MINISTERSTWO EDUKACJI NARODOWEJ
UNIWERSYTET WROCŁAWSKI
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

XIV OLIMPIADA INFORMATYCZNA
2006/2007

Olimpiada Informatyczna jest organizowana przy współudziale

PROKOM
SOFTWARE SA

WARSZAWA, 2007

MINISTERSTWO EDUKACJI NARODOWEJ
UNIWERSYTET WROCŁAWSKI
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

XIV OLIMPIADA INFORMATYCZNA
2006/2007

WARSZAWA, 2007

Autorzy tekstów:

Szymon Acedański
Marek Cygan
prof. dr hab. Zbigniew Czech
dr hab. Krzysztof Diks
dr Przemysława Kanarek
dr Marcin Kubica
mgr Jakub Pawlewicz
Marcin Pilipczuk
Jakub Radoszewski
prof. dr hab. Wojciech Rytter
mgr Piotr Stańczyk
dr Andrzej Walat
mgr Tomasz Waleń

Opracowanie i redakcja:

dr Przemysława Kanarek
Adam Iwanicki
Jakub Radoszewski

Skład:

Adam Iwanicki

Autorzy programów:

Szymon Acedański
Michał Brzozowski
Marek Cygan
Adam Iwanicki
Maciej Jaśkowski
mgr Anna Niewiarowska
mgr Paweł Parys
mgr Jakub Pawlewicz
Marcin Pilipczuk
Jakub Radoszewski
mgr Piotr Stańczyk
Szymon Wąsik
mgr Marek Żylak

Opracowanie i redakcja treści zadań:

dr Marcin Kubica

Tłumaczenie treści zadań:

dr Piotr Chrząstowski–Wachtel
dr hab. Krzysztof Diks
prof. dr hab. Paweł Idziak
mgr Tomasz Idziaszek
dr Przemysława Kanarek
dr Marcin Kubica
mgr Jakub Pawlewicz
Jakub Radoszewski
dr hab. Krzysztof Stencel
mgr Tomasz Waleń

Pozycja dotowana przez Ministerstwo Edukacji Narodowej.

Druk książki został sfinansowany przez PROKOM SOFTWARE SA

ISBN 978–83–922946–2–7

Spis treści

Krzysztof Diks

Wstęp

Drogi Czytelniku!

Już po raz czternasty przygotowaliśmy sprawozdanie z Olimpiady Informatycznej. Przepraszamy za opóźnienie, mamy jednak nadzieję, że jakość przygotowanych materiałów zrekompensuje to w pełni. W tym miejscu należy gorąco podziękować autorom wszystkich opracowań oraz redaktorom tego wydania: dr Przemce Kanarek z Uniwersytetu Wrocławskiego oraz Adamowi Iwanickiemu i Jakubowi Radoszewskiemu — studentom informatyki na Uniwersytecie Warszawskim.

Rok 2007 to kontynuacja sukcesów naszej Olimpiady. Po raz drugi z rzędu Polak został zwycięzcą Międzynarodowej Olimpiady Informatycznej. Dokonał tego Tomasz Kulczyński z Bydgoszczy, który okazał się najlepszy spośród 282 reprezentantów z ponad 70 krajów świata. Pozostali Polacy także spisali się znakomicie: Marcin Andrychowicz z Warszawy zdobył złoty medal, a Marcin Kurczych i Jakub Kallas zdobyli medale brązowe. Warty podkreślenia jest też fakt, że drużyna złożona z ubiegłorocznego zwycięzcy Międzynarodowej Olimpiady Informatycznej — Filipa Wolskiego — oraz jurorów Olimpiady — Marka Cygana i Marcina Pilipczuka — wygrała Akademickie Mistrzostwa Świata w Programowaniu Zespołowym.

Mijający rok 2007 to także czas startu młodszej „siostry” Olimpiady Informatycznej — Olimpiady Informatycznej Gimnazjalistów. Już wkrótce najlepsi gimnazjaliści wzmocnią szeregi „dużej” Olimpiady. Twórcom Olimpiady Gimnazjalistów, panom Ryszardowi Szubartowskiemu i Tadeuszowi Kuranowi, należą się podziękowania za wizję i tytaniczną pracę włożoną w jej przygotowanie.

W roku 2007 przeprowadziliśmy też warsztaty olimpijskie dla nauczycieli, które cieszyły się ogromnym zainteresowaniem. Podczas warsztatów nauczyciele mogli w praktyce zapoznać się z metodami pracy z uczniami uzdolnionymi informatycznie wychowawców wielu olimpijczyków — panów Andrzeja Dyrka i Ryszarda Szubartowskiego. Mamy nadzieję, że wszystkie te przedsięwzięcia przyczynią się do ugruntowania pozycji Olimpiady Informatycznej i do zainteresowania informatyką jeszcze większej liczby uczniów.

Życzę wszystkim przyjemnej lektury, a uczestnikom XV Olimpiady — wielu sukcesów.

Krzysztof Diks

Komitet Główny Olimpiady Informatycznej

Sprawozdanie z przebiegu XIV Olimpiady Informatycznej w roku szkolnym 2006/2007

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 2006/2007 odbyły się zawody XIV 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. 16 października 2006 r. rozesłano plakaty zawierające zasady organizacji zawodów I stopnia oraz zestaw 5 zadań konkursowych do 3721 szkół i zespołów szkół młodzieżowych ponadgimnazjalnych. Zawody I stopnia rozpoczęły się dnia 23 października 2006 r. Ostatecznym terminem nadsyłania prac konkursowych był 20 listopada 2006 roku.

Zawody II i III stopnia były dwudniowymi sesjami stacjonarnymi, poprzedzonymi jednodniowymi sesjami próbnymi. Zawody II stopnia odbyły się w siedmiu okręgach: Katowicach, Krakowie, Poznaniu, Rzeszowie, Toruniu, Warszawie, Wrocławiu oraz w Sopocie w dniach 06–08.02.2007 r., natomiast zawody III stopnia odbyły się w ośrodku firmy Combidata Poland S.A. w Sopocie, w dniach 27–31.03.2007 r.

Uroczystość zakończenia XIV Olimpiady Informatycznej odbyła się w dniu 31.03.2007 r. w Sali Posiedzeń Urzędu Miasta w Sopocie.

SKŁAD OSOBOWY KOMITETÓW OLIMPIADY INFORMATYCZNEJ

Komitet Główny

przewodniczący:
  dr hab. Krzysztof Diks, prof. UW (Uniwersytet Warszawski)
zastępcy przewodniczącego:
  prof. dr hab. Maciej M. Sysło (Uniwersytet Wrocławski)
  prof. dr hab. Paweł Idziak (Uniwersytet Jagielloński)
sekretarz naukowy:
  dr Marcin Kubica (Uniwersytet Warszawski)
kierownik Jury:
  dr hab. Krzysztof Stencel (Uniwersytet Warszawski)
kierownik organizacyjny:
  Tadeusz Kuran
członkowie:
  dr Piotr Chrząstowski–Wachtel (Uniwersytet Warszawski)
  prof. dr hab. Zbigniew Czech (Politechnika Śląska)
  dr Przemysława Kanarek (Uniwersytet Wrocławski)
  mgr Anna Beata Kwiatkowska (IV LO im. T. Kościuszki w Toruniu, UMK)
  prof. dr hab. Krzysztof Loryś (Uniwersytet Wrocławski)
  prof. dr hab. Jan Madey (Uniwersytet Warszawski)
  dr hab. inż. Jerzy Nawrocki, prof. PP (Politechnika Poznańska)
  prof. dr hab. Wojciech Rytter (Uniwersytet Warszawski)
  mgr Krzysztof J. Święcicki
  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
  mgr Tomasz Waleń (Uniwersytet Warszawski)

sekretarz Komitetu Głównego:
Monika Kozłowska–Zając (OEIiZK)

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

Komitety okręgowe

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

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

Komitet Okręgowy we Wrocławiu
przewodniczący:
prof. dr hab. Krzysztof Loryś (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. Joliot–Curie 15.

