow v_i$. Zatem po przypisaniu w kroku (c) otrzymujemy $v_1 \rightsquigarrow v_2 \rightsquigarrow \dots \rightsquigarrow v_{i+1}$.

Pewnego komentarza wymaga operacja wykonywana w kroku (b) pętli. Chcąc znaleźć odpowiednią pozycję j, możemy jakoś oznaczyć wszystkie te programy, które należą do $Win(v)$, a następnie szukać „od tyłu” największej takiej pozycji $j \leqslant i$, że v_j nie został oznaczony. Programy można oznaczać np. numerem iteracji i. Wtedy w dalszych iteracjach pętli takie oznaczenia nie będą już aktualne. Czas pojedynczego przeszukiwania z zastosowaniem tej metody ogranicza się przez $O(|Win(v)|)$, a więc czas wykonania całej pierwszej fazy wynosi $O(n+m)$, gdzie $m = \sum_{v=1}^{n} |Win(v)|$.

Mając dane uporządkowanie $v_1 \rightsquigarrow v_2 \rightsquigarrow \dots \rightsquigarrow v_n$, możemy przejść do drugiej fazy rozwiązania:

1. Przypisz $k := 1$.

2. Dla $i := 1, 2, \dots$, dopóki $i \leqslant k$, powtarzaj następujące operacje:

   (a) Znajdź największą pozycję j w ciągu $(v_1, v_2, \dots, v_n)$, taką że $v_j \notin Win(v_i)$, czyli $v_j \rightsquigarrow v_i$ lub $j = i$.

   (b) Przypisz $k := \max\{k, j\}$.

Po zakończeniu powyższego algorytmu szukanym zbiorem programów wygrywających jest $\{v_1, v_2, \dots, v_k\}$.

Załóżmy, że przed wykonaniem kroku pętli $L = \{v_{k+1}, v_{k+2}, \dots, v_n\}$ jest nadzbiorem zbioru wszystkich programów przegrywających. Ponieważ $i \leqslant k$, v_i jest programem wygrywającym. Jeżeli $v_j \rightsquigarrow v_i$, to v_j również jest wygrywający. Co więcej, mamy $v_1 \rightsquigarrow v_2 \rightsquigarrow \dots \rightsquigarrow v_j$, więc wszystkie programy $v_1, v_2, \dots, v_j$ są wygrywające. Zatem

$L \cap \{v_{j+1}, v_{j+2}, \ldots, v_n\}$ nadal jest nadzbiorem zbioru wszystkich programów przegrywających, stąd przypisanie w kroku (b). Pętla się kończy wykonywać, gdy $i > k$. Wtedy każdy program ze zbioru $V \setminus L$ wygrywa z każdym ze zbioru L, więc $V \setminus L = \{v_1, v_2, \ldots, v_k\}$ istotnie jest zbiorem wszystkich programów wygrywających.

Widzimy, że faza druga działa analogicznie do przedstawionego wcześniej schematu rozwiązania. Wystarczy przyjąć $W = \{v_1, v_2, \ldots, v_{i-1}\}$. Różnica tkwi w sposobie aktualizacji zbioru L, przez co może on być istotnie mniejszy od części wspólnej zbiorów $Win(v)$, po $v \in W$.

Przeszukiwanie w kroku (a) odbywa się analogicznie do tego w kroku (b) pierwszej fazy i działa w czasie $O(|Win(v_i)|)$. Zatem czas wykonania całej drugiej fazy wynosi $O(n+m)$. Tym samym całe rozwiązanie wzorcowe działa w czasie $O(n+m)$, czyli liniowym względem rozmiaru wejścia.

Powyższe rozwiązanie zostało zaimplementowane w `tur.c` oraz `tur1.pas`.

Rozwiązanie alternatywne

Inne optymalne rozwiązanie można uzyskać, bezpośrednio implementując podany na początku schemat algorytmu.

Zbiór L jest pamiętany na liście, w której programy występują w kolejności rosnących numerów. Programy, które nie występują w żadnym ze zbiorów W i L, przechowujemy w kolejce Q.

Na początku do kolejki Q wstawiamy programy (co najmniej jeden), o których wstępnie stwierdzamy, że są wygrywające. Takie programy można znaleźć np. poprzez symulację turnieju. Wszystkie programy, których nie wstawiliśmy do Q, umieszczamy w liście L.

W pojedynczym kroku pętli pobieramy z kolejki Q jakiś program wygrywający v. Obliczamy $L := L \cap Win(v)$, przeglądając obie listy równolegle w kolejności rosnących numerów i wybierając tylko te programy, które występują w obu listach. Programy, które są w ten sposób usuwane z listy L, są wygrywające, więc dodajemy je do kolejki Q. Algorytm się kończy, gdy kolejka Q jest pusta. Wtedy wszystkie programy nie występujące w L są wygrywające.

Rozwiązanie alternatywne zostało zaimplementowane w `turv1.cpp`.

Testy

Rozwiązania były oceniane na zestawie 16 testów. Testy 1a i 1b, 2a i 2b, 3a i 3b oraz 5a i 5b były pogrupowane w pary. W poniższym zestawieniu n jest liczbą programów, m liczbą znanych wyników partii, a w liczbą programów wygrywających.

nr testu	n	m	w	opis
1a	6	12	4	prosty losowy poprawnościowy
1b	1	0	1	przypadek szczególny $n = 1$
2a	10	40	6	prosty losowy poprawnościowy
2b	2	1	1	przypadek szczególny $n = 2$ i jeden zwycięzca
3a	20	153	4	losowy poprawnościowy
3b	20	0	20	przypadek szczególny $n = 20$ i żadnych krawędzi

nr testu	n	m	w	opis
4	134	5089	71	losowy poprawnościowy
5a	1018	11059	1016	losowy poprawnościowo–wydajnościowy
5b	100000	10	100000	wydajnościowy, bardzo mało krawędzi
6	1230	130141	1170	losowy poprawnościowo–wydajnościowy
7	1790	220000	70	losowy poprawnościowo–wydajnościowy
8	4055	530141	65	losowy poprawnościowo–wydajnościowy
9	100000	1000000	99995	losowy wydajnościowy, dużo zwycięzców
10	100000	1000000	4	losowy wydajnościowy, mało zwycięzców
11	1414	998990	2	strukturalny, wydajnościowy na pierwszą fazę
12	1415	998992	1414	strukturalny, wydajnościowy na drugą fazę

Zawody III stopnia

opracowania zadań

Wojciech Guzicki
Treść zadania, Opracowanie

Łukasz Kowalik
Program

Zgadywanka

Bajtazar stał się nałogowym hazardzistą. Wszystkie pieniądze przepuszcza w kasynie grając w „zgadywankę". Twierdzi, że można opracować system gry pozwalający wygrać z kasynem.

Jedna rozgrywka zgadywanki polega na tym, że losujemy kolejno 9 różnych liczb rzeczywistych z przedziału $(0;1)$, przy jednostajnym rozkładzie prawdopodobieństwa. Bezpośrednio po wylosowaniu każdej liczby należy od razu określić, którą co do wielkości jest to liczba. Oczywiście nie da się tego przewidzieć, trzeba próbować to zgadnąć. Jeżeli trafnie określi się uporządkowanie wszystkich 9 liczb, wygrywa się. W przeciwnym przypadku przegrywa się. Jeśli, na przykład, pierwszą wylosowaną liczbę określimy jako drugą co do wielkości, a potem wylosujemy jeszcze dwie liczby mniejsze od niej, to przegramy.

Zadanie

Pomóż Bajtazarowi! Zaprogramuj jak najlepszą strategię grania w zgadywankę. Napisz program, który rozegra wiele rozgrywek opisanej gry i wygra jak najwięcej razy. Ocena Twojego programu będzie tym wyższa, im więcej rozgrywek on wygra. Dokładniej, za każdą wygraną kasyno płaci graczowi 423,99 bajtalarów, natomiast za każdą przegraną pobiera od gracza 13,53 bajtalarów. Twój program rozegra 10^6 rozgrywek. Otrzymasz tyle punktów, ile wynosi całkowity uzyskany zysk podzielony przez 10^4 i zaokrąglony do najbliższej liczby całkowitej z przedziału $[0,100]$.

Musisz zaprogramować moduł zawierający następujące trzy procedury i funkcje:

- **`procedure`** `inicjalizuj` / **`void`** `inicjalizuj()` — *ta procedura zostanie wywołana tylko raz, na początku, przed rozegraniem wszystkich rozgrywek; możesz jej użyć do zainicjalizowania swoich struktur danych.*
- **`procedure`** `nowa_rozgrywka` / `void nowa_rozgrywka ()` — *ta procedura będzie wywołana na początku każdej rozgrywki. Możesz jej użyć by zainicjalizować zmienne związane z jedną rozgrywką.*
- **`function`** `kolejna_liczba (x : Double) : Integer` /
`int kolejna_liczba (double x)` — *ta funkcja dostaje jako parametr kolejną wylosowaną liczbę x; $0 < x < 1$, podaną z precyzją do 12 miejsc dziesiętnych. Funkcja powinna obliczyć, która co do wielkości jest to liczba wśród 9 liczb losowanych w aktualnej rozgrywce i zwrócić wynik. Jeśli rozgrywka jest przegrana, funkcja może zwrócić dowolną liczbę całkowitą od 1 do 9.*

Pliki (Pascal)

W katalogu `zga_pas` *znajdziesz następujące pliki:*

- `zga.pas` — *szkielet modułu grającego zawierający puste procedury i funkcje* `inicjalizuj`, `nowa_rozgrywka`, `kolejna_liczba`. *Powinieneś napisać kod tych funkcji.*

- `kasyno.pas` — *program generujący wiele rozgrywek (tyle ile określono w stałej* `ILE_ZESTAWOW`*) zgadywanki. Każda rozgrywka jest rozgrywana z użyciem procedur i funkcji modułu grającego* `zga.pas`*. Możesz użyć tego programu do testowania swojego modułu grającego.*
- `makefile` — *plik umożliwia skompilowanie programu* `kasyno.pas` *z dołączonym modułem* `zga.pas` *za pomocą polecenia* `make`*.*

Pliki (C/C++)

W katalogu `zga_c` / `zga_cpp` *znajdziesz następujące pliki:*

- `zga.h` — *plik zawierający nagłówki funkcji* `inicjalizuj`, `nowa_rozgrywka` *i* `kolejna_liczba`*.*
- `zga.c` / `zga.cpp` — *szkielet modułu grającego zawierający puste definicje funkcji zadeklarowanych w pliku nagłówkowym* `zga.h`*. Powinieneś napisać kod tych funkcji.*
- `kasyno.c` / `kasyno.cpp` — *program generujący wiele rozgrywek (tyle ile określono w stałej* `ILE_ZESTAWOW`*) zgadywanki. Każda rozgrywka jest rozgrywana z użyciem procedur i funkcji modułu grającego* `zga.c` / `zga.cpp`*. Możesz użyć tego programu do testowania swojego modułu grającego.*
- `makefile` — *plik umożliwia skompilowanie programu* `kasyno.c` / `kasyno.cpp` *z dołączonym modułem* `zga.c` / `zga.cpp` *za pomocą polecenia* `make`*.*

Rozwiązanie

Wynikiem Twojej pracy powinien być tylko jeden plik `zga.c` *albo* `zga.cpp` *albo* `zga.pas`*.*

Przykład

Przykładowa interakcja z programem może wyglądać następująco (pochyłym krojem zaznaczono liczby zwracane przez funkcję `kolejna_liczba`*; rozegrano 2 rozgrywki):*

Rozgrywka 1

`0.043319026120`
1
`0.933801434041`
8
`0.992050359372`
9
`0.145189967093`
4
`0.518803250649`
6

`0.093583048537`
2
`0.764309529654`
7
`0.338653748790`
5
`0.119437652934`
3 Wygrana

Rozgrywka 2

`0.164020610610`
2
`0.205594263575`
3
`0.042637391231`
1
`0.147521628974`
1 Przegrana
`0.946549875333`
1
`0.772946216573`
1
`0.152956276544`
1
`0.539653928563`
1
`0.552047936535`
1

Rozwiązanie

Zanim rozwiążemy zadanie w całej ogólności, popatrzmy na rozwiązanie w kilku prostszych przypadkach. Niech zatem najpierw gra polega na losowaniu pary różnych liczb (x, y) z przedziału $(0, 1)$. Narzuca się następująca strategia: jeśli $x < \frac{1}{2}$, to umieszczamy x na pierwszym miejscu (a więc będziemy musieli umieścić y na drugim miejscu), jeśli zaś $x \geqslant \frac{1}{2}$, to umieszczamy x na drugim miejscu (i umieścimy y na pierwszym miejscu). Czy jest to najlepsza strategia? Przyjrzyjmy się strategiom podobnym, ale liczbę $\frac{1}{2}$ zastąpmy inną liczbą.

Ustalmy liczbę a taką, że $0 < a < 1$. Tę liczbę a nazwiemy *progiem*. Losujemy kolejno dwie liczby x i y. Strategia polega na tym, że jeśli $x < a$, to umieszczamy x na pierwszym miejscu, a y na drugim; jeśli zaś $x \geqslant a$, to umieszczamy x na drugim miejscu i y na pierwszym. Popatrzmy, dla jakiego progu a prawdopodobieństwo końcowego sukcesu będzie największe. Zastosujemy tzw. prawdopodobieństwo geometryczne. Zdarzeniem elementarnym będzie para liczb (x, y) z przedziału $(0, 1)$; zbiór zdarzeń elementarnych Ω będzie więc kwadratem jednostkowym (bez dwóch boków i przekątnej):

$$\Omega = \{(x,y) : \ x,y \in (0,1) \ \wedge \ x \neq y\}.$$

Prawdopodobieństwo dowolnego zdarzenia $A \subseteq \Omega$ otrzymujemy dzieląc pole zdarzenia A przez pole całego zbioru Ω:

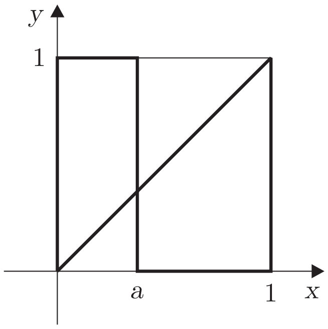
$$P(A) = \frac{Pole(A)}{Pole(\Omega)}.$$

Ponieważ w naszym przypadku $Pole(\Omega) = 1$, więc po prostu $P(A) = Pole(A)$.

Popatrzmy na zdarzenie A_a (rys. 1) polegające na tym, że strategia z progiem a zastosowana do pary (x,y) da sukces:

$$A_a = \{(x,y) : (x < a \wedge x < y) \vee (x \geqslant a \wedge y < x)\}.$$

Nietrudno obliczyć, że $Pole(A_a) = \frac{1}{2} + a - a^2$. Zatem prawdopodobieństwo zdarzenia A_a będzie największe, jeśli $a = \frac{1}{2}$ i wynosi ono wtedy $\frac{3}{4}$.



Rysunek 1: Zdarzenie A_a

Rozwiążmy podobne zadanie dla trzech liczb. Losujemy więc kolejno trzy różne liczby: (x,y,z). Narzuca się następująca strategia:

- Jeśli $x < \frac{1}{3}$, to umieszczamy x na pierwszym miejscu. Następnie, jeśli $y < x$, to grę oczywiście przegraliśmy; jeśli zaś $y > x$, to stosujemy strategię dla dwóch liczb z progiem wypadającym w środku przedziału $(x,1)$: jeśli $y < \frac{x+1}{2}$, to umieszczamy y na drugim miejscu, jeśli zaś $y \geqslant \frac{x+1}{2}$, to umieszczamy y na trzecim miejscu. Oczywiście liczbę z umieszczamy na wolnym miejscu.

- Jeśli $\frac{1}{3} \leqslant x < \frac{2}{3}$, to umieszczamy x na drugim miejscu. Następnie, jeśli $y < x$, to umieszczamy y na pierwszym miejscu, jeśli zaś $y > x$, to umieszczamy y na trzecim miejscu. Liczbę z umieszczamy na pozostałym wolnym miejscu.

- Jeśli $x \geqslant \frac{2}{3}$, to umieszczamy x na trzecim miejscu. Następnie, jeśli $y > x$, to umieszczamy y gdziekolwiek — grę już przegraliśmy. Jeśli zaś $y < x$, to stosujemy strategię dla dwóch liczb z progiem w środku przedziału $(0,x)$: jeśli $y < \frac{x}{2}$, to umieszczamy y na pierwszym miejscu, jeśli zaś $y \geqslant \frac{x}{2}$, to umieszczamy y na drugim miejscu. Liczbę z umieszczamy na pozostałym wolnym miejscu.

Czy ta strategia jest optymalna? Okazuje się, że nie. Spróbujemy znaleźć lepszą. Przede wszystkim wybierzmy dwa *progi* a_1 i a_2 zamiast $\frac{1}{3}$ i $\frac{2}{3}$. Oznaczmy $a = a_1$ i $b = 1 - a_2$, czyli progami będą liczby a i $1 - b$, a nasza strategia wygląda następująco:

- Jeśli $x < a$, to umieszczamy x na pierwszym miejscu. Następnie, jeśli $y < x$, to grę oczywiście przegraliśmy; jeśli zaś $y > x$, to stosujemy strategię dla dwóch liczb z progiem wypadającym w środku przedziału $(x, 1)$: jeśli $y < \frac{x+1}{2}$, to umieszczamy y na drugim miejscu, jeśli zaś $y \geqslant \frac{x+1}{2}$, to umieszczamy y na trzecim miejscu. Oczywiście liczbę z umieszczamy na pozostałym wolnym miejscu.
- Jeśli $a \leqslant x < 1 - b$, to umieszczamy x na drugim miejscu. Następnie, jeśli $y < x$, to umieszczamy y na pierwszym miejscu, jeśli zaś $y > x$, to umieszczamy y na trzecim miejscu. Liczbę z umieszczamy na pozostałym wolnym miejscu.
- Jeśli $x \geqslant 1 - b$, to umieszczamy x na trzecim miejscu. Następnie, jeśli $y > x$, to umieszczamy y gdziekolwiek — grę już przegraliśmy. Jeśli zaś $y < x$, to stosujemy strategię dla dwóch liczb z progiem w środku przedziału $(0, x)$: jeśli $y < \frac{x}{2}$, to umieszczamy y na pierwszym miejscu, jeśli zaś $y \geqslant \frac{x}{2}$, to umieszczamy y na drugim miejscu. Liczbę z umieszczamy na pozostałym wolnym miejscu.

Musimy obliczyć prawdopodobieństwo zdarzenia $A_{a,b}$ polegającego na tym, że strategia z progami a i $1 - b$ zastosowana do trójki liczb (x, y, z) da sukces. Znów skorzystamy z prawdopodobieństwa geometrycznego. Tym razem zbiorem zdarzeń elementarnych będzie sześcian (otwarty) o objętości 1:

$$\Omega = \big\{(x,y,z):\ x,y,z \in (0,1)\ \wedge x \neq y\ \wedge x \neq z\ \wedge\ y \neq z\big\}.$$

Prawdopodobieństwem dowolnego zdarzenia $A \subseteq \Omega$ będzie tym razem objętość bryły A. Popatrzmy na nasze zdarzenie $A_{a,b}$. Jest ono sumą sześciu wielościanów:

$$A_{a,b} = W_1 \cup W_2 \cup W_3 \cup W_4 \cup W_5 \cup W_6,$$

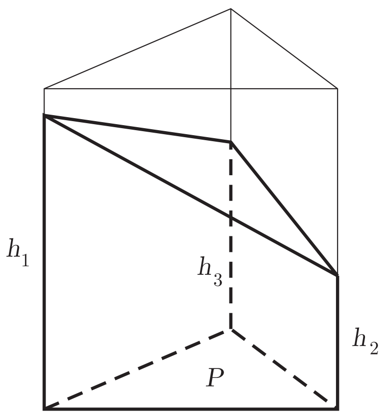
gdzie

$$\begin{aligned}
W_1 &= \big\{(x,y,z) \in \Omega:\ x < a\ \wedge\ x < y < \tfrac{1+x}{2}\ \wedge\ y < z\big\},\\
W_2 &= \big\{(x,y,z) \in \Omega:\ x < a\ \wedge\ y \geqslant \tfrac{1+x}{2}\ \wedge\ x < z < y\big\},\\
W_3 &= \big\{(x,y,z) \in \Omega:\ a \leqslant x < 1 - b\ \wedge\ y < x\ \wedge\ z > x\big\},\\
W_4 &= \big\{(x,y,z) \in \Omega:\ a \leqslant x < 1 - b\ \wedge\ x < y\ \wedge\ z < x\big\},\\
W_5 &= \big\{(x,y,z) \in \Omega:\ x \geqslant 1 - b\ \wedge\ \tfrac{x}{2} \leqslant y < x\ \wedge\ z < y\big\},\\
W_6 &= \big\{(x,y,z) \in \Omega:\ x \geqslant 1 - b\ \wedge\ y < \tfrac{x}{2}\ \wedge\ y < z < x\big\}.
\end{aligned}$$

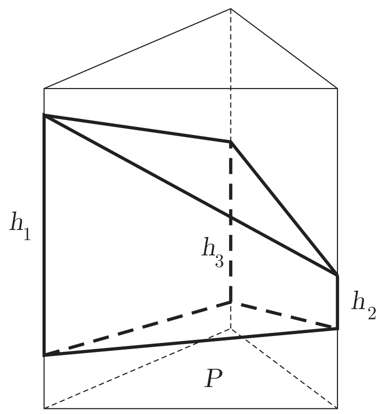
Obliczymy objętość każdego z tych wielościanów.

W tym celu skorzystamy ze wzoru na objętość graniastosłupa trójkątnego ściętego (rys. 2). Jeśli graniastosłup prosty o podstawie trójkątnej o polu P zetniemy płaszczyzną w taki sposób, że krawędzie boczne będą miały długości h_1, h_2 i h_3, to objętość tego graniastosłupa ściętego wyraża się wzorem

$$V = P \cdot \frac{h_1 + h_2 + h_3}{3}$$



Rysunek 2: Graniastosłup trójkątny prosty ścięty z jednej strony



Rysunek 3: Graniastosłup trójkątny prosty ścięty z obu stron

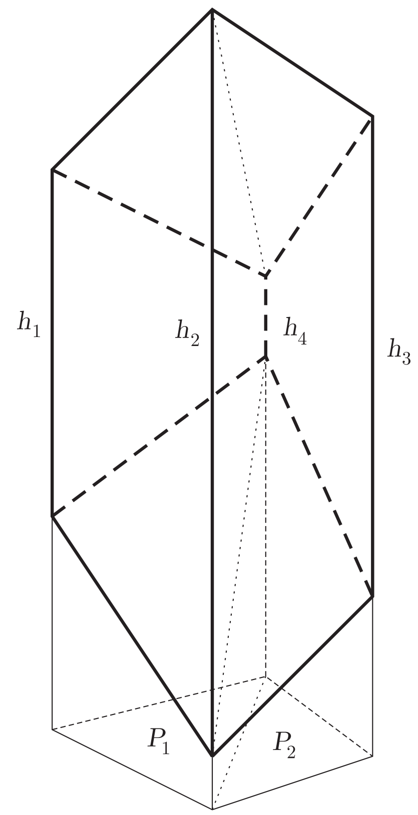
(por. [16], str. 212).

Ten sam wzór będzie wyrażał objętość graniastosłupa trójkątnego prostego, ściętego z obu stron (rys. 3). Dokładniej, jeśli mamy wielościan, którego krawędzie boczne są równoległe, mają długości h_1, h_2 i h_3 oraz którego przekrój płaski płaszczyzną przecinającą prostopadle te krawędzie boczne ma pole P, to objętość tego wielościanu wyraża się tym samym wzorem:

$$V = P \cdot \frac{h_1 + h_2 + h_3}{3}.$$

Objętość graniastosłupa czworokątnego, ściętego z obu stron, obliczamy w podobny sposób: rozcinamy go na dwa graniastosłupy trójkątne i dodajemy objętości tych dwóch graniastosłupów. Objętość graniastosłupa czworokątnego przedstawionego na rysunku 4 wyraża się zatem wzorem

$$V = P_1 \cdot \frac{h_1 + h_2 + h_4}{3} + P_2 \cdot \frac{h_2 + h_3 + h_4}{3}.$$



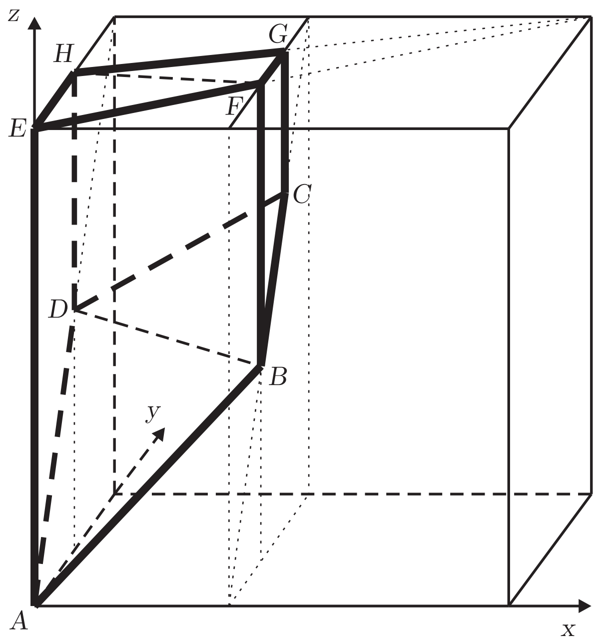
Rysunek 4: Graniastosłup czworokątny prosty ścięty z obu stron

Narysujmy teraz każdy z tych sześciu wielościanów. Okazuje się, że są one graniastosłupami ściętymi (z jednej lub obu stron). Oto te graniastosłupy:

- **Wielościan** W_1 **(rys. 5)**
  Płaszczyzna $BCGF$ ma równanie $x = a$.

$$|EH| = \tfrac{1}{2},$$
$$|GF| = \tfrac{1-a}{2},$$
$$|AE| = 1,$$
$$|BF| = 1-a,$$
$$|CG| = \tfrac{1-a}{2},$$
$$|DH| = \tfrac{1}{2},$$
$$Pole(EFH) = \tfrac{a}{4},$$
$$Pole(HFG) = \tfrac{a(1-a)}{4},$$
$$V = \tfrac{a^3-3a^2+3a}{8}.$$

gdzie $|AB|$ oznacza długość odcinka AB.



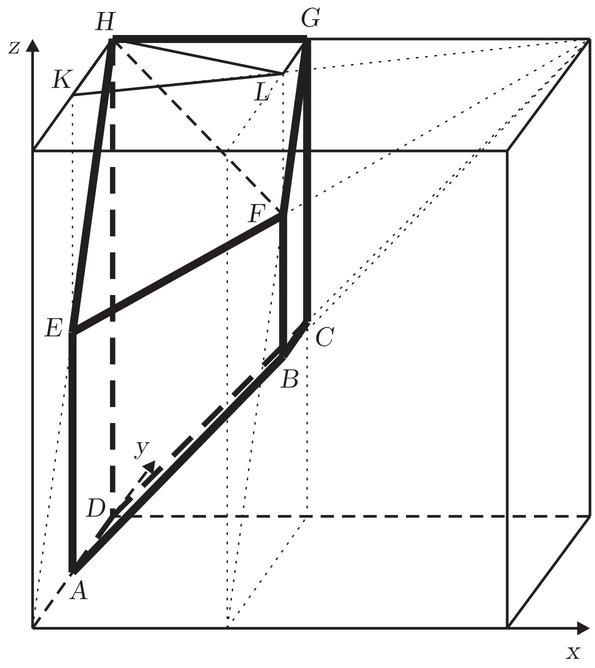
Rysunek 5: Wielościan W_1

- **Wielościan** W_2 **(rys. 6)**
  Płaszczyzna $BCGF$ ma równanie $x = a$.

$$|HK| = \tfrac{1}{2},$$
$$|GL| = \tfrac{1-a}{2},$$
$$|AE| = \tfrac{1}{2},$$
$$|BF| = \tfrac{1-a}{2},$$
$$|CG| = 1-a,$$
$$|DH| = 1,$$
$$Pole(KLH) = \tfrac{a}{4},$$
$$Pole(HLG) = \tfrac{a(1-a)}{4},$$
$$V = \tfrac{a^3-3a^2+3a}{8}.$$

- **Wielościan** W_3 **(rys. 7)**
  Płaszczyzna $ADHE$ ma równanie $x = a$. Płaszczyzna $BCGF$ ma równanie $x = 1-b$.

$$|AD| = 1-a,$$
$$|BC| = b,$$
$$|AE| = a,$$
$$|BF| = 1-b,$$
$$|CG| = 1-b,$$
$$|DH| = a,$$
$$Pole(ABD) = \tfrac{(1-a)(1-a-b)}{2},$$



Rysunek 6: Wielościan W_2

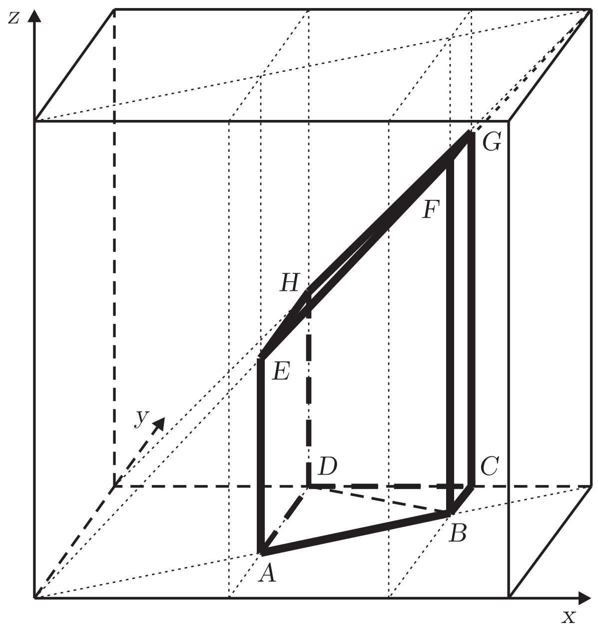
$$Pole(BCD) = \frac{b(1-a-b)}{2},$$
$$V = \frac{2a^3+2b^3-3a^2-3b^2+1}{6}.$$

- **Wielościan** W_4 **(rys. 8)**
  Płaszczyzna $ADHE$ ma równanie $x = a$. Płaszczyzna $BCGF$ ma równanie $x = 1-b$.

$$|HE| = a,$$
$$|GF| = 1-b,$$
$$|AE| = 1-a,$$
$$|BF| = b,$$
$$|CG| = b,$$
$$|DH| = 1-a,$$
$$Pole(HEF) = \frac{a(1-a-b)}{2},$$
$$Pole(BCD) = \frac{(1-b)(1-a-b)}{2},$$
$$V = \frac{2a^3+2b^3-3a^2-3b^2+1}{6}.$$

- **Wielościan** W_5 **(rys. 9)**
  Płaszczyzna $ADHE$ ma równanie $x = 1-b$.

$$|AD| = \frac{1-b}{2},$$
$$|BC| = \frac{1}{2},$$

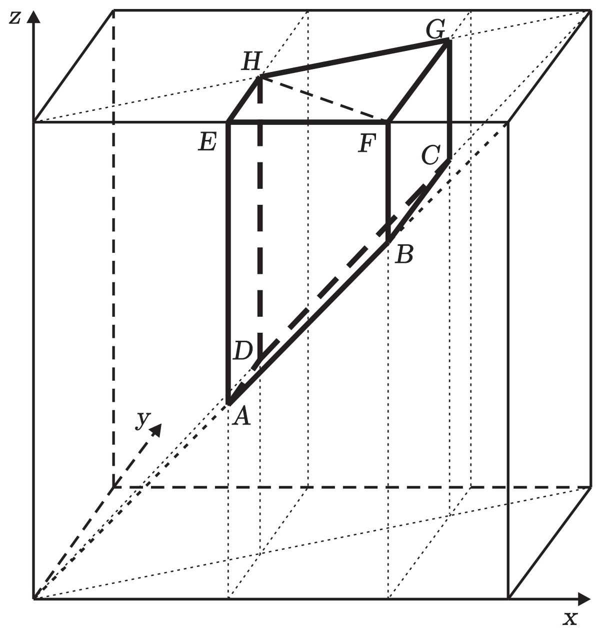


Rysunek 7: Wielościan W_3

$$
\begin{aligned}
|AE| &= \tfrac{1-b}{2},\\
|BF| &= \tfrac{1}{2},\\
|CG| &= 1,\\
|DH| &= 1-b,\\
Pole(ABD) &= \tfrac{b(1-b)}{4},\\
Pole(BCD) &= \tfrac{b}{4},\\
V &= \tfrac{b^3-3b^2+3b}{8}.
\end{aligned}
$$

- **Wielościan W_6 (rys. 10)**
  Płaszczyzna $ADHE$ ma równanie $x = 1-b$.

$$
\begin{aligned}
|AL| &= \tfrac{1-b}{2},\\
|BK| &= \tfrac{1}{2},\\
|AE| &= 1-b,\\
|BF| &= 1,\\
|CG| &= \tfrac{1}{2},\\
|DH| &= \tfrac{1-b}{2},\\
Pole(ABL) &= \tfrac{b(1-b)}{4},\\
Pole(BKL) &= \tfrac{b}{4},\\
V &= \tfrac{b^3-3b^2+3b}{8}.
\end{aligned}
$$



Rysunek 8: Wielościan W_4

Po dodaniu objętości wielościanów od W_1 do W_6 otrzymamy

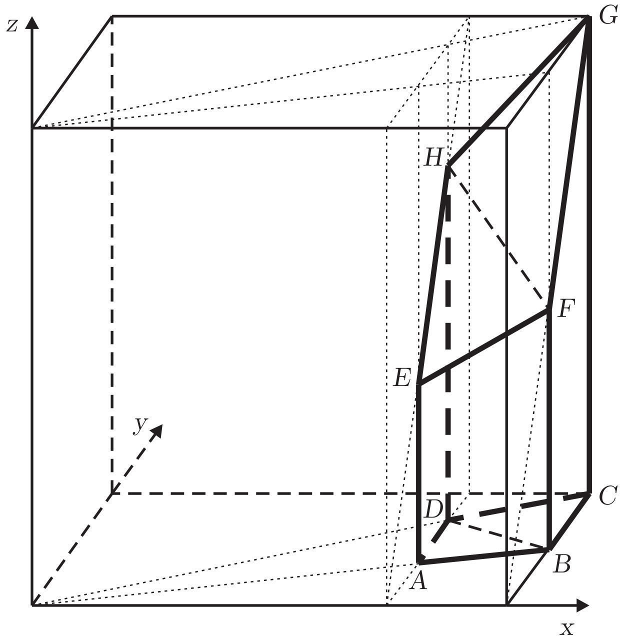
$$P(A_{a,b}) = \frac{11a^3 - 21a^2 + 9a + 2}{6} + \frac{11b^3 - 21b^2 + 9b + 2}{6}.$$

Prawdopodobieństwo $P(A_{a,b})$ będzie największe dla takich wartości $a, b \in (0, \frac{1}{2})$, dla których wielomian $f(x) = 11x^3 - 21x^2 + 9x + 2$ przyjmuje wartość największą. Obliczmy zatem pochodną tego wielomianu:

$$f'(x) = 33x^2 - 42x + 9 = 3(11x^2 - 14x + 3).$$

Pochodna ta ma miejsce zerowe w punkcie $x = \frac{3}{11}$ i można łatwo sprawdzić, że w tym punkcie wielomian $f(x)$ ma maksimum lokalne; jest to jednocześnie największa wartość tego wielomianu w przedziale $(0,1)$. Maksymalna wartość prawdopodobieństwa $P(A_{a,b})$ jest równa $\frac{377}{726} \approx 0{,}519284$. Jednym progiem jest $a = \frac{3}{11}$, drugim jest $1 - b = 1 - \frac{3}{11} = \frac{8}{11}$. Warto zauważyć, że dla $a = b = \frac{1}{3}$ prawdopodobieństwo sukcesu jest równe $\frac{83}{162} \approx 0{,}512345$. Różnica wynosi ok. $\frac{7}{1000}$. Przy milionie doświadczeń jest już wyraźna.

Zauważmy, że progi $\frac{3}{11}$ i $\frac{8}{11}$ można też wyznaczyć łatwiej. Zastanówmy się, gdzie umieścić pierwszą liczbę x. Dokładniej: jakie jest prawdopodobieństwo sukcesu, jeśli umieścimy ją na pierwszym miejscu? Oczywiście wtedy y i z muszą być większe od x (prawdopodobieństwo tego zdarzenia wynosi $(1-x)^2$) i wtedy odniesiemy sukces z prawdopodobieństwem $\frac{3}{4}$. Zatem, jeśli umieścimy liczbę x na pierwszym miejscu, to prawdopodobieństwo sukcesu będzie równe $\frac{3}{4}(1-x)^2$. Jeśli natomiast umieścimy x na drugim miejscu, to sukces odniesiemy wtedy, gdy $y < x$ i $z > x$ (prawdopodobieństwo tego zdarzenia wynosi $x(1-x)$) lub



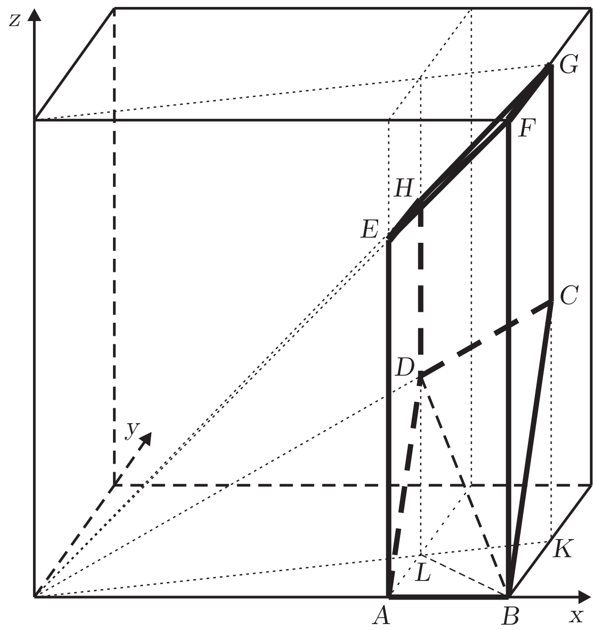
Rysunek 9: Wielościan W_5

$y > x$ i $z < x$ (prawdopodobieństwo sukcesu wynosi też $x(1-x)$). Łącznie w tym przypadku prawdopodobieństwo sukcesu wynosi $2x(1-x)$. Zatem opłaca się umieścić x na pierwszym miejscu wtedy i tylko wtedy, gdy pierwsze prawdopodobieństwo jest większe od drugiego, czyli wtedy i tylko wtedy, gdy

$$\frac{3}{4}(1-x)^2 > 2x(1-x).$$

Rozwiązując tę nierówność, dostajemy $x < \frac{3}{11}$. A więc na pierwszym miejscu opłaca się umieszczać liczby mniejsze od $\frac{3}{11}$; stąd pierwszym progiem jest $a = \frac{3}{11}$. W podobny sposób, porównując prawdopodobieństwa sukcesu, gdy umieścimy x na drugim i trzecim miejscu, otrzymamy drugi próg $\frac{8}{11}$. Zauważmy, że do tego potrzebne było prawdopodobieństwo sukcesu w grze, w której losujemy dwie liczby. Spostrzegawczy Czytelnik z pewnością zauważył, że wielokrotnie korzystaliśmy w tym rozumowaniu z pojęcia prawdopodobieństwa warunkowego.