Komitet Okręgowy w Toruniu:
przewodniczący:
prof. dr hab. Adam Jakubowski (Uniwersytet Mikołaja Kopernika w Toruniu)
zastępca przewodniczącego:
dr Mirosława Skowrońska (Uniwersytet Mikołaja Kopernika w Toruniu)
sekretarz:
dr Barbara Klunder (Uniwersytet Mikołaja Kopernika w Toruniu)
członkowie:
dr Andrzej Kurpiel (Uniwersytet Mikołaja Kopernika w Toruniu)
mgr Anna Beata Kwiatkowska (IV Liceum Ogólnokształcące w Toruniu)

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

Komitet Okręgowy w Poznaniu:
przewodniczący:
dr hab. inż. Maciej Drozdowski, prof. PP (Politechnika Poznańska)
zastępca przewodniczącego:
dr Jacek Marciniak (Uniwersytet Adama Mickiewicza w Poznaniu)
sekretarz:
mgr Ewa Nawrocka (Politechnika Poznańska)
członkowie:
inż. Władysław Bodzek (Politechnika Poznańska)
inż. Piotr Gawron (Politechnika Poznańska)
mgr inż. Przemysław Wesołek (Politechnika Poznańska)
Szymon Wąsik (Politechnika Poznańska)

Siedzibą Komitetu Okręgowego w Poznaniu jest Instytut Informatyki Politechniki Poznańskiej, ul. Piotrowo 2.

Górnośląski Komitet Okręgowy
przewodniczący:
prof. dr hab. Zbigniew Czech (Politechnika Śląska w Gliwicach)
zastępca przewodniczącego:
mgr inż. Jacek Widuch (Politechnika Śląska w Gliwicach) sekretarz:
mgr inż. Tomasz Wesołowski (Politechnika Śląska w Gliwicach)
członkowie:
mgr inż. Przemysław Kudłacik (Politechnika Śląska w Gliwicach)
mgr inż. Krzysztof Simiński (Politechnika Śląska w Gliwicach)
mgr inż. Tomasz Wojdyła (Politechnika Śląska w Gliwicach)

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

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

Siedzibą Komitetu Okręgowego w Krakowie jest Katedra Algorytmiki Uniwersytetu Jagiellońskiego w Krakowie, ul. Gronostajowa 3.

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

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

Jury Olimpiady Informatycznej

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

Szymon Acedański
Michał Brzozowski
Marek Cygan
mgr Krzysztof Dulęba
mgr Tomasz Idziaszek
Adam Iwanicki
Maciej Jaśkowski
Marian Kędzierski
Karol Kurach
Marek Marczykowski
mgr Anna Niewiarowska
mgr Paweł Parys
mgr Jakub Pawlewicz
Marcin Pilipczuk
Michał Pilipczuk
Jakub Radoszewski
mgr Piotr Stańczyk
Wojciech Tyczyński
Tomasz Warchoł
Szymon Wąsik
Bogdan Yakovenko
mgr Marek Żylak

ZAWODY I STOPNIA

W zawodach I stopnia XIV Olimpiady Informatycznej wzięło udział 825 zawodników. Decyzją Komitetu Głównego zdyskwalifikowano 16 zawodników. Powodem dyskwalifikacji była niesamodzielność rozwiązań zadań konkursowych.

Decyzją Komitetu Głównego do zawodów zostało dopuszczonych 34 uczniów gimnazjów i jeden uczeń szkoły podstawowej. Byli to uczniowie z następujących szkół:

Gimnazjum nr 24 przy III LO	Gdynia	8 uczniów
Gimnazjum nr 16	Szczecin	3
Gimnazjum Towarzystwa Salezjańskiego	Oświęcim	2
Gimnazjum i Liceum Akademickie	Toruń	2
Gimnazjum nr 7	Gdańsk	1 uczeń
Gimnazjum nr 17	Gdynia	1
Gimnazjum przy ZS nr 2	Hrubieszów	1
Gimnazjum nr 24	Katowice	1
Gimnazjum nr 1 im. St. Konarskiego	Kraków	1
Gimnazjum nr 18	Kraków	1
Szkoła Podstawowa nr 50	Lublin	1
Gimnazjum nr 2	Olsztyn	1
Gimnazjum	Opinogóra Górna	1
Gimnazjum Akademickie WSIiZ	Rzeszów	1
Publiczne Gimnazjum	Skórcz	1
Gimnazjum nr 5	Słupsk	1
KNG nr 5 im. św. Jana Bosko	Sosnowiec	1
Gimnazjum nr 1	Wadowice	1
Gimnazjum nr 13 im St. Staszica	Warszawa	1
Gimnazjum nr 11 im. Jana Paderewski	Warszawa	1
Gimnazjum nr 49	Warszawa	1
Gimnazjum nr 123 im. Jana Pawła II	Warszawa	1
Gimnazjum nr 86	Warszawa	1
Gimnazjum Dwujęzyczne nr 49	Wrocław	1

Kolejność województw pod względem liczby uczestników była następująca:

małopolskie	127 zawodników	dolnośląskie	38
mazowieckie	103	wielkopolskie	33
pomorskie	93	świętokrzyskie	30
śląskie	92	łódzkie	29
podkarpackie	63	lubuskie	26
kujawsko–pomorskie	43	lubelskie	23
podlaskie	41	warmińsko–mazurskie	15
zachodniopomorskie	39	opolskie	13

Jeden zawodnik uczęszczał do Stuyvesant High School w Nowym Jorku.
W zawodach I stopnia najliczniej reprezentowane były szkoły:

V LO im. Augusta Witkowskiego	Kraków	64 uczniów
III LO im. Marynarki Wojennej RP	Gdynia	40
XIV LO im. Stanisława Staszica	Warszawa	38
I LO im. Adama Mickiewicza	Białystok	21
VIII LO im. M. Skłodowskiej–Curie	Katowice	16
V Liceum Ogólnokształcące	Bielsko–Biała	13
VI LO im. J. i J. Śniadeckich	Bydgoszcz	13
XIII Liceum Ogólnokształcące	Szczecin	12
VI LO im. Wacława Sierpińskiego	Gdynia	11
I LO im. Tadeusza Kościuszki	Legnica	10
Gimnazjum nr 24 przy III LO	Gdynia	8
IV LO im. Tadeusza Kościuszki	Toruń	8
LO Zakonu Pijarów	Kraków	7
VIII LO im. Adama Mickiewicza	Poznań	7
III LO im. Adama Mickiewicza	Tarnów	7

IV LO im. H. Sienkiewicza	Częstochowa	6
VII LO im. Mikołaja Kopernika	Częstochowa	6
II LO im. Jana III Sobieskiego	Kraków	6
XXVII LO im. T. Czackiego	Warszawa	6
VI LO im. Tadeusza Rejtana	Warszawa	5
V LO im. Jana III Sobieskiego	Białystok	4
I LO im. C. K. Norwida	Bydgoszcz	4
I LO im. Władysława Jagiełły	Dębica	4
Ogólnokształcące Liceum Jezuitów	Gdynia	4
I Liceum Ogólnokształcące	Głogów	4
I LO im. T. Kościuszki	Gorzów Wlkp.	4
VIII LO im S. Wyspiańskiego	Kraków	4
I LO im. Mikołaja Kopernika	Krosno	4
I LO im. T. Kościuszki	Łomża	4
I LO im. Mikołaja Kopernika	Łódź	4
Społeczne LO nr 5	Milanówek	4
Gimnazjum Towarzystwa Salezjańskiego	Oświęcim	4
III Liceum Ogólnokształcące	Płock	4
LO im. St. Małachowskiego	Płock	4
LO św. Marii Magdaleny	Poznań	4
II Liceum Ogólnokształcące	Przemyśl	4
VI LO im. J. Kochanowskiego	Radom	4
II LO im. Adama Mickiewicza	Słupsk	4
I LO im. Bolesława Krzywoustego	Słupsk	4
X Liceum Ogólnokształcące	Szczecin	4
II LO im. St. Staszica	Tarnowskie Góry	4
Zespół Szkół Elektrycznych	Włocławek	4
III LO im. Stefana Żeromskiego	Bielsko–Biała	3
V Liceum Ogólnokształcące	Gdańsk	3
I LO im. M. Kopernika	Gdańsk	3
I LO im. E. Dembowskiego	Gliwice	3
I LO im. M. Kromera	Gorlice	3
II Liceum Ogólnokształcące	Gorzów Wlkp.	3
II Liceum Ogólnokształcące	Grudziądz	3
I LO im. M. Kopernika	Jarosław	3
I LO im. B. Nowodworskiego	Kraków	3
LO im. Marii Konopnickiej	Legionowo	3
I LO im. Stanisława Staszica	Lublin	3
II LO im. J. Zamoyskiego	Lublin	3
II LO im. Marii Konopnickiej	Opole	3
Zespół Szkół Elektrycznych	Opole	3
I LO im. Karola Marcinkowskiego	Poznań	3
Zespół Szkół Komunikacji	Poznań	3
II Liceum Ogólnokształcące	Poznań	3
I Liceum Ogólnokształcące	Racibórz	3
Zespół Szkół Elektronicznych	Radom	3
II Liceum Ogólnokształcące	Radomsko	3
IV LO im. Mikołaja Kopernika	Rzeszów	3
Liceum Ogólnokształcące WSIiZ	Rzeszów	3
I LO im. KEN	Sanok	3
II LO im. Emilii Plater	Sosnowiec	3
Gimnazjum nr 16	Szczecin	3
I LO im. Kazimierza Brodzińskiego	Tarnów	3
Zespół Szkół nr 36 im. M. Kasprzaka	Warszawa	3
XIV LO im. Polonii Belgijskiej	Wrocław	3
VII LO	Wrocław	3
III LO im. Adama Mickiewicza	Wrocław	3
I Liceum Ogólnokształcące	Zielona Góra	3

Najliczniej reprezentowane były miasta:

Kraków	98 uczniów	Poznań	24
Warszawa	70	Katowice	23
Gdynia	66	Bydgoszcz	18
Białystok	31	Bielsko–Biała	16
Szczecin	27	Częstochowa	15

Kielce	15	Koszalin	4
Wrocław	14	Krosno	4
Gdańsk	13	Limanowa	4
Rzeszów	13	Łomża	4
Toruń	13	Milanówek	4
Legnica	11	Oświęcim	4
Lublin	11	Racibórz	4
Tarnów	11	Radomsko	4
Kielce	10	Wodzisław Śląski	4
Łódź	10	Żary	4
Opole	9	Bełchatów	3
Płock	9	Chorzów	3
Słupsk	9	Elbląg	3
Gorzów Wlkp.	7	Gliwice	3
Radom	7	Gorlice	3
Przemyśl	6	Jastrzębie Zdrój	3
Włocławek	6	Konin	3
Zielona Góra	6	Legionowo	3
Głogów	5	Łowicz	3
Sanok	5	Nowy Sącz	3
Sosnowiec	5	Olsztyn	3
Tarnowskie Góry	5	Ostrowiec Świętokrzyski	3
Dębica	4	Siedlce	3
Grudziądz	4	Skarżysko–Kamienna	3
Jarosław	4	Wadowice	3
Jasło	4		

Zawodnicy uczęszczali do następujących klas:

do klasy V	szkoły podstawowej	1
do klasy I	gimnazjum	2
do klasy II		12
do klasy III		20
do klasy I	szkoły średniej	157
do klasy II		336
do klasy III		275
do klasy IV		3

3 zawodników nie podało informacji o klasach, do których uczęszczali.

W zawodach I stopnia zawodnicy mieli do rozwiązania pięć zadań: „Atrakcje turystyczne”, „Biura”, „Drzewa”, „Osie symetrii” i „Zapytania”.

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

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

- **ATR** — Atrakcje turystyczne

	ATR	
	liczba zawodników	czyli
100 pkt.	32	4,0%
75–99 pkt.	24	3,0%
50–74 pkt.	18	2,2%
1–49 pkt.	57	7,0%
0 pkt.	85	10,5%
brak rozwiązania	593	73,3%

- **BIU** — Biura

	BIU	
	liczba zawodników	czyli
100 pkt.	144	17,8%
75–99 pkt.	44	5,4%
50–74 pkt.	67	8,3%
1–49 pkt.	131	16,2%
0 pkt.	113	14,0%
brak rozwiązania	310	38,3%

- **DRZ** — Drzewa

	DRZ	
	liczba zawodników	czyli
100 pkt.	5	0,6%
75–99 pkt.	0	0%
50–74 pkt.	86	10,6%
1–49 pkt.	466	57,6%
0 pkt.	94	11,7%
brak rozwiązania	158	19,5%

- **OSI** — Osie symetrii

	OSI	
	liczba zawodników	czyli
100 pkt.	167	20,6%
75–99 pkt.	42	5,2%
50–74 pkt.	84	10,4%
1–49 pkt.	134	16,6%
0 pkt.	83	10,2%
brak rozwiązania	299	37,0%

- **ZAP** — Zapytania

	ZAP	
	liczba zawodników	czyli
100 pkt.	83	10,3%
75–99 pkt.	18	2,2%
50–74 pkt.	33	4,1%
1–49 pkt.	435	53,8%
0 pkt.	170	21,0%
brak rozwiązania	70	8,6%

W sumie za wszystkie 5 zadań konkursowych:

SUMA	liczba zawodników	czyli
500 pkt.	3	0,4%
375–499 pkt.	39	4,8%
250–374 pkt.	120	14,8%
125–249 pkt.	148	18,3%
1–124 pkt.	418	51,7%
0 pkt.	81	10,0%

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 381 zawodników, którzy osiągnęli w zawodach I stopnia wynik nie mniejszy niż 99 pkt.

Siedmiu zawodników nie stawiło się na zawody. W zawodach II stopnia uczestniczyło 374 zawodników.

Zawody II stopnia odbyły się w dniach 06–08 lutego 2007 r. w siedmiu stałych okręgach oraz w Sopocie:

- w Gliwicach — 20 zawodników z następujących województw:
  - małopolskie (1)
  - śląskie (19)
- w Krakowie — 60 zawodników z następujących województw:
  - małopolskie (60)
- w Poznaniu — 34 zawodników z następujących województw:
  - lubuskie (6)
  - wielkopolskie (14)
  - zachodniopomorskie (14)
- w Rzeszowie — 53 zawodników z następujących województw:
  - lubelskie (8)
  - małopolskie (6)
  - podkarpackie (27)
  - śląskie (1)
  - świętokrzyskie (11)
- w Toruniu — 31 zawodników z następujących województw:
  - kujawsko–pomorskie (17)
  - lubelskie (2)
  - mazowieckie (2)
  - podlaskie (5)
  - warmińsko–mazurskie (4)
  - zachodniopomorskie (1)
- w Warszawie — 68 zawodników z następujących województw:
  - lubelskie (2)
  - mazowieckie (46)
  - podkarpackie (1)
  - podlaskie (19)
- we Wrocławiu — 51 zawodników z następujących województw:
  - dolnośląskie (12)
  - lubuskie (6)
  - łódzkie (8)
  - małopolskie (2)
  - opolskie (5)
  - śląskie (17)
  - wielkopolskie (1)
- w Sopocie — 57 zawodników z następujących województw:
  - mazowieckie (1)
  - pomorskie (55)
  - zachodniopomorskie (1)