Mając teraz obliczone również prawdopodobieństwo sukcesu w grze, w której losujemy trzy liczby, możemy w ten sam sposób wyznaczyć progi w grze z czterema liczbami. Umieśćmy zatem pierwszą liczbę x na pierwszym miejscu i obliczmy prawdopodobieństwo sukcesu: trzy pozostałe liczby muszą być większe od x i wtedy uzyskamy sukces z prawdopodobieństwem $\frac{377}{726}$. Łącznie prawdopodobieństwo sukcesu w tym przypadku jest równe $\frac{377}{726}(1-x)^3$. Jeśli natomiast umieścimy x na drugim miejscu, to jedna z pozostałych liczb musi być mniejsza od x i dwie muszą być większe. Prawdopodobieństwo tego zdarzenia wynosi $3x(1-x)^2$: wystarczy zastosować wzór na prawdopodobieństwo tego, że w trzech próbach Bernoulliego uzyskamy dokładnie jeden sukces: sukcesem będzie tu wylosowanie



Rysunek 10: Wielościan W_6

liczby mniejszej od x, porażką wylosowanie liczby większej od x. Jednak wylosowanie tych liczb nie wystarczy: musimy umieścić prawidłowo obie liczy większe od x i prawdopodobieństwo tego zdarzenia wynosi $\frac{3}{4}$. Zatem prawdopodobieństwo całkowitego sukcesu w tym przypadku wynosi $\frac{3}{4}\cdot 3x(1-x)^2 = \frac{9}{4}x(1-x)^2$. Liczbę x należy więc umieścić na pierwszym miejscu wtedy i tylko wtedy, gdy

$$\frac{377}{726}(1-x)^3 > \frac{9}{4}x(1-x)^2.$$

Rozwiążmy tę nierówność:

$$\begin{gathered} 4\cdot 377\cdot(1-x) > 726\cdot 9\cdot x,\\ 2\cdot 377\cdot(1-x) > 363\cdot 9\cdot x,\\ 754-754x > 3267x,\\ 4021x < 754,\\ x < \frac{754}{4021}. \end{gathered}$$

Zatem pierwszym progiem jest $a = \frac{754}{4021} \approx 0{,}187516$. Podobnie wyznaczamy pozostałe progi: $\frac{1}{2}$ i $\frac{3267}{4021} \approx 0{,}812484$. Mamy zatem optymalną strategię w grze z czterema liczbami. Losujemy cztery różne liczby (x,y,z,t). Nasza strategia wygląda następująco:

- Jeśli $x < \frac{754}{4021}$, to umieszczamy x na pierwszym miejscu i pozostałe liczby umieszczamy zgodnie ze strategią dla trzech liczb z progami: $x+\frac{3}{11}(1-x)$ i $x+\frac{8}{11}(1-x)$. Inaczej mówiąc wywołujemy procedurę rekurencyjnie dla trzech liczb i przedziału $(x,1)$.

- Jeśli $\frac{754}{4021} \leqslant x < \frac{1}{2}$, to umieszczamy x na drugim miejscu. Jeśli któraś z następnych liczb jest mniejsza od x, to umieszczamy ją na pierwszym miejscu; jeśli jest większa od x, to stosujemy do niej strategię dla dwóch licz i przedziału $(x,1)$ — próg zatem wynosi $\frac{x+1}{2}$.

- Jeśli $\frac{1}{2} \leqslant x < \frac{3267}{4021}$, to umieszczamy x na trzecim miejscu i postępujemy analogicznie do przypadku (2).

- Jeśli $x \geqslant \frac{3267}{4021}$, to umieszczamy x na czwartym miejscu i postępujemy analogicznie jak w pierwszym przypadku.

Widzimy, że strategia jest wyznaczona jednoznacznie przez progi. Jeśli losujemy m liczb, to pierwszą liczbę umieścimy na miejscu wyznaczonym przez progi dla m liczb. Umieszczenie każdej liczby dzieli przedział na mniejsze przedziały. Jeśli wylosujemy liczbę z pewnego przedziału (x,y) i mamy tam do dyspozycji k miejsc, to tę liczbę umieszczamy na miejscu wyznaczonym przez progi w grze z k liczbami dla przedziału (x,y). Napisanie odpowiedniego programu jest dość łatwe. Zadanie sprowadza się więc do wyznaczenia progów w grach z m liczbami dla $m \leqslant 9$.

Niestety, nie widać prostego sposobu wyznaczenia prawdopodobieństwa sukcesu już w grze z czterema liczbami, a jest ono potrzebne do wyznaczenia progów w grze z pięcioma liczbami. Potem prawdopodobieństwo sukcesu w grze z pięcioma liczbami będzie potrzebne do wyznaczenia progów dla 6 liczb i tak dalej.

Spróbujmy teraz rozwiązać zadanie ogólnie. Losujemy m liczb i jako pierwszą wylosowaliśmy liczbę $x \in (0,1)$. Dajemy ją do jednego z przedziałów:

$$[a_0,a_1), \quad [a_1,a_2), \quad [a_2,a_3), \quad \ldots, \quad [a_{k-1},a_k), \quad [a_k,a_{k+1}), \quad \ldots, \quad [a_{m-1},a_m).$$

O liczbach $a_0,a_1,\ldots,a_m$ zakładamy, że

$$0 = a_0 < a_1 < a_2 < a_3 < \ldots < a_{k-1} < a_k < a_{k+1} < \ldots < a_{m-1} < a_m = 1.$$

Te liczby tak jak poprzednio nazywamy progami.

Przez P_m będziemy oznaczać prawdopodobieństwo tego, że przy optymalnej strategii (tzn. przy optymalnym wyborze progów) m losowych liczb z przedziału $(0,1)$ uporządkujemy we właściwy sposób.

Zakładamy, że znane są już liczby P_i dla $i < m$. Chcemy wyznaczyć optymalne progi przy podziale na m przedziałów i obliczyć P_m.

Przypuśćmy, że liczbę x damy do k-tego przedziału, tzn. $x \in [a_{k-1},a_k)$. Aby wszystkie liczby udało się uporządkować właściwie, muszą być spełnione dwa warunki:

1. Wśród pozostałych liczb ma być dokładnie $k-1$ liczb mniejszych od x i $m-k$ liczb większych od x. Prawdopodobieństwo tego zdarzenia obliczamy ze wzoru na liczbę sukcesów w schemacie Bernoulliego: sukcesem jest wylosowanie liczby mniejszej od x (prawdopodobieństwo sukcesu wynosi x); porażką — wylosowanie liczby większej od x (prawdopodobieństwo porażki wynosi zatem $1-x$). Prawdopodobieństwo uzyskania dokładnie $k-1$ sukcesów w $m-1$ próbach Bernoulliego wynosi $\binom{m-1}{k-1}x^{k-1}(1-x)^{m-k}$.

2. Zarówno liczby mniejsze od x, jak i większe od x, musimy uporządkować poprawnie. Prawdopodobieństwo koniunkcji tych zdarzeń niezależnych wynosi $P_{k-1} \cdot P_{m-k}$.

Łącznie, prawdopodobieństwo poprawnego uporządkowania wszystkich liczb wyniesie w tym przypadku

$$\binom{m-1}{k-1} \cdot P_{k-1} \cdot P_{m-k} \cdot x^{k-1}(1-x)^{m-k}.$$

Przypuśćmy teraz, że liczbę x umieszczamy w następnym przedziale, tzn. $x \in [a_k, a_{k+1})$. Rozumując analogicznie, znajdujemy wzór na prawdopodobieństwo poprawnego uporządkowania wszystkich liczb w tym przypadku:

$$\binom{m-1}{k} \cdot P_k \cdot P_{m-k-1} \cdot x^k(1-x)^{m-k-1}.$$

Próg a_k jest wyznaczony optymalnie, jeśli nierówność $x < a_k$ jest równoważna nierówności

$$\binom{m-1}{k-1} \cdot P_{k-1} \cdot P_{m-k} \cdot x^{k-1}(1-x)^{m-k} > \binom{m-1}{k} \cdot P_k \cdot P_{m-k-1} \cdot x^k(1-x)^{m-k-1}.$$

Mianowicie liczbę x wolimy zakwalifikować do przedziału $[a_{k-1}, a_k)$ niż do przedziału $[a_k, a_{k+1})$ wtedy i tylko wtedy, gdy prawdopodobieństwo sukcesu w pierwszym przypadku jest większe niż w drugim przypadku. Rozwiążmy tę nierówność:

$$\binom{m-1}{k-1} \cdot P_{k-1} \cdot P_{m-k} \cdot x^{k-1}(1-x)^{m-k} > \binom{m-1}{k} \cdot P_k \cdot P_{m-k-1} \cdot x^k(1-x)^{m-k-1},$$

$$\frac{(m-1)!}{(k-1)! \cdot (m-k)!} \cdot P_{k-1} \cdot P_{m-k} \cdot (1-x) > \frac{(m-1)!}{k! \cdot (m-1-k)!} \cdot P_k \cdot P_{m-k-1} \cdot x,$$

$$\frac{1}{m-k} \cdot P_{k-1} \cdot P_{m-k} \cdot (1-x) > \frac{1}{k} \cdot P_k \cdot P_{m-k-1} \cdot x,$$

$$k \cdot P_{k-1} \cdot P_{m-k} \cdot (1-x) > (m-k) \cdot P_k \cdot P_{m-k-1} \cdot x,$$

$$x < \frac{k \cdot P_{k-1} \cdot P_{m-k}}{k \cdot P_{k-1} \cdot P_{m-k} + (m-k) \cdot P_k \cdot P_{m-k-1}}.$$

Stąd otrzymujemy wzór na optymalny próg a_k:

$$a_k = \frac{k \cdot P_{k-1} \cdot P_{m-k}}{k \cdot P_{k-1} \cdot P_{m-k} + (m-k) \cdot P_k \cdot P_{m-k-1}}. \tag{1}$$

Teraz wyznaczymy prawdopodobieństwo sukcesu P_m przy porządkowaniu m liczb. Pokażemy trzy sposoby postępowania. Rozwiązanie wykorzystujące „matematykę wyższą” polega na zastosowaniu rachunku całkowego. Można pokazać, że prawdopodobieństwo P_m wyraża się wzorem

$$P_m = \sum_{i=0}^{m-1} \binom{m-1}{i} \cdot P_i \cdot P_{m-1-i} \cdot \int_{a_i}^{a_{i+1}} x^i(1-x)^{m-1-i}\,dx.$$

Czytelnikowi znającemu elementy rachunku całkowego pokażemy teraz, w jaki sposób można z tego wzoru otrzymać wzór rekurencyjny na P_m. Mianowicie

$$\int x^i(1-x)^{m-1-i}\,dx \quad = \int \textstyle\sum_{j=0}^{m-1-i} (-1)^j \binom{m-1-j}{j} x^{i+j}\,dx =$$

$$= \sum_{j=0}^{m-1-i} (-1)^j \binom{m-1-i}{j} \int x^{i+j}\, dx =$$
$$= \sum_{j=0}^{m-1-i} (-1)^j \binom{m-1-i}{j} \cdot \frac{x^{i+j+1}}{i+j+1} + C.$$

Zatem

$$\int_{a_i}^{a_{i+1}} x^i (1-x)^{m-1-i}\, dx = \sum_{j=0}^{m-1-i} (-1)^j \binom{m-1-i}{j} \cdot \frac{a_{i+1}^{i+j+1} - a_i^{i+j+1}}{i+j+1},$$

skąd dostajemy ostatecznie

$$P_m = \sum_{i=0}^{m-1} \sum_{j=0}^{m-1-i} (-1)^j \cdot \binom{m-1}{i} \cdot \binom{m-1-i}{j} \cdot P_i \cdot P_{m-1-i} \cdot \frac{a_{i+1}^{i+j+1} - a_i^{i+j+1}}{i+j+1}. \qquad (2)$$

Teraz już widzimy, w jaki sposób można otrzymać wszystkie wartości progów i prawdopodobieństw P_m. Na początku kładziemy

$$P_0 = P_1 = 1 \quad \text{oraz} \quad P_2 = \frac{3}{4}.$$

Progi dla $m = 2$ też znamy:

$$a_0 = 0, \quad a_1 = \frac{1}{2}, \quad a_2 = 1.$$

Następnie dla kolejnych $m = 3, \ldots, 9$ najpierw obliczamy wartości progów ze wzoru (1), a następnie wartość P_m ze wzoru (2). Napisanie krótkiego programu, który oblicza te wartości, jest już sprawą rutyny programistycznej. Do niniejszego opracowania jest dołączony taki program o nazwie `progi.pas`. A oto przybliżone wartości progów i prawdopodobieństw, otrzymane za pomocą tego programu:

$m = 1$	$P_1 = 1$	$a_0 = 0,$ $a_1 = 1.$
$m = 2$	$P_2 = 0,75$	$a_0 = 0,$ $a_1 = 0,5,$ $a_2 = 1.$
$m = 3$	$P_3 = 0,519284$	$a_0 = 0,$ $a_1 = 0{,}272727,$ $a_2 = 0{,}727273,$ $a_3 = 1.$

$m=4$	$P_4=0{,}343584$	$a_0=0$, $a_1=0{,}187516$, $a_2=0{,}5$, $a_3=0{,}812484$, $a_4=1$.
$m=5$	$P_5=0{,}221406$	$a_0=0$, $a_1=0{,}141935$, $a_2=0{,}380976$, $a_3=0{,}619024$, $a_4=0{,}858065$, $a_5=1$.
$m=6$	$P_6=0{,}140173$	$a_0=0$, $a_1=0{,}114166$, $a_2=0{,}306086$, $a_3=0{,}5$, $a_4=0{,}693914$, $a_5=0{,}885834$, $a_6=1$.
$m=7$	$P_7=0{,}087637$	$a_0=0$, $a_1=0{,}095446$, $a_2=0{,}255776$, $a_3=0{,}417492$, $a_4=0{,}582508$, $a_5=0{,}744224$, $a_6=0{,}904554$, $a_7=1$.
$m=8$	$P_8=0{,}054279$	$a_0=0$, $a_1=0{,}081992$, $a_2=0{,}219591$, $a_3=0{,}358326$, $a_4=0{,}5$, $a_5=0{,}641674$, $a_6=0{,}780409$, $a_7=0{,}918008$, $a_8=1$.

$m = 9$ $\qquad P_9 = 0{,}033375$ $\qquad a_0 = 0,$
$a_1 = 0{,}071858,$
$a_2 = 0{,}192359,$
$a_3 = 0{,}313750,$
$a_4 = 0{,}437932,$
$a_5 = 0{,}562068,$
$a_6 = 0{,}686250,$
$a_7 = 0{,}807641,$
$a_8 = 0{,}928142,$
$a_9 = 1.$

Teraz pokażemy drugą metodę wyznaczenia progów i prawdopodobieństw P_m. Pomysł polega na tym, by obliczyć prawdopodobieństwo P_m w sytuacji, w której losujemy liczby z dużego zbioru $\{0, 1, 2, \ldots, n\}$, a następnie przejść do granicy dla $n \to \infty$. Okaże się, że nie jest to bardzo trudne.

Przypuśćmy więc, że wszystkie progi i prawdopodobieństwa P_i zostały obliczone dla liczb i mniejszych od m. Wartości progów a_0, …, a_m obliczamy ze wzoru (1). Musimy obliczyć P_m. Zauważmy na wstępie, że wszystkie progi i prawdopodobieństwa P_m są liczbami wymiernymi. Wynika to natychmiast ze wzorów rekurencyjnych (1) i (2). Weźmy więc dużą liczbę n, przy tym taką, by wszystkie liczby $a_i n$ były całkowite.

Przypuśćmy teraz, że za pierwszym razem wylosowaliśmy liczbę k. Załóżmy następnie, że $a_i n < k \leqslant a_{i+1} n$, czyli umieszczamy liczbę k w przedziale o numerze $i+1$. Aby udało się uporządkować wszystkie liczby, musimy w dalszym ciągu wylosować i liczb mniejszych od k i pomyślnie uporządkować wszystkie liczby mniejsze od k i wszystkie liczby większe od k. Prawdopodobieństwo tego wynosi

$$\binom{m-1}{k} \cdot P_i \cdot P_{m-1-i} \cdot \frac{k^i (n-k)^{m-1-i}}{n^m}.$$

Sumując po wszystkich k z przedziału $[a_i n + 1, a_{i+1} n]$, a następnie po wszystkich przedziałach i przechodząc do granicy, otrzymamy

$$P_m = \lim_{n\to\infty} \sum_{i=0}^{m-1} \binom{m-1}{k} \cdot P_i \cdot P_{m-1-i} \cdot \sum_{k=a_i n+1}^{a_{i+1} n} \frac{k^i (n-k)^{m-1-i}}{n^m}.$$

Przekształcamy wzór na P_m:

$$\begin{aligned} P_m &= \lim_{n\to\infty} \sum_{i=0}^{m-1} \binom{m-1}{k} \cdot P_i \cdot P_{m-1-i} \cdot \sum_{k=a_i n+1}^{a_{i+1} n} \frac{k^i (n-k)^{m-1-i}}{n^m} = \\ &= \sum_{i=0}^{m-1} \binom{m-1}{k} \cdot P_i \cdot P_{m-1-i} \cdot \lim_{n\to\infty} \sum_{k=a_i n+1}^{a_{i+1} n} \frac{k^i (n-k)^{m-1-i}}{n^m} = \\ &= \sum_{i=0}^{m-1} \binom{m-1}{k} \cdot P_i \cdot P_{m-1-i} \cdot \end{aligned}$$

$$\cdot \lim_{n\to\infty}\left(\sum_{k=0}^{a_i n}\frac{k^i(n-k)^{m-1-i}}{n^m}-\sum_{k=0}^{a_{i+1}n}\frac{k^i(n-k)^{m-1-i}}{n^m}\right)=$$

$$= \sum_{i=0}^{m-1}\binom{m-1}{k}\cdot P_i\cdot P_{m-1-i}\cdot$$

$$\cdot\left(\lim_{n\to\infty}\sum_{k=0}^{a_i n}\frac{k^i(n-k)^{m-1-i}}{n^m}-\lim_{n\to\infty}\sum_{k=0}^{a_{i+1}n}\frac{k^i(n-k)^{m-1-i}}{n^m}\right).$$

Dla dowolnej liczby c obliczymy teraz granicę

$$\lim_{n\to\infty}\sum_{k=0}^{cn}\frac{k^i(n-k)^{m-1-i}}{n^m}.$$

Najpierw ze wzoru dwumianowego Newtona dostajemy

$$k^i(n-k)^{m-1-i}=\sum_{j=0}^{m-1-i}\binom{m-1-i}{j}\cdot(-1)^j\cdot n^{m-1-i-j}k^{i+j},$$

czyli

$$\frac{k^i(n-k)^{m-1-i}}{n^m}=\sum_{j=0}^{m-1-i}\binom{m-1-i}{j}\cdot(-1)^j\cdot\frac{k^{i+j}}{n^{i+j+1}}.$$

Zatem

$$\begin{aligned}\sum_{k=0}^{cn}\frac{k^i(n-k)^{m-1-i}}{n^m} &= \sum_{k=0}^{cn}\sum_{j=0}^{m-1-i}\binom{m-1-i}{j}\cdot(-1)^j\cdot\frac{k^{i+j}}{n^{i+j+1}}=\\ &= \sum_{j=0}^{m-1-i}\sum_{k=0}^{cn}\binom{m-1-i}{j}\cdot(-1)^j\cdot\frac{k^{i+j}}{n^{i+j+1}}=\\ &= \sum_{j=0}^{m-1-i}\binom{m-1-i}{j}\cdot\frac{(-1)^j}{n^{i+j+1}}\cdot\sum_{k=0}^{cn}k^{i+j}.\end{aligned}$$

Zajmijmy się teraz sumą $\sum_{k=0}^{r} k^p$, gdzie p i r są liczbami naturalnymi. Okazuje się, że istnieje wielomian $W_p(x)$ stopnia $p+1$ taki, że

$$\sum_{k=0}^{r}k^p=W(r).$$

Przykłady takich wielomianów są znane ze szkoły (są to typowe zadania na indukcję):

$$\begin{aligned}\sum_{k=0}^{n}k &= 1+2+\ldots+n=\frac{n(n+1)}{2}=\frac{1}{2}\cdot n^2+\frac{1}{2}\cdot n,\\ \sum_{k=0}^{n}k^2 &= 1^2+2^2+\ldots+n^2=\frac{n(n+1)(2n+1)}{6}=\frac{1}{3}\cdot n^3+\frac{1}{2}\cdot n^2+\frac{1}{6}\cdot n,\end{aligned}$$

$$\sum_{k=0}^{n} k^3 = 1^3+2^3+\ldots+n^3 = \frac{n^2(n+1)^2}{4} = \frac{1}{4}\cdot n^4 + \frac{1}{2}\cdot n^3 + \frac{1}{4}\cdot n^2.$$

Zatem wielomiany $W_p(x)$ mają postać:

$$\begin{aligned}
W_1(x) &= \frac{1}{2}x^2+\frac{1}{2}x,\\
W_2(x) &= \frac{1}{3}x^3+\frac{1}{2}x^2+\frac{1}{6}x,\\
W_3(x) &= \frac{1}{4}x^4+\frac{1}{2}x^3+\frac{1}{4}x.
\end{aligned}$$

Te trzy wielomiany znamy ze szkoły. Potrzebne nam będą następne. Można je znaleźć w książce [20] lub wyznaczyć np. za pomocą programu Derive. Oto one:

$$\begin{aligned}
W_4(x) &= \frac{1}{5}x^5+\frac{1}{2}x^4+\frac{1}{3}x^3-\frac{1}{30}x,\\
W_5(x) &= \frac{1}{6}x^6+\frac{1}{2}x^5+\frac{5}{12}x^4-\frac{1}{12}x^2,\\
W_6(x) &= \frac{1}{7}x^7+\frac{1}{2}x^6+\frac{1}{2}x^5-\frac{1}{6}x^3+\frac{1}{42}x,\\
W_7(x) &= \frac{1}{8}x^8+\frac{1}{2}x^7+\frac{7}{12}x^6-\frac{7}{24}x^4+\frac{1}{12}x^2,\\
W_8(x) &= \frac{1}{9}x^9+\frac{1}{2}x^8+\frac{2}{3}x^7-\frac{7}{15}x^5+\frac{2}{9}x^3-\frac{1}{30}x,\\
W_9(x) &= \frac{1}{10}x^{10}+\frac{1}{2}x^9+\frac{3}{4}x^8-\frac{7}{10}x^6+\frac{1}{2}x^4-\frac{3}{20}x^2.
\end{aligned}$$

Zauważamy, że te wielomiany mają następującą postać:

$$W_p(x) = \frac{1}{p}\cdot x^{p+1} + V_p(x),$$

gdzie V_p jest wielomianem stopnia p. Zatem

$$\sum_{k=0}^{cn} k^p = \frac{c^{p+1}n^{p+1}}{p+1} + V_p(cn).$$

Stąd wynika, że

$$\lim_{n\to\infty} \frac{1}{n^{p+1}}\cdot\sum_{k=0}^{cn} k^p = \lim_{n\to\infty} \frac{c^{p+1}}{p+1} + \frac{V_p(cn)}{n^{p+1}} = \frac{c^{p+1}}{p+1},$$

gdyż wielomian $V_p(cn)$ jest stopnia p i $\lim\limits_{n\to\infty}\frac{V_p(cn)}{n^{p+1}} = 0$. Zatem

$$\lim_{n\to\infty}\sum_{k=0}^{cn}\frac{k^i(n-k)^{m-1-i}}{n^m} = \lim_{n\to\infty}\sum_{j=0}^{m-1-i}\binom{m-1-i}{j}\cdot\frac{(-1)^j}{n^{i+j+1}}\cdot\sum_{k=0}^{cn}k^{i+j} =$$

$$= \sum_{j=0}^{m-1-i} (-1)^j \cdot \binom{m-1-i}{j} \cdot \lim_{n\to\infty} \frac{1}{n^{i+j+1}} \cdot \sum_{k=0}^{cn} k^{i+j}.$$
$$= \sum_{j=0}^{m-1-i} (-1)^j \cdot \binom{m-1-i}{j} \cdot \frac{c^{i+j+1}}{i+j+1}.$$

Stąd dostajemy

$$\lim_{n\to\infty} \sum_{k=0}^{a_{i+1}n} \frac{k^i(n-k)^{m-1-i}}{n^m} = \sum_{j=0}^{m-1-i} (-1)^j \cdot \binom{m-1-i}{j} \cdot \frac{a_{i+1}^{i+j+1}}{i+j+1},$$
$$\lim_{n\to\infty} \sum_{k=0}^{a_i n} \frac{k^i(n-k)^{m-1-i}}{n^m} = \sum_{j=0}^{m-1-i} (-1)^j \cdot \binom{m-1-i}{j} \cdot \frac{a_i^{i+j+1}}{i+j+1},$$

czyli

$$\lim_{n\to\infty} \sum_{k=0}^{a_{i+1}n} \frac{k^i(n-k)^{m-1-i}}{n^m} - \lim_{n\to\infty} \sum_{k=0}^{a_i n} \frac{k^i(n-k)^{m-1-i}}{n^m} =$$
$$= \sum_{j=0}^{m-1-i} (-1)^j \cdot \binom{m-1-i}{j} \cdot \frac{a_{i+1}^{i+j+1} - a_i^{i+j+1}}{i+j+1}.$$

Podstawiając tę różnicę granic do wzoru na P_m, otrzymujemy wzór rekurencyjny (2).

$$P_m = \sum_{i=0}^{m-1} \sum_{j=0}^{m-1-i} (-1)^j \cdot \binom{m-1}{i} \cdot \binom{m-1-i}{j} \cdot P_i \cdot P_{m-1-i} \cdot \frac{a_{i+1}^{i+j+1} - a_i^{i+j+1}}{i+j+1}. \qquad (2)$$

Wreszcie naszkicujemy trzeci sposób wyznaczenia prawdopodobieństwa P_m. Jeśli znamy już progi w grze z m liczbami, to piszemy program symulujący naszą grę. Wykonujemy np. milion (lub lepiej 10 milionów) prób i zliczamy, ile było udanych. Stosunek liczby udanych prób do wszystkich możemy przyjąć za prawdopodobieństwo sukcesu. Tę metodę przyjęto w programie wzorcowym.

Karol Cwalina
Treść zadania, Opracowanie

Łukasz Kowalik
Program

Misie-Patysie

Bolek i Lolek bawią się w grę o swojsko brzmiącej nazwie misie-patysie. Aby zagrać w misie-patysie wystarczy mieć trochę **misiów** *oraz równie nieokreśloną liczbę* **patysiów**, *które tworzą razem* **pulę gry**. *Trzeba też wybrać dowolną liczbę naturalną, którą zgodnie z tradycją nazywa się* **MP-ograniczeniem**.

Pierwszy ruch należy do Bolka, który może zabrać z puli pewną dodatnią liczbę misiów albo patysiów. Może też jednocześnie wziąć i misie, i patysie, ale tylko jeśli jednych i drugich bierze tyle samo i więcej od zera. Oczywiście nie można zabrać więcej misiów niż jest ich w puli — podobnie z patysiami. Co więcej ani jednych, ani drugich nie można wziąć więcej niż wynosi MP-ograniczenie.

Kolejne ruchy wykonywane są na przemian według tych samych reguł. Wygrywa ten z graczy, po którego ruchu pula pozostanie pusta. Twoim celem jest pomóc Bolkowi w odniesieniu zwycięstwa nad Lolkiem.

Zadanie

Zadanie polega na napisaniu modułu, który po skompilowaniu z odpowiednim programem grającym będzie grał jako Bolek. Na potrzeby tego zadania otrzymasz **uproszczony** *program grający, który pozwoli Ci przetestować rozwiązanie. Twój moduł powinien zawierać następujące dwie procedury (funkcje):*

- **procedure** `poczatek (m, p, mpo : LongInt)` *lub*
  void `poczatek (int m, int p, int mpo)`
  Ta procedura (funkcja) będzie wywołana tylko raz, na początku rozgrywki. Możesz jej użyć by zainicjalizować zmienne lub struktury danych potrzebne podczas rozgrywki. Parametry `m`, `p` *oraz* `mpo` *zawierają odpowiednio liczbę misiów i patysiów w puli gry oraz MP-ograniczenie. Liczby Misiów i Patysiów w puli będą zawsze dodatnie i niewiększe niż 10^7. MP-ograniczenie będzie dodatnie i niewiększe niż 10^6.*

- **procedure** `ruch_bolka (m, p : LongInt; var bm, bp : LongInt)` *lub*
  void `ruch_bolka (int m, int p, int *bm, int *bp)`
  Ta procedura (funkcja) powinna wyznaczyć kolejny ruch Bolka. Przekazane jej parametry `m` *i* `p` *zawierają odpowiednio liczbę misiów i patysiów w puli gry pozostałych po ostatnim ruchu Lolka. Wartości parametrów przekazanych przy pierwszym wywołaniu procedury (funkcji)* `ruch_bolka` *są takie same jak odpowiednie wartości parametrów przekazanych do procedury (funkcji)* `poczatek`. *Przed zakończeniem wykonania procedury zmienne* `bm`, `bp` *(*`*bm` *i* `*bp` *w języku C/C++) powinny zawierać, odpowiednio, liczbę misiów i patysiów wziętych przez Bolka z puli.*

Twój moduł nie może otwierać żadnych plików ani korzystać ze standardowego wejścia/wyjścia. Jeśli twój program wykona niedozwoloną operację, w tym np. procedura `ruch_bolka` *zwróci ruch niezgodny z zasadami gry, jego działanie zostanie przerwane. W tym przypadku dostaniesz 0 punktów za dany test.*

Jeśli Twój program przegra rozgrywkę, dostaniesz 0 punktów za test. Jeśli wygra, dostaniesz maksymalną przewidzianą za dany test liczbę punktów. Dla każdego testu, na którym zostanie uruchomiony Twój program, Bolek może wygrać, niezależnie od ruchów Lolka.

Pliki (Pascal)

W katalogu `mis_pas` *znajdziesz następujące pliki:*

- `mis.pas` – *szkielet modułu grającego zawierający puste procedury* `poczatek` *i* `ruch_bolka`*. Powinieneś napisać kod tych procedur.*
- `graj.pas` – *uproszczony program generujący rozgrywkę. Ruchy Lolka wykonywane są zgodnie z bardzo prostą strategią, natomiast ruchy Bolka wyznaczane są przez wywoływanie procedur modułu grającego* `mis.pas`*. Możesz użyć tego programu do testowania swojego modułu grającego.*

Pliki (C/C++)

W katalogu `mis_c/mis_cpp` *znajdziesz następujące pliki:*

- `mis.h` – *plik zawierający nagłówki funkcji* `poczatek` *i* `ruch_bolka`*.*
- `mis.c/mis.cpp` – *szkielet modułu grającego zawierający puste definicje funkcji zadeklarowanych w pliku nagłówkowym* `mis.h`*. Powinieneś napisać kod tych funkcji.*
- `graj.c/graj.cpp` – *uproszczony program generujący rozgrywkę. Ruchy Lolka wykonywane są zgodnie z bardzo prostą strategią, natomiast ruchy Bolka wyznaczane są przez wywoływanie procedur modułu grającego* `mis.c/mis.cpp`*. Możesz użyć tego programu do testowania swojego modułu grającego.*

Rozwiązanie

Wynikiem Twojej pracy powinien być tylko jeden plik `mis.c`, `mis.cpp` *lub* `mis.pas`*, ale nie kilka równocześnie.*

Przykład

Rozgrywka może przebiegać następująco:

Wywołanie	*Opis*
`poczatek(7, 2, 3);`	*jest 7 Misiów, 2 Patysie, a MP-ograniczenie=3*
`ruch_bolka (7, 2, bm, bp);` *lub*	*pierwszy ruch*
`ruch_bolka (7, 2, &bm, &bp);`	*Bolek bierze z puli 1 misia i 1 patysia; zostaje 6 misiów i 1 patyś*
`ruch_bolka (3, 1, bm, bp);` *lub*	*Lolek wziął z puli 3 misie; zostały 3 misie i 1 patyś*
`ruch_bolka (3, 1, &bm, &bp);`	*Bolek bierze z puli 1 misia; zostają 2 misie i 1 patyś*
`ruch_bolka (1, 0, bm, bp);` *lub*	*Lolek wziął z puli 1 misia i 1 patysia; został 1 miś i 0 patysiów*
`ruch_bolka (1, 0, &bm, &bp);`	*Bolek bierze z puli 1 misia i wygrywa*

Rozwiązanie

Żeby z powodzeniem grać w Misie-Patysie, musimy w każdym momencie gry potrafić określić optymalny ruch. Optymalny to znaczy taki, by niezależnie od gry naszego przeciwnika, rozgrywka zakończyła się naszą wygraną. Spróbujmy więc dokonać krótkiego przeglądu metod, którymi moglibyśmy się kierować przy wyborze ruchu.

Najprostszą strategią, banalną w implementacji, jest strategia losowa. Jak jednak później pokażemy, stosujący ją gracz jest niemal pewnym przegranym w starciu z rozsądnie grającym przeciwnikiem.

Inną metodą, często spotykaną w próbach rozwiązań problemów decyzyjnych, jest poszukiwanie algorytmu zachłannego. Algorytm taki pobieżnie analizuje dostępne mu dane i wybiera ten z możliwych ruchów, który wydaje się najkorzystniejszy w aktualnej sytuacji. Przykładem zagadnienia, dla którego istnieje intuicyjny algorytm zachłanny, jest problem wydawania reszty: dysponując monetami o określonych nominałach, chcemy wydać określoną kwotę pieniędzy w jak najmniejszej liczbie monet. Zachłanna metoda postępowania polega w tym przypadku na braniu najwyższych nominałów, które nie powodują przekroczenia kwoty do wydania. Niestety w naszej grze ciężko wskazać, czym mielibyśmy się kierować przy wartościowaniu ruchów.

Istnieje jeszcze trzecia, niezawodna metoda, polegająca na przeglądaniu całego drzewa gry, czyli symulowaniu wszystkich możliwych rozgrywek i wyborze takiego ruchu, który we wszystkich rozgrywkach umożliwiał odniesienie zwycięstwa. Właściwie przy próbie analizy dowolnej popularnej gry, prędzej czy później, dochodzi się do tej metody — tak jest w przypadku szachów lub go. Być może trudna i pracochłonna analiza jest właśnie przyczyną sukcesu tych gier, gdyż gwarantują w ten sposób ciekawą i nieprzewidywalną rozgrywkę.

Przeszukiwanie drzewa gry

Jak już wspomnieliśmy, metoda ta jest niesłychanie pracochłonna, pomimo to spróbujmy zapisać odpowiedni algorytm, a następnie go zmodyfikować. Dla wygody niech μ oznacza MP-ograniczenie.

function *czyWygrywajaca*(m, p : **longint**) : **boolean**;

```
  { m — liczba misiów, p — liczba patysiów w puli }
  { true gdy zaczynający ma dla tej puli strategię wygrywającą }
var i : longint;
begin
  if m=0 and p=0 then exit(false);
  for i := 1 to MIN(m, μ) do
    if not czyWygrywajaca(m-i, p) then exit(true);
  for i := 1 to MIN(p, μ) do
    if not czyWygrywajaca(m, p-i) then exit(true);
  for i := 1 to MIN(m, p, μ) do
    if not czyWygrywajaca(m-i, p-i) then exit(true);
  czyWygrywajaca := false;
end;

procedure ruch_bolka(m, p : longint; var bm, bp : longint);
var i : longint;
begin
  { próbujemy zabrać misie... }
  for i := 1 to MIN(m, μ) do
    if not czyWygrywajaca(m-i, p) then begin
      bm := i; bp := 0;
      exit;
    end;

  { ...a teraz patysie... }
  for i := 1 to MIN(p, μ) do
    if not czyWygrywajaca(m, p-i) then begin
      bm := 0; bp := i;
      exit;
    end;

  { ...a teraz jedne i drugie }
  for bm := 1 to MIN(m, p, μ) do
    if not czyWygrywajaca(m-bm, p-bp) then begin
      bm := i; bp := i;
      exit;
    end;
  { błąd: nie istnieje ruch wygrywający dla Bolka }
end;
```

Program realizujący metodę przeszukiwania całego drzewa gry został przedstawiony w pliku `mis3.pas`.

Wprawdzie tak zapisany algorytm jest poprawny, ale jego złożoność jest wykładnicza. Dzieje się tak, bo choć liczba wszystkich możliwych stanów gry wynosi $(m+1)(p+1)$, to funkcja *czyWygrywajaca* jest wywoływana bardzo wiele razy dla tych samych argumentów, a każde takie wywołanie oznacza kolejne wywołania. Taka rozrzutność jest niepotrzebna — raz obliczone wyniki mogą zostać zapamiętane, a to już prowadzi do rozwiązania o złożoności $O(mp\mu)$.

Algorytm dynamiczny

Wspomniane usprawnienie, polegające na zapamiętywaniu wyników funkcji *czyWygrywajaca*, nosi nazwę spamiętywania i jest bardzo bliskie innej ważnej technice programowania zwanej **programowaniem dynamicznym**.

Zauważmy, że przedstawiony wyżej algorytm prowadzi do wywołań funkcji dla wszystkich możliwych par argumentów, nic nie stoi więc na przeszkodzie, byśmy w fazie inicjalizacji naszej gry zbudowali tablicę, która gromadziłaby wyniki wszystkich możliwych wywołań funkcji, a dopiero potem przystąpili do rozgrywki. W naszym przypadku odpowiedź dla kolejnych (większych) pul określamy znając wyniki dla niektórych poprzednich (mniejszych) — i to jest właśnie istota programowania dynamicznego.

Niestety, choć osiągnęliśmy już wiele, dysponujemy bardzo wolnym rozwiązaniem, o skrajnie dużych wymaganiach pamięciowych, które się nawet zwiększyły po wprowadzeniu spamiętywania/programowania dynamicznego. Musimy więc dokonać pewnych zmian w naszym algorytmie — na początku zmienimy schemat obliczeń dynamicznych.

W podejściu wzorowanym na metodzie obliczania funkcji *czyWygrywająca*, aby poznać wynik dla kolejnej puli, musimy „zebrać” rezultaty dla mniejszych pul. Tym razem spróbujmy natomiast „rozpropagować” pewien wynik już w momencie jego otrzymania. W tym celu zauważmy, że pula (m,p), składająca się z m misiów i p patysiów, jest wygrywająca jeśli dla pewnego $1 \leqslant i \leqslant \mu$ choć jedna z pul $(m-i,p)$, $(m,p-i)$, $(m-i,p-i)$ jest przegrywająca — zatem po znalezieniu puli przegrywającej (m,p) od razu możemy uznać pule $(m+i,p)$, $(m,p+i)$ i $(m+i,p+i)$, gdzie $1 \leqslant i \leqslant \mu$, za wygrywające.

To prowadzi do następującego programu, który został przedstawiony w pliku `mis2.pas`.

```
var czyWygrywajaca : array [0..MaxM,0..MaxP] of boolean = {false};
  { MaxM, MaxP — maksymalne liczby misiów i patysiów w puli }
  { całą tablicę wypełniamy wartościami false }

procedure poczatek(m, p, μ : longint);
var i, im, ip : longint;
begin
  for im := 0 to MaxM do
    for ip := 0 to MaxP do
      if not czyWygrywajaca then begin
        for i := 1 to MIN(μ, MaxM-im) do
          czyWygrywajaca[im+i,ip] := true;
        for i := 1 to MIN(μ, MaxP-ip) do
          czyWygrywajaca[im,ip+i] := true;
        for i := 1 to MIN(μ, MaxM-im, MaxP-ip) do
          czyWygrywajaca[im+i,ip+i] := true;
      end;
end;
```

```
procedure ruch_bolka(m, p : longint; var bm, bp : longint);
  { treść tej procedury pozostaje prawie taka sama }
  { nawiasy po czyWygrywajaca powinny być kwadratowe, a nie okrągłe }
```

Zauważmy, że w każdej kolumnie tablicy nie więcej niż co $(\mu+1)$-sza pozycja jest przegrywająca, bo każdej takiej odpowiada μ pozycji wygrywających. Tym samym złożoność tego algorytmu wynosi $O(mp)$ i jak się przekonamy, jest on doskonałym punktem wyjścia do dalszych badań. Musimy jednak znaleźć zupełnie nowe spojrzenie na zadanie, które pozwoli nam usprawnić algorytm.

Przypadki szczególne

Bardzo często rozważenie przypadków szczególnych może nas naprowadzić na trop właściwego rozwiązania. My rozważymy dwa takie przypadki:

a) gdy pula składa się z samych Misiów;

b) gdy $\mu = \infty$, czyli innymi słowy nie ma MP-ograniczenia.

Przypadek a) jest bardzo prosty — dwóch graczy na przemian wybiera liczby naturalne ze zbioru $\{1,\ldots,\mu\}$, a zwycięzcą jest ten, który doprowadzi do tego, by suma wybranych liczb była równa m. Odrobina praktyki pozwala zauważyć, że Bolek (zaczynający gracz) jest skazany na porażkę wtedy i tylko wtedy, gdy m dzieli się przez $(\mu+1)$.

Dzieje się tak, gdyż Lolek może tak grać, by suma jego liczby i liczby wybranej chwilę wcześniej przez przeciwnika była równa $(\mu+1)$. Z drugiej strony, gdy $(\mu+1)$ nie dzieli m, to Bolek ma strategię wygrywającą, gdyż może wybrać liczbę m **mod** $(\mu+1)$ i postawić Lolka w sytuacji gracza zaczynającego, gdy nowe $m := m - (m \textbf{ mod } (\mu+1))$ dzieli się przez $(\mu+1)$.

To już coś — potrafimy dobrze grać w przypadku jednowymiarowym. Ale czy wnosi to cokolwiek do oryginalnych Misiów-Patysiów? Na szczęście tak: sugeruje nam, że czasami można swoimi ruchami dopełniać ruch przeciwnika do długości $(\mu+1)$, a w rezultacie przestać się przejmować MP-ograniczeniem.