W zawodach II stopnia najliczniej reprezentowane były szkoły:

V LO im. Augusta Witkowskiego	Kraków	53 uczniów
III LO im. Marynarki Wojennej RP	Gdynia	37
XIV LO im. Stanisława Staszica	Warszawa	28
I LO im. Adama Mickiewicza	Białystok	16
VIII LO im. M. Skłodowskiej–Curie	Katowice	11
VI LO im. J. i J. Śniadeckich	Bydgoszcz	9
XIII Liceum Ogólnokształcące	Szczecin	8

V Liceum Ogólnokształcące	Bielsko–Biała	6
Gimnazjum nr 24 przy III LO	Gdynia	5
I LO im. Stefana Żeromskiego	Kielce	5
III LO im. Adama Mickiewicza	Tarnów	5
XXVII LO im. T. Czackiego	Warszawa	5
V LO im. Jana III Sobieskiego	Białystok	4
I LO im. T. Kościuszki	Gorzów Wlkp.	4
II LO im. Jana III Sobieskiego	Kraków	4
I LO im. Tadeusza Kościuszki	Legnica	4
I LO im. Wł. Broniewskiego	Bełchatów	3
Ogólnokształcące Liceum Jezuitów	Gdynia	3
I LO im. E. Dembowskiego	Gliwice	3
II LO im. J. Zamoyskiego	Lublin	3
I LO im. Mikołaja Kopernika	Łódź	3
LO św. Marii Magdaleny	Poznań	3
II LO im. Adama Mickiewicza	Słupsk	3

Najliczniej reprezentowane były miasta:

Kraków	65 uczniów	Wrocław	5
Gdynia	46	Legnica	4
Warszawa	41	Łódź	4
Białystok	23	Opole	4
Szczecin	12	Rzeszów	4
Katowice	11	Bełchatów	3
Bydgoszcz	10	Częstochowa	3
Bielsko–Biała	7	Gdańsk	3
Kielce	7	Gliwice	3
Poznań	7	Konin	3
Tarnów	7	Sosnowiec	3
Gorzów Wlkp.	6	Toruń	3
Lublin	6	Wodzisław Śląski	3
Słupsk	6	Zielona Góra	3

W dniu 6 lutego odbyła się sesja próbna, na której zawodnicy rozwiązywali nie liczące się do ogólnej klasyfikacji zadanie „Grzbiety i doliny” . W dniach konkursowych zawodnicy rozwiązywali zadania: „Megalopolis”, „Powódź”, „Skalniak”, „Tetris Attack”, każde oceniane maksymalnie po 100 punktów.

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

- **GRZ — próbne** — Grzbiety i doliny

	GRZ — próbne	
	liczba zawodników	czyli
100 pkt.	148	39,6%
75–99 pkt.	48	12,8%
50–74 pkt.	30	8,0%
1–49 pkt.	43	11,5%
0 pkt.	75	20,1%
brak rozwiązania	30	8,0%

- **MEG** — Megalopolis

	MEG	
	liczba zawodników	czyli
100 pkt.	34	9,1%
75–99 pkt.	5	1,3%
50–74 pkt.	2	0,5%
1–49 pkt.	283	75,7%
0 pkt.	22	5,9%
brak rozwiązania	28	7,5%

- **POW** — Powódź

	POW	
	liczba zawodników	czyli
100 pkt.	14	3,7%
75–99 pkt.	2	0,5%
50–74 pkt.	4	1,1%
1–49 pkt.	277	74,1%
0 pkt.	11	2,9%
brak rozwiązania	66	17,7%

- **SKA** — Skalniak

	SKA	
	liczba zawodników	czyli
100 pkt.	4	1,1%
75–99 pkt.	2	0,5%
50–74 pkt.	3	0,8%
1–49 pkt.	66	17,7%
0 pkt.	173	46,2%
brak rozwiązania	126	33,7%

- **TET** — Tetris Attack

	TET	
	liczba zawodników	czyli
100 pkt.	19	5,1%
75–99 pkt.	1	0,3%
50–74 pkt.	27	7,2%
1–49 pkt.	67	17,9%
0 pkt.	168	44,9%
brak rozwiązania	92	24,6%

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

SUMA	liczba zawodników	czyli
400 pkt.	0	0%
300–399 pkt.	3	0,8%
200–299 pkt.	18	4,8%
100–199 pkt.	54	14,4%
1–99 pkt.	292	78,1%
0 pkt.	7	1,9%

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

ZAWODY III STOPNIA

Zawody III stopnia odbyły się w ośrodku firmy Combidata Poland S.A. w Sopocie w dniach od 27 do 31 marca 2007 r.

Do zawodów III stopnia zakwalifikowano 72 najlepszych uczestników zawodów II stopnia, którzy uzyskali wynik nie mniejszy niż 102 pkt.

Zawodnicy uczęszczali do szkół w następujących województwach:

pomorskie	18 zawodników	łódzkie	2
małopolskie	12	podkarpackie	2
śląskie	9	świętokrzyskie	2
mazowieckie	8	lubelskie	1 zawodnik
wielkopolskie	7	podlaskie	1
dolnośląskie	5	warmińsko–mazurskie	1
kujawsko–pomorskie	3	zachodniopomorskie	1

Niżej wymienione szkoły miały w finale więcej niż jednego zawodnika:

III LO im. Marynarki Wojennej RP	Gdynia	16
V LO im. Augusta Witkowskiego	Kraków	9
XIV LO im. Stanisława Staszica	Warszawa	6
V Liceum Ogólnokształcące	Bielsko–Biała	3
VI LO im. J. i J. Śniadeckich	Bydgoszcz	3
VIII LO im. M. Skłodowskiej–Curie	Katowice	3
II LO im. Jana III Sobieskiego	Kraków	3
II Liceum Ogólnokształcące	Gorzów Wlkp.	2
I LO im. T. Kościuszki	Gorzów Wlkp.	2
I LO im. Stefana Żeromskiego	Kielce	2

27 marca odbyła się sesja próbna, na której zawodnicy rozwiązywali nie liczące się do ogólnej klasyfikacji zadanie „Koleje”. W dniach konkursowych zawodnicy rozwiązywali zadania: „Gazociągi”, „Egzamin na prawo jazdy”, „Klocki”, „Odważniki” i „Waga czwórkowa”, 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:

- **KOL — próbne** — Koleje

	KOL — próbne	
	liczba zawodników	czyli
100 pkt.	6	8,3%
75–99 pkt.	9	12,5%
50–74 pkt.	14	19,5%
1–49 pkt.	17	23,6%
0 pkt.	9	12,5%
brak rozwiązania	17	23,6%

- **GAZ** — Gazociągi

	GAZ	
	liczba zawodników	czyli
100 pkt.	19	26,4%
75–99 pkt.	0	0%
50–74 pkt.	5	6,9%
1–49 pkt.	13	18,1%
0 pkt.	35	48,6%
brak rozwiązania	0	0%

- **EGZ** — Egzamin na prawo jazdy

	EGZ	
	liczba zawodników	czyli
100 pkt.	3	4,2%
75–99 pkt.	0	0%
50–74 pkt.	1	1,4%
1–49 pkt.	1	1,4%
0 pkt.	34	47,2%
brak rozwiązania	33	45,8%

- **KLO** — Klocki

	KLO	
	liczba zawodników	czyli
100 pkt.	7	9,7%
75–99 pkt.	2	2,8%
50–74 pkt.	0	0%
1–49 pkt.	28	38,9%
0 pkt.	25	34,7%
brak rozwiązania	10	13,9%

- **ODW** — Odważniki

	ODW	
	liczba zawodników	czyli
100 pkt.	38	52,8%
75–99 pkt.	4	5,6%
50–74 pkt.	1	1,4%
1–49 pkt.	14	19,4%
0 pkt.	14	19,4%
brak rozwiązania	1	1,4%

- **WAG** — Waga czwórkowa

	WAG	
	liczba zawodników	czyli
100 pkt.	10	13,9%
75–99 pkt.	1	1,4%
50–74 pkt.	0	0%
1–49 pkt.	19	26,4%
0 pkt.	27	37,5%
brak rozwiązania	15	20,8%

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

SUMA	liczba zawodników	czyli
500 pkt.	1	1,4%
375–499 pkt.	2	2,8%
250–374 pkt.	11	15,3%
125–249 pkt.	24	33,3%
1–124 pkt.	27	37,5%
0 pkt.	7	9,7%

W dniu 31 marca 2007 roku, w Sali Posiedzeń Urzędu Miasta w Sopocie, ogłoszono wyniki finału XIV Olimpiady Informatycznej 2006/2007 i rozdano nagrody ufundowane przez: PROKOM Software S.A., Wydawnictwa Naukowo–Techniczne, Ogólnopolską Fundację Edukacji Komputerowej i Olimpiadę Informatyczną.

Poniżej zestawiono listę wszystkich laureatów i finalistów:

(1) **Tomasz Kulczyński**, 3 klasa, VI LO im. J. i J. Śniadeckich w Bydgoszczy, 500 pkt., laureat I miejsca
(2) **Marcin Andrychowicz**, 2 klasa, XIV LO im. Stanisława Staszica w Warszawie, 400 pkt., laureat I miejsca
(3) **Jakub Kallas**, 3 klasa, III LO im. Marynarki Wojennej RP w Gdyni, 400 pkt., laureat I miejsca
(4) **Marcin Kurczych**, 3 klasa, I LO im. Stefana Żeromskiego w Kielcach, 349 pkt., laureat II miejsca
(5) **Jarosław Gomułka**, 3 klasa, XIV LO im. Polonii Belgijskiej we Wrocławiu, 345 pkt., laureat II miejsca
(6) **Piotr Niedźwiedź**, 3 klasa, VIII LO im. M. Skłodowskiej–Curie w Katowicach, 345 pkt., laureat II miejsca
(7) **Jarosław Błasiok**, 1 klasa, VIII LO im. M. Skłodowskiej–Curie w Katowicach, 341 pkt., laureat II miejsca
(8) **Maciej Klimek**, 2 klasa, II Liceum Ogólnokształcące w Gorzowie Wlkp., 316 pkt., laureat II miejsca
(9) **Marcin Kościelnicki**, 2 klasa, I LO im. Juliusza Słowackiego w Chorzowie, 315 pkt., laureat II miejsca
(10) **Rafał Józefowicz**, 3 klasa, II LO w Olsztynie, 306 pkt., laureat II miejsca
(11) **Michał Jastrzębski**, 3 klasa, XIV LO im. Stanisława Staszica w Warszawie, 300 pkt., laureat II miejsca
(12) **Damian Karasiński**, 3 klasa, I LO im. T. Kościuszki w Gorzowie Wlkp., 300 pkt., laureat II miejsca
(13) **Łukasz Wołochowski**, 2 klasa, III LO im. Marynarki Wojennej RP w Gdyni, 298 pkt., laureat II miejsca
(14) **Jacek Migdał**, 3 klasa, V Liceum Ogólnokształcące w Bielsku–Białej, 276 pkt., laureat III miejsca
(15) **Wojciech Baranowski**, 2 klasa, III LO im. Marynarki Wojennej RP w Gdyni, 249 pkt., laureat III miejsca
(16) **Marek Rogala**, 2 klasa, Ogólnokształcące Liceum Jezuitów w Gdyni, 249 pkt., laureat III miejsca
(17) **Marcin Babij**, 3 klasa, IX LO we Wrocławiu, 234 pkt., laureat III miejsca
(18) **Błażej Osiński**, 3 klasa, VI LO im. J. i J. Śniadeckich w Bydgoszczy, 209 pkt., laureat III miejsca
(19) **Piotr Kufel**, 3 klasa, XXVIII LO im. J. Kochanowskiego w Warszawie, 208 pkt., laureat III miejsca

(20) **Mateusz Klimek**, 3 klasa, II Liceum Ogólnokształcące w Gorzowie Wlkp., 206 pkt., laureat III miejsca
(21) **Aleksandra Lipiec**, 3 klasa, V LO im. Augusta Witkowskiego w Krakowie, 200 pkt., laureat III miejsca
(22) **Robert Obryk**, 2 klasa, V LO im. A. Witkowskiego w Krakowie, 200 pkt., laureat III miejsca
(23) **Filip Wieczorek**, 3 klasa, III LO im. Marynarki Wojennej RP w Gdyni, 200 pkt., laureat III miejsca
(24) **Tomasz Żurkowski**, 3 klasa, I Liceum Ogólnokształcące w Swarzędzu, 200 pkt., laureat III miejsca
(25) **Mateusz Baranowski**, 2 klasa, III LO im. Marynarki Wojennej RP w Gdyni, 199 pkt., laureat III miejsca

Lista pozostałych finalistów w kolejności alfabetycznej:

- Paweł Blokus, 3 klasa, III LO im. Marynarki Wojennej RP w Gdyni
- Michał Dereziński, 3 klasa, XIV LO im. Stanisława Staszica w Warszawie
- Łukasz Dudek, 3 klasa, II LO im. Jana III Sobieskiego w Krakowie
- Dominik Dudzik, 2 klasa, V LO im. Augusta Witkowskiego w Krakowie
- Artur Dwornik, 3 klasa, I LO im. T. Kościuszki w Koninie
- Przemysław Gajda, 3 klasa, V LO im. Augusta Witkowskiego w Krakowie
- Krzysztof Gogolewski, 3 klasa, I LO im. Hugona Kołłątaja w Krotoszynie
- Konrad Gołuchowski, 3 klasa, VIII LO im. M. Skłodowskiej–Curie w Katowicach
- Bartosz Górski, 3 klasa, VIII LO im. Władysława IV w Warszawie
- Jan Inowolski, 3 klasa, XIV LO im. Stanisława Staszica w Warszawie
- Michał Iwaniuk, 1 klasa, XIV LO im. Stanisława Staszica w Warszawie
- Wojciech Jamrozy, 2 klasa, II Liceum Ogólnokształcące w Mielcu
- Piotr Janik, 3 klasa, V LO im. Augusta Witkowskiego w Krakowie
- Jacek Jendrej, 2 klasa, III LO im. Unii Lubelskiej w Lublinie
- Piotr Kieć, 3 klasa, ZSO Liceum Ogólnokształcące w Kamiennej Górze
- Marek Kiszkis, 3 klasa, III LO im. Marynarki Wojennej RP w Gdyni
- Jerzy Kozera, 3 klasa, VI LO im. J. i J. Śniadeckich w Bydgoszczy
- Bolesław Kulbabiński, 2 klasa, I LO im. Stefana Żeromskiego w Kielcach
- Bartosz Lewiński, 2 klasa, III LO im. Marynarki Wojennej RP w Gdyni
- Tomasz Marmołowski, 3 klasa, III LO im. Marynarki Wojennej RP w Gdyni
- Przemysław Mazur, 2 klasa, II LO im. Jana III Sobieskiego w Krakowie
- Łukasz Mazurek, 2 klasa, XIV LO im. Stanisława Staszica w Warszawie
- Mirosław Michalski, 2 klasa, III LO im. Marynarki Wojennej RP w Gdyni
- Cezary Myczka, 3 klasa, I LO im. T. Kościuszki w Gorzowie Wlkp.
- Jakub Oćwieja, 1 klasa, V Liceum Ogólnokształcące w Bielsku–Białej
- Maciej Pacut, 2 klasa, III LO im. Stefana Żeromskiego w Bielsku–Białej
- Szymon Pałka, 2 klasa, V LO im. Augusta Witkowskiego w Krakowie
- Jonasz Pamuła, 2 klasa, V LO im. Augusta Witkowskiego w Krakowie
- Szymon Piechowicz, 3 klasa, II LO im. Adama Mickiewicza w Słupsku
- Michał Pilch, 3 klasa, I LO im. Adama Mickiewicza w Białystoku
- Mateusz Piwnicki, 3 klasa, I LO im. Tadeusza Kościuszki w Wieluniu
- Robert Pohnke, 2 klasa, III LO im. Marynarki Wojennej RP w Gdyni
- Piotr Polesiuk, 2 klasa, I Liceum Ogólnokształcące w Wałbrzychu
- Tomasz Roda, 3 klasa, III LO im. Marynarki Wojennej RP w Gdyni
- Rafał Ruciński, 1 klasa, XIII Liceum Ogólnokształcące w Szczecinie
- Jan Rudol, 2 klasa, II LO im. Jana III Sobieskiego w Krakowie
- Marek Sapota, 3 klasa, III LO im. Marynarki Wojennej RP w Gdyni
- Michał Stachurski, 2 klasa, I LO im. M. Kopernika w Jarosławiu
- Damian Straszak, 2 klasa, I Liceum Ogólnokształcące w Raciborzu
- Marcin Walas, 3 klasa, V Liceum Ogólnokształcące w Bielsku–Białej
- Oskar Wantoła, 3 klasa, III LO im. Marynarki Wojennej RP w Gdyni
- Eryk Wieliczko, 1 klasa, III LO im. Marynarki Wojennej RP w Gdyni
- Jarosław Wódka, 3 klasa, I LO im. Wł. Broniewskiego w Bełchatowie
- Marek Wróbel, 3 klasa, V LO im. Augusta Witkowskiego w Krakowie