Aby to pokazać, opiszemy najpierw grę dualną do Misiów-Patysiów, po czym na jej przykładzie pokażemy jak zapomnieć o MP-ograniczeniu. O Misiach-Patysiach można mianowicie myśleć w następujący sposób: na nieskończonej szachownicy, zajmującej pierwszą ćwiartkę układu współrzędnych, postawiono hetmana. W pojedynczym ruchu może się on przesunąć w lewo, w dół lub po skosie w lewo-dół, nie więcej jednak niż o μ. Gracze na przemian wykonują ruchy, a wygrywa ten z nich, który doprowadzi hetmana w lewy-dolny róg szachownicy.

Ta gra jest faktycznie tożsama z naszą, gdyż możemy się umówić, że wiersz i kolumna (numerowane od 0), na których stoi hetman, odpowiadają liczbie misiów i patysiów w puli w danym momencie. Ruch figury zaś odpowiada zabraniu pewnej ich liczby.

Po takim przygotowaniu pokażemy, że zachodzi następujący

Lemat 1 *Niech* $\overline{m} = m \textbf{ mod } (\mu+1)$, $\overline{p} = p \textbf{ mod } (\mu+1)$.
Wówczas $czyWygrywająca(m,p) = czyWygrywająca(\overline{m},\overline{p})$.

Dowód Myśląc o dualiźmie pomiędzy grami, podzielmy szachownicę na kwadraty o boku $(\mu+1)$ i określmy następującą strategię gry:

- gdy przeciwnik w swoim ostatnim ruchu przeniósł hetmana pomiędzy dwoma kwadratami, wykonujemy taki ruch, by w efekcie tego przesunięcia i ruchu przeciwnika zmienił się kwadrat, ale nie położenie hetmana w kwadracie;
- w przeciwnym przypadku wykonujemy taki ruch, jaki byśmy wykonali stojąc na polu $(\overline{m},\overline{p})$, ew. dowolny, jeśli stoimy w lewym-dolnym rogu kwadratu (to się może zdarzyć tylko wtedy, gdy nie będziemy zaczynać z pozycji wygrywającej).

Zauważmy, że jeśli $czyWygrywająca(\overline{m},\overline{p}) = \textbf{true}$, to powyższa strategia pokazuje, że $czyWygrywająca(m,p) = \textbf{true}$. Na grę możemy bowiem wtedy patrzeć jak na rozgrywkę toczoną wewnątrz pojedynczego kwadratu, której celem jest dojście do jego lewego-dolnego rogu — wystarczy pominąć te pary ruchów, które nie zmieniają położenia hetmana w kwadracie. A założyliśmy, że dla gry w kwadracie istnieje strategia wygrywająca dla zaczynającego. Podobnie można pokazać, że jeśli:

$$czyWygrywająca(\overline{m},\overline{p}) = \textbf{false}, \text{ to } czyWygrywająca(m,p) = \textbf{false}.$$

Dopóki bowiem Bolek zmienia kwadrat, w którym stoi figura, Lolek wykorzystuje pierwszy punkt strategii, więc gdy pierwszy z graczy wykonuje ruchy, to stale $czyWygrywająca(\overline{m},\overline{p}) = \textbf{false}$; w pewnym momencie jednak Bolek musi wykonać ruch wewnątrz kwadratu, przesuwając się do pola (m',p') takiego, że $czyWygrywająca(\overline{m'},\overline{p'}) = \textbf{true}$, a wtedy już Lolek będzie się znajdował w sytuacji opisanej we wcześniejszym akapicie. ■

Tym samym wystarczy wyznaczenie kawałka $czyWygrywająca[0..\mu,0..\mu]$, a w nim nie ma już sensu pojęcie MP-ograniczenia, gdyż w rozważanych pulach nie będzie nigdy więcej niż μ misiów i μ patysiów. Jednocześnie złożoność rozwiązania zmniejszyła się do $O(min(\mu^2,mp))$.

Brak MP-ograniczenia

Jak się okazało, przypadek ten w sposób jak najbardziej naturalny wypłynął w naszych rozważaniach.

Przypatrzmy się teraz sposobowi, w jaki wypełniamy tablicę *czyWygrywajaca*. Na początek przyjmijmy definicję: ***k*-tą przekątną** macierzy A nazywamy taki zbiór jej pozycji $a_{i,j}$, że $i-j=k$. Procedurę *poczatek* możemy przepisać w następującej postaci:

```
procedure poczatek (m, p, μ : longint);
  { wypełnia pola czyWygrywająca[i,j] dla i, j ⩽ μ }
var im, ip : longint
begin
  wypełnij tablicę czyWygrywajaca wartościami UNDEF
  for im := 0 to μ do begin
    niech ip będzie najmniejsze nieujemne takie, że czyWygrywajaca[im,ip]
    jest UNDEF;
    czyWygrywajaca[im,ip] := false;
```

```
9:      „zaznacz” im-ty wiersz: wartości UNDEF zamień na true;
10:     „zaznacz” ip-tą kolumnę: wartości UNDEF zamień na true;
11:     „zaznacz” (im − ip)-tą przekątną: wartości UNDEF zamień na true;
12:   end;
13: end;
```

W tym momencie jesteśmy już o krok od rozwiązania liniowego.

Przede wszystkim zauważmy, że skoro w każdej kolumnie tablicy jest tylko jedno pole z wartością **false**, moglibyśmy zrezygnować z zapisywania wartości **true** w tablicy, a zamiast tego dla każdego numeru m kolumny zapisywać numer p wiersza takiego, że $czyWygrywająca(m, p) =$ **false**. „Zaznaczenie” wiersza, kolumny lub przekątnej można by wtedy wykonać w czasie stałym.

Co więcej, ponieważ $czyWygrywająca(m, p) = czyWygrywająca(p, m)$, to w chwili znalezienia odpowiedniej pary (m, p) możemy jednocześnie „zaznaczać” pola dla przypadku (p, m). Takie postępowanie pozwala z kolei łatwo znajdować ip (7. wiersz kodu), gdyż pole (im, ip) musi leżeć na następnej przekątnej tablicy w stosunku do poprzednio znalezionej pary.

Całość zmian prowadzi do algorytmu wzorcowego o czasowej i pamięciowej złożoności $O(\mu)$.

Odpowiednie programy zapisane są w plikach `mis.pas`, `mis.c` i `mis.cpp`.

Inne rozwiązania

Wszystkie poprawne rozwiązania są prawdopodobnie mniej lub bardziej zbliżone do przedstawionego przez nas rozwiązania wzorcowego, ew. do którejś z przedstawionych wyżej jego nieefektywnych wersji.

Spośród rozwiązań niepoprawnych ciężko wybrać jakieś szczególnie interesujące — godne zauważenia jest chyba tylko, że ponieważ w każdej kolumnie i wierszu co $(\mu + 1)$-sze pole jest wygrywające, to prawdopodobieństwo zwycięstwa w przypadku posługiwania się strategią losową wynosi mniej więcej $(\frac{1}{\mu})^{liczba\ ruchów}$, a więc jest skrajnie małe. Śmiało można wysunąć hipotezę, że równie ciężko w ten sposób zwyciężyć, co ułożyć puzzle rzucając nimi o ścianę. Aby umożliwić wszystkim chętnym wypróbowanie własnej cierpliwości, strategia losowa została zaimplementowana w pliku `misb2.pas` — nie radzimy jednak czekać na zwycięstwo tego programu dla dużych testów.

Testy

Rozwiązania zawodników były sprawdzane na zestawie 12 testów, z których `1a` i `1b` oraz `4a` i `4b` zostały zgrupowane — aby dostać punkty za grupę, należało zaliczyć oba testy z grupy.

nr testu	m	p	μ	opis
1a	2	2	5	Poprawnościowy; $p < \mu$.
1b	5	5	5	Poprawnościowy.
2	100	800	15	Wydajnościowy (odsiewa wykładnicze).
3	501	501	1	j.w.
4a	50000	20	1000	Odsiewa $O(mp)$ bez dynamicznej alokacji pamięci.
4b	20	50000	1000	j. w.
5	10^5	10^5	50	Wydajnościowy (odsiewa $O(mp)$).
6	123456	654321	98765	Wydajnościowy.
7	10^7	1	10^6	j. w.
8	99999	99999	10^6	j. w.
9	10^6	10^6	$2 \cdot 10^6$	j. w.
10	10^7	10^7	10^6	Wydajnościowy, maksymalny.

Rozwiązanie wykładnicze przechodzi testy 1a, 1b. Algorytmy o złożoności $\Theta(mp)$, bez dynamicznego przydziału pamięci — 1a, 1b, 2, 3; $\Theta(mp)$ z dynamicznym przydziałem pamięci — 1a, 1b, 2, 3, 4a, 4b; $\Theta(\mu^2)$ wszystkie testy do 5 włącznie. Testy 6-10 przechodzi jedynie rozwiązanie wzorcowe.

Karol Cwalina | **Piotr Sankowski**
Treść zadania, Opracowanie | Program

Wschód-Zachód

Umowy międzynarodowe podpisane przez Wielce Rozległe Państwo nakładają na nie obowiązek tranzytowy — musi ono umożliwić przewiezienie koleją odpadów nuklearnych z elektrowni u wschodniego sąsiada do punktów utylizacji za zachodnią granicą. Względy ekologiczne nakazują taką organizację ruchu, by pociągi z odpadami jak najszybciej opuściły terytorium kraju.

Sieć kolejowa w tym państwie ma bardzo szczególną strukturę. Składa się ona z n miast-węzłów kolejowych oraz $n-1$ dwukierunkowych odcinków torów, łączących miasta-węzły. Między każdymi dwoma miastami istnieje możliwość przejazdu. Ponadto istnieje taki odcinek sieci, którego końce nie są miastami granicznymi, oraz każdy przejazd z miasta na granicy wschodniej do miasta na granicy zachodniej prowadzi przez ten odcinek.

Wszystkie pociągi z odpadami przyjeżdżają na granicę wschodnią tego samego dnia przed świtem, przy czym każdy z nich do innego miasta. Ze względu na niebezpieczeństwo wypadku pociągi jeżdżą tylko w dzień i po żadnym odcinku nie może jednocześnie jechać więcej niż jeden pociąg z odpadami, natomiast dowolnie wiele takich pociągów może oczekiwać w mieście-węźle. Dodatkowo przejechanie jednego odcinka zajmuje pociągowi jeden dzień. Ruch musi być tak zorganizowany, by każdy pociąg z odpadami dojechał do innego przejścia na granicy zachodniej.

Ile co najmniej dni pociągi z odpadami muszą spędzić na terytorium Wielce Rozległego Państwa?

Zadanie

Zadanie polega na napisaniu programu, który:

- *wczyta ze standardowego wejścia opis sieci kolejowej i przejść granicznych, do których przyjechały pociągi z odpadami;*
- *wyznaczy minimalną liczbę dni, jakie musi trwać tranzyt;*
- *wypisze znalezioną liczbę na standardowe wyjście.*

Wejście

Pierwsza linia wejścia zawiera trzy pooddzielane pojedynczymi odstępami liczby naturalne $1 \leqslant n, w, z \leqslant 10^6$, $n \geqslant w+z+2$. Liczba n oznacza liczbę miast-węzłów (są one ponumerowane od 1 do n), zaś w i z oznaczają liczby przejść granicznych odpowiednio na granicy wschodniej i zachodniej. Przejścia na granicy wschodniej oznaczone są liczbami $1, \dots, w$, zaś na zachodniej liczbami $n-z+1, \dots, n$.

W kolejnych $n-1$ liniach znajduje się opis odcinków sieci kolejowej. Każda linia opisu zawiera dwie różne liczby naturalne oddzielone pojedynczym odstępem, $1 \leqslant a, b \leqslant n$. Są to numery miast-węzłów połączonych odcinkiem torów.

W $n+1$-ej linii znajduje się jedna liczba naturalna p, $1 \leqslant p \leqslant w$, $1 \leqslant p \leqslant z$, oznaczająca liczbę pociągów z odpadami. W następnej (i ostatniej) linii wejścia znajduje się p, pooddzielanych pojedynczymi odstępami, różnych liczb naturalnych, z których każda jest nie większa niż *w. Są to numery przejść granicznych na granicy wschodniej, do których przyjechały pociągi z odpadami.*

Wyjście

Pierwsza i jedyna linia wyjścia powinna zawierać dokładnie jedną liczbę całkowitą, równą minimalnej liczbie dni, jakie odpady muszą spędzić na terytorium państwa.

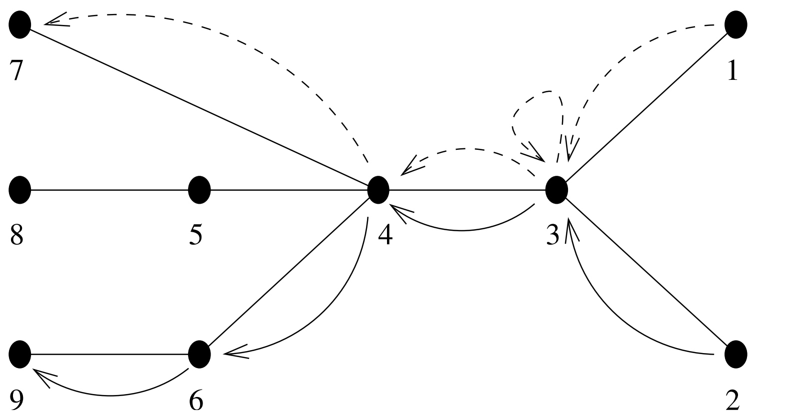
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
4
```



Strzałki przedstawiają ruch pociągów w kolejnych dniach dla jednej z optymalnych organizacji ruchu kolejowego — pętla oznacza, że danego dnia pociąg stał w mieście-węźle.

Rozwiązanie

Chwila namysłu pozwala wyrobić w sobie przekonanie, że stawiane przed nami zadanie jest w istocie problemem grafowym. I tak faktycznie jest: miasta-węzły są wierzchołkami grafu, zaś połączenia kolejowe jego krawędziami. To może w pewien sposób ukierunkować nasze poszukiwania rozwiązania, bo dla grafów opracowano już bardzo wiele efektywnych algorytmów, np. przechodzenia wszerz lub w głąb. Niemniej jednak, problem wydaje się opierać nachalnym próbom zastosowania jednego z nich — sprawia wrażenie, jakby wymagał metody znacznie rozwijającej algorytmy poszukiwania najkrótszych ścieżek.

Na szczęście okazuje się że to nieprawda, a to czego nam brakuje, to zrozumienia szczególnej struktury tego grafu. W treści zadania możemy znaleźć dwie ważne informacje:

1. z każdego miasta-węzła można dojechać do każdego innego, czyli innymi słowy graf jest **spójny**;
2. w n-węzłowej sieci kolejowej jest dokładnie $n-1$ dwukierunkowych odcinków torów.

Te dwa fakty wskazują, że graf ma strukturę acyklicznego grafu spójnego, czyli drzewa. To jest intuicyjnie oczywiste, gdy wyobrazimy sobie, że graf nie jest cały dany w jednej chwili, tylko stopniowo się rozbudowuje. Najpierw mamy n wierzchołków, a potem w $n-1$ krokach dokładamy po jednej krawędzi. Zatem, startujemy w sytuacji, gdy graf ma n spójnych składowych, a kończymy tylko z jedną składową (spójna składowa to dowolny maksymalny zbiór wierzchołków takich, że pomiędzy każdymi dwoma istnieje ścieżka). Ponieważ dołożenie krawędzi może zmniejszyć liczbę spójnych składowych grafu co najwyżej o 1, to aby dołożenie $n-1$ krawędzi uczyniło graf spójnym, każda kolejna krawędź musi łączyć dwie różne składowe, co wyklucza powstanie cyklu.

To „odkrycie" jest milowym krokiem na naszej drodze do rozwiązania zadania, bo oznacza, że każdy pociąg na jeden tylko sposób może przejechać pomiędzy każdymi dwoma miastami (oczywiście z dokładnością do wyboru miejsc postojów). W efekcie każdy odcinek toru ma jednoznacznie określone końce: wschodni i zachodni, a każda trasa pociągu przebiega odcinki od końca wschodniego w kierunku końca zachodniego (nie rozważamy cykli, gdyż można je zastąpić wielodniowym pobytem w jednym mieście-węźle).

Dla wygody nazwijmy wyróżniony odcinek sieci jej **gardłem**, a jego końce niech to będą **węzeł wschodni** i **węzeł zachodni**. Wszystkie miasta leżące na wschód od węzła zachodniego nazwijmy **miastami wschodnimi**, zaś leżące na zachód od węzła wschodniego — **miastami zachodnimi**.

Symulacja

Ponieważ nic już więcej nie wyczytamy z treści zadania, nadszedł czas na poszukiwanie rozwiązania. Zastanówmy się najpierw, jak mogłaby wyglądać organizacja ruchu kolejowego w takiej sieci, po czym spróbujemy zasymulować ten proces w naszym grafie.

Trasa każdego pociągu dzieli się niewątpliwie na dwie fazy — w pierwszej pociąg dojeżdża do węzła zachodniego (i w tej fazie może być zmuszony do postojów we wschodnich miastach-węzłach), w drugiej zaś już bez postojów może jechać do wybranego miasta na granicy zachodniej. Postawiony wymóg, by pociągi dojechały do różnych miast na granicy zachodniej, oznacza, że trzeba jeszcze określić, do którego miasta powinien udać się każdy pociąg wyjeżdżający z węzła zachodniego. To jest jednak oczywiste: ostatni pociąg powinien pojechać do najbliższego, przedostatni do drugiego najbliższego, a pierwszy do p-tego najbliższego takiego miasta.

Pozostaje jeszcze pytanie, jaką strategię należy przyjąć w pierwszej fazie. Odpowiedź brzmi banalnie i można ją ująć w trzech słowach: byle do przodu. W ewidentny sposób opłaca się by pociąg jechał, jeśli tylko potrzebny mu tor jest wolny. Gdy wiele pociągów potrzebuje pewnego toru, to oczywiście może przejechać tylko jeden, a pozostałe muszą czekać do następnego dnia. Ponieważ nie jest powiedziane, który pociąg ma dojechać do którego miasta na granicy zachodniej, możemy przepuścić dowolny z oczekujących pociągów.

To już pozwala nam naszkicować pierwszą wersję rozwiązania:

```
znajdź węzeł zachodni oraz dla każdego miasta wschodniego v znajdź
następujące po nim na trasie do węzła zachodniego miasto nast[v]
dla każdego miasta v policz odległości dist[v] od węzła zachodniego
uporządkuj miasta wschodnie w kolejności rosnących odległości
od węzła zachodniego

ile_jeszcze_jedzie := p;
dzien := 0;
odp := 0;
while ile_jeszcze_jedzie > 0 do begin
   Inc(dzien);
   for v ∈ miasta wschodnie do
      if ile_w[v] > 0 then begin
         Dec(ile_w[v]); Inc(ile_w[nast[v]]);
      end;
   if ile_w[węzeł zachodni] = 1 then begin
      odp := MAX(odp, dzien+dist[miasto docelowe tego pociągu]);
      ile_w[węzeł zachodni] := 0;
      Dec(ile_jeszcze_jedzie);
   end;
end;
```

Choć szkic ten jest daleko niepełny, bo w szczególności nie potrafimy jeszcze znaleźć węzła zachodniego, to pozwala nam ocenić tę próbę. Gołym okiem widać, że ten algorytm działa w czasie $\Omega(n^2)$, czyli jest zdecydowanie zbyt wolny.

Niewątpliwie wewnętrzna pętla **for** zamiast przeglądać wszystkie miasta wschodnie, mogłaby ograniczyć się tylko do tych, z których choć jeden pociąg chce wyruszyć na trasę. To jednak nie poprawia złożoności.

Co więc jest słabym punktem wybranego przez nas podejścia? O tym w następnej części, zatytułowanej „Rozwiązanie wzorcowe”.

Rozwiązanie wzorcowe

Wystarczy chwila zastanowienia, by na naszego wroga numer jeden wytypować symulację. Choć wiernie oddaje to, jak odbywa się ruch kolejowy, jest zbyt szczegółowa — nam wystarczy wiedzieć kiedy kolejne pociągi dojeżdżają do węzła zachodniego.

Spróbujmy teraz trochę „pogdybać”. Być może rozważenie pewnych przypadków szczególnych pozwoli nam znaleźć drogę do rozwiązania.

Gdyby... tylko jeden pociąg przewoził odpady, dzień wjazdu do węzła zachodniego byłby równy odległości (mierzonej liczbą odcinków torów) miasta na granicy wschodniej, z którego wyjechał pociąg od węzła zachodniego.

Gdyby... były dwa pociągi, moglibyśmy napotkać na pewne trudności w przypadku, gdy odległości od węzła zachodniego obu miast, z których wyjeżdżają pociągi, były takie same. Wtedy w jednym czasie próbowałyby przejechać przez wspólny odcinek toru — co najmniej jeden taki istnieje, np. gardło sieci — więc jeden z pociągów zmuszony byłby czekać. Ale potem oba mogłyby bez przeszkód dojechać do węzła zachodniego. Tym samym, jeden z pociągów dojechałby po czasie odpowiadającym odległości jaką przebył, a drugi następnego dnia. Zauważmy, że możemy na tę sytuację spojrzeć jeszcze w ten sposób, że jeden z pociągów spędza pierwszy dzień w swoim mieście na granicy wschodniej, co uniemożliwia spotkanie z drugim pociągiem, więc eliminuje też konieczność oczekiwania na zwolnienie toru.

Gdyby... powtórzyć to rozumowanie dla większej liczby pociągów? Moglibyśmy dla każdego pociągu policzyć odległość jaką musi przebyć do węzła zachodniego i potraktować te odległości jako pierwsze przybliżenia czasów dojazdu do tego węzła. Oczywiście pozostaje wymóg tego, że pociągi muszą dojeżdżać tam w różnych dniach, czyli czasy musiałyby zostać „rozrzucone”: dla każdego powtarzającego się czasu zastąpilibyśmy kolejne powtórzenia najmniejszymi „niezajętymi jeszcze” czasami.

Np. $\{2,2,2,4,7\}$ przeszłoby w $\{2,3$ `(powstało z 2)`$,4,5$ `(powstało z 2)`$,7\}$.

Nie ulega wątpliwości, że tak otrzymany plan jest optymalny, o ile tylko można go zrealizować. Na szczęście to nie przedstawia większych trudności, bo wystarczy, by każdy z pociągów odpowiednią liczbę dni (różnica pomiędzy przydzielonym mu momentem wjazdu, a odległością od węzła zachodniego) stał w mieście, z którego wyrusza. Dzięki temu żadne dwa pociągi nie spotkają się na trasie i nie będzie w związku z tym już żadnych opóźnień.

Możemy zatem zapisać szkic rozwiązania wzorcowego:

```
znajdź węzeł zachodni
dla każdego miasta granicznego policz odległości dist[v] od węzła zachodniego

ile_zachodnich := 0;
for v ∈ miasta na granicy zachodniej do begin
   Inc(ile_zachodnich);
   na_zachodzie[ile_zachodnich] := dist[v];
end;
Sort(na_zachodzie);

ile_wschodnich := 0;
nastepny_dzien := 0;
for v ∈ miasta, z których ruszają pociągi do begin
   Inc(ile_wschodnich);
   na_wschodzie[ile_wschodnich] := dist[v];
end;
Sort(na_wschodzie);
{ a teraz będzie rozrzucanie }
for i := 1 to p do begin
   na_wschodzie[i] := MAX(na_wschodzie[i], nastepny_dzien);
```

```
      nastepny_dzien := na_wschodzie[i]+1;
   end;

   odp := 0;
   for i := 1 to p do
      odp := MAX(odp, na_wschodzie[i]+na_zachodzie[p-i+1]);
```

To podejście jest już zdecydowanie lepsze od pierwszego. Wprawdzie nie powiedzieliśmy jeszcze jak znaleźć węzeł zachodni, ale reszta algorytmu ma złożoność taką jak sortowanie, czyli $O(n \log n)$, a nawet $O(n)$, jeśli zastosujemy sortowanie kubełkowe, gdyż znalezienie odległości od węzła zachodniego można wykonać dowolnym przeszukiwaniem w czasie $O(n)$.

Znajdowanie węzła zachodniego

Zajmiemy się teraz ostatnim, pomijanym do tej pory, aspektem naszego zadania, mianowicie problemem znajdowania węzła zachodniego. Na początek poszukajmy jakichś szczególnych własności tego węzła z punktu widzenia jadącego pociągu z odpadami.

Oto dość prosta i silna cecha: jest to taki węzeł, że przybywszy do niego z miasta na granicy wschodniej, można dojechać z niego do każdego miasta na granicy zachodniej, ale do żadnego na granicy wschodniej (nie rozważamy tras wielokrotnie odwiedzających pewne miasto). W przypadku gdyby istniało kilka takich wierzchołków, za węzeł zachodni może zostać uznany dowolny z nich.

Spostrzeżenie to można wykorzystać na kilka sposobów. Ciekawym pomysłem jest np. „zwijanie” drzewa sieci od strony wschodniej: odcinamy wszystkie miasta, z wyjątkiem granicznych zachodnich, o stopniu 1 oraz te, którym w tym postępowaniu stopień się zmniejszy do 1. Intuicyjnie wydaje się, że proces ten powinien doprowadzić do odcięcia wszystkich miast wschodnich i w efekcie do „wskazania” węzła zachodniego. W rzeczywistości poprawne zaimplementowanie tego pomysłu wymaga dużej dozy uwagi, choć prowadzi do krótkiego i efektywnego algorytmu. Takie rozwiązanie zostało przedstawione w pliku `wsc.cpp`.

Inną metodą, prowadzącą do wykrycia węzła zachodniego, jest przejście drzewa w głąb z dowolnego miasta na granicy wschodniej i określanie dla każdego wierzchołka, czy wszystkie znajdujące się poniżej liście odpowiadają miastom z granicy zachodniej. Najpłycej znajdujący się w drzewie przeszukiwania wierzchołek, dla którego to zajdzie, będzie dobrym węzłem zachodnim.

Oba pomysły można zrealizować w liniowej złożoności — w ten sposób upadł ostatni bastion, z którym musieliśmy się zmierzyć w tym zadaniu.

Inne rozwiązania

Inne rozwiązania, to przede wszystkim nieoptymalne realizacje podejścia wzorcowego. Wśród nich są zarówno działające w czasie $O(n \log n)$ rozwiązania „prawie optymalne” (np. wykorzystujące algorytm Quicksort zamiast sortowania kubełkowego), jak i dalece nieoptymalne kwadratowe, odpowiadające naszym pierwszym próbom znalezienia rozwiązania.

Nawet te można jednak „poprawić”, stosując dość prostą, lecz często skuteczną optymalizację. Jeżeli ze stacji odjeżdżają pociągi w kolejnych dniach, to nie pamiętamy ich wszystkich, lecz tylko dzień odjazdu pierwszego i ich liczbę. Działa to szybko, gdy pociągów jest wiele w porównaniu do liczby torów, a wszystkie mają do przebycia podobne odległości. Choć algorytm nadal jest kwadratowy, może przejść wiele testów, zwłaszcza losowych — rozwiązanie to zostało zaimplementowane w pliku `wsc2.cpp`.

Oprócz rozwiązań poprawnych istnieje oczywiście całe mnóstwo niepoprawnych, z naszego punktu widzenia zupełnie bezwartościowych.

Testy

Zadanie testowane było na zestawie 13 danych testowych.

Testy zostały wygenerowane przy pomocy programu `wscingen.cpp`. Testy 1.-10. generowane są poprzez wygenerowanie dwóch drzew podobnych rozmiarów, a następnie połączenie ich korzeni przy pomocy krawędzi — gardła sieci. Rozważmy las drzew, na początku składający się z pewnej ilości drzew jednoelementowych. Drzewo możemy wygenerować łącząc pewną liczbę drzew ze zbioru w większe drzewo. Taką operację łączenia powtarzamy aż zostanie tylko jedno drzewo. Generowane testy dzielą się na dwie klasy, ze względu na sposób łączenia.

FIFO Do tworzonego węzła podpinane są kolejne w numeracji węzły, a nowy węzeł dostaje kolejny wolny numer — w ten sposób możemy otrzymać pełne drzewo.

Random Do tworzonego węzła podpinane są losowe węzły.

Testy 11.-13. zostały zaprojektowane specjalnie po to, by odrzucić „udoskonalone” algorytmy kwadratowe. Odległości od miast na granicy wschodniej do węzła zachodniego są w nich bardzo różne, w związku z czym pociągi rzadziej jeżdżą w kolejnych dniach po tym samym torze. W testach tych zachodni węzeł jest bezpośrednio połączony z wszystkimi miastami na zachodniej granicy.

W teście 11. miasta wschodnie tworzą jedną długą linię kolejową, na której co drugie miasto jest graniczne (czyli każdym odcinkiem pociągi jeżdżą co drugi dzień). W teście 12. znajduje się drzewo binarne, przy czym jeden syn jest podpięty bezpośrednio, a drugi za pośrednictwem pewnej liczby miast. W teście 13. mamy jedną długą linię, do której w losowych miejscach podpięte są miasta ze wschodniej granicy.

Nr	typ testu	*n*	*p*	wynik
0	Test przykładowy.	9	2	4
1	Mały test poprawnościowy – Random.	16	4	6
2	Pełne drzewa binarne – FIFO.	510	100	114
3	Średni test poprawnościowy – Random.	12235	5000	5003
4	Średni test poprawnościowy – FIFO.	33520	10000	10019
5	Test wydajnościowy, pełne drzewo binarne – FIFO.	131070	10000	10030
6	Test wydajnościowy – FIFO.	122261	10000	10012
7	Test wydajnościowy – Random.	199781	40000	40011
8	Test wydajnościowy – Random.	366762	100000	100004
9	Test wydajnościowy – FIFO.	453318	100000	100011
10	Test wydajnościowy – Random.	1000000	100000	100008
11	Test wydajnościowy o specyficznej strukturze.	1000000	300000	700000
12	Test wydajnościowy o specyficznej strukturze.	1000000	32769	508438
13	Test wydajnościowy o specyficznej strukturze.	1000000	100000	799998

Piotr Chrząstowski
Treść zadania, Opracowanie

Rafał Rusin
Program

Wyspy

Bajtocja jest oblana oceanem. Na jej terenie znajdują się jeziora. Na tych jeziorach wyspy, na tych wyspach zdarzają się dalsze jeziorka, a na nich wysepki i tak dalej. Ocean ma stopień zero. Bajtocja, jako wyspa ma stopień 1. Jeziora na wyspach Bajtocji stopień 2, itd., czyli jezioro ma stopień $w+1$, jeśli znajduje się na wyspie stopnia w, a wyspa ma stopień $j+1$, jeśli znajduje się na jeziorze stopnia j. Wynika stąd oczywiście, że wszystkie stopnie wysp są nieparzyste, a jezior i oceanu parzyste.

Wszystkie jeziora i wyspy mają linie brzegowe w kształcie wielokątów o prostopadłych kolejnych bokach (równoległych do osi układu współrzędnych), a ich wierzchołki mają współrzędne całkowite. Żadne dwie linie brzegowe nie przecinają się, ani nie stykają się.

Mając dane kontury linii brzegowych, wyznacz maksymalny stopień wyspy/jeziora w Bajtocji.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opisy linii brzegowych wysp i jezior,*
- *obliczy maksymalny stopień jeziora/wyspy,*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu wejścia zapisana jest jedna dodatnia liczba całkowita n, liczba linii brzegowych, $1 \leqslant n \leqslant 40000$. Linie brzegowe są opisane w kolejnych wierszach, po jednej w wierszu. Każdy z tych wierszy zawiera nieujemne liczby całkowite pooddzielane pojedynczymi odstępami. Pierwsza liczba w wierszu to k, parzysta liczba punktów tworzących linię brzegową, $4 \leqslant k \leqslant 10000$. Dalej w wierszu znajduje się k liczb: $x_1, x_2, \ldots, x_k$, $0 \leqslant x_i \leqslant 10^8$. Kolejne punkty tworzące linię brzegową to: (x_1,x_2), (x_3,x_2), (x_3,x_4), (x_5,x_4), $\ldots (x_{k-1},x_k)$, (x_1,x_k). Są podane we współrzędnych kartezjańskich oraz opisują linię brzegową lewoskrętnie (czyli idąc z punktu i do $i+1$, wnętrze mamy po lewej stronie). Linie brzegowe są podane w takiej kolejności, że:

- *linia brzegowa każdego jeziora jest podana zawsze po linii brzegowej wyspy, na której się znajduje,*
- *linia brzegowa każdej wyspy jest podana zawsze po linii brzegowej jeziora, na którym się znajduje.*

Do opisania całej mapy użyto nie więcej niż 200000 punktów.

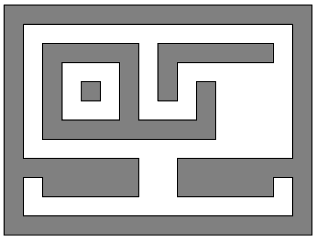
Wyjście

Twój program powinien wypisać w pierwszym i jedynym wierszu wyjścia jedną liczbę całkowitą: maksymalny stopień jeziora/wyspy.

Przykład

Dla danych wejściowych:

```
6
4 1 0 17 12
16 10 4 16 11 2 4 8 2 3 3 2 1 16 3 15
2
8 8 10 3 5 12 8 11 6
4 4 6 7 9
4 6 8 5 7
6 10 9 15 10 9 7
```



poprawnym wynikiem jest:

```
5
```

Rozwiązanie

Zróbmy kilka początkowych spostrzeżeń. Po pierwsze możemy się skupić jedynie na odcinkach poziomych. Ponieważ każdy koniec odcinka pionowego jest jednocześnie końcem pewnego odcinka poziomego to skupiając się tylko na odcinkach poziomych nie przeoczymy żadnego fragmentu brzegu. Patrząc tylko na odcinki poziome, ponumerowane w kolejności ich występowania na brzegu, możemy sobie w jednoznaczny sposób odtworzyć położenie brakujących odcinków pionowych.

Kolejnym spostrzeżeniem jest to, że jeśli z lewego końca pewnego odcinka spojrzymy w górę, to albo nie zobaczymy już żadnej linii brzegowej i wtedy ten odcinek jest brzegiem Bajtocji i ma stopień 1, albo zobaczymy brzeg bezpośrednio nad nami. Są wtedy trzy przypadki:

- Odcinek, który zobaczyliśmy bezpośrednio nad nami należy do tego samego brzegu, co nasz odcinek i wtedy oczywiście ograniczamy ten sam obszar.
- Odcinek ten należy do innego brzegu i jest skierowany na prawo. Wtedy oznacza to, że jego wnętrze jest po jego drugiej stronie i tamten odcinek należy do brzegu innego obszaru, ale mającego ten sam stopień, co nasz obszar.
- Odcinek ten należy do innego brzegu i jest skierowany na lewo. Wtedy oznacza to, że jesteśmy podobszarem obszaru otoczonego tamtym brzegiem, zatem nasz stopień będzie o 1 większy.

W rzeczywistości można nawet nieco uprościć rozumowanie: to, czy odcinek sąsiedni z góry jest z tego samego brzegu nie ma znaczenia, liczy się tylko jego zwrot: jeśli jest w prawo, to nie zmieniamy stopnia, jeśli jest w lewo, to nasz stopień będzie o 1 większy.

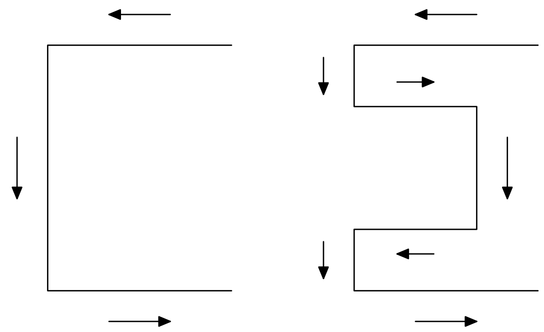
Formalny dowód tych spostrzeżeń wymaga dość głębokiej wiedzy z dziedziny topologii i wykracza poza program szkolny; na szczęście są to fakty na tyle intuicyjne, że można je w miarę sensownie uzasadnić.

Zrobimy to metodą indukcyjną. Każdą linię brzegową można przedstawić jako kształt powstały z prostokąta przez wciskanie albo wyciskanie z niego kawałków brzegu. Przy każdym takim wciśnięciu lub wyciśnięciu dodajemy prostopadłe krawędzie łączące wciśnięty odcinek z resztą brzegu (lub w razie potrzeby usuwamy kawałki brzegu, jeśli nie ma z czym łączyć). Dowód teraz polega na tym, żeby pokazać, że dla linii brzegowych będących prostokątami zachodzi teza (baza indukcji), a następnie, że pojedyncze wciśnięcie odcinka nie zmienia prawdziwości tezy. Indukcja będzie zatem względem liczby wciśnięć potrzebnych do uzyskania zadanego kształtu linii brzegowych.

To, że dla linii brzegowych będących prostokątami teza zachodzi, powinno być dość oczywiste. Prosta indukcja względem liczby prostokątów powinna nas o tym przekonać i tę część uzasadnienia pominiemy.

Wracając do uzasadnienia naszej tezy zauważmy, że wciśnięcie odcinka poziomego nie zmienia jej prawdziwości: nie może się on wszakże przeciąć z żadnym innym odcinkiem, więc jego sąsiedztwo z góry się nie zmieni. Pozostanie on też sąsiadem dla swoich „kolegów” z dołu. Nikt się między nich „nie wetnie” i nie rozerwie tej zależności między stopniami, którą ustaliliśmy na podstawie założenia indukcyjnego.

W przypadku odcinków pionowych dodane nowe poziome krawędzie mogą przysłonić sąsiedztwo pionowe dla odcinków, które znajdują się na dole. Ze względu na symetrię sytuacji ustalmy, że „wciskamy” w stronę prawą odcinek znajdujący się na lewej krawędzi konturu, czyli krawędzi skierowanej do dołu — patrz rysunek poniżej. Dla pozostałych 3 przypadków, kiedy lewą krawędź konturu wciskamy w lewą stronę oraz gdy prawą krawędź konturu wciskamy w stronę prawą bądź lewą, rozumowanie przebiega analogicznie.



Kluczową obserwacją jest tu spostrzeżenie, że wciśnięcie takie zawsze prowadzi do sytuacji, w której nad odcinkiem skierowanym w prawo będzie odcinek skierowany w lewo, tak jak przed wciśnięciem, i nad każdym odcinkiem skierowanym w lewo będzie odcinek skierowany w prawo. Zatem w ramach jednego obszaru zawsze zwroty sąsiednich w pionie odcinków będą różne. Jeśli Czytelnik został tu przekonany, to dobrze, choć nie jest to ścisły dowód.

Natomiast przypadek, kiedy jeden obszar jest podobszarem drugiego rozpoznajemy po tym, że każdy odcinek podobszaru, jest jednakowo skierowany, jak dowolny jego sąsiad z obszaru go otaczającego. Sąsiedztwo rozumiemy tu jako najbliższy odcinek przecięty przez prostą prostopadłą do badanego odcinka.

Aby zrealizować sprawdzenie przedstawionych warunków, zastosujemy narzucającą się tu technikę *zamiatania*, opisywaną już w materiałach z poprzednich olimipad[1]. Technika ta, bardzo typowa dla wielu zadań geometrii obliczeniowej, wprowadza porządek przy przeglądaniu badanych obiektów na płaszczyźnie i pozwala nie uronić żadnego z nich, organizując ich przetwarzanie w efektywny sposób. Interesujące nas poziome odcinki tworzące linie brzegowe posortujemy od lewej do prawej względem ich współrzędnych x-owych, a w przypadku równych współrzędnych x-owych, względem współrzędnych y-kowych od góry do dołu. Każdy odcinek będzie zatem występował dwukrotnie: raz reprezentowany przez lewą współrzędną, a raz przez prawą. Następnie pionową miotłą będziemy zamiatali płaszczyznę od lewej do prawej, pobierając (i w miarę potrzeby usuwając) odcinki po kolei, zgodnie z naszym porządkiem. W każdym momencie w pionowej miotle będą się znajdowały te odcinki, które ją aktualnie przecinają, a punktami zatrzymania się miotły w marszu od lewej do prawej będą x-owe współrzędne odcinków. Zauważmy, że zgodnie z warunkami zadania wszystkie współrzędne y znajdujące się w miotle muszą być różne.

Pierwszy odcinek, który napotkamy będzie oczywiście częścią linii brzegowej Bajtocji — najbardziej na zachód wysuniętym cyplem. Może zresztą być kilka równie daleko wysuniętych odcinków, co nam w niczym nie przeszkodzi; będziemy tylko pamiętać, żeby przetwarzać takie odcinki o równych lewych współrzędnych z góry na dół. Pierwszemu odcinkowi nadamy stopień $d = 1$ i włożymy do pustej miotły. Dalsze odcinki poziome będziemy przetwarzali zgodnie z następującą zasadą. Jeżeli napotkamy prawy koniec odcinka, to odnajdujemy ten odcinek w miotle i usuwamy. Jeżeli natomiast napotkamy lewy koniec odcinka, to patrzymy w górę. Niech s będzie najbliższym od góry sąsiadem badanego odcinka o znajdującym się nad jego lewym końcem. Niech brzeg, do którego należy odcinek s ma stopień d. Zachodzą teraz dwa przypadki:

- jeśli zwrot s jest taki sam, jak zwrot o (czyli na prawo), to umieszczamy w miotle odcinek o z numerem poziomu $d+1$. Oznacza to, że nasze wnętrze jest bezpośrednim podwnętrzem wnętrza ogarniętego przez brzeg, do którego należy odcinek s;

- jeśli zwrot s jest przeciwny do zwrotu odcinka o, to umieszczamy w miotle odcinek o z numerem poziomu d. Oznacza to, że nasze wnętrze jest wnętrzem sąsiednim do wnętrza ogarniętego przez brzeg, do którego należy odcinek s.

Pozostają jeszcze szczegóły implementacyjne. Miotła powinna się dać szybko zainicjalizować, a ponadto sprawnie wykonywać następujące operacje:

- odszukanie odcinka w miotle o największej współrzędnej nieprzekraczającej zadanego y, bądź stwierdzenie że takiego odcinka nie ma;

- wstawienie do miotły zadanego odcinka;

- usunięcie z miotły zadanego odcinka.

Musimy też pamiętać, żeby do miotły wstawiać odcinki z zaznaczeniem ich stopnia i zwrotu.

[1]Patrz „Łamane płaskie" — V OI, „Lodowisko" — VI OI, „Kopalnia złota" — VIII OI

Przyjmijmy następujące oznaczenia. Niech n oznacza liczbę linii brzegowych, k_i liczbę punktów tworzących i-tą linię brzegową, zaś $m = \sum_{i=1}^{n} k_i$ liczbę wszystkich punktów. Zauważmy, że w pesymistycznym przypadku liczba odcinków przechowywanych w miotle będzie rzędu m.

Jeśli zaimplementujemy miotłę na przykład w uporządkowanej tablicy, to choć samo wyszukiwanie odcinka względem posortowanych współrzędnych y zajmie nam czas rzędu $\log m$, to operacje wstawiania i usuwania odcinka w pesymistycznym czasie będą rzędu m. Przy $m/2$ odcinkach dostaniemy algorytm kwadratowy.

Podobne kłopoty będziemy mieli ze strukturami listowymi. Tam też nie sposób jest zrealizować w modelu list prostych wszystkich potrzebnych nam operacji w czasie mniejszym niż liniowy, a to oznacza, że koszt całego algorytmu też będzie kwadratowy.

Dobrą strukturą dla miotły są np. drzewa AVL, o których można się dowiedzieć np. z książki [14]. Implementacja ich jest jednak dość uciążliwa, choć niektórzy ćwiczą ją sobie w ramach przygotowań do olimpiady. Za pomocą drzew AVL każdą z potrzebnych operacji daje się zrealizować w pesymistycznym czasie rzędu $\log m$. Ponieważ dla każdego z m końców odcinków będziemy wykonywali każdą z tych operacji co najwyżej raz, więc łączny koszt tej fazy wynosi $O(m \log m)$. Wczytanie danych zajmuje czas liniowy ze względu na m, a początkowe posortowanie którymś z szybkich algorytmów sortujących daje się zrobić w czasie rzędu $m \log m$. Zatem taki jest rząd złożoności całego algorytmu. Oczywiście odcinki wkładamy do miotły z zaznaczeniem ich stopnia oraz zwrotu.

W programie wzorcowym, znajdującym się na płycie, zastosowany został wybieg implementacyjny znacznie upraszczający sprawę. Zastosowano bowiem strukturę `<map>` ze standardowej biblioteki STL języka `C++`. Tam po prostu za darmo dostajemy logarytmiczną implementację potrzebnych nam operacji. Oczywiście znajomość odpowiednich bibliotek oszczędzi nam też problemu szybkiego posortowania danych.

Testy

Opisy testów zawierają umowne nazwy figur.

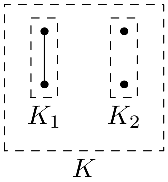
nr	opis
0	przykładowy test z treści zadania
1	spirala z prostokącikami
2	mała szachownica
3	mały ciąg prostokątów zawartych w sobie - "rura"
4	spirala z prostokącikami
5	krzywa Hilberta st. 4 z prostokącikami
6	"drzewo" prostokątów
7	"rura", w niej spirala, w spirali "rura"
8	duża szachownica, "klucz" i spirala
9	"klucz" i "drzewo" złożone z prostokątów, szachownic, spiral i "rura"
10	spirala, "rura" i klucz
11	"klucz" i spirala
12	"klucz" i spirala

Tomasz Waleń **Rafał Rusin**
Treść zadania, Opracowanie Program

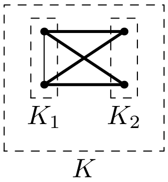
Kaglony

Kaglony *to narodowa ulubiona potrawa mieszkańców Bajtocji. Kaglony mają bardzo charakterystyczną budowę. Glon składający się z jednej komórki jest kaglonem. Mając dwa kaglony* K_1 *i* K_2*, można je połączyć w następujący sposób:*

- *biorąc wszystkie komórki z* K_1 *i* K_2*, oraz wszystkie połączenia z* K_1 *i* K_2*,*



- *biorąc wszystkie komórki z* K_1 *i* K_2*, wszystkie połączenia z* K_1 *i* K_2*, oraz dodając nowe połączenia: każdą komórkę z* K_1 *łączymy z każdą komórką z* K_2*.*



Otrzymujemy w wyniku nowy kaglon K*.*

Niestety niedawno wrogie państwo Bitocji rozpoczęło sprzedaż glonów imitujących kaglony. Glony te są na tyle podobne, że na pierwszy rzut oka trudno odróżnić je od oryginału, dlatego też rząd Bajtocji poprosił Cię o napisanie programu, który umożliwiłby sprawdzanie czy dany glon jest kaglonem.

Zadanie

Napisz program który:

- *wczyta ze standardowego wejścia opisy glonów,*
- *sprawdzi, które z nich są poprawnymi kaglonami,*
- *zapisze na standardowym wyjściu odpowiedź.*

Wejście

W pierwszym wierszu standardowego wejścia zapisana jest jedna liczba całkowita k*,* $1 \leqslant k \leqslant 10$*, liczba badanych glonów. W kolejnych wierszach zapisane jest* k *opisów glonów. Pojedynczy opis ma następującą postać: w pierwszym wierszu zapisane są dwie liczby* *całkowite* n *i* m*, oddzielone pojedynczym odstępem,* $1 \leqslant n \leqslant 10\,000$*,* $0 \leqslant m \leqslant 100\,000$*, odpowiednio liczba komórek i liczba połączeń. Komórki są ponumerowane od* 1 *do* n*. W kolejnych* m *wierszach opisane są połączenia, w każdym z tych wierszy zapisano dwie liczby całkowite* a*,* b *oddzielone pojedynczym odstępem,* $a \neq b$*,* $1 \leqslant a,b \leqslant n$*, oznaczające, że komórki* a *i* b *są połączone. Każde połączenie wymienione jest jeden raz.*

Wyjście

Na standardowym wyjściu należy zapisać k *wierszy. W* i*–tym wierszu należy zapisać jedno słowo:*

- `TAK` — *jeśli* i*–ty glon jest poprawnym kaglonem,*
- `NIE` — *w przeciwnym przypadku.*

Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

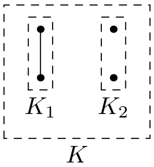
```
TAK
NIE
TAK
```

Rozwiązanie

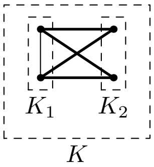
Najprostsze rozwiązanie

Dobrym dowodem na to, że dany graf jest kaglonem, może być drzewo opisujące sposób otrzymania grafu. Liście tego drzewa odpowiadają wierzchołkom grafu, a węzły wewnętrzne odpowiadają operacjom, które łączą grafy z poddrzew. Zgodnie z treścią zadania istnieją dwie metody łączenia poddrzew:

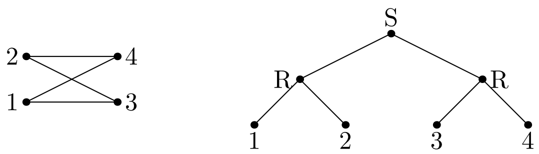
- równoległa — łączymy grafy z poddrzew, nie dodając żadnych dodatkowych krawędzi;



- szeregowa — łączymy grafy z poddrzew, ale tym razem dodajemy krawędzie pomiędzy każdą parą wierzchołków z różnych poddrzew.



Na rysunku 1 przedstawiono *ka–drzewo* dla przykładowego grafu.



Rysunek 1: Przykładowe ka–drzewo

Dopełnieniem grafu $G = (V, E)$ będziemy nazywać graf $G' = (V, E')$ o tym samym zbiorze wierzchołków V i krawędziach $E' = \{(u, v) : (u, v) \notin E\}$.

Łatwo zauważyć, że gdy dla kaglonu G, ostatnią operacją było połączenie:

- równoległe — graf G składa się z co najmniej dwóch spójnych składowych,
- szeregowe — graf G' (dopełnienie G) składa się z co najmniej dwóch spójnych składowych.

Jeśli graf G ($|G| > 1$) nie spełnia żadnego z tych warunków, czyli jednocześnie G i G' są spójne — to graf G nie jest kaglonem.

Powyższe rozumowanie prowadzi do następującego algorytmu na sprawdzanie, czy graf G jest kaglonem:

function CzyKaglon(G)
begin
    if $|G|$=1 **then** **return** TRUE;
    wyznacz spójne składowe G — $G_1, \ldots, G_k$;
    if $k > 1$ **then**
        return CzyKaglon(G_1) **and** … **and** CzyKaglon(G_k);
    else begin
        wyznacz G' — dopełnienie grafu G;
        wyznacz spójne składowe G' — $G'_1, \ldots, G'_j$
        oraz odpowiadające im podgrafy G — $G_1, \ldots, G_j$

```
        if j = 1 then
            return FALSE;
        else
            return CzyKaglon(G_1) and ... and CzyKaglon(G_j);
    end
end
```

Niestety takie rozwiązanie jest dosyć wolne, wymaga $O(n^3)$ czasu oraz $O(n^2)$ pamięci (w pesymistycznym przypadku graf G' może mieć rozmiar $\Omega(n^2)$).

Rozwiązanie wzorcowe

Problem rozpoznawania kaglonów można rozwiązać nawet w czasie $O(n+m)$, jednak takie rozwiązanie jest dosyć skomplikowane. Dalej opiszemy rozwiązanie o trochę gorszej złożoności czasowej, jednak znacznie prostsze. Rozwiązanie składa się z dwóch kroków:

- wyznaczenia pewnego obiektu kombinatorycznego, który miałby świadczyć o tym, że graf jest ka–glonem;
- a następnie weryfikacji poprawności takiego świadka (jeśli graf nie jest ka–glonem, to w tym kroku się o tym przekonamy).

Bardzo dobrym świadkiem na to, że graf jest ka–glonem, mogłoby być ka–drzewo. Niestety efektywne wyznaczenie ka–drzewa jest zadaniem dosyć skomplikowanym. W naszym rozwiązaniu świadkiem będzie taka permutacja wierzchołków grafu, która odpowiada kolejności liści w pewnym ka–drzewie badanego grafu. Taką permutację będziemy nazywać *permutacją faktoryzującą*. W rozdziale *Weryfikacja* opisany jest algorytm, który pozwala sprawdzić w czasie liniowym czy dla danej permutacji wierzchołków istnieje odpowiadające mu ka–drzewo.

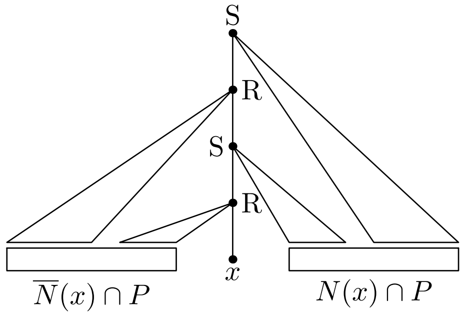
Obliczenie permutacji faktoryzującej

Początkowo nie dysponujemy żadną wiedzą o kolejności wierzchołków w permutacji faktoryzującej, czyli rozpoczynamy obliczenia z $P = (V)$. W trakcie obliczeń uzyskujemy dodatkowe informacje na temat kolejności wierzchołków, jednak nadal nie znamy dokładnego ich uporządkowania, w takiej sytuacji częściowo obliczoną permutację możemy reprezentować jako sekwencję rozłącznych podzbiorów wierzchołków:

$$P = (C_1, C_2, \ldots, C_k),\ \text{gdzie } C_i \subseteq V$$

Oczywiście każdy wierzchołek $v \in V$ należy do dokładnie jednego ze zbiorów C_i. Naszym celem jest takie przekształcenie P, by każdy ze zbiorów C_i miał rozmiar 1, czyli P stał się permutacją wierzchołków V.

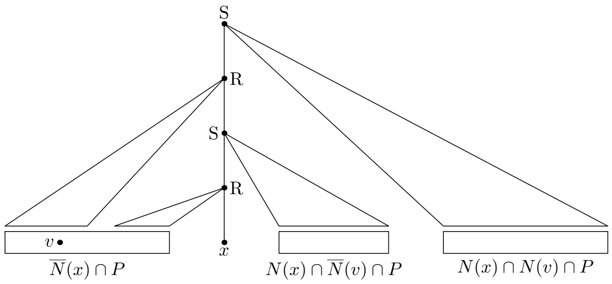
W pierwszym kroku algorytmu (gdy sekwencja P jest po prostu zbiorem wierzchołków), wybieramy dowolny wierzchołek x i dzielimy P na $(P \cap \overline{N}(x), \{x\}, P \cap N(x))$, gdzie $N(x)$ oznacza zbiór sąsiadów wierzchołka x, a $\overline{N}(x)$ zbiór tych wierzchołków, które nie sąsiadują z x. Dlaczego możemy tak arbitralnie ustalić kolejność wierzchołków? Zauważmy, że dowolne ka–drzewo możemy tak uporządkować, by zawierało liście właśnie w tej kolejności — $P \cap \overline{N}(x), \{x\}, P \cap N(x)$.



Rysunek 2: Podział zbioru P na $P \cap \overline{N}(x), \{x\}, P \cap N(x)$

Jednak taki podział wierzchołków nie jest jeszcze wystarczający — nie doprowadziliśmy jeszcze do sytuacji w której następne sprawdzenia będą wykonywane rekurencyjnie.

Tak więc należy uporządkować zbiory $P \cap \overline{N}(x), P \cap N(x)$, tak by każdy z nich zawierał wierzchołki z poddrzew wyznaczonych przez przodków wierzchołka x w ka–drzewie. Wybierając dowolny wierzchołek v z $\overline{N}(x)$ możemy podzielić wierzchołki z $N(x)$ na te, które nie są połączone z v i na te, które sąsiadują z v — czyli $N(x) \cap \overline{N}(v)$ i $N(x) \cap N(v)$. Powyższe rozumowanie przedstawione jest na rysunku 3. Podobny podział zbioru $\overline{N}(x)$ można uzyskać używając jako wierzchołków rozdzielających $v \in N(x)$.



Rysunek 3: Wynik operacji $Polepsz(v, (\overline{N}(x), \{x\}, N(x)))$

Bardziej formalnie procedura podziału zbiorów na wierzchołki poddrzew ma następującą postać:

```
function Polepsz(v, P)
begin
    for C_i ∈ P do
        if C_i ∩ N(v) ≠ C_i and C_i ∩ N(v) ≠ ∅ then
            zastąp C_i, przez zbiory C_i ∩ N̄(v), C_i ∩ N(v) (w takiej kolejności)
end
```

Gdy wykonamy już podziały dla wszystkich wierzchołków z $v \in N(x), \overline{N}(x)$, zbiory $C_i \in P$ będą odpowiadać wierzchołkom poddrzew o korzeniach będących przodkami wierzchołka x. Stąd dla każdego z poddrzew (zbiorów C_i) można rekurencyjnie obliczyć odpowiadającą mu permutację. Dzięki wielokrotnemu zastosowaniu powyższej metody uzyskujemy coraz dokładniejszą informację na temat permutacji P. Pełny kod procedury obliczania permutacji faktoryzującej ma następującą postać:

```
function Kaglon(x,P)
begin
    if |P|=1 then  return (x);
    P:=(N̄(x),{x},N(x));
    niech A mniejszy ze zbiorów N̄(x), N(x), a B większy;
    for x ∈ A do
        Polepsz(x, P)
    end;
    for C_i ∈ P ∩ B do
        niech y będzie dowolnym wierzchołkiem należącym do C_i;
        Polepsz(y, P);
        C'_i:=Kaglon(y, C_i);
        zastąp C_i przez C'_i w P
    end;
    for C_i ∈ P ∩ A do
        niech y będzie dowolnym wierzchołkiem należącym do C_i;
        C'_i:=Kaglon(y, C_i);
        zastąp C_i przez C'_i w P
    end;
    return P
end
```

Procedura $Polepsz(x,P)$ wymaga czasu proporcjonalnego do $N(x)$. Teraz wystarczy zauważyć, że każdy wierzchołek może być argumentem dla procedury *Polepsz* co najwyżej $O(\log n)$ razy — wierzchołek ma szansę na wielokrotne wykorzystanie tylko, gdy należy do mniejszego ze zbiorów $\overline{N}(x)$, $N(x)$ — czyli zbioru A. Stąd całkowity czas potrzebny na obliczenie permutacji faktoryzującej jest rzędu $O(n+m\log n)$.

Weryfikacja

O dwóch wierzchołkach x, y mówimy, że są bliźniakami ($twins(x,y)$), jeśli $N(x) = N(y)$ lub $N(x) \cup \{x\} = N(y) \cup \{y\}$. Gdy przeanalizujemy postać ka–drzewa, można zauważyć, że dwa wierzchołki x, y są bliźniakami wtedy i tylko wtedy, gdy są braćmi w ka–drzewie. Stąd pojęcie "bliźniaków" może być pomocne w weryfikacji poprawności permutacji faktoryzującej — wystarczy przeglądać permutację z lewa na prawo i lokalizować sąsiadujące wierzchołki.

```
i:=0;
z:=x_1; { pierwszy wierzchołek w permutacji P }
while i < n−1 and z ≠ x_n do begin
```

```
    if twins(z, prev(z)) then
        usuń prev(z) z P;
        i:=i+1
    else if twins(z, next(z)) then
        usuń z z P;
        z:=next(z);
        i:=i+1
    else
        z:=next(z)
end;
if P=1 then
    return TRUE;
else
    return FALSE;
```

Jeśli permutacja zostanie zredukowana do pojedynczego wierzchołka oznacza to, że permutacja odpowiada pewnemu ka–drzewu, a więc graf G jest kaglonem. Weryfikacja permutacji wymaga czasu $O(n+m)$.

Testy

Zadanie testowane było na zestawie 10 danych testowych, których opisy zawiera poniższa tabela.

Nr	k	max n	max m
1	10	8	14
2	10	169	216
3	10	10000	40
4	10	9910	99000
5	10	9910	99125
6	10	9910	99002
7	10	10000	99000
8	10	9521	95210
9	10	10000	99894
10	10	450	52364

Jakub Pawlewicz **Paweł Wolff**
Treść zadania, Opracowanie Program

Maksymalne rzędy permutacji

Permutacją n-elementową nazywamy różnowartościową funkcję

$$\pi : \{1,2,\ldots,n\} \longmapsto \{1,2,\ldots,n\}.$$

Rzędem permutacji π nazywamy najmniejsze takie $k \geqslant 1$, że dla wszystkich $i = 1,2,\ldots,n$ zachodzi:

$$\underbrace{\pi(\pi(\ldots(\pi}_{k \text{ razy}}(i))\ldots)) = i$$

Na przykład, rzędem trzyelementowej permutacji $\pi(1) = 3, \pi(2) = 2, \pi(3) = 1$ jest 2, bo $\pi(\pi(1)) = 1, \pi(\pi(2)) = 2, \pi(\pi(3)) = 3$.

Dla zadanego n rozważmy permutacje n-elementowe o największym możliwym rzędzie. Na przykład maksymalny rząd permutacji pięcioelementowej wynosi 6. Przykładem permutacji pięcioelementowej, której rząd wynosi 6 jest $\pi(1) = 4, \pi(2) = 5, \pi(3) = 2, \pi(4) = 1, \pi(5) = 3$.

Spośród wszystkich permutacji n-elementowych o maksymalnym rzędzie chcemy znaleźć permutację najwcześniejszą (w porządku leksykograficznym). Dokładniej, mówimy, że permutacja n-elementowa π jest wcześniejsza niż permutacja n-elementowa σ, gdy istnieje takie i, że $\pi(j) = \sigma(j)$ dla argumentów $j < i$ oraz $\pi(i) < \sigma(i)$. Najwcześniejszą permutacją pięcioelementową o rzędzie 6 jest $\pi(1) = 2, \pi(2) = 1, \pi(3) = 4, \pi(4) = 5, \pi(5) = 3$.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia zestaw liczb całkowitych $n_1, n_2, \ldots, n_d$,*
- *dla każdej liczby n_i (dla $i = 1,2,\ldots,d$) wyznaczy najwcześniejszą n_i-elementową permutację o maksymalnym rzędzie,*
- *wypisze na standardowe wyjście wyznaczone permutacje.*

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna dodatnia liczba całkowita d, $1 \leqslant d \leqslant 10$. W kolejnych d wierszach znajdują się dodatnie liczby całkowite $n_1, n_2, \ldots, n_d$, po jednej w wierszu, $1 \leqslant n_i \leqslant 10\,000$.

Wyjście

Twój program powinien wypisać na standardowe wyjście d wierszy. Wiersz nr i powinien zawierać ciąg liczb całkowitych oddzielonych pojedynczymi odstępami, będący ciągiem wartości $\pi(1), \pi(2), \ldots, \pi(n_i)$ najwcześniejszej permutacji n_i-elementowej o maksymalnym rzędzie.

Przykład

Dla danych wejściowych:
```
2
5
14
```
poprawnym wynikiem jest:
```
2 1 4 5 3
2 3 1 5 6 7 4 9 10 11 12 13 14 8
```

Rozwiązanie

Podstawowe pojęcia i fakty

Zanim przystąpimy do omawiania problemu, należy przypomnieć parę podstawowych pojęć i faktów.

NWD i NWW

Definicja 1 *Największym wspólnym dzielnikiem* liczb całkowitych $a_1, \ldots, a_k$ nazwiemy największą liczbę całkowitą, która dzieli wszystkie a_i, dla $i = 1, \ldots, k$:

$$\mathrm{NWD}(a_1, \ldots, a_k) = \max\{d : d \mid a_i \text{ dla wszystkich } i = 1, \ldots, k\}.$$

Definicja 2 Powiemy, że liczby całkowite a i b są *względnie pierwsze*, jeśli nie mają wspólnego dzielnika większego od 1, czyli $\mathrm{NWD}(a,b) = 1$.

Definicja 3 *Najmniejszą wspólną wielokrotnością* liczb całkowitych $a_1, \ldots, a_k$ nazwiemy najmniejszą dodatnią liczbę całkowitą, która jest podzielna przez każdą z liczb a_i, dla $i = 1, \ldots, k$.

$$\mathrm{NWW}(a_1, \ldots, a_k) = \min\{d > 0 : a_i \mid d \text{ dla wszystkich } i = 1, \ldots, k\}.$$

Fakt 1 Zachodzą następujące własności NWW:

(i) Dla liczb całkowitych $b, c, a_1, \ldots, a_k$:

$$\mathrm{NWW}(b, c, a_1, \ldots, a_k) = \mathrm{NWW}(\mathrm{NWW}(b,c), a_1, \ldots, a_k).$$

(ii) Jeśli a i b są względnie pierwsze, to:

$$\mathrm{NWW}(a,b) = ab.$$

Permutacje, rozkłady na cykle, rząd permutacji

Definicja 4 *Cyklem długości c* w permutacji π nazwiemy taki ciąg indeksów $(i_1 \cdots i_c)$, że $\pi(i_1) = i_2, \pi(i_2) = i_3, \ldots, \pi(i_{c-1}) = i_c, \pi(i_c) = i_1$.

Definicja 5 *Rozkładem na cykle* permutacji n–elementowej π nazwiemy takie rozbicie zbioru $\{1,\ldots,n\}$ na ciągi $(i_{1,1}\cdots i_{1,c_1})(i_{2,1}\cdots i_{2,c_2})\cdots(i_{l,1}\cdots i_{l,c_l})$, że każdy z ciągów $(i_{j,1}\cdots i_{j,c_j})$, dla $j=1,\ldots,l$, jest cyklem długości c_j w permutacji π.

Przykład 1 Dla permutacji $\pi(1)=4, \pi(2)=5, \pi(3)=2, \pi(4)=1, \pi(5)=3$ rozkład na cykle wynosi: $(1\,4)(2\,5\,3)$.

Teraz w inny sposób zdefiniujemy rząd permutacji.

Definicja 6 Permutację powstałą przez k–krotne złożenie n–elementowej permutacji π ze sobą, dla $k \geqslant 1$, oznaczamy π^k:

$$\pi^k(i) = \underbrace{\pi(\pi(\ldots(\pi}_{k \text{ razy}}(i))\ldots)) \qquad \text{dla } i=1,\ldots,n.$$

Definicja 7 n–elementową permutację *identycznościową* oznaczamy przez id:

$$\mathrm{id}(i) = i \qquad \text{dla } i=1,\ldots,n.$$

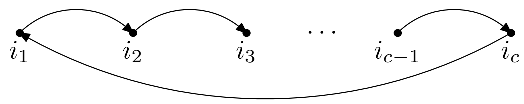
Definicja 8 *Rzędem* permutacji π nazywamy najmniejsze $k \geqslant 1$, dla którego zachodzi $\pi^k \equiv \mathrm{id}$:

$$\mathrm{rz}\,\pi = \min\{k \geqslant 1 : \pi^k \equiv \mathrm{id}\}.$$

Aby zrozumieć istotę problemu, jaka zaszyta jest w treści zadania, pokażemy w jaki sposób liczy się rząd permutacji.

Lemat 1 *Niech permutacja π zawiera cykl $(i_1\, i_2 \cdots i_c)$ długości c. W permutacji π^k, $k \geqslant 1$, elementy cyklu przejdą na siebie wtedy i tylko wtedy, gdy k jest wielokrotnością c:*

$$\pi^k(j) = j \textit{ dla } j \in \{i_1,\ldots,i_c\} \textit{ wtw, gdy } c \mid k.$$



Rysunek 1: Cykl w permutacji

Dowód Na cykl w permutacji możemy patrzeć jak na graf, którego krawędzie są postaci $i \longrightarrow \pi(i)$ (Rysunek 1). Oczywiste jest, że każda ścieżka z danego wierzchołka do tego samego wierzchołka będzie miała długość, która jest wielokrotnością długości cyklu. ∎

Fakt 2 Rząd permutacji π, której cykle są długości $c_1,\ldots,c_l$, wyraża się wzorem:

$$\mathrm{rz}\,\pi = \mathrm{NWW}(c_1,\ldots,c_l).$$

Dowód Z lematu 1 wynika, że w permutacji π^k, j–ty cykl, $j=1,\ldots,l$, przejdzie na siebie wtedy i tylko wtedy, gdy k jest wielokrotnością c_j. Zatem rzędem π będzie najmniejsza wspólna wielokrotność $c_1,\ldots,c_l$. ∎

Przykład 2 Permutacja z przykładu 1 ma cykle długości 2 i 3, więc jej rząd wynosi $\mathrm{NWW}(2,3)=6$.

Postać maksymalnego rzędu permutacji n–elementowej

W tej sekcji przyjrzymy się jaką postać ma maksymalny rząd permutacji.

Definicja 9 Skończony ciąg liczb całkowitych dodatnich $a_1, \ldots, a_l$ nazwiemy *podziałem* liczby n, jeśli zachodzi:

$$\sum_{i=1}^{l} a_i = n.$$

Z faktu 2 wynika, że musimy szukać permutacji n–elementowej, w taki sposób, żeby $\mathrm{NWW}(c_1, \ldots, c_l)$ było największe, gdzie $c_1, \ldots, c_l$ oznaczają długości cyklów permutacji. Zatem szukanie maksymalnego rzędu sprowadza się do maksymalizowania wartości $\mathrm{NWW}(c_1, \ldots, c_l)$ po wszystkich podziałach $c_1, \ldots, c_l$ liczby n.

Przechodzimy do omówienia własności podziałów n o maksymalnej NWW. Posłużymy się oczywistą nierównością:

Fakt 3 Dla liczb całkowitych $2 \leqslant x < y$ zachodzi nierówność:

$$x + y < xy. \tag{11}$$

Dowód $2 < y \Leftrightarrow y + 2 < 2y \Leftrightarrow y < 2(y-1) \Rightarrow y < x(y-1) \Leftrightarrow x + y < xy.$ ■

Wśród podziałów n o maksymalnej NWW, jak się później okaże, będą nas interesowały podziały o największej liczbie jedynek. Poniższy lemat mówi o podstawowych własnościach takich podziałów.

Lemat 2 *Podział $c_1, \ldots, c_l$ liczby n o maksymalnej* NWW *i o największej liczbie jedynek spełnia warunki:*

(i) Dla każdego $i = 1, \ldots, l$, c_i jest dodatnią potęgą liczby pierwszej lub jedynką.

(ii) Każde dwie liczby podziału c_i i c_j są względnie pierwsze, dla $i, j = 1, \ldots, l$, $i \neq j$.

Dowód Dowody obu punktów są nie wprost. Zakładamy, że podział $c_1, \ldots, c_l$ liczby n jest podziałem o maksymalnej NWW i ma największą liczbę jedynek wśród takich podziałów.

(i) Załóżmy, że $c_i > 1$ jest iloczynem liczb względnie pierwszych: $c_i = ab$, $\mathrm{NWD}(a,b) = 1$, $2 \leqslant a < b$. Z (11) mamy $a + b < ab$. Zatem jeśli zamiast jednej liczby ab podziału n weźmiemy liczby a, b oraz $ab - a - b$ jedynek otrzymamy nowy podział n, który będzie miał więcej jedynek. Ponadto NWW otrzymanego podziału się nie zmieni, gdyż $\mathrm{NWW}(a,b) = ab$. Mamy więc nowy podział o maksymalnej NWW i większej liczbie jedynek. Otrzymana sprzeczność dowodzi, że $c_i = p^{\alpha}$ dla pewnej liczby pierwszej p i $\alpha \geqslant 1$.

(ii) Załóżmy, że dla pewnych $i \neq j$ jest $\mathrm{NWD}(c_i, c_j) > 1$. Na podstawie (i) wiemy, że c_i i c_j są potęgami liczb pierwszych. Ponieważ mają wspólny dzielnik, więc muszą być potęgami tej samej liczby pierwszej. Przyjmijmy $c_i = p^{\alpha}$ i $c_j = p^{\beta}$. Bez straty ogólności załóżmy, że $\alpha \leqslant \beta$, wtedy $\mathrm{NWW}(c_i, c_j) = c_j$. Zatem możemy w podziale n zamiast c_i wziąć c_i jedynek nie zmieniając NWW. Sprzeczność dowodzi, że $\mathrm{NWD}(c_i, c_j) = 1$.

■

Bezpośrednio z lematu 2 wynika twierdzenie.

Twierdzenie 3 *Podział n o maksymalnej* NWW *i największej liczbie jedynek składa się z jedynek oraz liczb postaci $p_1^{\alpha_1},\ldots,p_k^{\alpha_k}$, gdzie $p_1,\ldots,p_k$ są różnymi liczbami pierwszymi, a $\alpha_1,\ldots,\alpha_k$ dodatnimi liczbami całkowitymi.*

Przykład 3 Weźmy wszystkie podziały 22 o maksymalnej NWW, która wynosi 420. Są to:

$$22 = 4+5+6+7 = 3+3+4+5+7 = 1+2+3+4+5+7 = 1+1+1+3+4+5+7.$$

W pierwszym podziale 6 rozkłada się na iloczyn liczb względnie pierwszych 2 i 3. W drugim i trzecim znajdujemy pary liczb, które nie są względnie pierwsze: $3,3$ oraz $2,4$. Ostatni podział spełnia warunki lematu 2. Każda liczba różna od jedynki jest potęgą liczby pierwszej: $2^2,3^1,5^1,7^1$, a zatem według twierdzenia 3 ma największą liczbę jedynek wśród podziałów o maksymalnej NWW.

Wniosek 4 *Maksymalny rząd permutacji n–elementowej wyraża się wzorem:*

$$p_1^{\alpha_1}\cdots p_k^{\alpha_k},$$

gdzie $p_1,\ldots,p_k$ są różnymi liczbami pierwszymi, $\alpha_1,\ldots,\alpha_k$ dodatnimi liczbami całkowitymi takimi, że:

$$\sum_{i=1}^{k} p_i^{\alpha_i} \leqslant n.$$

Permutacje najmniejsze leksykograficznie

Pokażemy jak konstruować permutacje najmniejsze leksykograficznie wśród permutacji o zadanych długościach cykli.

Twierdzenie 5 *Najmniejszą leksykograficznie permutacją rozkładającą się na cykle długości $c_1 \leqslant c_2 \leqslant \ldots \leqslant c_l$ jest permutacja π wyrażająca się wzorami (dla $r = 1,\ldots,l$):*

$$\pi(C_{r-1}+j) = \begin{cases} C_{r-1}+j+1 & \textit{dla } j = 1,\ldots,c_r-1, \\ C_{r-1}+1 & \textit{dla } j = c_r, \end{cases} \tag{12}$$

gdzie

$$C_0 = 0 \textit{ i } C_r = \sum_{i=1}^{r} c_i, \textit{ dla } r > 0.$$

Przyjrzyjmy się, co mówi to twierdzenie. Wzory (12) przedstawiają zapis cyklu c_r–elementowego. Zatem konstrukcja permutacji najmniejszej leksykograficznie polega na zapisaniu kolejnych cykli od najkrótszego do najdłuższego, gdzie cykl długości c jest postaci $(j \quad j+1 \quad j+2 \cdots j+c-1)$.

Dowód Udowodnimy indukcyjnie po i tezę:

> Wartości $\pi(1),\dots,\pi(i)$ najmniejszej leksykograficznie permutacji π, wśród permutacji rozkładających się na cykle długości $c_1 \leqslant \dots \leqslant c_l$, spełniają wzór (12). (13)

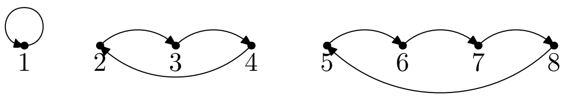
Dla $i = 0$ teza (13) jest oczywista. Niech $i \geqslant 1$, wtedy $i = C_{r-1} + j$ dla pewnych $1 \leqslant r \leqslant l$ i $1 \leqslant j \leqslant c_r$. Załóżmy, że wartości $\pi(1),\dots,\pi(i-1)$ są wyznaczone przez wzory (12). Zobaczmy co oznacza to założenie. $\pi(1),\dots,\pi(C_{r-1})$ reprezentują cykle długości $c_1,\dots,c_{r-1}$, więc pozostaje nam już tylko utworzyć cykle długości $c_r,\dots,c_l$. Ponadto zbiór wartości wynosi $\{\pi(1),\dots,\pi(C_{r-1})\} = \{1,\dots,C_{r-1}\}$. Pozostałe znane wartości to $\pi(C_{r-1}+1),\dots,\pi(C_{r-1}+j-1)$, które wynoszą odpowiednio $C_{r-1}+2,\dots,C_{r-1}+j$. Zatem dostępne wartości, które może przyjąć $\pi(i)$, wynoszą $C_{r-1}+1$ oraz $C_{r-1}+j+1,\dots,n$. Najmniejszą z nich jest $C_{r-1}+1$. Jeśli przyjmiemy

$$\pi(i) = C_{r-1}+1$$

utworzymy cykl $(C_{r-1}+1 \quad C_{r-1}+2 \ \cdots \ C_{r-1}+j)$ o długości j. Najmniejszy cykl jaki możemy utworzyć ma długość c_r. Zatem możemy przyjąć $\pi(i) = C_{r-1}+1$ tylko wtedy, gdy $j = c_r$. Jeśli $j < c_r$, to za $\pi(i)$ przyjmujemy drugą najmniejszą możliwą wartość:

$$\pi(i) = C_{r-1}+j+1.$$

■



Rysunek 2: Najmniejsza leksykograficznie permutacja o długościach cykli $1,3,4$

Przykład 4 Najmniejszą leksykograficznie permutacją, która rozkłada się na cykle długości $1,3,4$ jest

$$\pi(1)=1, \pi(2)=3, \pi(3)=4, \pi(4)=2, \pi(5)=6, \pi(6)=7, \pi(7)=8, \pi(8)=5$$

(Rysunek 2).

Umiemy już konstruować permutacje najmniejsze leksykograficznie o zadanych długościach cykli. Jeśli dodatkowo mamy możliwość decydowania jakie są długości cykli permutacji, to chcemy wiedzieć, który zestaw długości cykli umożliwi utworzenie permutacji najmniejszej leksykograficznie.

Lemat 6 *Dane są dwa ciągi $c_1 \leqslant \dots \leqslant c_l$ oraz $c'_1 \leqslant \dots \leqslant c'_{l'}$ takie, że*

$$n = \sum_{i=1}^{l} c_i = \sum_{i=1}^{l'} c'_i,$$

i o tej własności, że ciąg $c_1,\ldots,c_l$ jest mniejszy leksykograficznie od ciągu $c'_1,\ldots,c_{l'}$, tzn., że istnieje takie r, że $c_i = c'_i$ dla $i = 1,\ldots,r-1$ oraz $c_r < c'_r$. Wśród permutacji, której cykle są długości $c_1,\ldots,c_l$ lub $c'_1,\ldots,c'_{l'}$ najmniejszą leksykograficznie jest najmniejsza leksykograficznie permutacja o cyklach długości $c_1,\ldots,c_l$.

Dowód Jest to wniosek z twierdzenia 5. Wystarczy wziąć najmniejszą leksykograficznie permutację dla ciągu $c_1,\ldots,c_l$ oraz dla ciągu $c'_1,\ldots,c'_{l'}$ i sprawdzić, że ta pierwsza jest mniejsza leksykograficznie. ■

Wniosek 7 *Jeśli ciąg $c_1,\ldots,c_l$ ma więcej jedynek niż ciąg $c'_1,\ldots,c'_{l'}$, to najmniejsza leksykograficznie permutacja o cyklach długości $c_1,\ldots,c_l$ jest mniejsza od najmniejszej leksykograficznie permutacji o cyklach długości $c'_1,\ldots,c'_{l'}$.*

Na podstawie twierdzenia 3 i wniosku 7 możemy sformułować twierdzenie, które mówi nam w jakiej klasie permutacji szukać rozwiązania naszego zadania.

Twierdzenie 8 *Permutacja n–elementowa o największym rzędzie, która jest najmniejsza leksykograficznie, składa się z cykli długości $1,\ldots,1,p_1^{\alpha_1},\ldots,p_k^{\alpha_k}$, dla pewnych różnych liczb pierwszych $p_1,\ldots,p_k$ i dodatnich liczb całkowitych $\alpha_1,\ldots,\alpha_k$ takich, że:*

$$\sum_{i=1}^{k} {p_i}^{\alpha_i} \leqslant n.$$

Szukanie maksymalnego rzędu

Wiemy jaka jest postać szukanej permutacji n–elementowej. Pozostaje pytanie jak znaleźć maksymalny rząd. Możemy to zrobić stosując podejście programowania dynamicznego. Od teraz $p_1, p_2, p_3,\ldots$ oznaczają kolejne liczby pierwsze.

Definicja 10 Maksymalny rząd permutacji n–elementowej, w której występują tylko cykle o długości 1 lub postaci p^α, dla $p \leqslant p_k$, oznaczamy przez $R_{n,k}$.

Bezpośrednio z tej definicji wynika, że wartości $R_{n,k}$ będą rosły wraz z wzrostem n i k:

Fakt 4

$$R_{n_1,k_1} \leqslant R_{n_2,k_2} \quad \text{dla } k_1 \leqslant k_2,\ i\ n_1 \leqslant n_2.$$

Do wyliczania $R_{n,k}$ w sposób systematyczny wykorzystujemy twierdzenie:

Twierdzenie 9 *Wartości $R_{n,k}$ możemy wyliczać rekurencyjnie według wzoru:*

$$R_{n,k} = \max\{R_{n,k-1}\} \cup \{p_k^\alpha R_{n-p_k^\alpha,k-1} \,:\, 1 < p_k^\alpha \leqslant n\}, \tag{14}$$

przy warunkach brzegowych

$$R_{n,0} = 1.$$

Dowód Permutacja może nie zawierać cyklu postaci p_k^α co daje $R_{n,k-1}$. Jeśli jednak zawiera cykl postaci p_k^α, to maksymalny rząd n–elementowej permutacji z cyklami o długościach postaci p^β, $p \leqslant p_k$, będzie wynosił tyle, co maksymalny rząd permutacji mającej mniej o p_k^α elementów z cyklami o długościach postaci p^β, $p \leqslant p_{k-1}$, pomnożony przez długość cyklu p_k^α. ■

Żeby znaleźć maksymalny rząd permutacji n–elementowej wystarczy wyliczyć $R_{n,k}$ dla odpowiednio dużego k. Musi ono być na tyle duże, że powiększenie k nie powiększy już $R_{n,k}$. Zatem algorytm na szukanie maksymalnego rzędu permutacji n–elementowej wygląda następująco:

for i := 0 **to** n **do**
    $R_{i,0}$:= 1
    for k := 1 **to** „odpowiednio duże k” **do**
        $R_{i,k}$:= $\max\{R_{i,k-1}\} \cup \{p_k^{\alpha} R_{i-p_k^{\alpha},k-1} : 1 < p_k^{\alpha} \leqslant i\}$
maksymalny rząd permutacji n–elementowej wynosi $R_{n,k}$,
  gdzie k jest „odpowiednio duże”

Ograniczanie k

Co oznacza „odpowiednio duże k”? Przyjmijmy parę oznaczeń.

Definicja 11 Niech K_n oznacza najmniejsze k takie, że dla wszystkich $k' \geqslant k$ jest $R_{n,k'} = R_{n,k}$.

Definicja 12 Niech $\overline{K}_n$ oznacza najmniejsze k takie, że dla wszystkich $k' \geqslant k$ i dowolnego $n' \leqslant n$ jest $R_{n',k'} = R_{n',k}$.

Wartość $\overline{K}_n$ można też opisać za pomocą K_n:

$$\overline{K}_n = \max_{0 \leqslant n' \leqslant n} K_{n'}.$$

Możemy teraz powiedzieć, że przez “odpowiednio duże k” rozumiemy dowolne k, o którym wiemy, że jest większe od $\overline{K}_n$.

Jak duże jest K_n? Na pewno zachodzi $p_{K_n} \leqslant n$. W ten sposób dla $n = 10\,000$ dostajemy bardzo słabe ograniczenie $\overline{K}_n \leqslant 1229$, gdyż $p_{1229} = 9973 < 10\,000 < 10\,007 = p_{1230}$. Do „rozsądnego” ograniczania K_n pomocny jest następujący lemat.

Lemat 10 *Niech $n \geqslant 1$. Niech h będzie najmniejszą liczbą spełniającą jeden z warunków:*

(i) $p_{h+1} > n$,

(ii) $h \geqslant \overline{K}_{n-1}$ *i* $R_{n,h} \geqslant nR_{n-p_{h+1},h}$,

wtedy h jest ograniczeniem na K_n:

$$K_n \leqslant h.$$

Dowód Jeżeli $p_{h+1} > n$, to dla każdego $k' > h$ będzie $p_{k'} > n$, a co za tym idzie, ze wzoru (14) mamy $R_{n,k'} = R_{n,k'-1}$. Czyli $R_{n,k'} = R_{n,h}$ dla każdego $k' \geqslant h$, zatem $K_n \leqslant h$.

Załóżmy teraz, że zachodzi (ii). Udowodnimy indukcyjnie, że dla $k' > h$ jest

$$R_{n,k'} = R_{n,h}. \tag{15}$$

Zakładamy, że $R_{n,k'-1} = R_{n,h}$. Jeżeli $p_{k'} > n$, to równość (15) otrzymujemy bezpośrednio z (14). Zatem niech $p_{k'} \leqslant n$. Weźmy α takie, że $p_{k'}^{\alpha} \leqslant n$. Z tego, że $h \geqslant \overline{K}_{n-1}$ mamy

$$p_{k'}^{\alpha} R_{n-p_{k'}^{\alpha},k'-1} = p_{k'}^{\alpha} R_{n-p_{k'}^{\alpha},h}.$$

Zachodzi $n - p_{k'}^{\alpha} \leqslant n - p_{h+1}$, więc z faktu 4 mamy

$$p_{k'}^{\alpha} R_{n-p_{k'}^{\alpha},h} \leqslant p_{k'}^{\alpha} R_{n-p_{h+1},h}.$$

Dalej szacujemy:

$$p_{k'}^{\alpha} R_{n-p_{h+1},h} \leqslant n R_{n-p_{h+1},h} \leqslant R_{n,h}.$$

W końcu z założenia indukcyjnego mamy

$$R_{n,h} = R_{n,k'-1}.$$

Podsumowując powyższe rozumowanie otrzymujemy:

$$p_{k'}^{\alpha} R_{n-p_{k'}^{\alpha},k'-1} \leqslant R_{n,k'-1},$$

skąd wynika, że $R_{n,k'} = R_{n,k'-1} = R_{n,h}$, a to kończy dowód indukcyjny.

■

Z pomocą lematu 10 możemy termin „odpowiednio duże k” zastąpić warunkiem:

$$p_{k+1} > i \vee (k \geqslant \overline{K}_{i-1} \wedge R_{i,k} \geqslant i R_{i-p_{k+1},k}). \tag{16}$$

Możemy teraz uzupełnić algorytm:

$\overline{K}_0 := 0$
for $i := 0$ **to** n **do**
    $R_{i,0} := 1$
    $k := 0$
    while $p_{k+1} \leqslant i \wedge (k < \overline{K}_{i-1} \vee R_{i,k} < i R_{i-p_{k+1},k})$ **do**
        $k := k+1$
        $R_{i,k} := \max\{R_{i,k-1}\} \cup \{p_k^{\alpha} R_{i-p_k^{\alpha},k-1} : 1 < p_k^{\alpha} \leqslant i\}$
    while $k > 0 \vee R_{i,k} = R_{i,k-1}$ **do** $k := k-1$ { Szukamy wartości K_i }
    $K_i := k$, $\overline{K}_i := \max(\overline{K}_{i-1}, K_i)$
maksymalny rząd permutacji n–elementowej wynosi R_{n,K_n}

Okazuje się, że w tym algorytmie dla $n \leqslant 10\,000$ największe k dla jakiego będziemy liczyć $R_{i,k}$ wynosi 99. Natomiast największe K_i wynosi 70. Wynika stąd, że za ograniczenie k wystarczyło przyjąć dowolną liczbę nie mniejszą od 70.

Reprezentacja $R_{n,k}$

Jak duże mogą być $R_{n,k}$? Okazuje się, że są na tyle duże, że trzeba implementować duże liczby całkowite. Przy czym na dużych liczbach potrzebujemy tylko operacji:

- dodawania,
- mnożenia przez liczbę z przedziału $[1, 10000]$,
- porównywania.

Zauważmy, że nie potrzebujemy wypisywania dużej liczby i za podstawę możemy wziąć potęgę dwójki. Najwygodniej za podstawę jest wziąć 2^{16}. Rozmiar pamiętanej liczby można ustalać dynamicznie przy zapamiętywaniu nowo wyliczanej wartości $R_{n,k}$. W przypadku, gdy chcemy przydzielić pamięć statycznie, trzeba określić maksymalny rozmiar liczby jaką będziemy pamiętać. W tym celu musimy sprawdzić jakie jest największe $R_{n,k}$. Największe $R_{n,k}$ wynosi tyle co rząd permutacji n–elementowej dla $n = 10\,000$. Można eksperymentalnie sprawdzić, że wartość ta wynosi w przybliżeniu $1.8 \cdot 2^{454} < 2^{464} = 2^{16\cdot 29}$. Zatem można reprezentować wartości $R_{n,k}$ przez 29–cyfrowe liczby o podstawie 2^{16}.

Przy takiej reprezentacji pamięć potrzebna na zapamiętanie wszystkich wartości $R_{n,k}$ wynosi $2 \cdot 29 \cdot 10\,000 \cdot 70 = 40\,600\,000 \approx 40$M.

Wypisanie szukanej permutacji

Gdy już mamy wyliczony maksymalny rząd R_{n,K_n} permutacji n–elementowej, pozostaje wypisanie permutacji najmniejszej leksykograficznie. Wpierw musimy odtworzyć długości cykli tej permutacji. Oznaczmy:

$$C_{i,k} = \begin{cases} 0 & \text{jeśli } R_{i,k} = R_{i,k-1} \\ p_k^{\alpha} & \text{jeśli } R_{i,k} = p_k^{\alpha} R_{i-p_k^{\alpha},k-1} \end{cases}$$

wtedy algorytm wyznaczania długości cykli jest następujący:

```
i := n
for k := K_n downto 1 do
    if C_{i,k} > 0 then zapamiętaj długość cyklu C_{i,k}
    i := i − C_{i,k}
do zapamiętanych długości cykli dodaj i cykli długości 1
```

Wartości $C_{i,k}$ możemy sobie wcześniej zapamiętać podczas wyliczania wartości $R_{i,k}$.

Mając długości cykli możemy już wypisać permutację najmniejszą leksykograficznie. Sortujemy otrzymany ciąg liczb długości cykli od najmniejszej do największej otrzymując ciąg $c_1 \leqslant \ldots \leqslant c_l$, a następnie stosujemy twierdzenie 5.

Wiele wartości n

Dla testu zawierającego wiele wartości n bierzemy to największe. Dla niego stosujemy przedstawiony algorytm do wyliczania wartości $R_{n,k}$ oraz $C_{n,k}$. Mając te wartości możemy wypisać szukane permutacje dla wszystkich n znajdujących się w teście.

Inne rozwiązania

Typ `double` zamiast dużych liczb całkowitych

Można sprawdzić, że dla $n \leqslant 10\,000$ przy wyliczaniu wartości $R_{n,k}$ wszystkie porównania dotyczą liczb różniących się względnie o co najmniej 10^{-5}. Sugeruje to reprezentowanie wartości $R_{n,k}$ przez typ `double`. Rzeczywiście stosując zamiast dużych liczb całkowitych liczby zmiennoprzecinkowe podwójnej precyzji otrzymamy poprawny program dla danych wejściowych w tym zadaniu.

Pamiętanie części wartości $R_{n,k}$

Zauważmy, że we wzorze (14) wartość $R_{n,k}$ zależy tylko od wartości $R_{i,k-1}$, gdzie $i \leqslant n$. Zatem, aby wyznaczyć wszystkie wartości $R_{i,k}$ wystarczy, że będziemy pamiętać wartości $R_{i,k-1}$. Jeśli będziemy liczyć w odpowiedniej kolejności, to wystarczy pamiętać tylko $n+1$ wartości:

for i := 0 **to** n **do** R_i := 1 { Inicjacja dla $k = 0$ }
for k := 1 **to** „odpowiednio duże k” **do**
  for i := n **downto** 0 **do**
    $R_i := \max\{R_i\} \cup \{p_k^{\alpha} R_{i-p_k^{\alpha}} : 1 < p_k^{\alpha} \leqslant i\}$
maksymalny rząd permutacji n–elementowej wynosi R_n

Oczywiście, żeby odtworzyć później długości cykli trzeba zapamiętać już wszystkie wartości $C_{i,k}$.

Testy

Zostało przygotowanych piętnaście testów. Maksymalne n w każdym z testów wynosiło kolejno: 5, 10, 20, 79, 789, 2003, 4567, 7890, 8945, 10000, 9878, 9991, 510, 2021, 3705. W ostatnich pięciu testach dane zostały tak dobrane, aby wyłapać rozwiązania nakładające zbyt duże ograniczenie na k.

Pogromcy Algorytmów 2003

opracowania zadań

Przestawione literki

Niektóre dzieci nie potrafią wymawiać wszystkich liter, niektóre zaś czasem wymawiają litery poprawnie, a czasem nie. Kamil mówi czasem T zamiast K, ale zamiast T nigdy nie mówi K. Podobnie zamiast G mówi czasem D. Natomiast zamiast R mówi czasem L, a kiedy indziej F. Oczywiście czasami zdarza się, że wymawia literkę poprawnie. Tata Kamila zawsze się zastanawia, ile rzeczywistych słów może oznaczać wyraz, który wypowiedział jego syn (nie zastanawia się nad tym, czy są to poprawne wyrazy w języku polskim).

Zadanie

Napisz program, który:

- *wczyta (ze standardowego wejścia) to, co powiedział Kamil,*
- *obliczy ile różnych słów może to oznaczać,*
- *wypisze wynik (na standardowe wyjście).*

Wejście

W pierwszym i jedynym wierszu wejścia zapisano słowo wypowiedziane przez Kamila. Dla uproszczenia przyjmujemy, że słowo to zapisano za pomocą jedynie wielkich liter alfabetu angielskiego i jego długość wynosi co najwyżej 20.

Wyjście

Twój program powinien wypisać (na standardowe wyjście) tylko jeden wiersz zawierający tylko jedną liczbę całkowitą, równą liczbie różnych słów, które może oznaczać słowo wypowiedziane przez Kamila.

Przykład

Dla danych wejściowych:
`FILIPEK`
poprawnym wynikiem jest:
`4`

Julka

Julka zaskoczyła wczoraj w przedszkolu swoją wychowawczynię rozwiązując następującą zagadkę:

> Klaudia i Natalia mają razem 10 jabłek, ale Klaudia ma o 2 jabłka więcej niż Natalia. Ile jabłek ma każda z dziewczynek?

Julka odpowiedziała bez namysłu: Klaudia ma 6 jabłek, natomiast Natalia ma 4 jabłka.

Wychowawczyni postanowiła sprawdzić, czy odpowiedź Julki nie była przypadkowa i powtarzała zagadkę, za każdym razem zwiększając liczby jabłek w zadaniu. Julka zawsze odpowiadała prawidłowo. Zaskoczona wychowawczyni chciała kontynuować „badanie" Julki, ale przy bardzo dużych liczbach sama nie potrafiła szybko rozwiązać zagadki. Pomóż pani przedszkolance i napisz program, który będzie podpowiadał jej rozwiązania.

Zadanie

Napisz program, który:

- *wczyta (ze standardowego wejścia) liczbę jabłek, które mają razem obie dziewczynki oraz liczbę mówiącą, o ile więcej jabłek ma Klaudia,*
- *obliczy, ile jabłek ma Klaudia i ile jabłek ma Natalia,*
- *wypisze wynik (na standardowe wyjście).*

Wejście

Wejście składa się z dwóch wierszy. Pierwszy wiersz zawiera liczbę wszystkich jabłek posiadanych przez dziewczynki, natomiast drugi — liczbę mówiącą, o ile więcej jabłek ma Klaudia. Obie liczby są całkowite i dodatnie. Wiadomo, że dziewczynki mają razem nie więcej niż 10^{100} *(1 i 100 zer) jabłek. Jak widać, jabłka mogą być bardzo malutkie.*

Wyjście

Twój program powinien wypisać (na standardowe wyjście) w dwóch kolejnych wierszach dwie liczby całkowite, po jednej w wierszu. Pierwszy wiersz powinien zawierać liczbę jabłek Klaudii, natomiast drugi — liczbę jabłek Natalii. Wiadomo, że dziewczynki zawsze mają całe jabłka.

Przykład

Dla danych wejściowych:

```
10
2
```

poprawnym wynikiem jest:

```
6
4
```

Jasiek

Jasiek ma dopiero 6 lat, ale już przejawia liczne talenty. Bardzo lubi rysować i układać zagadki. Dzisiaj rano dostał od mamy kartkę w kratkę, ołówek i z wielką ochotą zabrał się do rysowania. Wszystkie rysunki Jaśka mają pewne wspólne cechy:

- *Jasiek zaczernia pełne kratki;*
- *jeżeli dwie zaczernione kratki dotykają się, to mają wspólny bok lub róg;*
- *są spójne, co oznacza, że między każdymi dwiema zaczernionymi kratkami istnieje ciąg zaczernionych kratek, w którym każde dwie kolejne kratki mają wspólny bok;*
- *nie ma białych dziur, czyli że z każdej białej kratki można narysować linię do brzegu kartki, która nigdy nie dotknie jakiejkolwiek zaczernionej kratki.*

W południe zadzwoniła mama i zapytała, co przedstawia dzisiejszy rysunek Jaśka. Maluch nie odpowiedział wprost, tylko opisał rysunek podając ciąg ruchów potrzebnych do obejścia zaczernionych kratek na brzegu rysunku, czyli takich, które mają co najmniej jeden wspólny róg z jakąś białą kratką. Jasiek ustalił kratkę początkową, a następnie podał ciąg kierunków, w których należy się posuwać, żeby obejść cały rysunek. Wiadomo, że Jasiek opisał rysunek w kierunku przeciwnym do ruchu wskazówek zegara. Mama była wielce zaskoczona złożonością rysunku, a w szczególności liczbą zaczernionych kratek. Czy potrafiłbyś na podstawie opisu Jaśka szybko obliczyć, ile jest zaczernionych kratek na rysunku?

Zadanie

Napisz program, który:

- *wczyta (ze standardowego wejścia) opis rysunku Jaśka,*
- *policzy liczbę wszystkich zaczernionych kratek,*
- *wypisze wynik (na standardowe wyjście).*

Wejście

Wejście składa się z szeregu wierszy, z których każdy zawiera tylko jeden znak. Wiersz pierwszy zawiera dużą literę P, natomiast wiersz ostatni — dużą literę K. Litera P oznacza początek opisu, a litera K jego koniec. W każdym z pozostałych wierszy (jeżeli takie są) zapisano jedną literę N, W, S lub E, gdzie N oznacza północ, W — zachód, S — południe, a E — wschód. Każdy wiersz wejścia odpowiada pewnej kratce na brzegu rysunku. Wiersz pierwszy i ostatni odpowiadają tej samej kratce, od której zaczyna się i w której kończy się opis. Litera w wierszu różnym od wiersza pierwszego i ostatniego mówi, w którym kierunku należy pójść, żeby przejść do kolejnej kratki brzegowej przy obchodzeniu rysunku przeciwnie do ruchu wskazówek zegara. Opis Jaśka nie jest nadmiarowy, tzn. kończy się po obejściu całego rysunku i dotarciu do kratki początkowej. Długość opisu nigdy nie przekracza 20000 liter.

Wyjście

Twój program powinien wypisać (na standardowe wyjście) tylko jeden wiersz z jedną liczbą całkowitą równą liczbie zaczernionych kratek na rysunku Jaśka.

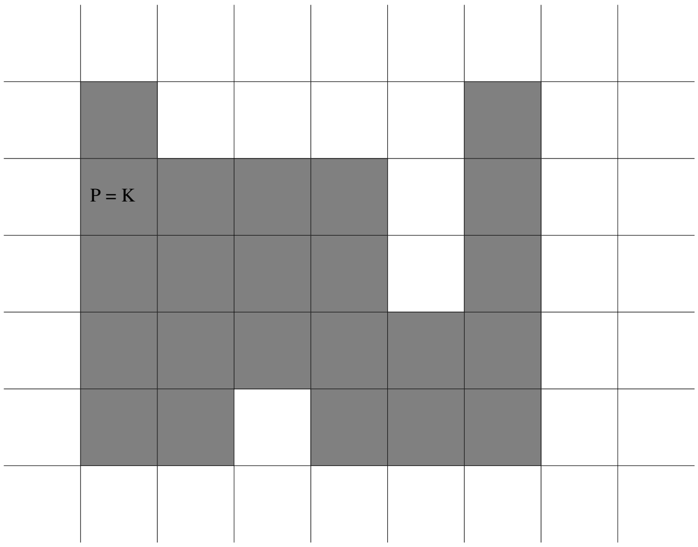
Przykład

Dla danych wejściowych:

```
P
S
S
S
E
N
E
E
S
E
E
N
N
N
N
S
S
S
W
W
N
N
W
W
W
N
S
K
```

poprawnym wynikiem jest:

```
23
```



Rozwiązanie

Nietrudno zauważyć, że rysunek Jaśka, to wielokąt o bokach równoległych do osi układu współrzędnych i wierzchołkach o współrzędnych całkowitoliczbowych, a naszym celem jest napisanie programu, który policzy pole tego wielokąta. Najprościej do tego celu zastosować wzór Greeniego. Jeżeli (x_j, y_j) są kolejnymi wierzchołkami na obwodzie wielokąta W, przy obchodzeniu W w kierunku przeciwnym do ruchu wskazówek zegara, $j = 0, 1, \ldots, n-1$, oraz $x_0 = x_n$ i $y_0 = y_n$, to pole W wyraża się wzorem:

$$\text{Pole}(W) = 0.5(a_0 + a_1 + \ldots + a_{n-1}),$$

gdzie $a_i = x_i y_{i+1} - x_{i+1} y_i$. Tak więc cała trudność w tym zadaniu sprowadza się do wyznaczenia kolejnych punktów kratowych (o współrzędnych całkowitoliczbowych) na obwodzie wielokąta przy założeniu, że startujemy z punktu $(0,0)$. W tym celu przyjmijmy, że wielokąt będziemy obchodzili przesuwając wzdłuż jego brzegu kostkę 2×1 i przyjmując, że ostatnio rozważany wierzchołek na obwodzie wielokąta jest w środku prawego, dłuższego brzegu kostki. Dla danych z przykładu kostkę na początku przykładamy w taki sposób, że jej pierwszy kwadracik pokrywa się z kwadratem oznaczonym literą P, natomiast drugi kwadrat jest bezpośrednio pod pierwszym. Oto program realizujący opisany powyżej pomysł.

```
var
  pierwszy, r1, r2: char; { pierwszy i dwa kolejne kierunki w obejściu W }
  x1, y1: longint; { ostatni rozważany punkt na obwodzie }
  x2, y2: longint; { kolejny punkt na obwodzie }
  P : longint; { obliczane Pole }
  ostatni_obrot : boolean;
begin
  readln(r1);
```

```
readln(r2);
if r2 = 'K' then writeln(1) { tylko jedna kratka }
else
begin
  pierwszy := r2;
  P := 0;
  x1 := 0; y1 := 0; ostatni_obrot := false;
  repeat
    r1 := r2; readln(r2);
    if r2 = 'K' then
    begin
    { trzeba jeszcze wrócić do punktu wyjścia }
      r2 := pierwszy; ostatni_obrot := true
    end;
    { badamy kolejne dwa ruchy, żeby odpowiednio przesunąć kostkę }
    { pole liczymy razy dwa; na końcu podzielimy przez dwa }
    case r1 of
       'E': case r2 of
              'E': begin x2 := x1 + 1; y2 := y1 end;
              'N': begin x2 := x1 + 1; y2 := y1 + 1; P := P + 1 end;
              { pomijamy róg typu 'EN' - ucinamy pół kratki }
              'W': begin x2 := x1; y2 := y1 + 1; P := P + 2 end;
              { ucinamy całą kratkę }
              'S': begin x2 := x1; y2 := y1 end
            end;
       'N': case r2 of
              'N': begin x2 := x1; y2 := y1 + 1 end;
              'W': begin x2 := x1 - 1; y2 := y1 + 1; P := P + 1 end;
              { ucinamy pół kratki }
              'S': begin x2 := x1 - 1; y2 := y1; P := P + 2 end;
              { ucinamy całą kratkę }
              'E': begin x2 := x1; y2 := y1 end
            end;
       'W': case r2 of
              'W': begin x2 := x1 - 1; y2 := y1 end;
              'S': begin x2 := x1 - 1; y2 := y1 - 1; P := P + 1 end;
              { ucinamy pół kratki }
              'E': begin x2 := x1; y2 := y1 - 1; P := P + 2 end;
              { ucinamy całą kratkę }
              'N': begin x2 := x1; y2 := y1 end
            end;
       'S': case r2 of
              'S': begin x2 := x1; y2 := y1 - 1 end;
              'E': begin x2 := x1 + 1; y2 := y1 - 1; P := P + 1 end
              { ucinamy pół kratki }
              'N': begin x2 := x1 + 1; y2 := y1; P := P + 2 end
```

```
                { ucinamy całą kratkę }
                'W': begin x2 := x1; y2 := y1 end
              end
      end;
      P := P + x1*y2 - x2*y1;
      x1 := x2; y1 := y2;
    until ostatni_obrot;
    P := P div 2;
    writeln(P)
  end
end.
```

W oczywisty sposób powyższy program działa w czasie liniowym.

Krzysztof Diks
Treść zadania, Opracowanie

Krzysztof Diks
Program

Dyzio

Dyzio jest przyjacielem Jaśka i też lubi zagadki. Oto zagadka, z którą Dyzio przyszedł do Jaśka:

> Jaśku, masz tu sznurek, który trzeba pociąć na mniejsze kawałki. Nie powiem Ci wprost, jak to zrobić, ale popatrz na ten ciąg zer (*0*) i jedynek (*1*). Jedynka na początku oznacza, że sznurek trzeba przeciąć na pół. Jeśli jednak pierwszą cyfrą byłoby zero, to byłaby to jedyna cyfra w ciągu i oznaczałaby, że nie musisz już nic robić — chcę mieć sznurek w całości. Jeśli jednak musisz przeciąć sznurek, to po pierwszej jedynce zapisałem, co zrobić z lewym kawałkiem (stosując te same reguły, co dla całego sznurka), a następnie zapisałem co zrobić z prawym kawałkiem (cały czas trzymając się tych samych zasad zapisu). Zawsze musisz najpierw pociąć lewy kawałek, a dopiero potem możesz zabrać się do prawego. A teraz tnij i powiedz, ile minimalnie cięć trzeba wykonać, żeby otrzymać najkrótszy kawałek.

Niestety mama chowa przed Jaśkiem nożyczki, ale szczęśliwie pod ręką był komputer i Jasiek szybko napisał program symulujący cięcie sznurka. Czy Ty też potrafisz napisać taki program?

Zadanie

Napisz program, który:

- *wczyta (ze standardowego wejścia) opis sposobu cięcia sznurka,*
- *policzy, ile minimalnie cięć trzeba wykonać, żeby dostać (pierwszy) najkrótszy kawałek,*
- *wypisze wynik (na standardowe wyjście).*

Wejście

Pierwszy wiersz wejścia zawiera liczbę całkowitą n ($1 \leqslant n \leqslant 20\,000$). W drugim wierszu wejścia zapisano dokładnie jedno słowo zero-jedynkowe (ciąg zer i jedynek bez znaków odstępu między nimi) długości n — opis sposobu cięcia sznurka dostarczony przez Dyzia.

Wyjście

Twój program powinien wypisać (na standardowe wyjście) tylko jeden wiersz, zawierający tylko jedną liczbę całkowitą, równą minimalnej liczbie cięć, które trzeba wykonać, żeby dostać najkrótszy kawałek.

Przykład

Dla danych wejściowych:
```
9
110011000
```
poprawnym wynikiem jest:
```
4
```

Rozwiązanie

Każdy, nawet początkujący algorytmik, zauważy bez trudu, że wejściowy napis zero-jedynkowy koduje obejście pewnego drzewa binarnego metodą prefiksową (znaną też z angielska jako „preorder”). Jedynki odpowiadają odwiedzaniu węzłów wewnętrznych (czyli tych różnych od `nil`), a zera – odwiedzaniu węzłów zewnętrznych (czyli tych reprezentowanych przez `nil`). Każdy węzeł zewnętrzny (0) odpowiada jednemu, niepodzielnemu kawałkowi sznurka. Naszym celem jest znalezienie w drzewie pierwszego „od lewej” węzła zewnętrznego (czyli 0) o największej głębokości i policzenie, ile poprzedza go węzłów wewnętrznych (czyli 1–ek).

Sercem rozwiązania jest rekurencyjna procedura `obejdz_drzewo`. Wywołujemy ją zawsze dla węzła, do którego wchodzimy po raz pierwszy. Po wejściu do takiego węzła liczymy jego głębokość. Gdy jest to węzeł wewnętrzny (czyli odpowiadający 1), to zwiększamy o 1 licznik dotychczas odwiedzonych węzłów wewnętrznych (napotkanych jedynek). W przypadku wejścia do węzła zewnętrznego (odpowiada on niepodzielnemu już kawałkowi sznurka), aktualizujemy informacje o dotychczas najgłębszym węźle wewnętrznym (im głębiej położony węzeł, tym krótszy jest odpowiadający mu sznurek).
Oto formalny zapis procedury `obejdz_drzewo`. Wykorzystujemy w niej następujące zmienne globalne:
`odw` – liczba dotychczas odwiedzonych węzłów
`gl` – aktualna głębokość w drzewie
`kod[1..n]` – tablica zerojedynkowa z zakodowanym obejściem drzewa
`ile_1` – liczba dotychczas przejrzanych 1–ek (węzłów wewnętrznych)
`max_gl` – dotychczas maksymalna głębokość węzła zewnętrznego (odpowiadającemu 0)
`ile_1` – liczba 1–ek (węzłów wewnętrznych, czyli cięć sznurka) poprzedzających pierwsze 0 na głębokości `max_gl`

```
procedure obejdz_drzewo;
begin
  odw := odw + 1;
  gl := gl + 1;
  if kod[odw] = '1' then
  { węzeł wewnętrzny }
  begin
    ile_1 := ile_1 + 1;
    obejdz_drzewo; { obejście lewego poddrzewa }
    obejdz_drzewo  { obejście prawego poddrzewa }
  end
```

```
    else
    { węzeł zewnętrzny }
      if gl > max_gl then
      begin
        max_gl := gl;
        poprz_1 := ile_1
      end;
    gl := gl - 1 { powrót do węzła na mniejszej głębokości }
  end;
```

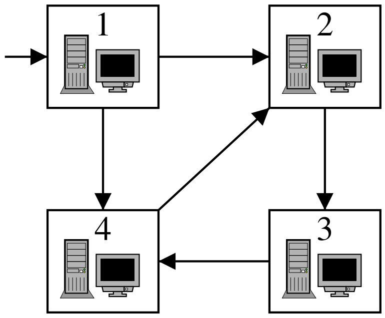
Wywołanie tej procedury dla całego kodu wejściowego, z odpowiednio zainicjowanymi zmiennymi globalnymi, daje nam poprawne rozwiązanie (wartość zmiennej `poprz_1`). Nietrudno zauważyć, że w każdym wywołaniu `obejdz_drzewo` przesuwamy się w kodzie o jedną pozycję w prawo. Dlatego złożoność podanego rozwiązania jest liniowa.

Krzysztof Onak **Krzysztof Onak**
Treść zadania, Opracowanie Program

Bajtocka Agencja Informacyjna

Bajtocka Agencja Informacyjna (BAI) posiada n komputerów zorganizowanych w sieć. Komputery są ponumerowane liczbami od 1 do n, a komputer o numerze 1 jest serwerem. Komputery są połączone za pomocą jednokierunkowych kanałów informacyjnych, które łączą pary komputerów. Cała sieć jest skonstruowana tak, że z serwera można przesłać — bezpośrednio lub pośrednio — informacje do każdego innego komputera.

Gdy BAI zdobywa nową wiadomość, to zostaje ona umieszczona na serwerze, a następnie rozpropagowana w sieci. Szef agencji zastanawia się, co stałoby się w przypadku, gdyby jeden z komputerów przestał zupełnie działać, np. wyleciał w powietrze w wyniku ataku terrorystycznego. Wówczas mogłoby się okazać, że nowo zdobyte informacje nie docierałyby do któregoś z pozostałych komputerów, gdyż uszkodzony komputer był pośrednikiem nie do uniknięcia. Komputery, których awaria mogłaby doprowadzić do takiej sytuacji, nazwiemy **komputerami krytycznymi**. *Na przykład w sytuacji przedstawionej na poniższym rysunku komputerami krytycznymi są komputery o numerach 1 i 2 — 1 jest serwerem, natomiast każda informacja przesyłana z serwera do komputera 3 musi przejść przez komputer 2.*



Zadanie

Napisz program, który:

- *wczyta opis sieci ze standardowego wejścia,*
- *znajdzie wszystkie komputery krytyczne,*
- *wypisze numery komputerów krytycznych na standardowe wyjście.*

Wejście

W pierwszym wierszu znajdują się dwie liczby całkowite, n i m, oddzielone pojedynczym odstępem. Liczba n to liczba komputerów w sieci, $2 \leqslant n \leqslant 5\,000$, a m to liczba kanałów informacyjnych, $n-1 \leqslant m \leqslant 200\,000$. Każdy z kolejnych m wierszy opisuje pojedynczy kanał *informacyjny i składa się z dwóch liczb całkowitych oddzielonych pojedynczym odstępem. Są to odpowiednio a i b* ($1 \leqslant a,b \leqslant n$ *i* $a \neq b$)*, co oznacza, że kanał przesyła informacje z komputera o numerze a do komputera o numerze b. Możesz założyć, że nie ma dwóch kanałów informacyjnych, które zaczynają się i kończą w tych samych punktach.*

Wyjście

Wyjście powinno się składać z dwóch wierszy. W pierwszym wierszu powinna znaleźć się jedna liczba k — liczba komputerów krytycznych. W drugim powinny znaleźć się numery komputerów krytycznych pooddzielane pojedynczymi odstępami, wymienione w kolejności rosnącej.

Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
2
1 2
```

Rozwiązanie

Grafolandia

Na początku przenosimy się rutynowo z Bajtocji do świata grafów. Mamy zatem dany graf skierowany G, w którym wierzchołki odpowiadają komputerom, a krawędzie kanałom informacyjnym. Nasz graf ma, zgodnie z treścią zadania, n wierzchołków i m krawędzi. Ponadto jeden z wierzchołków grafu, nazwijmy go *źródłem*, jest wyróżniony i istnieją skierowane ścieżki ze źródła do każdego z pozostałych wierzchołków. Zadajemy teraz pytanie z osobna dla każdego z wierzchołków, czy po jego usunięciu z grafu nadal będą istniały ścieżki ze źródła do każdego z nieusuniętych wierzchołków. Jeśli nie, to wierzchołek ten jest *krytyczny*.

Algorytm bezpośredni

Najprostszy algorytm, to po prostu usuwanie z grafu G po jednym wierzchołku i sprawdzanie, czy nadal da sie dotrzeć do pozostałych wierzchołków ze źródła. Jego złożoność czasowa jest rzędu $O(nm)$, ale okazuje się, że zadanie można rozwiązać za pomocą algorytmu o lepszej złożoności czasowej.

Drzewo DFS i numeracja preorder

W naszym grafie G skonstruujemy teraz drzewo T przechodzenia w głąb (powszechnie stosowany angielski skrót, który określa to drzewo, to *DFS = depth first search*) o korzeniu w źródle i ponumerujemy wierzchołki liczbami od 1 do n w kolejności, w jakiej odwiedzamy je po raz pierwszy (to właśnie jest kolejność *preorder*). O co dokładnie chodzi? Startując ze źródła, wykonujemy rekurencyjnie w każdym wierzchołku następujące czynności:

- zaznacz, że v jest już odwiedzony, gdzie v to aktualny wierzchołek;
- nadaj v pierwszy nieużyty jeszcze numer ze zbioru $\{1, 2, \ldots, n\}$;
- po kolei dla każdego w będącego sąsiadem v (tzn. dla każdego wierzchołka, do którego istnieje krawędź z v) sprawdź, czy jest już odwiedzony, a jeśli nie, to dodaj do konstruowanego drzewa T krawędź z v do w i wykonaj rekurencyjnie czynności na w.

Przykładowe drzewo DFS z numeracją wierzchołków preorder można zobaczyć na rysunku 1.

Może się zdarzyć, że dla grafu G i ustalonego wierzchołka startowego otrzymujemy wiele różnych drzew przechodzenia w głąb w zależności od tego, w jakiej kolejności odwiedzamy sąsiadów, ale okazuje się, że wszystkie te drzewa mają interesujące nas własności i wystarczy wziąć dowolne z nich.

Łatwo zauważyć, że drzewo DFS T jest podgrafem początkowego grafu G. Dokładniej, ma taki sam zbiór wierzchołków, a zbiór krawędzi T jest podzbiorem krawędzi grafu G. W związku z tym możemy podzielić krawędzie G na *drzewowe*, czyli te, które należą też do drzewa T, i *niedrzewowe*, czyli te, które do drzewa T już nie należą.

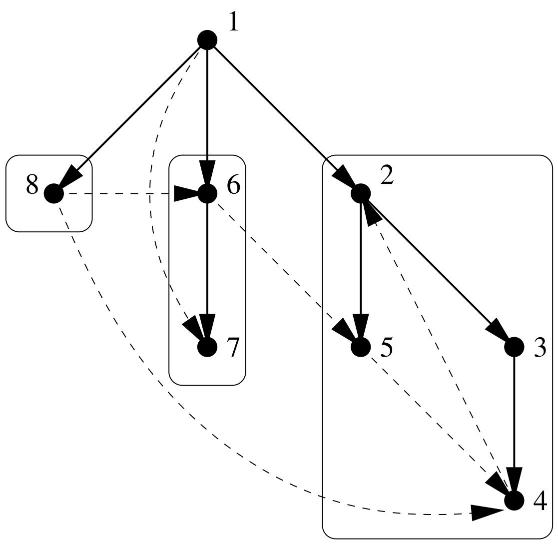
Od tej pory będziemy utożsamiać wierzchołki G z numeracją preorder, którą właśnie im nadaliśmy.

Zauważmy, że jeśli w grafie G istnieje krawędź z v do w, to $w < v$ lub w jest potomkiem v w drzewie. W szczególności oznacza to, że jeśli jakiś wierzchołek ma w drzewie T kilku synów, to krawędzie niedrzewowe pomiędzy poddrzewami o korzeniach w synach mogą prowadzić tylko od tych powstałych później do tych powstałych wcześniej. Przykład na rysunku 1.

Obliczanie wierzchołków krytycznych

Mówiąc inaczej, wierzchołek v jest krytyczny, gdy do jakiegoś innego wierzchołka w nie da się dojść ze źródła nie przechodząc przez v. Wynika stąd, że w musi być potomkiem v w drzewie T. Można wywnioskować, że v jest krytyczny wtedy i tylko wtedy, gdy do co najmniej jednego z jego synów w drzewie T nie da się dojść, nie przechodząc przez niego.

Wystarczy zatem, że ograniczymy się do problemu, czy do danego wierzchołka w da się dojść, omijając jego ojca v w drzewie T. Zastanówmy się, co możemy powiedzieć o wierzchołkach, przez które da się dojść do w. Korzystając z wcześniej zauważonego faktu dotyczącego drzew DFS, do w nie istnieją w grafie G ścieżki z wierzchołków u takich, że $u < w$ i u nie jest przodkiem w w T. Niech $s = s_1 s_1 \ldots s_k$ będzie ścieżką prostą ze źródła do w w T. Wówczas $s_1 = 1$, $s_{k-1} = v$ i $s_k = w$. Zastanówmy się, czy do w da się dojść z jednego z wierzchołków s_i, gdzie $1 \leqslant i < k-1$, przechodząc po drodze tylko po wierzchołkach u



Rysunek 1: Przykładowy graf z drzewem DFS. Krawędzie drzewowe są pogrubione, a niedrzewowe przerywane. Wierzchołki ponumerowane są w kolejności preorder. Dodatkowo zostały zaznaczone poddrzewa wierzchołka o numerze 1.

takich, że $u \geqslant w$. Jeśli tak, to v można pominąć na drodze ze źródła 1, a jeśli nie, to jest to niemożliwe i v jest krytyczny.

Przy najprostszej implementacji znów dostajemy oszacowanie czasu działania $O(nm)$. Ale my podejdziemy do problemu inaczej. Ustalmy wierzchołek w. Załóżmy, że dla każdego wierzchołka u z zakresu od w do n mamy obliczony pierwszy wierzchołek, oznaczmy go przez $t_w(u)$, na ścieżce ze źródła 1 do w, z którego można dotrzeć do u, przechodząc w międzyczasie tylko po wierzchołkach o numerach większych lub równych w. Wówczas ojca wierzchołka w można pominąć na drodze ze źródła wtedy i tylko wtedy, gdy $t_w(w)$ nie jest ojcem w.