- Michał Wszołek, 3 klasa, V LO im. Augusta Witkowskiego w Krakowie
- Łukasz Zatorski, 2 klasa, X Liceum Ogólnokształcące we Wrocławiu
- Artur Żylinski, 3 klasa, III LO im. Marynarki Wojennej RP w Gdyni

Komitet Główny Olimpiady Informatycznej przyznał następujące nagrody rzeczowe:

(1) puchar ufundowany przez Olimpiadę Informatyczną przyznano zwycięzcy XIV Olimpiady — Tomaszowi Kulczyńskiemu,

(2) roczny abonament na książki ufundowany przez Wydawnictwa Naukowo–Techniczne przyznano zwycięzcy XIV Olimpiady — Tomaszowi Kulczyńskiemu,

(3) złote, srebrne i brązowe medale ufundowane przez Olimpiadę Informatyczną przyznano odpowiednio laureatom I, II i III miejsca,

(4) laptopy (3 szt.) ufundowane przez Prokom Software S.A. przyznano laureatom I miejsca,

(5) aparaty cyfrowe ufundowane przez Prokom Software S.A. i Olimpiadę Informatyczną przyznano laureatom II miejsca,

(6) odtwarzacze mp4 ufundowane przez Prokom Software S.A. przyznano laureatom III miejsca,

(7) pamięci pendrive ufundowane przez Ogólnopolską Fundację Edukacji Komputerowej przyznano wszystkim finalistom,

(8) książki ufundowane przez Wydawnictwa Naukowo–Techniczne przyznano wszystkim uczestnikom zawodów finałowych.

Ogłoszono komunikat o powołaniu reprezentacji Polski na:

- Międzynarodową Olimpiadę Informatyczną IOI'2007, która odbędzie się w Chorwacji, w miejscowości Zagrzeb w terminie 15–22 sierpnia 2007 r.:

  (1) Tomasz Kulczyński
  (2) Marcin Andrychowicz
  (3) Jakub Kallas
  (4) Marcin Kurczych

  rezerwowi:

  (5) Jarosław Gomułka
  (6) Piotr Niedźwiedź

- Olimpiadę Informatyczną Krajów Europy Środkowej CEOI'2007, która odbędzie się w Czechach w miejscowości Brno, w terminie 1–7 lipca 2007 r.:

  (1) Tomasz Kulczyński
  (2) Marcin Andrychowicz
  (3) Jakub Kallas
  (4) Marcin Kurczych

  rezerwowi:

  (5) Jarosław Gomułka
  (6) Piotr Niedźwiedź

- Bałtycką Olimpiadę Informatyczną, która odbędzie się w Niemczech w miejscowości Güstrow, w terminie 24–28 kwietnia 2007 r.:

  (1) Jarosław Błasiok
  (2) Maciej Klimek
  (3) Marcin Kościelnicki
  (4) Łukasz Wołochowski
  (5) Wojciech Baranowski
  (6) Marek Rogala

  rezerwowi:

  (7) Robert Obryk
  (8) Mateusz Baranowski

- **Obóz czesko–polsko–słowacki, Czechy — Praga, 10–16 czerwca 2007 r.:**
  - członkowie reprezentacji oraz zawodnicy rezerwowi powołani na Międzynarodową Olimpiadę Informatyczną (IOI'2007) i Olimpiadę Informatyczną Krajów Europy Środkowej (CEOI'2007).
- **Obóz im. A. Kreczmara, Nowy Sącz, 22 lipca — 4 sierpnia 2007 r.:**
  - reprezentanci na międzynarodowe zawody informatyczne, zawodnicy rezerwowi oraz wszyscy laureaci i finaliści, którzy uczęszczają do klas niższych niż maturalna.

Sekretariat wystawił łącznie 25 zaświadczeń o uzyskaniu tytułu laureata i 47 zaświadczeń o uzyskaniu tytułu finalisty XIV Olimpiady Informatycznej.

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

- Ireneusz Bujnowski (I LO w Białymstoku)
  - Michał Pilch — finalista
- Czesław Drozdowski (XIII LO w Szczecinie)
  - Rafał Ruciński — finalista
- Beata Duszyńska (I LO im. Wł. Broniewskiego w Bełchatowie)
  - Jarosław Wódka — finalista
- Andrzej Dyrek (V LO im. A. Witkowskiego w Krakowie)
  - Dominik Dudzik — finalista
  - Przemysław Gajda — finalista
  - Piotr Janik — finalista
  - Aleksandra Lipiec — laureatka III miejsca
  - Robert Obryk —laureat III miejsca
  - Szymon Pałka — finalista
  - Jonasz Pamuła — finalista
  - Michał Wszołek — finalista
- Grzegorz Gutowski (V LO im. A. Witkowskiego w Krakowie)
  - Piotr Janik — finalista
  - Marek Wróbel — finalista
- Wiesław Jakubiec (LO im. S. Żeromskiego w Bielsku–Białej)
  - Maciek Pacut — finalista
- Witold Jarnicki (Instytut Matematyki Uniwersytetu Jagiellońskiego w Krakowie)
  - Łukasz Dudek — finalista
  - Przemysław Mazur — finalista
  - Jan Rudol — finalista
- Anna Kamoda (I LO im. T. Kościuszki w Koninie)
  - Artur Dwornik — finalista
- Rafał Kosmalski (X LO we Wrocławiu)
  - Łukasz Zatorski — finalista
- Anna Kowalska (V LO w Bielsku–Białej)
  - Jacek Migdał — laureat III miejsca
  - Jakub Oćwieja — finalista
- Maria Król (I LO im. M. Kopernika w Jarosławiu)
  - Michał Stachurski — finalista
- Krzysztof Magiera (student Wydziału Informatyki AGH w Krakowie)
  - Łukasz Dudek — finalista
- Marek Nitkiewicz (II LO w Olsztynie)
  - Rafał Józefowicz — laureat II miejsca
- Rafał Nowak (Instytut Informatyki Uniwersytetu Wrocławskiego)
  - Jarosław Gomułka — laureat II miejsca
- Alfred Ortyl (II LO im. M. Kopernika w Mielcu)

  - Wojciech Jamrozy — finalista
- Jakub Piasecki (I LO w Swarzędzu)
  - Tomek Żurkowski — laureat III miejsca
- Adam Pucia (Pałac Młodzieży w Katowicach)
  - Piotr Niedźwiedź — laureat II miejsca
- Włodzimierz Raczek (V LO w Bielsku–Białej)
  - Jacek Migdał — laureat III miejsca
  - Jakub Oćwieja — finalista
- Jerzy Saran (III LO w Lublinie)
  - Jacek Jendrej — finalista
- Hanna Stachera (XIV LO im. St. Staszica w Warszawie)
  - Marcin Andrychowicz — laureat I miejsca
  - Michał Dereziński — finalista
  - Łukasz Mazurek — finalista
- Piotr Szczeciński (I LO im. Hugona Kołłątaja w Krotoszynie)
  - Krzysztof Gogolewski — finalista
- Ryszard Szubartowski (III LO im. Marynarki Wojennej RP w Gdyni)
  - Mateusz Baranowski — laureat III miejsca
  - Wojciech Baranowski — laureat III miejsca
  - Paweł Blokus — finalista
  - Jakub Kallas — laureat I miejsca
  - Marek Kiszkis — finalista
  - Bartosz Lewiński — finalista
  - Tomasz Marmołowski — finalista
  - Mirosław Michalski — finalista
  - Robert Pohnke — finalista
  - Tomasz Roda — finalista
  - Marek Sapota — finalista
  - Oskar Wantoła — finalista
  - Filip Wieczorek — laureat III miejsca
  - Eryk Wieliczko — finalista
  - Łukasz Wołochowski — laureat II miejsca
  - Artur Żylinski — finalista
- Paweł Walter (V LO im. A. Witkowskiego w Krakowie)
  - Dominik Dudzik — finalista
  - Robert Obryk — laureat III miejsca
  - Jonasz Pamuła — finalista
- Iwona Waszkiewicz (VI LO im. J. i J. Śniadeckich w Bydgoszczy)
  - Jerzy Kozera — finalista
  - Tomasz Kulczyński — laureat I miejsca
  - Błażej Osiński — laureat III miejsca
- Jacek Złydach (V LO im. A. Witkowskiego w Krakowie)
  - Michał Wszołek — finalista

Zgodnie z decyzją Komitetu Głównego z dn. 30 marca 2007 r. nauczyciele — opiekunowie naukowi laureatów i finalistów — otrzymają nagrody pieniężne.
Programy wzorcowe w postaci elektronicznej i testy użyte do sprawdzania rozwiązań zawodników będą dostępne na stronie Olimpiady Informatycznej: www.oi.edu.pl.

Warszawa, 29 czerwca 2007 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) Komitet Główny zastrzega sobie prawo do opublikowania rozwiązań zawodników, którzy zostali zakwalifikowani do następnego etapu, zostali wyróżnieni lub otrzymali tytuł laureata.

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

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

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

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

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

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

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

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

§4 KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

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

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

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

(4) Komitet dokonuje zmian w swoim składzie za zgodą Organizatora.

(5) Komitet powołuje i rozwiązuje komitety okręgowe Olimpiady.

(6) Komitet:

  (a) opracowuje szczegółowe „Zasady organizacji zawodów”, które są ogłaszane razem z treścią zadań zawodów I stopnia Olimpiady,

  (b) powołuje i odwołuje członków Jury Olimpiady, które jest odpowiedzialne za sprawdzenie zadań,

  (c) udziela wyjaśnień w sprawach dotyczących Olimpiady,

  (d) ustala listy laureatów i wyróżnionych uczestników oraz kolejność lokat,

  (e) przyznaje uprawnienia i nagrody rzeczowe wyróżniającym się uczestnikom Olimpiady,

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

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

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

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

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

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

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

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

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

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

(16) Przewodniczący:

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

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

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

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

§5 KOMITETY OKRĘGOWE

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

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

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

§6 PRZEBIEG OLIMPIADY

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

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

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

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

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

§7 UPRAWNIENIA I NAGRODY

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

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

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

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

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

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

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

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

§8 FINANSOWANIE OLIMPIADY

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

§9 PRZEPISY KOŃCOWE

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

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

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

Warszawa, 8 września 2006 r.

Komitet Główny Olimpiady Informatycznej

Zasady organizacji zawodów w roku szkolnym 2006/2007

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 2006/2007.

§1 WSTĘP

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

§2 ORGANIZACJA OLIMPIADY

(1) Olimpiadę przeprowadza Komitet Główny Olimpiady Informatycznej.

(2) Olimpiada Informatyczna jest trójstopniowa.

(3) W Olimpiadzie Informatycznej mogą brać indywidualnie udział uczniowie wszystkich typów szkół ponadgimnazjalnych i szkół średnich dla młodzieży dających możliwość uzyskania matury. W Olimpiadzie mogą również uczestniczyć — za zgodą Komitetu Głównego — uczniowie szkół podstawowych, gimnazjów, zasadniczych szkół zawodowych i szkół zasadniczych.

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

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

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

(7) Do zawodów II stopnia zostanie zakwalifikowanych 350 uczestników, których rozwiązania zadań I stopnia zostaną ocenione najwyżej; do zawodów III stopnia — 60 uczestników, których rozwiązania zadań II stopnia zostaną ocenione najwyżej. Komitet Główny może zmienić podane liczby zakwalifikowanych uczestników co najwyżej o 20%.

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

(9) Komitet Główny zastrzega sobie prawo do opublikowania rozwiązań zawodników, którzy zostali zakwalifikowani do następnego etapu, zostali wyróżnieni lub otrzymali tytuł laureata.

(10) Terminarz zawodów:

- zawody I stopnia — 23.10–20.11.2006 r.

  ogłoszenie wyników:
  - w witrynie Olimpiady — 18.12.2006 r.,
  - pocztą — 28.12.2006 r.
- zawody II stopnia — 06–08.02.2007 r.

  ogłoszenie wyników:
  - w witrynie Olimpiady — 19.02.2007 r.
  - pocztą — 02.03.2007 r.
- zawody III stopnia — 27–31.03.2007 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 20 listopada 2006 r. Komitet Główny nie ponosi odpowiedzialności za brak możliwości przekazania rozwiązań przez witrynę w sytuacji nadmiernego obciążenia lub awarii serwisu. Odbiór przesyłki zostanie potwierdzony przez Komitet Główny zwrotnym listem elektronicznym (prosimy o zachowanie tego listu). Brak potwierdzenia może oznaczać, że rozwiązanie nie zostało poprawnie zarejestrowane. W tym przypadku zawodnik powinien przesłać swoje rozwiązanie przesyłką poleconą za pośrednictwem zwykłej poczty. Szczegóły dotyczące sposobu postępowania przy przekazywaniu zadań i związanej z tym rejestracji będą podane w witrynie.
- Pocztą, przesyłką poleconą, na adres:

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

w nieprzekraczalnym terminie nadania do 20 listopada 2006 r. (decyduje data stempla pocztowego). Prosimy o zachowanie dowodu nadania przesyłki.

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

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

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

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

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

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

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

(7) Programy w *C/C++* będą kompilowane w systemie Linux za pomocą kompilatora GCC/G++ v. 4.1.1.
Programy w Pascalu będą kompilowane w systemie Linux za pomocą kompilatora FreePascal v. 2.0.4. Wybór polecenia kompilacji zależy od podanego rozszerzenia pliku w następujący sposób (np. dla zadania *abc*):

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

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

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

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

(10) Uczestnik korzystający z poczty zwykłej przysyła:

- Nośnik (dyskietkę lub CD-ROM) w standardzie dla komputerów PC, zawierający:

  - spis zawartości nośnika oraz dane osobowe zawodnika w pliku nazwanym `SPIS.TXT`,
  - do każdego rozwiązanego zadania — program źródłowy lub plik z danymi.