Okazuje się ponadto, że możemy obliczyć wartości $t_{w-1}(w-1),\ldots,t_{w-1}(n)$ na podstawie wartości $t_w(w),\ldots,t_w(n)$ w czasie $O(n)$, co da nam łączny czas działania algorytmu $O(n^2)$, jeśli zaczniemy z $w = n$ i zejdziemy do 2, obliczając kolejne wartości $t_w(\cdot)$. Na początku przyjmujemy $t_{w-1}(u) := t_w(u)$, dla $u = w, w+1, \ldots, n$. Następnie przeglądamy krawędzie wchodzące do $w-1$. Niech $I(w-1)$ będzie zbiorem wierzchołków, z których wychodzą krawędzie do $w-1$. Ustalamy już na stałe $t_{w-1}(w-1) := \min(\{u | u \in I(w-1) \wedge u < w-1\} \cup \{t_w(u) | u \in I(w-1) \wedge u > w-1\})$, rozpatrując tym samym dwa możliwe przypadki, bo albo da się dotrzeć do $w-1$ bezpośrednio z wierzchołków na ścieżce drzewowej z 1 do $w-1$, albo pośrednio przez inne wierzchołki, a skąd, to już mamy policzone. Pozostaje jeszcze zastanowić się, czy do niektórych wierzchołków z zakresu od w do n nie da się dojść z wcześniejszego wierzchołka na ścieżce z 1 do $w-1$, korzystając z $w-1$ po drodze. Mogą to być tylko wierzchołki będące potomkami $w-1$ w drzewie T, bo do pozostałych nie da się dotrzeć z $w-1$. Ponadto, jeśli dla jakiegoś wierzchołka u jest to prawda, to dla tego wierzchołka wartość $t_{w-1}(u)$, którą chcemy policzyć, jest równa $t_{w-1}(w-1)$. Wystarczy zatem przejść w głąb poddrzewa drzewa T o korzeniu w $w-1$ poprawiając wartości $t_{w-1}(u)$ na $\min\{t_{w-1}(w-1), t_{w-1}(u)\}$. Przy okazji warto zauważyć, że jeśli w jakimś wierzchołku wartość się w ten sposób nie polepsza, to również w całym jego poddrzewie nie zostanie polepszona, więc nie warto się dalej zagłębiać i otrzymujemy w ten sposób pewne przyspieszenie.

Lepsze rozwiązania

Zadanie można rozwiązać szybciej, a mianowicie istnieje algorytm, który wymaga czasu rzędu tylko $O(m\alpha(m,n))$. Nieco gorsze, ale może bardziej czytelne dla niektórych, szacowanie czasu działania tego algorytmu to $O(m\log^* n)$. Jest on trochę bardziej skomplikowany i korzysta, jak się można domyślić po oszacowaniu czasu działania, ze struktury danych dla zbiorów rozłącznych (więcej szczegółów na temat tych struktur można znaleźć w [14] i [17]). Rozmiar danych wejściowych w zadaniu został jednak specjalnie tak dobrany, żeby wystarczyło rozwiązanie działające w czasie $O(n^2)$.

Istnieją także algorytmy rozwiązujące nasz problem w czasie $O(m)$, lecz są one już na tyle skomplikowane, że trudno oczekiwać, żeby ktoś, nawet znając jeden z nich wcześniej, zdążył go poprawnie zaimplementować w czasie weekendu z Pogromcami Algorytmów, a jednocześnie zysk z tego byłby żaden, bo w praktyce nie sposób przy pomocy testów wychwycić teoretyczny asymptotyczny narzut związany z użyciem struktury dla zbiorów rozłącznych.

Dla dociekliwego Czytelnika zdradzamy, że słowem kluczowym, pod którym należy szukać powyższych algorytmów, jest „dominatory". Algorytmy te służą do wyznaczania właśnie dominatorów, ale nie będziemy już tutaj tłumaczyć, co to jest. Zdradzimy za to, że przydają się między innymi przy analizie przepływu sterowania w programach, a tym samym przy optymalizacji programów. Być może nawet Twój ulubiony kompilator, drogi Czytelniku, używa tych algorytmów.

Rozwiązania zawodników

Zawodnicy implementowali najczęściej podany na początku algorytm bezpośredni lub różne niepoprawne heurystyki. Niektórzy stosowali podany wyżej algorytm lub lepsze wersje o złożoności $O(m\log n)$ i $O(m\alpha(m,n))$. Część uczestników znalazła te algorytmy w literaturze (pozytywny efekt dydaktyczny Pogromców!). Jeszcze inni zastosowali algorytm bezpośredni wraz z uwzględnieniem *bardzo* dużej liczby specjalnych przypadków i heurystyk przyspieszających, wprawiając niejednokrotnie jury konkursu w zdumienie ogromem włożonej pracy.

Ciekawe zadanie

Dla Czytelnika pozostawiamy do rozwiązania nietrudne — już teraz — zadanie. Mamy dany graf skierowany, który jest jedną silnie spójną składową, tzn. z każdego wierzchołka można dotrzeć do każdego innego. Problem, który chcemy rozwiązać, to dla każdego wierzchołka grafu stwierdzić, czy po jego usunięciu nadal będziemy dysponować jedną silnie spójną składową.

Krzysztof Sikora Treść zadania, Opracowanie
Paweł Wolff Program

Nawiasy

Słowem nawiasowym *będziemy nazywali słowo złożone z dwóch rodzajów znaków: nawiasów otwierających, czyli „(", oraz nawiasów zamykających, czyli „)". Wśród wszystkich słów nawiasowych będziemy wyróżniać* **poprawne wyrażenia nawiasowe**. *Są to takie słowa nawiasowe, w których występujące nawiasy można połączyć w pary w taki sposób, że:*

- *każda para składa się z nawiasu otwierającego oraz (występującego dalej w słowie nawiasowym) nawiasu zamykającego,*
- *dla każdej pary fragment słowa nawiasowego zawarty między nawiasami tej pary zawiera tyle samo nawiasów otwierających co zamykających.*

Na słowie nawiasowym można wykonywać operacje:

- **zamiany**, *która zamienia i-ty nawias w słowie na przeciwny,*
- **sprawdzenia**, *która sprawdza, czy słowo nawiasowe jest poprawnym wyrażeniem nawiasowym.*

Na pewnym słowie nawiasowym wykonywane są kolejno operacje zamiany lub sprawdzenia.

Zadanie

Napisz program, który:

- *wczyta (ze standardowego wejścia) słowo nawiasowe oraz ciąg operacji kolejno wykonywanych na tym słowie,*
- *dla każdej operacji sprawdzenia (występującej we wczytanym ciągu operacji) stwierdzi, czy bieżące słowo nawiasowe jest poprawnym wyrażeniem nawiasowym,*
- *wypisze wynik (na standardowe wyjście).*

Wejście

W pierwszym wierszu wejścia znajduje się jedna liczba całkowita n ($1 \leqslant n \leqslant 30\,000$) oznaczająca długość słowa nawiasowego. W drugim wierszu znajduje się n nawiasów bez znaków odstępu między nimi. W trzecim wierszu znajduje się jedna liczba całkowita m ($1 \leqslant m \leqslant 1\,000\,000$) oznaczająca liczbę operacji wykonywanych na słowie nawiasowym. W każdym z kolejnych m wierszy znajduje się jedna liczba całkowita. Jeśli w $(k+3)$-cim wierszu (dla $1 \leqslant k \leqslant m$) występuje liczba 0, to znaczy, że k-tą z kolei operacją wykonywaną na słowie nawiasowym jest operacja sprawdzenia. Jeśli zaś jest to liczba całkowita p spełniająca $1 \leqslant p \leqslant n$, to znaczy, że operacją tą jest operacja zamiany p-tego nawiasu na przeciwny.

Wyjście

Twój program powinien wypisać w kolejnych wierszach (standardowego wyjścia) wyniki kolejnych operacji sprawdzenia. Jeśli bieżące słowo nawiasowe jest poprawnym wyrażeniem nawiasowym, to należy wypisać słowo `TAK`*, w przeciwnym przypadku słowo* `NIE`*. (Na wyjściu powinno pojawić się tyle wierszy, ile operacji sprawdzenia zadano na wejściu.)*

Przykład

Dla danych wejściowych:

```
4
()((
4
4
0
2
0
```

poprawnym wynikiem jest:

```
TAK
NIE
```

Rozwiązanie

Wprowadzenie

Algorytm dynamiczny, to algorytm pozwalający obliczać funkcję f dla danych wejściowych x, które mogą zmieniać się w czasie. Bardziej formalnie implementuje on następujące operacje:

- *inicjalizacja* — inicjalizacja x
- *zmiana* — zmiana x
- *sprawdzanie* — obliczenie $f(x)$

Nasze zadanie polega na skonstruowaniu takiego właśnie algorytmu. Danymi wejściowymi jest słowo nawiasowe. Funkcja f odpowiada na pytanie, czy słowo jest poprawnym wyrażeniem nawiasowym, a operacje to:

- *inicjalizacja* — inicjalizacja struktury danych dla słowa wejściowego
- *zmiana* — zamiana i-tego nawiasu w słowie na przeciwny
- *sprawdzanie* — obliczenie wartości funkcji f

Rozwiązanie naiwne

Bardzo łatwo można zaimplementować operację *sprawdzanie* korzystając z następującego lematu.

Lemat 1 *Słowo s jest poprawnym wyrażeniem nawiasowym wtedy i tylko wtedy, gdy dla każdego prefiksu p słowa s liczba nawiasów zamykających w p jest nie większa niż liczba nawiasów otwierających w p oraz liczba nawiasów otwierających i zamykających w słowie s jest taka sama.*

Niestety złożoność obliczeniowa takiej implementacji jest liniowa ze względu na długość słowa wejściowego, a przez to niewystarczająca, gdyż cały algorytm działa wówczas w czasie $O(mn)$.

Rozwiązanie wzorcowe

Lemat 2 *Niech $s = s_1 \cdot s_2$ oraz l, p, l_1, p_1, l_2, p_2 oznaczają, odpowiednio, liczbę niesparowanych lewych nawiasów w słowie s, niesparowanych prawych nawiasów w słowie s, niesparowanych lewych nawiasów w słowie s_1, niesparowanych prawych nawiasów w słowie s_1, niesparowanych lewych nawiasów w słowie s_2 i niesparowanych prawych nawiasów w słowie s_2. Wtedy jeśli $l_1 \leqslant p_2$, to*

$$l = l_2,\ p = p_1 + p_2 - l_1,$$

w przeciwnym przypadku

$$l = l_1 + l_2 - p_2,\ p = p_1.$$

Definicja 1 Niech wartością funkcji $nawiasy(s)$ będzie para (l, p) wtedy i tylko wtedy, gdy słowo s ma l niesparowanych lewych i p niesparowanych prawych nawiasów.

Lemat 3 *Słowo s jest poprawnym wyrażeniem nawiasowym wtedy i tylko wtedy, gdy*

$$nawiasy(s) = (0, 0)$$

Funkcję *nawiasy* można obliczyć w czasie liniowym korzystając z lematu 2. Oprócz tego daje się ją w prosty sposób „zdynamizować”. Jeśli podczas jej obliczania słowo jest dzielone na pół (z dokładnością do jednego znaku) i dodatkowo przechowane są wartości dla wszystkich obliczanych podsłów, to operacja *zmiana* da się wykonać w czasie $O(\log n)$ i polega ona na uaktualnieniu logarytmicznie wielu obliczonych wcześniej wartości. W sumie otrzymuje dynamiczny algorytm, którego operacje mają złożoności czasowe:

- *inicjalizacja* $O(n)$ — obliczenie $nawiasy(s)$ oraz inicjalizacja struktury danych (tablicy) przechowującej wyniki dla wszystkich obliczanych podsłów
- *zmiana* $O(\log n)$ — uaktualnienie struktury danych
- *sprawdzanie* $O(1)$ — pobranie wartości ze struktury danych

W sumie całe rozwiązanie działa w czasie $O(m \log n)$. Złożoność pamięciowa jest liniowa, gdyż jest co najwyżej $2|s|$ podsłów, dla których funkcja *nawiasy* jest wywoływana.

Wojciech Guzicki — Treść zadania, Opracowanie
Michał Adamaszek — Program

Zbrodnia na Piccadilly Circus

Sherlock Holmes prowadzi śledztwo w sprawie zbrodni na Piccadilly Circus. Holmes zastanawia się, jaka była maksymalna, a jaka minimalna liczba osób przebywających równocześnie na miejscu zbrodni w czasie, gdy mogła zostać ona popełniona. Scotland Yard przeprowadził szczegółowe śledztwo, przesłuchał wszystkie osoby, które były widziane na miejscu zbrodni i ustalił, o której godzinie pojawiły sie one na miejscu zbrodni, a o której je opuściły. Doktor Watson zaofiarował się pomóc przetworzyć dane zgromadzone przez Scotland Yard i wyznaczyć liczby, które interesują Sherlocka Holmesa, ma jednak z tym problemy. Pomóż mu!

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia przedział czasowy, w którym została popełniona zbrodnia oraz dane zgromadzone przez Scotland Yard,*
- *wyznaczy minimalną (może to być zero, chociaż dziwne, żeby w czasie zbrodni nikt nie przebywał w miejscu, w którym się dokonała, ale właśnie takimi sprawami zajmują się Holmes i Watson) i maksymalną liczbę osób, które równocześnie przebywały na miejscu zbrodni w przedziale czasu, gdy mogła ona zostać popełniona,*
- *wypisze wyniki na standardowe wyjście.*

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie liczby całkowite p i k, $0 \leqslant p \leqslant k \leqslant 1\,000\,000\,000$. Są to, odpowiednio, najwcześniejsza i najpóźniejsza chwila, kiedy mogła zostać popełniona zbrodnia. Drugi wiersz standardowego wejścia zawiera jedną liczbę całkowitą n, $3 \leqslant n \leqslant 5\,000$. Jest to liczba osób przesłuchanych przez Scotland Yard. W każdym z kolejnych n wierszy są zapisane po dwie liczby całkowite — w wierszu $i+2$ zapisane są liczby a_i i b_i oddzielone pojedynczym odstępem, $0 \leqslant a_i \leqslant b_i \leqslant 1\,000\,000\,000$. Są to, odpowiednio, chwila pojawienia się i-tej osoby na miejscu zbrodni i jej odejścia. Oznacza to, iż i-ta osoba przebywała na miejscu zbrodni przez cały czas od chwili a_i do chwili b_i (włącznie).

Wyjście

Twój program powinien wypisać na standardowe wyjście, w pierwszym wierszu i jedynym wierszu, dwie liczby całkowite oddzielone pojedynczym odstępem: minimalną i maksymalną liczbę osób, które były równocześnie na miejscu zbrodni, w czasie od chwili p do chwili k (włącznie).

Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
1 4
```

Rozwiązanie

Rozwiązanie zadania polega na kolejnym zliczaniu osób przebywających na miejscu zbrodni. Najpierw zapiszemy w tablicy chwilę przyjścia i odejścia każdej osoby. W tym celu definiujemy typ `Chwila`, będący parą liczb: czasu przyjścia lub odejścia i liczby 0 lub 1 (0 oznacza przyjście, 1 odejście). Następnie tworzymy tablicę zawierającą wszystkie takie chwile. Ze względów technicznych dodajemy jedną osobę, która przyszła w chwili p i odeszła w chwili k; na końcu od wyniku odejmiemy 1. Istnienie takiej osoby znacznie uprości warunki zakończenia pętli; jest to tzw. *wartownik*. A oto deklaracje używanych typów i rozważanej tablicy.

```
const
  MaxN = 5000;
type
  Chwila = record
        Czas : longint;
        Stan : Byte
  end;
  Tablica = array [1..2*MaxN+2] of Chwila;
var
  T : Tablica;
```

Następnie wczytujemy dane: czasy p i k początku i końca rozważanego okresu, w którym nastąpiła zbrodnia oraz liczbę n osób. Potem do tablicy `T` wczytujemy czasy przyjścia i odejścia kolejnych osób, na końcu dodając naszą dodatkową osobę (wartownika). Wreszcie sortujemy dane zapisane w tablicy `T`.

Oczywiście chwile wcześniejsze umieszczamy w tablicy wcześniej. Ponieważ uważamy, że w chwili przyjścia lub odejścia dana osoba przebywała na miejscu zdarzenia, więc chwila przyjścia jednej osoby poprzedza tę samą chwilę odejścia innej osoby. Dokładniej, przyjmujemy, że `T[i]` poprzedza `T[j]` wtedy i tylko wtedy, gdy:

- `T[i].Czas<T[j].Czas` lub
- `T[i].Czas=T[j].Czas` oraz `T[i].Stan<T[j].Stan`.

Po posortowaniu danych następuje zliczanie osób. Najpierw zliczamy wszystkie osoby, które przebywały na miejscu zbrodni w chwili p. Możemy to zrobić w następujący sposób. Indeks `i` oznacza numer kolejnej chwili w tablicy `T` (czyli kolejnej osoby); `liczba` oznacza liczbę osób przebywających w danej chwili na miejscu zdarzenia. Na początku oczywiście `liczba=0`. Przed chwilą p wraz z przyjściem kolejnej osoby zwiększamy licznik `liczba` o 1, wraz z odejściem zmniejszamy o 1. W chwili p tylko zliczamy osoby przychodzące. Dodanie wartownika było przydatne do tego, by pierwsza pętla zatrzymała się dokładnie w chwili p oraz po to, by mieć pewność, że policzymy jakieś osoby w ostatniej pętli.

```
i:=1;
liczba:=0;
while T[i].Czas<p do begin
  if T[i].Stan=0 then liczba:=liczba+1 else liczba:=liczba-1;
  i:=i+1
end;
while (T[i].Czas=p) and (T[i].Stan=0) do begin
  liczba:=liczba+1;
  i:=i+1
end;
```

Teraz zapamiętujemy najmniejszą i największą liczbę osób przebywających na miejscu zdarzenia (używamy zmiennych `min` i `max`). Następnie zliczamy wszystkie osoby, które przyszły lub odeszły przed chwilą k oraz osoby, które przyszły w chwili k. Obecność wartownika znów daje łatwy warunek zakończenia pętli, bowiem na pewno chwila k występuje w tablicy. Na końcu odejmiemy naszą dodatkową osobę (wartownika).

```
min:=liczba;
max:=liczba;
while T[i].Czas<k do begin
  if T[i].Stan=0 then liczba:=liczba+1
                 else liczba:=liczba-1;
  if liczba<min then min:=liczba;
  if liczba>max then max:=liczba;
  i:=i+1
end;
while (T[i].Czas=k) and (T[i].Stan=0) do begin
  liczba:=liczba+1;
  i:=i+1
end;
if liczba>max then max:=liczba;
min:=min-1;
max:=max-1;
```

Pozostanie tylko wypisanie wyniku. Ten program jest bardzo nieefektywny i można go przyspieszyć w kilku miejscach. W czasie wczytywania danych możemy pominąć wszystkie osoby, które odeszły z miejsca zdarzenia przed chwilą p lub przyszły po chwili k. Możemy też zmodyfikować ich czasy przyjścia (jeśli ktoś przyszedł przed chwilą p, to możemy przyjąć, że przyszedł w chwili p; podobnie z chwilą odejścia). Teraz na początku wystarczy zliczyć osoby, które przyszły w chwili p. Na ogół zmniejszy to również rozmiar danych w tablicy i przyspieszy czas sortowania. Następnie możemy przyjąć prostsze kryterium sortowania: sortujemy tylko według czasu, nie zwracając uwagi na to, czy jest to moment przyjścia czy odejścia. Będzie to wymagało modyfikacji procedury zliczania: w każdej chwili zliczamy osoby przychodzące i osoby odchodzące. Następnie do zmiennej `liczba` dodajemy liczbę osób przychodzących (modyfikując w razie potrzeby zmienną `max`), a potem odejmujemy liczbę osób odchodzących (modyfikując zmienną `min`). Tak zoptymalizowany program został przyjęty jako program wzorcowy.

Jakub Pawlewicz **Krzysztof Onak**
Treść zadania, Opracowanie Program

Superliczby w permutacji

Permutacja n-elementowa jest ciągiem n-elementowym składającym się z różnych liczb ze zbioru $\{1, 2, \ldots, n\}$. *Przykładowo, ciąg* $2, 1, 4, 5, 3$ *jest permutacją 5-elementową.*

W permutacjach liczb będą interesować nas najdłuższe rosnące podciągi. W przykładowej permutacji mają one długość 3 i istnieją dokładnie dwa takie podciągi, a mianowicie $2, 4, 5$ *oraz* $1, 4, 5$.

Superliczbą *nazwiemy każdą liczbę, która należy do któregokolwiek z najdłuższych rosnących podciągów. W permutacji* $2, 1, 4, 5, 3$ *superliczbami są* $1, 2, 4, 5$*, zaś liczba 3 superliczbą nie jest.*

Twoim zadaniem jest dla zadanej permutacji znaleźć wszystkie superliczby.

Zadanie

Napisz program, który:

- *wczyta permutację ze standardowego wejścia,*
- *znajdzie wszystkie superliczby,*
- *wypisze znalezione superliczby na standardowe wyjście.*

Wejście

Wejście składa się z dwóch wierszy. W pierwszym wierszu znajduje się jedna liczba n, $1 \leqslant n \leqslant 100\,000$. *W drugim wierszu znajduje się n liczb tworzących permutację n-elementową, pooddzielanych pojedynczymi odstępami.*

Wyjście

Wyjście powinno się składać z dwóch wierszy. W pierwszym wierszu powinna znaleźć się jedna liczba m — liczba superliczb w wejściowej permutacji. W drugim powinny znaleźć się superliczby, pooddzielane pojedynczymi odstępami i wymienione w kolejności rosnącej.

Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
4
1 2 4 5
```

Rozwiązanie

Rozwiązanie wzorcowe wywodzi się z algorytmu na szukanie najdłuższego rosnącego podciągu. Oznaczmy zadaną permutację przez $\pi_1, \ldots, \pi_n$. Weźmy k pierwszych elementów permutacji: $\pi_1, \ldots, \pi_k$. Interesują nas l–elementowe podciągi rosnące ciągu $\pi_1, \ldots, \pi_k$. Im mniejszy jest ostatni element takiego podciągu, tym łatwiej jest go później wydłużyć, aby nadal był rosnący.

Definicja 1 Przez $S_{k,l}$ oznaczamy najmniejszy element, na który może się kończyć l–elementowy podciąg rosnący ciągu $\pi_1, \ldots, \pi_k$. Jeśli taki podciąg nie istnieje, to przyjmujemy $S_{k,l} = +\infty$. Ponadto przyjmujemy $S_{k,0} = -\infty$. Nieskończoności będą pełniły rolę wartowników.

Definicja 2 Przez L_k oznaczamy największe l takie, że $S_{k,l} < +\infty$. Innymi słowy L_k oznacza długość najdłuższego podciągu rosnącego ciągu $\pi_1, \ldots, \pi_k$.

Twierdzenie 1 *Dla ustalonego* $0 \leqslant k \leqslant n$ *ciąg* $S_{k,l}$ *jest rosnący:*

$$-\infty = S_{k,0} < S_{k,1} < \ldots < S_{k,L_k} < S_{k,L_k+1} = +\infty, \tag{17}$$

a dla $k \geqslant 1$ *jest dokładnie jedno takie* $i \geqslant 1$, *że*

$$S_{k-1,i-1} < \pi_k < S_{k-1,i}$$

oraz zachodzi wzór rekurencyjny:

$$S_{k,l} = \begin{cases} \pi_k & \text{dla } l = i, \\ S_{k-1,l} & \text{dla } l \neq i, l \geqslant 0. \end{cases} \tag{18}$$

Dowód Dowód indukcyjny po k. Na początku mamy $S_{0,0} = -\infty$, $S_{0,l} = +\infty$ dla $l \geqslant 1$ oraz $L_0 = 0$. Zatem (17) jest spełnione dla $k = 0$.

Niech teraz $k \geqslant 1$. Z założenia indukcyjnego mamy:

$$-\infty = S_{k-1,0} < S_{k-1,1} < \ldots < S_{k-1,L_{k-1}} < S_{k-1,L_{k-1}+1} = +\infty.$$

Zatem istnieje dokładnie jedno takie $i \geqslant 1$, że $S_{k-1,i-1} < \pi_k < S_{k-1,i}$. Popatrzmy ile wynosi $S_{k,l}$. Rozważmy trzy przypadki.

1. $0 \leqslant l \leqslant i-1$. Wtedy $\pi_k > S_{k-1,l}$ i nie uda nam się zmniejszyć najmniejszego końcowego elementu l–elementowego podciągu rosnącego ciągu $\pi_1, \ldots, \pi_k$, który wynosi $S_{k-1,l}$, więc $S_{k,l} = S_{k-1,l}$.

2. $l = i$. Dokładając do $l-1$–elementowego podciągu kończącego się na $S_{k-1,i-1}$ element π_k dostajemy nowy rosnący podciąg l–elementowy. Ma on ostatni element mniejszy niż $S_{k-1,l}$, więc możemy zmniejszyć ostatni element rosnącego podciągu z $S_{k-1,l}$ na π_k. Zatem w tym przypadku mamy $S_{k,l} = \pi_k$.

3. $i < l$. Tym razem l–elementowy podciąg rosnący ciągu $\pi_1, \ldots, \pi_k$ nie może się kończyć na π_k, gdyż najmniejszy ostatni element $l-1$–elementowego podciągu rosnącego ciągu $\pi_1, \ldots, \pi_{k-1}$ wynosi $S_{k-1,l-1} > \pi_k$, więc mamy $S_{k,l} = S_{k-1,l}$.

Udowodniliśmy wzór (18). Przy okazji mamy

$$L_k = \begin{cases} L_{k-1}+1 & \text{dla } i = L_{k-1} \\ L_{k-1} & \text{w p.p.} \end{cases}$$

Porównując wartości $S_{k,l}$ z wartościami $S_{k-1,l}$ widzimy, że zmienia się tylko jedna pozycja $l = i$ i dla niej jest $S_{k,i-1} < S_{k,i} < S_{k,i+1}$, więc zachowana jest własność (17). ■

Z twierdzenia 1 wynika następujący iteracyjny algorytm na znajdowanie najdłuższego podciągu rosnącego. Utrzymujemy tablicę $S[0,\ldots,n+1]$ wartości $S_{k,l}$. Inicjujemy $S[0] := -\infty$ i $S[l] := +\infty$ dla $l = 1,\ldots,n+1$. Teraz wykonujemy n kroków, dla $k = 1,\ldots,n$. W k–tym kroku za pomocą wyszukiwania binarnego znajdujemy i takie, że $S[i-1] < \pi_k < S[i]$ i przypisujemy $S[i] := \pi_k$. Po wykonaniu n kroków największe L takie, że $S[L] < +\infty$, jest długością najdłuższego podciągu rosnącego. Algorytm ten działa w czasie $O(n \log n)$ i pamięci $O(n)$.

Niewielka modyfikacja powyższego algorytmu umożliwi nam wyliczanie dodatkowej informacji. Mianowicie dla każdego k, $1 \leqslant k \leqslant n$, możemy policzyć długość $E[k]$ najdłuższego podciągu rosnącego kończącego się na π_k. Wystarczy obok przypisania $S[i] := \pi_k$ dodać przypisanie $E[k] := i$. Pseudokod wyliczający tablicę $E[\cdot]$ wygląda następująco:

```
S[0] := −∞
forall 1 ⩽ l ⩽ n+1 do S[l] := +∞
for k := 1 to n do
   wyszukaj binarnie i takie, że S[i−1] < π_k < S[i]
   S[i] := π_k
   E[k] := i
```

Możemy zdefiniować również tablicę $B[\cdot]$. Niech $B[k]$ oznacza długość najdłuższego podciągu rosnącego zaczynającego się na π_k. Dla tak zdefiniowanych $B[\cdot]$ i $E[\cdot]$, długość najdłuższego podciągu rosnącego do jakiego należy π_k wynosi $B[k]+E[k]-1$. Zatem, żeby znaleźć wszystkie superliczby wystarczy znaleźć wszystkie k spełniające $B[k]+E[k]-1 = L$, gdzie L jest długością najdłuższego podciągu rosnącego ciągu $\pi_1,\ldots,\pi_n$.

Pozostaje kwestia znalezienia tablicy $B[\cdot]$. Metoda jest identyczna, jak w przypadku szukania tablicy $E[\cdot]$. Możemy skonstruować analogiczny algorytm szukania najdłuższych podciągów rosnących od tyłu. Będzie to po prostu szukanie najdłuższych podciągów malejących dla permutacji $\pi_n,\ldots,\pi_2,\pi_1$, co można w prosty sposób sprowadzić do szukania najdłuższych podciągów rosnących ciągu $-\pi_n,\ldots,-\pi_2,-\pi_1$. Zatem możemy wykorzystać istniejący kod dla tablicy $E[\cdot]$ do liczenia tablicy $B[\cdot]$, tylko trzeba ją odwrócić:

```
znajdź tablicę E'[·] dla ciągu −π_n, ..., −π_2, −π_1
forall 1 ⩽ k ⩽ n do B[k] := E'[n+1−k]
```

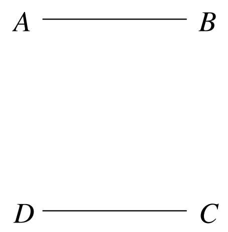
Przedstawione rozwiązanie zadania działa w czasie $O(n \log n)$ i pamięci $O(n)$.

Piotr Chrząstowski
Treść zadania, Opracowanie

Krzysztof Sikora
Program

Tańce gordyjskie Krzyśków

Na zakończenie tegorocznej edycji Pogromców Algorytmów Krzyśki odpowiedzialne za prawidłowy przebieg zawodów (KC, KD, KO, KS) postanowili odtańczyć radosny taniec gordyjski. Taniec gordyjski to tradycyjny bajtocki taniec tańczony przez dwie pary tancerzy. Początkowo tancerze stoją w wierzchołkach kwadratu $ABCD$, w dwóch parach: A–B i C–D. Każda z par rozciąga między sobą sznurek. Tak więc na początku oba sznurki są rozciągnięte poziomo i równolegle do siebie.



Początkowe ustawienie tancerzy.

Taniec składa się z ciągu ruchów, z których każdy może być ruchem następującego rodzaju:

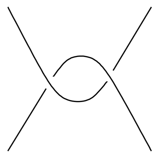
- *(S) Tancerze stojący w punktach B i C zamieniają się miejscami (nie puszczając swoich sznurków) w ten sposób, że tancerz stojący w punkcie B podnosi rękę ze sznurkiem do góry i idąc do punktu C przepuszcza tancerza idącego z punktu C do B przed sobą, pod swoją ręką.*
- *(R) Wszyscy tancerze wykonują obrót o 90 stopni w prawo nie puszczając sznurków, czyli tancerz, który stał w punkcie A idzie do punktu B, ten, który stał w punkcie B idzie do punktu C, ten, który stał w punkcie C idzie do punktu D, a ten, który stał w punkcie D idzie do punktu A.*

W trakcie tańca sznurki plączą się ze sobą, jednak na koniec tańca powinny zostać rozplątane i znowu być rozciągnięte poziomo i równolegle do siebie. Tancerze nie muszą przy tym stać na tych samych miejscach, na których stali na początku. Taniec ten wymaga od tancerzy dużej wprawy, gdyż w trakcie tańca sznurki mogą być bardzo splątane i ciąg ruchów, który prowadziłby do ich rozplątania i rozciągnięcia poziomo i równolegle do siebie, może być trudny do odgadnięcia.

Krzyśki to początkujący tancerze. Twoje zadanie polega na napisaniu programu, który pomógłby im zakończyć rozpoczęty taniec. Na podstawie ciągu już wykonanych ruchów, Twój program powinien wyznaczyć minimalną liczbę ruchów pozwalających zakończyć taniec.

Przykład

Przykładowo, po wykonaniu ciągu ruchów SS otrzymujemy następującą konfigurację:



Najkrótszy ciąg ruchów, który pozwala zakończyć taniec, ma długość 5 i jest nim RSRSS.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis ciągu wykonanych ruchów w tańcu,*
- *wyznaczy minimalną liczbę ruchów potrzebnych do rozplątania sznurków i rozciągnięcia ich poziomo i równolegle do siebie (po wykonaniu tych ruchów tancerze nie muszą znajdować się na swoich początkowych pozycjach),*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu standardowego wejścia zapisana jest jedna dodatnia liczba całkowita n równa liczbie wykonanych ruchów, $1 \leqslant n \leqslant 1000000$. W drugim wierszu zapisane jest jedno słowo długości n złożone z liter `S` *i/lub* `R`*. Kolejne litery tego słowa reprezentują kolejno wykonane ruchy w tańcu.*

Wyjście

Twój program powinien wypisać na standardowe wyjście, w pierwszym i jedynym wierszu, jedną liczbę całkowitą — minimalną liczbę ruchów (S i/lub R) koniecznych do rozplątania sznurków i rozciągnięcia ich poziomo i równolegle do siebie.

Przykład

Dla danych wejściowych:
```
2
SS
```
poprawnym wynikiem jest:
```
5
```

Rozwiązanie

Postępując w sposób opisany w treści zadania można nieźle zaplątać sznurki. Miłe z kolei jest to, że wykonując w odpowiedniej kolejności wspomniane operacje zawsze jesteśmy w stanie dojść do stanu wyjściowego, co wcale nie jest oczywiste i co wkrótce wykażemy. Pamiętajmy tu od razu o zastrzeżeniu czynionym w treści zadania, że to, kto trzyma sznurki, jest nieistotne, podobnie jak to, który sznurek jest gdzie i w którą stronę jest skierowany. Mówiąc ściślej, *abstrahujemy* od nieistotnych różnic — interesuje nas jedynie rodzaj otrzymanego węzła. Tak samo wyglądające węzły uznajemy za *równoważne* niezależnie od tego kto, gdzie i jak trzyma końce sznurków. Za to zwracamy uwagę na to, czy węzeł jest ułożony pionowo, czy poziomo.

O klasyfikację wszystkich takich węzłów, jakie zdefiniowano w zadaniu, jak również wielu innych, pokusił się znakomity matematyk John H. Conway (inicjał drugiego imienia jest o tyle istotny, że żyje w tej chwili dwóch matematyków o tym samym pierwszym imieniu i nazwisku — to dla tych, którzy zechcą w Internecie poszukać więcej o działalności „naszego" Conwaya, a warto!). John H. Conway znany jest szeroko, jako twórca gry LIFE, najpopularniejszego chyba automatu komórkowego. Jego twierdzenie o charakteryzacji tangli (tak nazwał węzły występujące w naszym zadaniu) jest niezwykłej wprost urody, a że może być podstawą rozwiązania, przedstawimy je tu z radością.

Do tego potrzebne będzie nam pojęcie *izomorfizmu*, jednego z podstawowych pojęć w całej matematyce. Intuicyjnie oznacza ono nierozróżnialność pewnych obiektów, jeśli ograniczymy się do zestawu operacji (lub ogólniej relacji), które będą określone na obiektach dwóch zbiorów. *Algebrą* $\mathcal{A} = \langle A, o_1, \ldots, o_n \rangle$ nazwiemy zbiór A wraz z zestawem operacji $o_1, \ldots, o_n$. Zbiór A nazwiemy *nośnikiem algebry* $\mathcal{A}$. Dwie algebry mogą być różne, mimo że ich nośniki są identyczne. Na przykład algebry $\langle R, + \rangle, \langle R, - \rangle, \langle R, +, * \rangle$ są wszystkie różne. Dla dwóch zadanych algebr dobrze jest wiedzieć, czy zachowują się tak samo. Oczywiście, aby algebry uznać za tak samo się zachowujące, konieczne jest, aby operacji w każdej z nich było tyle samo i żeby odpowiadające sobie operacje miały po tyle samo argumentów. Będziemy też żądali, żeby nośniki algebr były tak samo liczne, czyli żeby istniała funkcja różnowartościowa odwzorowująca jeden zbiór na drugi.

Jeżeli na przykład przyjmiemy za nośnik jednej algebry zbiór liczb rzeczywistych dodatnich, a drugiej ujemnych, a w obu algebrach byłaby określona jedna operacja dodawania, to można by znaleźć funkcję $f : R_+ \to R_-$, która tak sparuje elementy tych dwóch zbiorów, że wykonanie dodawania w jednym zbiorze da wynik, dla którego wartość tej funkcji będzie wynikiem dodawania obrazów argumentów. Innymi słowy $f(x_1 +_1 x_2) = f(x_1) +_2 f(x_2)$ (indeksy przy dodawaniu podkreślają to, że dodawanie odbywa się w innych dziedzinach, choć z punktu widzenia całego zbioru liczb rzeczywistych daje te same wyniki, co zwykłe dodawanie). Taka funkcja, to $f(x) = -x$. Równie dobra byłaby np. funkcja $g(x) = -2x$. Zauważmy też, że gdybyśmy dołączyli jeszcze operację odejmowania (nie zawsze określoną, bo czasami wynik ucieka poza zakres), to również nie sposób byłoby odróżnić wspomniane algebry: operacja odejmowania byłaby określona w zbiorze R_+ dla argumentów x_1, x_2 wtedy i tylko wtedy, gdy byłaby określona w zbiorze R_- dla argumentów $f(x_1), f(x_2)$ i w takim przypadku, podobnie jak dla dodawania, dawałaby izomorficzne wyniki dla izomorficznych argumentów, czyli $f(x_1 - x_2)$ byłoby równe $f(x_1) - f(x_2)$. Zauważmy jeszcze, że formalnie rzecz biorąc dodawania (podobnie zresztą jak odejmowania) w zbiorach R_+ i R_-, to dwie różne operacje. Ustalając izomorfizm dwóch algebr musimy wskazać, które elementy nośnika pierwszej algebry odpowiadają którym elementom drugiej (u nas odpowiedniość tę wyznacza np. funkcja $f(x) = -x$) oraz którym operacjom pierwszej algebry odpowiadają, które operacje drugiej algebry (u nas operacji $+_1$ odpowiada operacja $+_2$, a operacji $-_1$ operacja $-_2$). Powiemy zatem, że algebry $\langle R_+, +_1, -_1 \rangle$ oraz $\langle R_-, +_2, -_2 \rangle$ są izomorficzne, gdyż ustaliliśmy stosowne odpowiedniości między nośnikami i operacjami obu algebr.

Gdybyśmy dołączyli do zbioru operacji w obu algebrach mnożenie, to takie algebry nie byłyby izomorficzne, gdyż w algebrze drugiej mnożenie byłoby operacją niedającą rezultatu dla żadnych argumentów — wynik wykraczałby poza dziedzinę, chyba, że sztucznie określilibyśmy mnożenie w liczbach ujemnych niestandardowo, jako $x_1 *_2 x_2 = -x_1 x_2$. Wtedy znów algebry $\langle R_+, +_1, -_1, *_1 \rangle$ oraz $\langle R_+, +_2, -_2, *_2 \rangle$ byłyby izomorficzne dla operacji $*_1$ określonej standardowo jako mnożenie w R_+. Zauważmy jeszcze, że gdyby przyjąć standardowe mnożenie, jako operację w drugiej algebrze, to byłyby one rozróżnialne: wystarczyłoby zapytać o wynik mnożenia dla odpowiadających sobie elementów, np. w pierwszej algebrze $1 * 2$ byłoby równe 2, natomiast w drugiej $(-1) * (-2)$ byłoby działaniem nieokreślonym, zatem usłyszawszy obie odpowiedzi wiedzielibyśmy, o której algebrze rozmawiamy. Te algebry byłyby zatem nieizomorficzne.

Tutaj mieliśmy do czynienia z dość prostą funkcją ustalającą izomorfizm. Światy tych dwóch algebr były jak gdyby wzajemnym lustrzanym odbiciem względem funkcji $f(x) = -x$. Czasem taki izomorfizm nie jest oczywisty. Zauważmy, że algebry $\langle R_+, * \rangle$ oraz $\langle R, + \rangle$ są izomorficzne, a funkcją ustalającą taki izomorfizm jest dowolna z funkcji $f(x) = log_a(x)$, dla $1 \neq a > 0$. Funkcja logarytm jest bowiem różnowartościowa i „na", a ponadto zachodzi wzór $f(x*y) = f(x) + f(y)$. Przy okazji zauważmy, że podstawa logarytmu jest tu nieistotna: każda liczba dodatnia a różna od jedynki ustala izomorfizm tych dwóch algebr.

Zapytajmy jeszcze po co w ogóle rozważać izomorfizmy. Gdy dwie algebry są izomorficzne, może się okazać, że wykonywanie operacji w jednej z nich jest technicznie prostsze, niż w drugiej. Zatem mamy możliwość wykonywania operacji w tej z algebr, w której się to robi prościej. Jeśli tylko umielibyśmy w prosty sposób tłumaczyć z nośnika jednej algebry w drugą i na odwrót, to mając do wykonania działanie w algebrze „trudniejszej", moglibyśmy przetłumaczyć argumenty za pomocą funkcji f ustalającej odpowiedniość między nośnikami, wykonać prostsze działanie i za pomocą funkcji f^{-1}, odwrotnej do f, uzyskać wynik w oryginalnej algebrze.

Pamiętajmy, że funkcja odwrotna zawsze istnieje: wszak f musi być $1-1$ i „na".

Wszyscy się chyba zgodzimy, że dodaje się prościej, niż mnoży. Zatem jeśli umiemy szybko logarytmować i antylogarytmować (czyli podnosić do potęgi), to zamiast żmudnego mnożenia moglibyśmy postąpić następująco. Szukając wyniku mnożenia x przez y, znajdujemy ich logarytmy, dodajemy je do siebie i szukamy liczby, której logarytmem byłby otrzymany wynik dodawania. Tak zresztą postępowano przez parę wieków, od czasu gdy Napier wynalazł logarytmy i ułożył ich tablice, które pozwalały szybko znajdować zarówno logarytmy, jak i antylogarytmy. Działania te wykonywano w sposób przybliżony, ale do wielu celów wystarczająco dokładny. Na tej zasadzie działał też suwak logarytmiczny — podstawowe narzędzie inżynierów epoki przedkalkulatorowej.

Ten cały wstęp był potrzebny, aby przedstawić rozwiązanie naszego zadania za pomocą tej samej techniki. Zauważmy bowiem, że węzły tworzą algebrę: nośnikiem są wszystkie rodzaje węzłów, a operacjami dwie figury taneczne, które przekształcają węzły w węzły — obie są operacjami jednoargumentowymi. Wykorzystamy tu twierdzenie Conway'a, które skomplikowaną algebrę węzłów „tłumaczy" w prostą algebrę liczb wymiernych z nieskończonością i z dwiema szczególnymi operacjami. Pierwszy problem polega na znalezieniu odpowiedników figur tanecznych w lepiej wyobrażalnej algebrze liczb wymiernych. Rozszerzmy więc zbiór liczb wymiernych o nieskończoność i rozważmy zbiór $Q_\infty = Q \cup \infty$, gdzie Q jest zbiorem liczb wymiernych, jako nośnik naszej algebry. Zbiór Q_∞ nazwiemy zbiorem liczb superwymiernych. Operacjami, odpowiadającymi operacjom S i R, będą $s(x) = x + 1$ i $r(x) = -\frac{1}{x}$. Zakładamy tu, że $r(0) = \infty$, $r(\infty) = 0$, $s(\infty) = \infty$. Początkowy układ poziomych węzłów odpowiadał będzie liczbie 0. Nasza algebra $Q_\infty = \langle Q_\infty, r, s\rangle$ okazuje się być izomorficzna z algebrą węzłów, w której nośnikiem są wszystkie rodzaje węzłów możliwych do otrzymania za pomocą operacji R i S, a te dwie jednoargumentowe operacje są jedynymi rozważanymi. To jest właśnie treścią twierdzenia Conway'a, którego dowodu nie będziemy tu przytaczali. Wykonując w odpowiedniej kolejności operacje s i r możemy z zera uzyskać dowolną liczbę wymierną. Wykonując analogiczne ciągi operacji S i R możemy uzyskać wszystkie węzły zgodnie z treścią zadania. Każdy z tych węzłów może więc uzyskać unikalny numer zwany numerem Conway'a i będący odpowiadającą mu liczbą superwymierną. Różnym węzłom odpowiadają różne numery Conway'a i na odwrót: różnym liczbom superwymiernym odpowiadają różne węzły. Zauważmy przy okazji, że wykonanie dwóch kolejnych operacji r po sobie na liczbach superwymiernych jest identycznością, podobnie jak wykonanie dwóch operacji R na węzłach: dwukrotny obrót o 90 stopni jest symetrią środkową, która nie zmienia węzła, a jedynie zamienia tych, którzy trzymają końce i miejsce położenia końców.

Zobaczmy na kilku przykładach, jak to wszystko działa. Układ początkowy to 0. Wykonanie operacji R prowadzi nas do węzła „nieskończoność", jako że $r(0) = \infty$. Powtórne wykonanie R oznacza powrót do zera, bo $r(\infty) = 0$. Zatem węzły całkowicie rozplątane, poziome i pionowe, odpowiadają liczbom 0 i ∞. Przy okazji: dla układu pionowego wykonanie operacji S nie zmienia go, gdyż $s(\infty) = \infty$.

Jak wygląda sytuacja po wykonaniu pojedynczego S w stanie wyjściowym na węzłach? Ano węzeł, który uzyskujemy ma numer 1, bo $s(0) = 1$. Zgodnie z twierdzeniem Conway'a tylko jeden węzeł ma numer jeden i aby go rozplątać, należy za pomocą operacji s i r przekształcić jedynkę w zero. Zauważmy, że w tym celu wystarczy wykonać ciąg $r \cdot s$, bo $s(r(1)) = s(-1) = 0$. Łatwo sprawdzić, że faktycznie wykonanie kolejno operacji R, a potem S, doprowadzi nas do węzła wyjściowego.

Gdybyśmy wykonali dwa S-y, tak jak w przykładzie z treści zadania, otrzymalibyśmy węzeł o numerze 2. Najprostsza metoda rozwikłania go to taka, która za pomocą minimalnej liczby operacji r i s przekształci dwójkę w zero. Mamy więc $2 \xrightarrow{r} -\frac{1}{2} \xrightarrow{s} \frac{1}{2} \xrightarrow{r} -2 \xrightarrow{s} -1 \xrightarrow{s} 0$. Ponieważ ciąg operacji *rsrss* w algebrze liczb superwymiernych przekształca liczbę 2 w 0, więc ciąg operacji *RSRSS* w algebrze węzłów doprowadzi nas od węzła $S(S(0))$ do punktu wyjściowego.

Teraz rozplątać możemy każdy węzeł korzystając z izomorfizmu naszych algebr. Mając ciąg zaplątań złożony z operacji S i R obliczamy numer otrzymanego węzła przez wykonanie analogicznego ciągu operacji s i r na liczbach superwymiernych, a następnie przekształcamy otrzymany numer możliwie krótką sekwencją operacji s i r doprowadzającą do zera. Rozwiązaniem będzie długość tej sekwencji.

Sprowadziliśmy zatem nasz problem do następującego: jak mając daną liczbę superwymierną możliwie szybko uzyskać zero za pomocą operacji s i r? Algorytm, który liczby dodatnie będzie natychmiast zamieniał na ujemne za pomocą operacji r, a na liczbach ujemnych będzie wykonywał operację s tak długo, aż otrzymamy liczbę nieujemną, jest tu optymalny.

Wiadomo, że za wyjątkiem sytuacji, gdy zaczynamy od węzła ∞, czyli pionowych sznurków, ostatnim ruchem musi być s. Zatem będziemy atakowali zero od dołu: naszym celem powinno być osiągnięcie w miarę szybko ujemnej liczby całkowitej, a następnie wykonanie odpowiedniej liczby s-ów. Zrobimy to w taki sposób, że mianowniki kolejnych liczb ujemnych będą malały do jedności. Ostatnia liczba ułamkowa dodatnia powinna być postaci $\frac{1}{k}$ dla pewnego k i końcówka algorytmu będzie wyglądała tak, że po uzyskaniu $-k$, wykonamy k-krotnie operację s. Z kolei liczbę $\frac{1}{k}$ można uzyskać tylko za pomocą operacji s (inaczej bezsensownie wyskoczylibyśmy z „dobrej" ujemnej liczby całkowitej). Niżej pokażemy, że podana metoda daje zawsze minimalną liczbę ruchów rozplatującą węzeł.

Możemy więc zastosować następujący algorytm.

Niech l/m będzie liczbą wymierną lub minus nieskończonością reprezentującą zaplątanie sznurków.

1. `if` $l = 0$ `then` return
2. `else if` $l/m > 0 \vee m = 0$ `then` $(l,m) := (-m,l)$ // (operacja R)
3. `else` $l := l \bmod m$ (ciąg operacji S)

Udowodnimy, że ten algorytm prowadzi do rozplątania sznurków za pomocą minimalnej liczby ruchów.

Wzór 1 Dla liczb naturalnych $l > 0$ i $m > 0$

$$l = (-m) \bmod (l+m) \tag{19}$$

Dowód wzoru 1

$$l = (-m) \bmod (l+m) = -m - \left\lfloor \frac{-m}{l+m} \right\rfloor (l+m)$$

wtedy i tylko wtedy, gdy

$$l+m = -\left\lfloor \frac{-m}{l+m} \right\rfloor (l+m)$$

wtedy i tylko wtedy, gdy

$$1 = -\left\lfloor \frac{-m}{l+m} \right\rfloor$$

wtedy i tylko wtedy, gdy

$$1 = \left\lceil \frac{m}{l+m} \right\rceil$$

■

Lemat 1 Algorytm się zatrzymuje.

Dowód lematu 1 Pokażemy, że dla dowolnej liczby wymiernej ujemnej postaci $-\frac{l}{m}$, dla $l,m > 0$, wykonywanie kolejnych kroków algorytmu doprowadzi nas w końcu do wartości $x = 0$. Jeśli $m = 1$, to po l-krotnym wykonaniu S dostaniemy zero. Jeśli zaś $m > 1$, to wykonując kroki algorytmu dojdziemy do innej liczby wymiernej $-\frac{l'}{m'}$ takiej, że $m > m'$. Niech bowiem $x = -\frac{l}{m}$ oraz $l > 0$ i $m > 1$. Dla $x < 0$ wykonujemy ciąg operacji S i po jego wykonaniu dojdziemy do sytuacji, gdy x po raz pierwszy stanie się większe od zera. Niech wtedy $x = \frac{l_1}{m}$, gdzie $m > l_1 > 0$. Teraz wykonujemy R i mamy $x = -m/l_1$. Mianownik zmalał, a ułamek jest ujemny. Zatem zawsze po wykonaniu ciągu S, aż do uzyskania liczby dodatniej, zakończonego jednym R, dostaniemy ułamek ujemny o mniejszym mianowniku. Nie można w nieskończoność zmniejszać dodatniego mianownika. Zatem po skończonej liczbie kroków mianownik stanie się równy 1, a to jak wiemy doprowadza do zatrzymania całego algorytmu. ■

Lemat 2 Algorytm wykonuje minimalną liczbę kroków.

Dowód lematu 2 Rozpatrzmy możliwe ruchy dla $l \neq 0, m \geqslant 0$.

- $m = 0$ i S — oczywiście zły ruch, bo nie zmienia węzła.
- $m = 0$ i R — dobry, bo w jednym ruchu doprowadza nas do zera. Szybciej nie można.
- $l,m > 0$ i S — zły.
  Wykonajmy S, a potem tak jak każe algorytm i pokażmy, że ta sekwencja jest dłuższa niż najlepsza możliwa. Mamy bowiem

  $$l/m \to (l+m)/m \to -m/(l+m) \to ((-m) \bmod (l+m))/(l+m) =$$

  $$\text{(ze wzoru 19)} \ = l/(l+m) \to -(l+m)/l \to ((-(l+m)) \bmod l)/l = ((-m) \bmod l)/l$$

  czyli 5 kroków. Lepszą sekwencją, po której dochodzimy do tej samej liczby, jest $l/m \to -m/l \to ((-m) \bmod l)/l$, co wymaga 2 kroków.
- $l,m > 0$ i R — dobry.
- $m > 0, l < 0$ i S — dobry.
- $m > 0$ i $l < 0$ i R — zły.
  Niech bowiem $l_1 = -l > 0$, po wykonaniu R mamy $x = m/l_1$. Teraz wykonanie R nie ma sensu, gdyż wrócimy do stanu poprzedniego, a wykonanie S jest złe ($m/l_1 > 0$).

Teraz wystarczy jeszcze pokazać, że jakikolwiek zły krok nie prowadzi do optymalnego rozwiązania. Jeśli będziemy wykonywać tylko S dla liczby dodatniej, to nigdy nie dojdziemy do rozplątania, bo będziemy zwiększać liczbę dodatnią i nie osiągniemy zera. Załóżmy więc, że po złym kroku S (dla liczby dodatniej) kiedyś wykonamy R, czyli tak jak nakazuje algorytm. Wykonanie dobrego kroku po złym to już przypadek rozpatrywany wcześniej — zawsze można lepiej wyjść na niewykonywaniu tego ostatniego S, tylko pójściu na skróty. Jeśli więc S było wykonane dla liczby dodatniej, to tylko zwiększyło niepotrzebnie liczbę kroków konieczną do rozplątania. Dla operacji R jest jeszcze łatwiej, gdyż R^{2k} nic nie daje, a $R^{2k+1} = R$. Sensowne jest zatem wykonywanie jedynie pojedynczych R przetykanych S-ami.

Podsumowując: R-y należy wykonywać tylko jednokrotnie, a S-y tylko dla liczb ujemnych. I tak właśnie działa nasz algorytm, więc dochodzi najszybciej do stanu końcowego.

Wiemy już jak dojść do rozwiązania końcowego, pozostaje więc zliczyć ile potrzebujemy ruchów. Jeśli algorytm wykonuje operację R, to zwiększamy licznik o 1, gdy zaś S (właściwie ciąg operacji S) dodajemy do licznika $\left|\lfloor \frac{l}{m} \rfloor\right|$. Rzecz jasna zakresy danych wymuszały implementację własnej arytmetyki umożliwiającej obliczenie tych działań.

Złożoności Wykorzystując dowód lematu 2 widać, iż przy zaplątywaniu operacja S zwiększa liczbę kroków algorytmu o 3. Liczba operacji S w ciągu wejściowym jest oczywiście niewiększa niż jego długość. Złożoność czasowa algorytmu wynosi więc $O(n)$, gdzie n jest długością ciągu podanego na wejściu.
Złożoność pamięciowa jest $O(1)$.

Wojciech Guzicki
Treść zadania, Opracowanie

Marcin Mucha
Program

Tomki

Oprogramowanie używane w Pogromcach Algorytmów to dzieło Tomka W. W tym roku opiekuje się nim Tomek M. Tomek M. bardzo się stara, żeby konkurs przebiegał bez zarzutu. To jest powodem tego, że Tomek miewa ostatnio koszmarne sny. Wczoraj śniło mu się, że jest skoczkiem, który skacze po nieskończonej szachownicy. Ruchy, jakie może wykonywać skoczek, są opisane przez dwie pary liczb całkowitych: (a,b) i (c,d) — skoczek stojący na polu (x,y) może wykonać ruch na pole $(x+a, y+b)$, $(x-a, y-b)$, $(x+c, y+d)$ lub $(x-c, y-d)$. Tomek zastanawiał się (oczywiście we śnie), dla jakiego pola (x,y) różnego od $(0,0)$, na które może doskoczyć startując z pola $(0,0)$ (być może w wielu skokach), wartość $|x|+|y|$ jest najmniejsza.[1] Tomek obudził się zlany potem, bo nie mógł poradzić sobie z tym zadaniem i nie wiedział, czy nie oddali się od punktu $(0,0)$ tak daleko, że nie zdąży przygotować ostatniej tury Pogromców Algorytmów.

Zadanie

Twoje zadanie polega na napisaniu programu, który:

- *wczyta ze standardowego wejścia dwie pary liczb całkowitych (a,b) i (c,d), różne od $(0,0)$, opisujące ruchy skoczka,*
- *wyznaczy taką parę liczb całkowitych (x,y) różną od $(0,0)$, dla której skoczek może doskoczyć (być może w wielu skokach) z pola $(0,0)$ na pole (x,y) i dla której wartość $|x|+|y|$ jest najmniejsza,*
- *wypisze na standardowe wyjście wartość $|x|+|y|$.*

Wejście

W pierwszym i jedynym wierszu standardowego wejścia znajdują się cztery liczby całkowite: a, b, c i d, $-100\,000 \leqslant a,b,c,d \leqslant 100\,000$.

Wyjście

Twój program powinien wypisać na standardowe wyjście, w pierwszym i jedynym wierszu, jedną liczbę całkowitą równą $|x|+|y|$.

Przykład

Dla danych wejściowych:
```
13 4 17 5
```

[1] Uwaga: |p| oznacza wartość bezwględną liczby p, czyli p, gdy $p \geqslant 0$, i $-p$, gdy $p < 0$.

poprawnym wynikiem jest:
2

Rozwiązanie

Zadanie dotyczy problemu znajdowania najkrótszego niezerowego wektora w kracie. Wyjaśnimy najpierw, co to są kraty na płaszczyźnie.

Pojęcie wektora na płaszczyźnie jest dobrze znane ze szkoły. Jeśli mamy wektor $\overrightarrow{a}$, to jego współrzędne będziemy oznaczać symbolami a_1 i a_2; tak więc $\overrightarrow{a} = [a_1, a_2]$. Przypuśćmy teraz, że mamy dane dwa wektory $\overrightarrow{a}$ i $\overrightarrow{b}$ na płaszczyźnie. Oznaczmy przez $L(\overrightarrow{a}, \overrightarrow{b})$ następujący zbiór wektorów:

$$L(\overrightarrow{a}, \overrightarrow{b}) = \left\{m \cdot \overrightarrow{a} + n \cdot \overrightarrow{b} : \ m, n \in \mathbb{Z}\right\},$$

gdzie $\mathbb{Z}$ oznacza zbiór liczb całkowitych. Zbiór $L(\overrightarrow{a}, \overrightarrow{b})$ nazywamy *kratą* rozpiętą przez wektory $\overrightarrow{a}$ i $\overrightarrow{b}$. Będziemy zawsze zakładać, że wektory te nie są równoległe; w szczególności oba są niezerowe. Nazwę kraty wyjaśnią dwa przykłady.

Przykład 1 Niech $\overrightarrow{a} = [1,0]$ i $\overrightarrow{b} = [0,1]$. Wtedy wektory $\overrightarrow{a}$ i $\overrightarrow{b}$ rozpinają następującą kratę $L(\overrightarrow{a}, \overrightarrow{b})$:

$$L(\overrightarrow{a}, \overrightarrow{b}) = \left\{[m,n] : \ m, n \in \mathbb{Z}\right\}.$$

Jeśli wszystkie wektory kraty $L(\overrightarrow{a}, \overrightarrow{b})$ zaczepimy w początku układu współrzędnych, to ich końce będą w punktach o obu współrzędnych całkowitych, czyli tzw. punktach kratowych. Punkty te są punktami przecięcia linii przedstawionych na rysunku 1.

Przykład 2 Niech $\overrightarrow{a} = [2,0]$ i $\overrightarrow{b} = [1,2]$. Można łatwo zauważyć, że jeśli zaczepimy w początku układu współrzędnych wszystkie wektory kraty $L(\overrightarrow{a}, \overrightarrow{b})$ rozpiętej przez wektory $\overrightarrow{a}$ i $\overrightarrow{b}$, to końce tych wektorów będą punktami przecięcia linii przedstawionych na rysunku 2.

Każdemu wektorowi $\overrightarrow{a}$ przyporządkowujemy liczbę rzeczywistą nieujemną $\|\overrightarrow{a}\|$, nazywaną *długością* wektora $\overrightarrow{a}$. Długość wektora można definiować na wiele sposobów. Tu zajmiemy się dwoma sposobami.

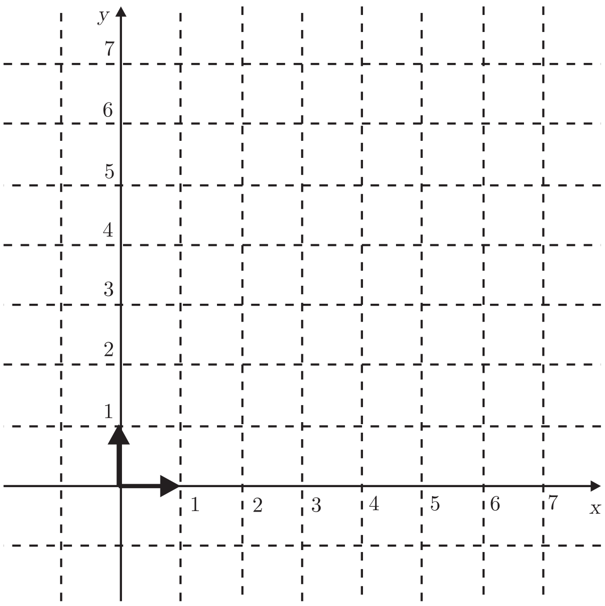
1. *Długość euklidesową* wektora $\overrightarrow{a} = [a_1, a_2]$ definiujemy wzorem

$$\|\overrightarrow{a}\| = \sqrt{a_1^2 + a_2^2}.$$

2. *Długość „miejską”* wektora $\overrightarrow{a} = [a_1, a_2]$ definiujemy wzorem

$$\|\overrightarrow{a}\| = |a_1| + |a_2|.$$

Można łatwo sprawdzić, że długość zdefiniowana tak, jak wyżej w punktach 1. i 2., spełnia następujące warunki:



Rysunek 1: Rysunek do przykładu 1

(a) $\|\vec{a}\| = 0 \Leftrightarrow \vec{a} = [0,0]$,

(b) $\|t \cdot \vec{a}\| = |t| \cdot \|\vec{a}\|$;

(c) $\|\vec{a} + \vec{b}\| \leqslant \|\vec{a}\| + \|\vec{b}\|$.

Udowodnimy teraz kilka prostych lematów.

Lemat 1 $\|\vec{a} + \vec{b}\| \geqslant \|\vec{a}\| - \|\vec{b}\|$.

Dowód Z warunków (b) i (c) wynika, że

$$\|\vec{a}\| = \|(\vec{a} + \vec{b}) + (-\vec{b})\| \leqslant \|\vec{a} + \vec{b}\| + \|-\vec{b}\| = \|\vec{a} + \vec{b}\| + \|\vec{b}\|,$$

skąd natychmiast wynika teza lematu. ■

Lemat 2 *Dla dowolnej liczby całkowitej* $r \neq 0$

$$L(\vec{a}, \vec{b}) = L(\vec{b}, \vec{a} - r \cdot \vec{b}).$$

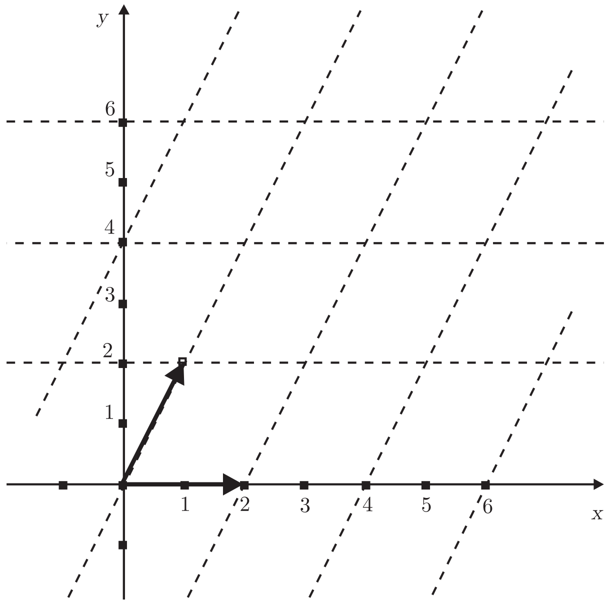
Dowód Przypuśćmy najpierw, że $\vec{c} \in L(\vec{a}, \vec{b})$. Istnieją wówczas liczby całkowite m i n takie, że

$$\vec{c} = m \cdot \vec{a} + n \cdot \vec{b}.$$

Wtedy

$$\vec{c} = m \cdot \vec{a} + n \cdot \vec{b} = n \cdot \vec{b} + mr \cdot \vec{b} + m \cdot \vec{a} - mr \cdot \vec{b} = \tag{20}$$

$$= (n + mr) \cdot \vec{b} + m \cdot (\vec{a} - r \cdot \vec{b}) \in L(\vec{b}, \vec{a} - r \cdot \vec{b}). \tag{21}$$



Rysunek 2: Rysunek do przykładu 2

Z drugiej strony, jeśli $\vec{c} \in L(\vec{b}, \vec{a} - r \cdot \vec{b})$, to istnieją liczby całkowite m i n takie, że

$$\vec{c} = m \cdot \vec{b} + n \cdot (\vec{a} - r \cdot \vec{b}).$$

Ale wtedy

$$\vec{c} = n \cdot \vec{a} + (m - nr) \cdot \vec{b} \in L(\vec{a}, \vec{b}),$$

co kończy dowód lematu. ■

Lemat 3 *Załóżmy, że dla każdej liczby całkowitej r zachodzi nierówność*

$$\|\vec{a} - r \cdot \vec{b}\| \geqslant \|\vec{b}\|.$$

Wtedy $\vec{b}$ *jest najkrótszym niezerowym wektorem kraty* $L(\vec{a}, \vec{b})$.

Dowód Niech $\vec{c} \in L(\vec{a}, \vec{b})$ oraz $\vec{c} \neq \vec{o}$. Chcemy pokazać, że $\|\vec{c}\| \geqslant \|\vec{b}\|$. Istnieją liczby całkowite m i n, nierówne jednocześnie zeru, takie, że $\vec{c} = m \cdot \vec{a} + n \cdot \vec{b}$. Mamy dwa przypadki.

Przypadek 1: $m = 0$. Wtedy $n \neq 0$, a więc $|n| \geqslant 1$. Zatem

$$\|\vec{c}\| = \|n \cdot \vec{b}\| = |n| \cdot \|\vec{b}\| \geqslant \|\vec{b}\|.$$

Przypadek 2: $m \neq 0$. Wykonujemy wtedy dzielenie z resztą liczby n przez liczbę m. Istnieją więc liczby całkowite t i s takie, że $n = tm + s$ oraz $0 \leqslant s < |m|$. Mamy wówczas $|m| - s \geqslant 1$ i korzystając z lematu 1, dostajemy:

$$\|\vec{c}\| \quad = \quad \|m \cdot \vec{a} + n \cdot \vec{b}\| = \|m \cdot \vec{a} + (mt + s) \cdot \vec{b}\| = \|m \cdot \vec{a} + mt \cdot \vec{b} + s \cdot \vec{b}\| \geqslant$$

$$
\begin{aligned}
&\geqslant \|m\cdot\overrightarrow{a}+mt\cdot\overrightarrow{b}\|-\|s\cdot\overrightarrow{b}\|=|m|\cdot\|\overrightarrow{a}+t\cdot\overrightarrow{b}\|-|s|\cdot\|\overrightarrow{b}\|=\\
&= |m|\cdot\|\overrightarrow{a}-(-t)\cdot\overrightarrow{b}\|-s\cdot\|\overrightarrow{b}\|\geqslant|m|\cdot\|\overrightarrow{b}\|-s\cdot\|\overrightarrow{b}\|=\big(|m|-s\big)\cdot\|\overrightarrow{b}\|\geqslant\\
&\geqslant \|\overrightarrow{b}\|, \qquad (22)
\end{aligned}
$$

co kończy dowód. ■

Nasze zadanie polega na znalezieniu najkrótszego niezerowego wektora kraty rozpiętej przez dwa dane wektory $\overrightarrow{a}$ i $\overrightarrow{b}$. Lematy 2 i 3 pokazują, że następujący algorytm jest poprawny:

if $\|\overrightarrow{a}\|<\|\overrightarrow{b}\|$ **then** $\overrightarrow{a}\leftrightarrow\overrightarrow{b}$;
repeat
  znajdź takie całkowite r, by liczba $t=\|\overrightarrow{a}-r\cdot\overrightarrow{b}\|$ była jak najmniejsza;
  if $t<\|\overrightarrow{b}\|$ **then begin**
    $\overrightarrow{c}:=\overrightarrow{a}-r\cdot\overrightarrow{b}$;
    $\overrightarrow{a}:=\overrightarrow{b}$;
    $\overrightarrow{b}:=\overrightarrow{c}$
  end
until $t\geqslant\|\overrightarrow{b}\|$;
return $\overrightarrow{b}$;

Jedyny niejasny fragment algorytmu dotyczy znajdowania takiej liczby całkowitej r, by liczba $t=\|\overrightarrow{a}-r\cdot\overrightarrow{b}\|$ była jak najmniejsza. Pokażemy teraz, w jaki sposób możemy to zrobić (przy obu definicjach długości wektora). Niech więc dane będą dwa wektory $\overrightarrow{a}=[a_1,a_2]$ i $\overrightarrow{b}=[b_1,b_2]$. Załóżmy przy tym, że $\overrightarrow{b}\neq[0,0]$. Przypomnijmy następnie, że $\lfloor x\rfloor$ oznacza największą liczbę całkowitą niewiększą od x, a $\lceil x\rceil$ najmniejszą liczbę całkowitą niemniejszą od x:

$$\lfloor x\rfloor\leqslant x<\lfloor x\rfloor+1,\qquad \lceil x\rceil-1<x\leqslant\lceil x\rceil.$$

Rozpatrzmy najpierw długość euklidesową. Wtedy

$$t=\|\overrightarrow{a}-r\cdot\overrightarrow{b}\|=\sqrt{(a_1-b_1r)^2+(a_2-b_2r)^2}.$$

Zdefiniujmy funkcję kwadratową f wzorem

$$f(x)=(a_1-b_1x)^2+(a_2-b_2x)^2=(b_1^2+b_2^2)x^2-2(a_1b_1+a_2b_2)x+(a_1^2+b_1^2).$$

Liczba t będzie najmniejsza dla takiej całkowitej wartości r, dla której funkcja f przyjmuje najmniejszą wartość. W tym celu wystarczy wybrać jako r liczbę całkowitą najbliższą współrzędnej x_w wierzchołka paraboli będącej wykresem funkcji f, a więc najbliższą liczbie

$$x_w=\frac{a_1b_1+a_2b_2}{b_1^2+b_2^2}.$$

A zatem $r=\lfloor x_w+\frac{1}{2}\rfloor$.

Rozpatrzmy teraz długość miejską. Wtedy

$$t=\|\overrightarrow{a}-r\cdot\overrightarrow{b}\|=|a_1-b_1r|+|a_2-b_2r|.$$

Zdefiniujmy funkcję f wzorem

$$f(x) = |a_1 - b_1 x| + |a_2 - b_2 x|.$$

Liczba t będzie najmniejsza dla takiej całkowitej wartości r, dla której funkcja f przyjmuje najmniejszą wartość. Zbadanie funkcji f wymaga rozpatrzenia kilku przypadków.

1. Jeśli $b_1 = 0$, to jako liczbę r musimy wziąć liczbę całkowitą najbliższą liczbie $\frac{a_2}{b_2}$, czyli wystarczy wziąć $r = \lfloor \frac{a_2}{b_2} + \frac{1}{2} \rfloor$.
2. Jeśli $b_1 = 0$, to analogicznie bierzemy $r = \lfloor \frac{a_1}{b_1} + \frac{1}{2} \rfloor$.
3. Jeśli $b_1 \neq 0$ oraz $b_2 \neq 0$, to zauważamy, że funkcja f przyjmuje najmniejszą wartość w jednym z dwóch punktów: $x_1 = \frac{a_1}{b_1}$ lub $x_2 = \frac{a_2}{b_2}$. Dokładniej, jeśli $|b_1| > |b_2|$, to wartością najmniejszą jest $f\left(\frac{a_1}{b_1}\right)$, jeśli $|b_1| = |b_2|$, to wartości w obu punktach są równe (funkcja f jest wtedy stała w całym przedziale o końcach $\frac{a_1}{b_1}$ i $\frac{a_2}{b_2}$), jeśli zaś $|b_1| < |b_2|$, to najmniejszą wartością funkcji f jest $f\left(\frac{a_2}{b_2}\right)$. Stąd wynika, że:
   (a) jeśli $|b_1| > |b_2|$, to jako r bierzemy tę z liczb $r_1 = \lfloor \frac{a_1}{b_1} \rfloor$ i $r_2 = \lceil \frac{a_1}{b_1} \rceil$, dla której wartość funkcji f jest mniejsza;
   (b) jeśli $|b_1| \leqslant |b_2|$, to jako r bierzemy tę z liczb $r_1 = \lfloor \frac{a_2}{b_2} \rfloor$ i $r_2 = \lceil \frac{a_2}{b_2} \rceil$, dla której wartość funkcji f jest mniejsza.

To kończy opis algorytmu. Algorytm ten pochodzi od Gaussa. Nie jest znany żaden ogólny algorytm wyznaczania najkrótszego niezerowego wektora w kracie n-wymiarowej (czyli rozpiętej w przestrzeni $\mathbb{R}^n$ przez n wektorów liniowo niezależnych) dla $n > 3$.

XV Międzynarodowa Olimpiada Informatyczna, Kenosha 2003

zadania

Krzysztof Diks, Marcin Kubica, Tomasz Waleń
Tłumaczenie

Porównywanie programów (Comparing code)

Zadanie

Racine Business Networks (RBN) pozwało do sądu Heuristic Algorithm Languages (HAL) oskarżając HAL o kradzież kodu źródłowego z systemu RBN Unix i wykorzystanie go w otwartym (Open software) systemie operacyjnym HALnix.

Zarówno RBN jak i HAL posługują się językiem programowania, w którym każda instrukcja znajduje się w osobnym wierszu i jest postaci:

`STOREA = STOREB + STOREC`,

gdzie `STOREA`, `STOREB`, `STOREC` *są nazwami zmiennych. W szczególności, nazwa pierwszej zmiennej znajduje się na początku wiersza, po niej następuje odstęp, znak równości, odstęp, nazwa drugiej zmiennej, odstęp, znak dodawania, odstęp, nazwa trzeciej zmiennej. Nazwa tej samej zmiennej może pojawiać się w danym wierszu więcej niż raz. Nazwy zmiennych składają się z co najwyżej 8 dużych liter ASCII (od „A” do „Z”).*

RBN twierdzi, że HAL skopiował z kodu źródłowego RBN ciąg kolejnych instrukcji modyfikując je tylko nieznacznie:

- *RBN twierdzi, że HAL zmienił nazwy niektórych zmiennych w celu zatuszowania swoich niecnych czynów. To znaczy, HAL skopiował fragment kodu z programu RBN i dla każdej zmiennej pojawiającej się w tym fragmencie zmienił jej nazwę (wszystkie jej wystąpienia), chociaż nowa nazwa może być taka sama jak poprzednia. Oczywiście żadne dwie różne nazwy zmiennych nie zostały zmienione na tę samą nazwę.*

- *RBN twierdzi również, że HAL mógł dopuścić się zmiany kolejności argumentów po prawej stronie niektórych instrukcji:* `STOREA = STOREB + STOREC` *mogło zostać zmienione na* `STOREA = STOREC + STOREB`.

- *RBN twierdzi, że tzw. „specjaliści” HALa byli zbyt ograniczeni, aby dodatkowo mataczyć i kolejność wierszy w ukradzionym podstępnie kodzie nie uległa zmianie.*

Mając dane kody źródłowe programów RBN i HAL, znajdź najdłuższy ciąg kolejnych wierszy w programie HALa, który mógłby pochodzić z ciągu kolejnych wierszy z programu RBN przy zastosowaniu opisanych powyżej przekształceń. Zwróć uwagę, że fragmenty programów nie muszą zaczynać się w wierszach o tych samych numerach.

Wejście

Pierwszy wiersz pliku wejściowego `code.in` *zawiera dwie liczby całkowite oddzielone pojedynczym odstępem, R i H ($1 \leqslant R \leqslant 1000$, $1 \leqslant H \leqslant 1000$). R jest liczbą wierszy w kodzie źródłowym programu RBN, a H jest liczbą wierszy w kodzie źródłowym HAL.*

Kolejne R wierszy to kod źródłowy programu RBN.
Następne H wierszy to kod źródłowy programu HAL.

Wyjście

Plik wyjściowy `code.out` *powinien zawierać jeden wiersz, a w nim jedną liczbę całkowitą: długość najdłuższego ciągu kolejnych wierszy w programie HAL, który mógłby pochodzić (po przekształceniu) z programu RBN.*

Przykład

Dla pliku wejściowego `code.in`*:*

```
4 3
RA = RB + RC
RC = D + RE
RF = RF + RJ
RE = RF + RF
HD = HE + HF
HM = HN + D
HN = HA + HB
```

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

```
2
```

Wiersze 1–2 w programie RBN są takie same jak wiersze 2–3 w programie HALa, przy zastosowaniu następującego przemianowania zmiennych w programie RBN: `RA` → `HM`, `RB` → `D`, `RC` → `HN`, `D` → `HA`, `RE` → `HB`. *Nie istnieje pasujący fragment składający się z 3 lub więcej wierszy.*

Krzysztof Diks, Marcin Kubica, Tomasz Waleń
Tłumaczenie

Ścieżki (Trail maintenance)

Zadanie

Krowy farmera Jana chcą przemieszczać się pomiędzy N ($1 \leqslant N \leqslant 200$) pastwiskami na farmie (ponumerowanymi $1 \dots N$) mimo, że pastwiska są przedzielone lasami. Krowy chcą utrzymywać system ścieżek pomiędzy pastwiskami tak, aby mogły podróżować z dowolnego pastwiska do dowolnego innego, chodząc tylko po pastwiskach i „utrzymywanych” ścieżkach. Ścieżki są dwukierunkowe.

Krowy nie potrafią wytyczać ścieżek. Zamiast tego utrzymują ścieżki, które wytyczyły dzikie zwierzęta. Każdego tygodnia krowy mogą zdecydować się na utrzymywanie dowolnie wybranych znanych ścieżek dzikich zwierząt.

Krowy są ciekawskie i na początku każdego tygodnia odkrywają jedną dziką ścieżkę. Potem muszą się zdecydować, które ścieżki utrzymywać w tym tygodniu tak, aby mogły przedostawać się z dowolnego pastwiska na dowolne inne. Krowy mogą korzystać tylko z aktualnie utrzymywanych ścieżek.

Krowy chcą zawsze zminimalizować sumę długości ścieżek, które muszą utrzymywać. Krowy mogą wybrać dowolny podzbiór ścieżek dzikich zwierząt, które chcą utrzymywać, niezależnie od tego, które z nich były utrzymywane w poprzednim tygodniu.

Ścieżki dzikich zwierząt (nawet te utrzymywane) nigdy nie są proste. Dwie ścieżki łączące te same pastwiska mogą mieć różne długości. Co więcej, jeśli dwie ścieżki się przecinają, krowy są tak skupione na swojej drodze, że nie przechodzą na żadną inną ścieżkę.

Na początku każdego z tygodni krowy opisują dziką ścieżkę, którą odkryły. Twój program musi wtedy wypisać minimalną sumaryczną długość ścieżek, które krowy muszą utrzymywać danego tygodnia tak, aby mogły przemieszczać się między dwoma dowolnymi pastwiskami, o ile jest to tylko możliwe.

Wejście

Pierwszy wiersz standardowego wejścia zawiera dwie liczby całkowite N i W, oddzielone odstępem. Liczba W jest liczbą tygodni, przez które program ma pomagać krowom ($1 \leqslant W \leqslant 6000$).

Dla każdego tygodnia wczytaj jeden wiersz opisujący odkrytą dziką ścieżkę. Taki wiersz zawiera 3 liczby całkowite oddzielone odstępami: końce ścieżki (numery pastwisk) i długość ścieżki (liczbę całkowitą z zakresu $1 \dots 10000$). Każda ścieżka łączy dwa różne pastwiska.

Wyjście

Zaraz po tym, gdy program dowiaduje się o nowoodkrytej dzikiej ścieżce powinien wypisać na standardowe wyjście jeden wiersz zawierający jedną liczbę całkowitą, równą minimalnej, sumarycznej długości ścieżek, które muszą być utrzymywane, aby krowy mogły przewędrować *z dowolnego pastwiska na dowolne inne. Jeśli jest to niemożliwe, program powinien wypisać „-1”.*

Twój program musi zakończyć swoje działanie po wypisaniu odpowiedzi dla danych z ostatniego tygodnia.

Przykład

Wejście	*Wyjście*	*Opis*
4 6		
1 2 10		
	-1	*nie da się połączyć 4 z pozostałymi pastwiskami*
1 3 8		
	-1	*nie da się połączyć 4 z pozostałymi pastwiskami*
3 2 3		
	-1	*nie da się połączyć 4 pozostałymi pastwiskami*
1 4 3		
	14	*utrzymuj 1 4 3, 1 3 8 i 3 2 3*
1 3 6		
	12	*utrzymuj 1 4 3, 1 3 6 i 3 2 3*
2 1 2		
	8	*utrzymuj 1 4 3, 2 1 2 i 3 2 3*
	Program kończy działanie	

Krzysztof Diks, Marcin Kubica, Tomasz Waleń
Tłumaczenie

Malejący ciąg (Reverse)

Zadanie

Rozważmy komputer z dwiema operacjami (w skrócie TOM — ang. Two-Operation Machine), który składa się z dziewięciu rejestrów ponumerowanych 1...9. W każdym rejestrze można zapamiętać nieujemną liczbę całkowitą z zakresu 0...1000. Dostępnymi operacjami komputera są:

- `S` *i j — Zapisz w rejestrze j zawartość rejestru i powiększoną o jeden.*
- `P` *i — Wypisz zawartość rejestru i.*

Dla danej liczby całkowitej N ($0 \leqslant N \leqslant 255$) wygeneruj program na maszynę TOM, który wypisze malejący ciąg liczb całkowitych N, $N-1$, $N-2$, ..., 0. Do programu zalicza się także ustalenie początkowych wartości rejestrów. Maksymalna liczba kolejnych operacji typu `S` *powinna być jak najmniejsza.*

Oto przykładowy program TOM i wynik jego działania (dla $N = 2$):

Operacja	*Zawartość rejestrów*									*Wypisywana wartość*
	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	
Inicjalizacja	0	2	0	0	0	0	0	0	0	
P 2	0	2	0	0	0	0	0	0	0	2
S 1 3	0	2	1	0	0	0	0	0	0	
P 2	0	2	1	0	0	0	0	0	0	1
P 1	0	2	1	0	0	0	0	0	0	0

Zestawy danych (ang. input cases) są ponumerowane od 1 do 16 i można je otrzymać za pomocą portalu konkursowego.

Dane

Pierwszy wiersz zawiera numer zestawu K. Drugi wiersz zawiera liczbę N.

Wyniki

Pierwszy wiersz pliku wyjściowego powinien zawierać napis „`FILE reverse` *K”, gdzie K jest numerem zestawu danych wejściowych.*

Drugi wiersz powinien zawierać dziewięć liczb pooddzielanych odstępami, odpowiadających początkowym wartościom rejestrów, w kolejności (rejestr 1, rejestr 2 itd.).

Kolejne wiersze pliku wyjściowego powinny zawierać uporządkowaną listę kolejnych operacji do wykonania, po jednej operacji w wierszu. Tak więc w wierszu 3-cim powinna znaleźć się *pierwsza operacja, itd. Ostatni wiersz powinien zawierać operację, która wypisuje 0. Każdy wiersz powinien zawierać poprawną operację. Instrukcje należy formatować w sposób pokazany poniżej.*

Przykład

Dla danych wejściowych:

```
1
2
```

prawidłowymi odpowiedziami są (niepełne punkty):

```
FILE reverse 1
0 2 0 0 0 0 0 0 0
P 2
S 1 3
P 3
P 1
```

i (pełne punkty):

```
FILE reverse 1
0 2 1 0 0 0 0 0 0
P 2
P 3
P 1
```

Ocenianie

Dla każdego zestawu danych punkty będą liczone z uwzględnieniem poprawności i optymalności programu TOM.

Poprawność: 20%

Program TOM jest poprawny, jeśli wykonuje po kolei nie więcej niż 131 operacji typu `S` *i ciąg wartości wypisanych przez ten program jest właściwy (zawiera dokładnie $N+1$ liczb całkowitych, zaczynając od N i kończąc na 0). Jeśli w wyniku wykonania operacji* `S` *nastąpi przepełnienie rejestru, to program zostanie uznany za niepoprawny.*

Optymalność: 80%

Optymalność poprawnego programu TOM jest mierzona w zależności od maksymalnej liczby kolejnych operacji typu `S`*, która powinna być jak najmniejsza. Punktacja będzie obliczana w zależności od różnicy między twoim programem TOM i najlepszym znanym programem TOM.*

Krzysztof Diks, Marcin Kubica, Tomasz Waleń
Tłumaczenie

Kontemplacja płotu (Seeing the boundary)

Zadanie

Farmer Jasio codziennie podziwia płot ogradzający jego farmę, a szczególnie rozkoszuje się widokiem słupków w płocie. Farma jest płaska jak deska i ma kształt kwadratu o wymiarach $N \times N$ metrów ($2 \leqslant N \leqslant 500000$). Jeden z rogów farmy ma współrzędne $(0,0)$, a przeciwległy róg ma współrzędne (N,N); boki farmy są równoległe do osi układu współrzędnych.

Słupki w płocie znajdują się w rogach, jak również wzdłuż płotu, w odstępach co jeden metr. Łącznie mamy więc $4N$ słupków. Słupki stoją pionowo, a ich grubość jest pomijalnie mała. Farmer Jasio chce wiedzieć, ile słupków można zobaczyć z zadanego miejsca na farmie.

Na farmie Jasia znajduje się R ($1 \leqslant R \leqslant 30000$) ogromnych głazów, które zasłaniają mu widok na niektóre ze słupków w jego płocie — Jaś jest niskiego wzrostu, a skały są zbyt wysokie, żeby mógł spojrzeć ponad nimi. Podstawa każdego głazu ma postać wypukłego wielokąta o niezerowym polu, którego wierzchołki mają współrzędne całkowitoliczbowe. Głazy wystają z ziemi dokładnie pionowo. Głazy nie zachodzą na siebie, ani nie stykają się ze sobą, jak również nie dotykają Farmera Jasia, ani płotu. Farmer Jaś również nie dotyka płotu, ani nie stoi na głazie, ani też nie siedzi w środku głazu.

Mając dane: wielkość farmy, położenie i wielkość głazów na farmie oraz miejsce, gdzie stoi Jaś, oblicz ile spośród słupków w płocie Jaś może zobaczyć. Jeśli wierzchołek skały leży na jednej linii pomiędzy Jasiem, a słupkiem, to zakładamy, że Jaś nie może zobaczyć takiego słupka.

Wejście

Pierwszy wiersz pliku wejściowego `boundary.in` *zawiera dwie liczby całkowite N i R oddzielone odstępem.*

Następny wiersz pliku wejściowego zawiera dwie liczby całkowite X i Y oddzielone odstępem — położenie farmera Jasia wewnątrz farmy.

Kolejne wiersze opisują R głazów:

- *Opis i-tego głazu rozpoczyna się od wiersza zawierającego jedną liczbę całkowitą p_i ($3 \leqslant p_i \leqslant 20$) — liczbę wierzchołków podstawy głazu.*
- *Każdy z kolejnych p_i wierszy zawiera współrzędne wierzchołka — dwie liczby całkowite X i Y oddzielone odstępem. Wierzchołki tworzące podstawę głazu są parami różne i są podane w kolejności odwrotnej do kierunku ruchu wskazówek zegara.*

Wyjście

Plik wyjściowy `boundary.out` *powinien zawierać jeden wiersz, a w nim jedną liczbę całkowitą — liczbę słupków w płocie, które może zobaczyć farmer Jaś.*

Przykład

Dla pliku wejściowego `boundary.in`*:*

```
100 1
60 50
5
70 40
75 40
80 40
80 50
70 60
```

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

```
319
```

Zauważ, że w podstawie głazu są współliniowe wierzchołki: (70,40), (75,40) i (80,40).

Krzysztof Diks, Marcin Kubica, Tomasz Waleń
Tłumaczenie

Zgadnij, która to krowa (Guess which cow)

Zadanie

Janko Rolnik jest szczęśliwym posiadaczem stada N ($1 \leqslant N \leqslant 50$) bardzo podobnych krów. Krowy są ponumerowane od 1 do N. Janko musi utulić krowę do snu w jej łóżeczku.

Krowy są rozróżniane na podstawie P ($1 \leqslant P \leqslant 8$) cech, ponumerowanych $1 \ldots P$. Każda cecha może przyjmować jedną z trzech wartości. Dla przykładu, kolor kolczyka w uchu może być żółty, zielony lub czerwony. Dla prostoty wartości każdej z cech są reprezentowane przez litery `X`, `Y`, *i* `Z`. *Każda para krów Janka różni się co najmniej jedną cechą.*

Napisz program, który znając cechy krów Janka, pomoże mu odpowiedzieć na pytanie, którą właśnie krowę układa do snu. Twój program może zadać Jankowi nie więcej niż 100 pytań postaci: Czy wartość cechy T rozpatrywanej krowy należy do zbioru S? Postaraj się, żeby Twój program zadawał możliwie jak najmniej pytań.

Wejście

Pierwszy wiersz pliku `guess.in` *zawiera dwie liczby całkowite N i P oddzielone odstępem.*

W każdym z następnych N wierszy opisano cechy jednej krowy — P liter pooddzielanych pojedynczymi odstępami. Pierwsza litera w wierszu to wartość cechy 1, itd. Drugi wiersz wejścia opisuje krowę nr 1, trzeci wiersz opisuje krowę nr 2, itd.

Interakcja

Faza pytanie/odpowiedź odbywa się za pomocą standardowego wejścia i wyjścia.

Każdy program zadaje pytania o krowę układaną do łóżeczka pisząc na standardowe wyjście wiersz, który składa się z litery `Q`, *odstępu, numeru cechy, odstępu, a następnie jednej lub więcej wartości oddzielonych pojedynczymi odstępami.*

Dla przykładu, „`Q 1 Z Y`*” oznacza „Czy wartością cechy 1 jest* `Z` *lub* `Y`*?” Cecha musi być liczbą całkowitą z zakresu $1 \ldots P$. Żadna z wartości nie może się pojawić więcej niż raz w jednym pytaniu i każda z nich musi być równa* `X`, `Y` *lub* `Z`.

Po zadaniu każdego pytania, program otrzymuje odpowiedź, wczytując jeden wiersz zawierający jedną liczbę całkowitą. Liczba 1 oznacza, że wartość cechy, o którą pytał program należy do podanego zbioru wartości; liczba 0 oznacza, że nie należy.

Na koniec program powinien wypisać jeden wiersz składający się z litery `C`, *odstępu, a następnie jednej liczby całkowitej, równej numerowi krowy układanej do łóżeczka przez Janka.*

Przykład

Dla pliku `guess.in`*:*

```
4 2
X Z
X Y
Y X
Y Y
```

interakcja mogłaby wyglądać następująco:

Wejście	*Wyjście*	*Objaśnienie*
	`Q 1 X Z`	
`0`		*To może być krowa 3 lub krowa 4.*
	`Q 2 Y`	
`1`		*To musi być krowa 4!*
	`C 4`	
koniec wykonywania programu		

Ocenianie

Poprawność: 30% punktów

Program, który wypisze poprawny numer krowy otrzyma pełną liczbę punktów za poprawność tylko wtedy, gdy tylko jedna krowa ma cechy zgodne z otrzymanymi odpowiedziami. Program, który zada więcej niż 100 pytań dla danego testu nie otrzyma żadnych punktów za ten test.

Liczba pytań: 70% punktów

Pozostałe punkty będą przyznane w zależności od liczby pytań potrzebnych do wyznaczenia właściwej krowy. Testy są tak przygotowane, żeby nagradzać minimalizowanie liczby pytań dla najgorszego przypadku. Rozwiązania prawie optymalne otrzymają częściową liczbę punktów.

Krzysztof Diks, Marcin Kubica, Tomasz Waleń
Tłumaczenie

Uciekające roboty (Amazing robots)

Zadanie

Jesteś dumnym właścicielem dwóch robotów, które są umieszczone w oddzielnych prostokątnych jaskiniach złożonych z identycznych kwadratów. Kwadrat $(1,1)$ *w jaskini jest kwadratem w lewym górnym rogu, który jest rogiem północno-zachodnim. W jaskini* i $(i = 1,2)$ *znajduje się* G_i $(0 \leqslant G_i \leqslant 10)$ *strażników próbujących złapać roboty. Roboty patrolują proste ścieżki w jedną i drugą stronę. Twoim zadaniem jest wyznaczenie ciągu rozkazów przekazywanych robotom, dzięki którym opuszczą one jaskinię, nie dając się złapać strażnikom.*

Na początku każdej z minut wysyłasz ten sam rozkaz do obu robotów. Każdy rozkaz jest nazwą kierunku: north, south, east, west (po polsku północ, południe, wschód, zachód). Robot przemieszcza się o jeden kwadrat w podanym kierunku, chyba że napotka ścianę, wtedy stoi w miejscu przez minutę. Robot wychodzi z jaskini opuszczając ją i od tego momentu ignoruje wszystkie rozkazy.

Strażnicy przemieszczają się o jeden kwadrat na początku każdej minuty, w tym samym czasie kiedy roboty. Każdy strażnik rozpoczyna swój patrol w podanym kwadracie i porusza się w podanym kierunku o jeden kwadrat na minutę, aż wykona o jeden mniej kroków niż wynosi podana liczba kwadratów na jego ścieżce. Wtedy natychmiast się obraca i kontynuuje swój patrol do kwadratu, od którego rozpoczął patrolowanie, po czym obraca się i zaczyna od początku. Strażnicy kończą patrolowanie, gdy oba roboty opuszczą jaskinie.

Ścieżki strażników nie będą przechodziły przez ściany ani nie będą wychodziły z jaskini. Pomimo, że ścieżki strażników mogą się przecinać, dwóch strażników nigdy nie znajdzie się w tym samym kwadracie pod koniec minuty i nigdy nie zamieni się zajmowanymi kwadratami podczas trwania minuty. Żaden strażnik nie rozpocznie swojego patrolu w tym samym kwadracie, w którym stoi na początku robot.

Strażnik łapie robota, jeśli pod koniec minuty zajmuje ten sam kwadrat, co robot, albo jeśli podczas minuty robot i strażnik zamienią się miejscami.

Mając dane opisy dwóch jaskiń (każda nie większa niż 20×20*) wraz z początkowymi kwadratami obu robotów oraz ścieżkami patrolowanymi przez strażników w każdej z jaskiń, wyznacz ciąg rozkazów dzięki którym roboty wyjdą z jaskini nie dając złapać się strażnikom. Należy zminimalizować czas po którym ostatni z robotów opuści jaskinię. Jeśli roboty opuszczą jaskinie w różnych momentach, czas wyjścia pierwszego robota nie ma znaczenia.*

Wejście

Pierwsza część pliku `robots.in` *opisuje pierwszą jaskinię, robota w niej przebywającego i strażników. Podobnie, druga część pliku opisuje drugą jaskinię, robota w niej przebywającego i strażników.*

Pierwszy wiersz wejścia zawiera dwie liczby całkowite R_1 i C_1 oddzielone odstępem — liczbę wierszy i kolumn w pierwszej jaskini.

Następne R_1 wierszy zawiera po C_1 znaków opisujących wygląd jaskini. Początkowy kwadrat robota jest zaznaczony literą „`X`*". Kropka („*`.`*") reprezentuje otwartą przestrzeń, a „*`#`*" oznacza ścianę. Każda jaskinia zawiera dokładnie jednego robota.*

Kolejny wiersz opisu zawiera jedną liczbę całkowitą G_1, liczbę strażników w pierwszej jaskini ($0 \leqslant G_1 \leqslant 10$).

Każdy z następnych G_1 wierszy opisuje ścieżkę patrolu pojedynczego strażnika. Opis składa się z trzech liczb całkowitych i jednej litery, oddzielonych pojedynczymi odstępami. Pierwsze dwie liczby określają wiersz i kolumnę kwadratu, na którym strażnik znajduje się na początku patrolu. Trzecia liczba określa liczbę kwadratów ($2 \ldots 4$), które tworzą ścieżkę patrolu. Litera określa kierunek, w którym strażnik jest zwrócony na początku swojej drogi: „`N`*", „*`S`*", „*`E`*", lub „*`W`*" (północ, południe, wschód, zachód).*

Opis drugiej jaskini jest w takim samym formacie jak opis pierwszej jaskini, ale być może z innymi wartościami.

Wyjście

Pierwszy wiersz pliku `robots.out` *powinien zawierać jedną liczbę całkowitą K ($K \leqslant 10000$), liczbę rozkazów, po której oba roboty opuszczą jaskinię unikając złapania. Jeśli taki ciąg rozkazów istnieje, to najkrótszy ciąg będzie miał nie więcej niż 10000 rozkazów. Następne K wierszy, to ciąg rozkazów, każdy zawierający jeden znak ze zbioru {„*`N`*", „*`S`*", „*`E`*", „*`W`*"}. Jeśli taki ciąg nie istnieje, wyjście powinno zawierać tylko jeden wiersz zawierający „*`-1`*".*

Oba roboty powinny znajdować się poza jaskinią po wykonaniu ostatniego z rozkazów. Ostatni z rozkazów powinien sprawić, że co najmniej jeden z robotów opuszcza jaskinię. Jeśli wiele różnych ciągów sprawia, że roboty opuszczają jaskinię w tym samym, minimalnym czasie, to dowolny z nich będzie zaakceptowany.

Przykład

Dla pliku wejściowego `robots.in`*:*

```
5 4
####
#X.#
#..#
...#
##.#
1
4 3 2 W
4 4
####
#...
#X.#
####
0
```

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

```
8
E
N
E
S
S
S
E
S
```

Ocenianie

Nie będą przyznawane częściowe punkty za testy, w których nie istnieje ciąg rozkazów wyprowadzający oba roboty poza jaskinie. Za inne testy będą przyznawane punkty częściowe zgodnie z opisem podanym poniżej.

Poprawność: 20% punktów

Plik wyjściowy jest uznawany za poprawny jeśli jest jest zgodny ze specyfikacją, zawiera nie więcej niż 10000 rozkazów, które sprawiają, że oba roboty opuszczają jaskinię, z czego co najmniej jeden z nich po ostatnim z rozkazów.

Optymalność: 80% punktów

Plik wyjściowy jest uznawany za optymalny, jeśli jest poprawny i nie istnieje krótszy poprawny ciąg rozkazów. Program, który generuje ciąg rozkazów nie będący ciągiem optymalnym nie uzyska punktów za optymalność.

X Bałtycka Olimpiada Informatyczna, Ventspils 2004

zadania

Przemysława Kanarek, Krzysztof Stencel
Tłumaczenie

Ciąg (Sequence)

Krótkie sformułowanie

Dany jest ciąg liczbowy. Twoim zadaniem jest skonstruowanie ciągu rosnącego, który najlepiej przybliża zadany ciąg. Najlepiej przybliżającym ciągiem jest ciąg o najmniejszym całkowitym odchyleniu od zadanego ciągu.

Formalne sformułowanie

Niech $t_1, t_2, \ldots, t_N$ *oznaczają liczby zadanego ciągu. Twoim zadaniem jest skonstruowanie rosnącego ciągu liczbowego* $z_1 < z_2 < \ldots < z_N$. *Całkowite odchylenie, tj. suma* $|t_1 - z_1| + |t_2 - z_2| + \cdots + |t_N - z_N|$, *powinno być najmniejsze z możliwych.*

Wejście

W pierwszym wierszu pliku tekstowego `seq.in` *znajduje się jedna liczba całkowita* N *(*$1 \leqslant N \leqslant 1000000$*). Każdy z następnych* N *wierszy zawiera jedną liczbę całkowitą, tj. kolejny element zadanego ciągu. Liczba* t_K *jest podana w* $(K+1)$*-szym wierszu. Każdy element ciągu spełnia nierówności* $0 \leqslant t_K \leqslant 2000000000$.

Wyjście

Pierwszy wiersz pliku wyjściowego `seq.out` *powinien zawierać minimalne całkowite odchylenie. Każdy z następnych* N *wierszy powinien zawierać jedną liczbę całkowitą, która jest kolejnym elementem rosnącego ciągu, który najlepiej przybliża zadany ciąg. Jeśli istnieje kilka ciągów o takim minimalnym odchyleniu, Twój program powinien wypisać jeden z nich.*

Przykład

Dla pliku wejściowego `seq.in`*:*

```
7
9
4
8
20
14
15
18
```

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

```
13
6
7
8
13
14
15
18
```

Przemysława Kanarek, Krzysztof Stencel
Tłumaczenie

Niecodzienna liczba (Unique number)

Dany jest ciąg dodatnich liczb całkowitych. Wiadomo, że jedna z liczb ciągu występuje w nim tylko raz, podczas gdy każda z pozostałych dokładnie k ($k > 1$) razy.

Napisz program, który znajdzie **liczbę występującą w ciągu dokładnie raz!**

Wejście

W pierwszym wierszu pliku tekstowego `unique.in` *znajduje się dodatnia liczba całkowita n ($n \leqslant 2000001$) oznaczająca liczbę elementów ciągu.*

W kolejnych n wierszach pliku znajduje się po jednej liczbie; w $(i+1)$-szym wierszu podany jest i-ty element ciągu. Żaden element ciągu nie przekracza 2147483647.

Wyjście

W jedynym wierszu pliku tekstowego `unique.out` *Twój program musi wypisać jedną liczbę — tę która występuje w ciągu dokładnie raz.*

Przykład

Dla pliku wejściowego `unique.in`*:*

```
13
537
295
210
413
413
210
413
210
413
210
537
537
537
```

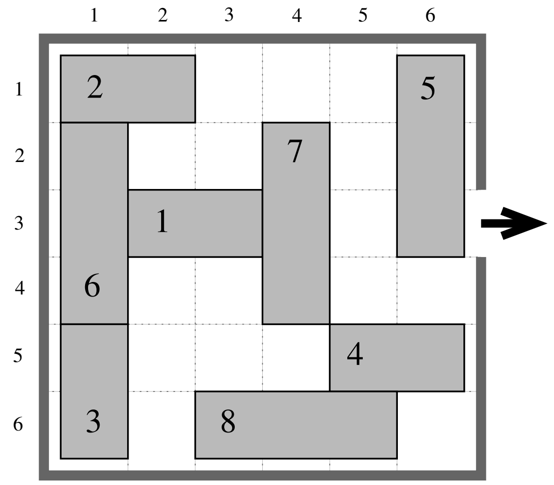
poprawnym wynikiem jest plik wyjściowy `unique.out`*:*

```
295
```

Przemysława Kanarek, Krzysztof Stencel
Tłumaczenie

Parking (Car park)

W schronisku, w którym odbywa się BOI 2004, jest garaż dla gości o kształcie kwadratu o wymiarach 6 na 6. Wiersze garażu są ponumerowane liczbami od 1 do 6, począwszy od najwyższego. Kolumny garażu także ponumerowane są liczbami od 1 do 6, począwszy od pierwszej kolumny z lewej. Z garażu jest tylko jeden wyjazd znajdujący się na prawym końcu trzeciego wiersza.



W garażu zaparkowano N samochodów. Pomiędzy nimi jest i Twój samochód, ale niełatwo nim stamtąd wyjechać, bo jest zablokowany przez inne samochody. Wraz z przyjaciółmi usiłujesz wydostać swój samochód z garażu. Możesz przy tym przestawiać inne samochody, przepychając je w przód lub w tył (szczęśliwie wszystkie są zaparkowane na luzie). Nie możesz jednak kierować lub skręcać żadnym samochodem (także swoim!).

Twoim zadaniem jest określenie, jaka jest minimalna liczba ruchów (jeden ruch to przepchnięcie jednego samochodu o 1 w przód lub w tył) pozwalających wydostać z garażu Twój samochód (ma on wymiary 2x1). Nie wolno przy tym wypchnąć z garażu żadnego innego samochodu, a Twój musi znaleźć się całkowicie poza kwadratem garażu.

Są tylko dwa typy samochodów: o wymiarach 2x1 (mieszczące się dokładnie na dwóch kwadratowych polach garażu) oraz o wymiarach 3x1 (mieszczące się na 3 polach). Samochody można przesuwać jedynie wzdłuż dłuższego boku.

W przykładzie przedstawionym na rysunku mamy $N = 8$ samochodów; Twój jest oznaczony numerem 1.

Podany niżej ciąg, to najkrótszy ciąg ruchów (18-tu) pozwalających wydostać Twój samochód z garażu:

$$4\leftarrow\leftarrow\leftarrow, 2\rightarrow, 6\uparrow, 3\uparrow, 8\leftarrow\leftarrow, 5\downarrow\downarrow\downarrow, 7\downarrow\downarrow, 1\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow.$$

Wejście

W pierwszym wierszu pliku `carpark.in` *znajduje się liczba samochodów N ($1 \leqslant N \leqslant 16$).*

W każdym z kolejnych N wierszy znajduje się opis samochodu oznaczonego numerem i. Opis składa się z czterech liczb całkowitych: l_i — długości samochodu, o_i — jego skierowania oraz x_i i y_i — współrzędnych początkowych (numeru kolumny i wiersza lewego-górnego pola) samochodu. Skierowanie $o_i = 1$ oznacza, że samochód jest zaparkowany poziomo (czyli, że jego dłuższy bok przebiega poziomo); inna wartość o_i oznacza, że samochód jest zaparkowany pionowo.

Powyższe liczby spełniają następujące ograniczenia: $l_i \in \{2,3\}$, $o_i \in \{0,1\}$, $1 \leqslant x_i, y_i \leqslant 6$.

Twój samochód jest opisany w drugim wierszu pliku wejściowego, zaraz po wierszu zawierającym liczbę wszystkich samochodów N. Samochód trzeba wyprowadzić z garażu przez jedyny istniejący wyjazd.

Wyjście

Plik wynikowy `carpark.out` *powinien zawierać tylko jedną liczbę całkowitą oznaczającą minimalną liczbę ruchów potrzebnych do wyprowadzenia Twojego samochodu z garażu.*

Jeżeli Twojego samochodu nie da się wyprowadzić z garażu, to plik wynikowy powinien zawierać liczbę -1.

Przykład

Dla pliku wejściowego `carpark.in`*:*

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

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

```
18
```

Przemysława Kanarek, Krzysztof Stencel
Tłumaczenie

Powtórzenia (Repeats)

Słowo s nazywamy (k,l)-**powtórzeniem**, *jeżeli powstaje przez k krotne złączenie pewnego słowa t o długości l ($l \geqslant 1$). Na przykład, słowo*

$s =$ `abaabaabaaba`

jest $(4,3)$-powtórzeniem, dla

$t =$ `aba`.

Słowo t nazywamy **zarodkiem** *słowa s. W powyższym przypadku zarodek t ma długość 3, a całe słowo s powstaje przez czterokrotne powtórzenie t.*

Napisz program pozwalający rozwiązać następujące zadanie. Na wejściu dane jest długie słowo u. Twój program musi znaleźć pewne (k,l)-powtórzenie, które występuje jako (ciągłe!) podsłowo w u, przy czym k musi być jak największe. Na przykład, słowo wejściowe

$u =$ `babbabaabaabaabab`

zawiera $(4,3)$-powtórzenie rozpoczynające się na pozycji 5, zaznaczone przez podkreślenie. Ponieważ u nie zawiera żadnego innego (ciągłego) podsłowa złożonego z więcej niż 4 powtórzeń, Twój program powinien zwrócić podkreślone słowo jako wynik.

Wejście

W pierwszym wierszu pliku wejściowego `repeats.in` *znajduje się liczba n oznaczająca długość słowa wejściowego ($1 \leqslant n \leqslant 50000$).*

W kolejnych n wierszach znajduje się słowo wejściowe złożone z liter `a` *i* `b`, *zapisane po jednym znaku w wierszu. Znaki podane są w takiej kolejności, w jakiej występują w słowie wejściowym.*

Wyjście

Plik wyjściowy `repeats.out` *musi zawierać trzy liczby całkowite, zapisane każda w oddzielnym wierszu. Opisują one znalezione przez Twój program (k,l)-powtórzenie w następujący sposób:*

- *w pierwszym wierszu znajduje się największa możliwa liczba powtórzeń k;*
- *w drugim wierszu znajduje się l — długość zarodka, który powtarza się w słowie wejściowym k razy;*
- *w trzecim, ostatnim wierszu, znajduje się numer znaku (pozycja) $1 \leqslant p \leqslant n$ w słowie wejściowym, od którego zaczyna się znalezione (k,l)-powtórzenie.*

Przykład

Dla pliku wejściowego `repeats.in`*:*

```
17
b
a
b
b
a
b
a
a
b
a
a
b
a
a
b
a
b
```

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

```
4
3
5
```

Przemysława Kanarek, Krzysztof Stencel
Tłumaczenie

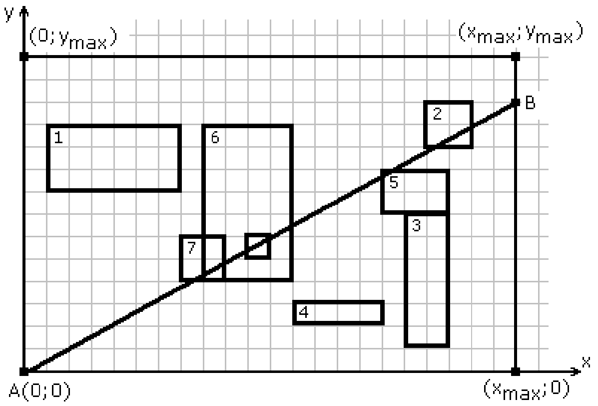
Prostokąty (Rectangles)

Danych jest N prostokątów położonych na płaszczyźnie. Boki prostokątów są równoległe do osi układu współrzędnych. Prostokąty mogą na siebie nachodzić lub zawierać się jedne w drugich. Współrzędne wierzchołków prostokątów są dodatnimi liczbami całkowitymi, nieprzekraczającymi odpowiednio $x_{\max}$ *(współrzędna x) oraz* $y_{\max}$ *(współrzędna y).*

Rozważmy odcinek rozpoczynający się w punkcie $A(0,0)$ *i kończący się w pewnym punkcie B. Współrzędne punktu B (drugiego końca odcinka) spełniają następujące warunki:*

- *są liczbami całkowitymi;*
- *albo B należy do odcinka* $[(0,y_{\max}),(x_{\max},y_{\max})]$*, albo B należy do odcinka* $[(x_{\max},0),(x_{\max},y_{\max})]$*;*

Odcinek AB może przecinać niektóre z prostokątów (powiemy, że odcinek przecina prostokąt, nawet wówczas, gdy przechodzi jedynie przez jeden z jego wierzchołków).



Na rysunku jest przedstawionych 8 prostokątów. Odcinek AB przecina pięć spośród nich.

Zadanie

Napisz program znajdujący maksymalną liczbę prostokątów, które może przecinać odcinek AB, oraz obliczający współrzędne punktu B. Jeżeli istnieje kilka rozwiązań, program powinien znaleźć dowolne z nich.

Wejście

W pierwszym wierszu pliku wejściowego `rect.in` *znajdują się trzy liczby całkowite:* $x_{\max}$, $y_{\max}$ $(0 \leqslant x_{\max}, y_{\max} \leqslant 10^9)$ *oraz N* $(1 \leqslant N \leqslant 10000)$*. W kolejnych N wierszach znajdują się opisy prostokątów. Każdy składa się z czterech liczb całkowitych: współrzędnych lewego-dolnego wierzchołka* (x_{bl} *i* y_{bl}) *oraz prawego-górnego wierzchołka* (x_{tr} *i* y_{tr}) *prostokąta.*

Wyjście

W pierwszym i jedynym wierszu pliku wyjściowego `rect.out` *powinny znaleźć się trzy liczby całkowite. Pierwsza z nich powinna być maksymalną liczbą prostokątów, jakie może przeciąć odcinek AB. Kolejne dwie, to współrzędne x i y punktu B, dla którego osiągana jest maksymalna liczba przecięć.*

Przykład

Dla pliku wejściowego `rect.in`*:*

```
22 14 8
1 8 7 11
18 10 20 12
17 1 19 7
12 2 16 3
16 7 19 9
8 4 12 11
7 4 9 6
10 5 11 6
```

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

```
5 22 12
```

Przemysława Kanarek, Krzysztof Stencel
Tłumaczenie

Statki (Ships)

Plansza do gry w statki składa się z $N \times N$ pól. Każde pole jest zajęte przez jakiś statek albo jest puste. Jeśli dwa sąsiednie pola (tzn. mające wspólną krawędź) są zajęte, to muszą one należeć do tego samego statku. Pola należące do różnych statków nie mają żadnych wspólnych punktów. Wyporność statku to liczba pól zajętych przez ten statek.

Na rysunku 1 pola należące do statków są zaczernione. Na planszy widać jeden statek 29-tonowy, trzy statki 7-tonowe, dwa statki 4-tonowe i trzy jednotonowe.

Zadanie

Napisz program, który na podstawie opisu statków na planszy policzy liczbę statków i wyporność każdego z nich.

Wejście

W pierwszej linii pliku ships.in znajduje się jedna dodatnia liczba całkowita N ($N < 30000$).

W każdej z następnych N linii podana jest informacja o jednym wierszu planszy. W tych liniach opisano kolejne grupy pól należących do statków. Opis grupy może mieć jeden z dwóch formatów:

- `numer_kolumny`
  Taka grupa składa się z jednego pola w kolumnie o numerze `numer_kolumny`*, przy czym pole leżące bezpośrednio na lewo od kolumny* `numer_kolumny` *i pole leżące bezpośrednio na prawo od kolumny* `numer_kolumny` *nie należą żadnych do statków.*
- `numer_pierwszej_kolumny-numer_ostatniej_kolumny`
  Taka grupa składa się z wszystkich pól w kolumnach o numerach od `numer_pierwszej_kolumny` *do* `numer_ostatniej_kolumny` *włącznie, przy czym pole bezpośrednio na lewo od kolumny* `numer_pierwszej_kolumny` *i pole bezpośrednio na prawo od kolumny* `numer_ostatniej_kolumny` *nie należą do żadnych statków.*

Opisy poszczególnych grup są oddzielone przecinkami. Każda linia jest zakończona średnikiem. W tych liniach nie ma odstępów. Jeśli w jakimś wierszu planszy nie ma pól należących do statków, odpowiednia linia pliku wejściowego składa się wyłącznie ze średnika. Wiadomo, że liczba wszystkich statków nie przekracza 1000, a wyporność żadnego statku nie jest większa niż 1000 ton.

Wyjście

Twój program powinien zapisać wynik do pliku `ships.out`*. W każdej linii tego pliku muszą znaleźć się dwie liczby całkowite oddzielone odstępem. Pierwsza z nich to wyporność, a druga* *to liczba statków mających tę wyporność. Wyporności muszą być wypisane w porządku malejącym. Daną wyporność można wypisać tylko wtedy, gdy jest co najmniej jeden statek o tej wyporności.*

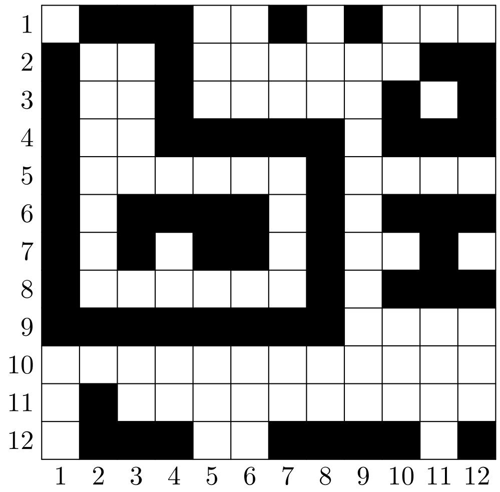
Przykład

Dla pliku wejściowego `ships.in`*:*

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

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

```
29 1
7 3
4 2
1 3
```



Rys. 1.

Wojciech Rytter
Treść zadania, Opracowanie

Przemysława Kanarek, Krzysztof Stencel
Tłumaczenie

Waga (Scales)

Masz wagę szalkową o równych ramionach, zbiór odważników i pewien obiekt. Masy odważników to 1, 3, 9, 27, 81, ..., czyli masa każdego odważnika jest nieujemną potęgą trojki. Dla każdej liczby całkowitej $k \geqslant 0$ jest dokładnie jeden odważnik o masie 3^k. Masa obiektu to dodatnia liczba całkowita m. Na lewej szalce położono obiekt do zważenia. Twoim zadaniem jest położyć pewne odważniki na lewej i prawej szalce tak, aby waga była w równowadze.

Zadanie

Napisz program, który:

- *odczyta m, tzn. masę obiektu z pliku tekstowego* `scales.in`*,*
- *wyznaczy odważniki, które należy położyć na obu szalkach,*
- *zapisze wyniki do pliku tekstowego* `scales.out`*.*

Wejście

Pierwszy wiersz pliku tekstowego `scales.in` *zawiera jedną liczbę całkowitą m, $1 \leqslant m \leqslant 10^{100}$.*

Wyjście

Plik wyjściowy `scales.out` *powinien zawierać dwa wiersze. Pierwszy wiersz tego pliku powinien zawierać informację o odważnikach na lewej szalce. Pierwsza liczba w tym wierszu musi być nieujemna i ma to być liczba odważników do położenia na lewej szalce. Następnie należy wypisać w porządku rosnącym masy odważników z tej szalki. Liczby należy oddzielić pojedynczymi spacjami.*

Drugi wiersz tego pliku powinien zawierać informację o odważnikach na prawej szalce w takim samym formacie.

Przykład

Dla danych wejściowych:
```
42
```
prawidłową odpowiedzią jest:
```
3 3 9 27
1 81
```
Dla danych wejściowych:
```
30
```
prawidłową odpowiedzią jest:

```
0
2 3 27
```

Rozwiązanie

Zadanie można zinterpretować jako zapis liczby naturalnej m w specjalnej reprezentacji trójkowej: w postaci tej bazą jest liczba 3, a cyfry są ze zbioru {-1, 0, 1}. Oznaczmy tę reprezentację przez $repr(m)$.

Mamy $repr(m) = (a_0, a_1, a_2, \ldots, a_k)$, gdzie $a_i \in \{-1,\ 0,\ +1\}$, $a_k \neq 0$ oraz

$$m = \sum_{i=0}^{k} a_i \cdot 3^i$$

Mając obliczoną reprezentację, na prawej szalce kladziemy odważniki 3^i dla których $a_i = 1$, natomiast na lewej szalce kładziemy obiekt o masie m i odważniki 3^i dla $a_i = -1$.

Oznaczmy przez $[m]_3$ normalną reprezentację trójkową liczby m

W algorytmie mamy do czynienia z bardzo dużymi liczbami tak, że dodatkowym utrudnieniem jest implemnatcja arytmetyki na reprezentacjach dziesiętnych i trójkowych dużych liczb.

Obserwacja. Rozwiązanie opiera się na następującym spostrzeżeniu. Załóżmy, że mamy, liczbę naturalną $m > 0$ i liczbę x taką, że $[x]_3$ składa się z (samych) $k+1$ jedynek oraz $x \geqslant m$. Jeśli $[x+m]_3 = (b_0, b_1, b_2, \ldots, b_k)$, to $repr(m) = (b_0 - 1, b_1 - 1, b_2 - 1, \ldots b_k - 1)$, z dokładnością do pominiętych nieznaczących ostatnich zer.

Opis algorytmu.

Obliczmy najmniejszą liczbę x, taką że $[x]_3$ składa się z samych jedynek oraz $x \geqslant m$.
Niech $y = m + x$.
Obliczmy $[y]_3 = (b_0, b_1, b_2, \ldots, b_k)$.
return $repr(m) = (b_0 - 1, b_1 - 1, b_2 - 1, \ldots, b_k - 1)$

Przykład.
Niech $m = 42$. Mamy $[42]_3 = (0,2,1,1)$, oraz $x = 121$, $[x]_3 = (1,1,1,1,1)$,
$y = m + x = 42 + 121 = 163$, gdzie $y = (1,0,0,0,2)$.
Tak więc $repr(42) = (0, -1, -1, -1,\ 1)$.

A więc na prawej szalce kładziemy odważnik 2^4, a na lewej obiekt o masie 42 i odważniki 2^1, 2^2, 2^3.

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

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

[13] L. Banachowski, A. Kreczmar, W. Rytter. *Analiza algorytmów i struktur danych*. WNT, Warszawa, 1987.

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

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

[16] I. N. Bronsztejn, K. A. Siemiendiajew. *Matematyka. Poradnik encyklopedyczny*. PWN, Warszawa, wydanie XIV, 1997.

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

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

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

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

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

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

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

[24] W. Lipski, W. Marek. *Analiza kombinatoryczna*. PWN, Warszawa, 1986.

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

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

[27] Günter Rote. Crossing the bridge at night. In *Bulletin of the EATCS 78*, 2002.

[28] R. Sedgewick. *Algorithms*. Addison-Wesley, 1988.

[29] M. M. Sysło. *Algorytmy*. WSiP, Warszawa, 1998.

[30] M. M. Sysło. *Piramidy, szyszki i inne konstrukcje algorytmiczne*. WSiP, Warszawa, 1998.

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

[32] R. J. Wilson. *Wprowadzenie do teorii grafów*. PWN, Warszawa, 1998.

[33] N. Wirth. *Algorytmy + struktury danych = programy*. WNT, Warszawa, 1999.

Niniejsza publikacja stanowi wyczerpujące źródło informacji o zawodach XI Olimpiady Informatycznej, przeprowadzonych w roku szkolnym 2003/2004. Książka zawiera informacje dotyczące organizacji, regulaminu oraz wyników zawodów. Zawarto w niej także opis rozwiązań wszystkich zadań konkursowych. Dodatkowo w książce znalazły się opisy rozwiązań trudniejszych zadań z konkursu programistycznego *Pogromcy Algorytmów*, którego współorganizatorem jest Olimpiada Informatyczna, portal Gazeta.pl i firma PROKOM SOFTWARE SA. Do książki dołączony jest dysk CD-ROM zawierający wzorcowe rozwiązania i testy do wszystkich zadań Olimpiady oraz większości zadań z Pogromców.

Książka zawiera też zadania z XV Międzynarodowej Olimpiady Informatycznej oraz X Bałtyckiej Olimpiady Informatycznej.

XI 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 83–917700–5–2