  Na nośniku nie powinno być żadnych podkatalogów.

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

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

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

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

(14) Od dnia 4 grudnia 2006 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 8 grudnia 2006 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 8 grudnia 2006 r. nie będą rozpatrywane.

§4 UPRAWNIENIA I NAGRODY

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

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

(3) Uprawnienia określone w p. 1. i 2. przysługują na zasadach określonych w rozporządzeniu MENiS z dnia 7 września 2004 r. w sprawie warunków i sposobu oceniania, klasyfikowania i promowania uczniów i słuchaczy oraz przeprowadzania sprawdzianów i egzaminów w szkołach publicznych. (Dz. U. z 2004 r. Nr 199, poz. 2046, § 18 i § 56) wraz z późniejszymi zmianami zawartymi w Rozporządzeniu MENiS z dnia 14 czerwca 2005 r. (Dz. U. z 2005 r. Nr 108, poz. 905).

(4) Laureaci i finaliści Olimpiady mają ułatwiony lub wolny wstęp do tych szkół wyższych, których senaty podjęły uchwały w tej sprawie, zgodnie z przepisami ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym, na zasadach zawartych w tych uchwałach (Dz.U. z 2005 r. Nr 164 poz. 1365).

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

(6) Komitet Główny ustala skład reprezentacji Polski na XIX Międzynarodową Olimpiadę Informatyczną w 2007 roku na podstawie wyników zawodów III stopnia i regulaminu tej Olimpiady Międzynarodowej.

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

(8) Wyznaczeni przez Komitet Główny reprezentanci Polski na olimpiady międzynarodowe oraz finaliści, którzy nie są w ostatniej programowo klasie swojej szkoły, zostaną zaproszeni do nieodpłatnego udziału w VIII Obozie Naukowo–Treningowym im. Antoniego Kreczmara, który odbędzie się w czasie wakacji 2007 r.

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

§5 PRZEPISY KOŃCOWE

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

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

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

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

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

Zawody I stopnia

opracowania zadań

Zbigniew J. Czech **Piotr Stańczyk**
Treść zadania, Opracowanie **Program**

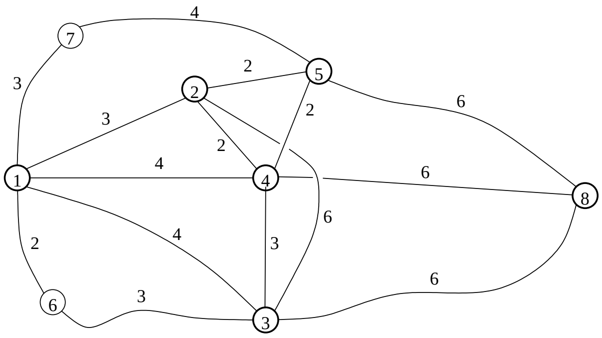
OI, Etap I

Atrakcje turystyczne

Bajtazar jedzie z Bitowic do Bajtogrodu. Po drodze chce odwiedzić kilka wybranych miejscowości, w których znajdują się interesujące zabytki, dobre restauracje czy inne atrakcje turystyczne. Kolejność odwiedzania wybranych miejscowości nie jest całkowicie obojętna. Na przykład, Bajtazar wolałby nie wspinać się na wieżę zamku Bitborku po sutym obiedzie zjedzonym w Cyfronicach, a do Zipowic (na słynną kawę Compresso) chciałby wpaść raczej po obiedzie, a nie przed. Jednak kolejność odwiedzania wybranych miejscowości nie jest całkowicie ustalona i do pewnego stopnia elastyczna. Ze względu na obłędne ceny benzyny, Bajtazar chce tak zaplanować trasę przejazdu, żeby była jak najkrótsza. Pomóż mu wyznaczyć długość najkrótszej trasy spełniającej jego wymagania.

Sieć drogowa składa się z n *miejscowości i łączących je* m *dróg. Miejscowości są ponumerowane od 1 do* n*, a drogi od 1 do* m*. Każda droga łączy dwie różne miejscowości i jest dwukierunkowa. Drogi spotykają się tylko w miejscowościach (w których mają końce) i nie przecinają się poza nimi. Drogi mogą prowadzić przez estakady i tunele. Każda droga ma określoną długość. Parę miejscowości może łączyć co najwyżej jedna bezpośrednia droga.*

Oznaczmy przez k *liczbę wybranych miejscowości, które chce odwiedzić Bajtazar. Numeracja miast jest taka, że Bitowice mają numer 1, Bajtogród numer* n*, a miejscowości, które chce odwiedzić Bajtazar, mają numery* $2, 3, \ldots, k+1$.



Na rysunku przedstawiono przykładową sieć dróg. Powiedzmy, że Bajtazar chce odwiedzić miejscowości 2, 3, 4 i 5, przy czym miejscowość 2 chce odwiedzić przed miejscowością 3, a miejscowości 4 i 5 po miejscowości 3. Wówczas najkrótsza trasa biegnie przez miejscowości 1, 2, 4, 3, 4, 5, 8 i ma ona długość 19.

Zauważmy, że miejscowość 4 pojawia się na tej trasie przed i po miejscowości 3. Jednak przed odwiedzeniem miejscowości 3 Bajtazar nie zatrzyma się w niej, gdyż takie przyjął ograniczenia. Nie oznacza to jednak, że w ogóle nie może wcześniej przejeżdżać przez tę miejscowość.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis sieci drogowej, listę wybranych miejscowości, które chce odwiedzić Bajtazar oraz ograniczenia, co do kolejności, w jakiej chce je odwiedzić,*
- *wyznaczy długość najkrótszej trasy, przechodzącej przez wszystkie wybrane przez Bajtazara miejscowości w odpowiedniej kolejności,*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu standardowego wejścia znajdują się trzy liczby całkowite n*,* m *i* k *pooddzielane pojedynczymi odstępami,* $2 \leqslant n \leqslant 20\,000$, $1 \leqslant m \leqslant 200\,000$, $0 \leqslant k \leqslant 20$*; ponadto jest spełniona nierówność* $k \leqslant n-2$.

Kolejne m *wierszy zawiera opisy dróg, po jednej w wierszu. Wiersz* $i+1$*-szy zawiera trzy liczby całkowite* p_i*,* q_i *i* l_i*, pooddzielane pojedynczymi odstępami,* $1 \leqslant p_i < q_i \leqslant n$, $1 \leqslant l_i \leqslant 1\,000$*. Liczby te oznaczają drogę łączącą miejscowości* p_i *i* q_i*, o długości* l_i*. Możesz założyć, że dla każdych danych testowych można z Bitowic dojechać do Bajtogrodu oraz do wszystkich miejscowości, które Bajtazar chce odwiedzić.*

W $m+1$*-szym wierszu znajduje się jedna liczba całkowita* g*,* $0 \leqslant g \leqslant \frac{k\cdot(k-1)}{2}$*. Jest to liczba ograniczeń dotyczących kolejności odwiedzania wybranych przez Bajtazara miast. Ograniczenia te są podane w kolejnych* g *wierszach, po jednym w wierszu. Wiersz* $m+i+1$*–szy zawiera dwie liczby całkowite* r_i *i* s_i *oddzielone pojedynczym odstępem,* $2 \leqslant r_i \leqslant k+1$,

$2 \leqslant s_i \leqslant k+1$, $r_i \neq s_i$. Para liczb r_i i s_i oznacza, że Bajtazar chce odwiedzić miejscowość r_i przed odwiedzeniem miejscowości s_i. Nie oznacza to, że nie może przejechać przez s_i przed odwiedzeniem r_i ani że nie może przejechać przez r_i po odwiedzeniu s_i, jednak nie będzie się on wtedy zatrzymywał ani zwiedzał żadnych atrakcji turystycznych. Możesz założyć, że dla każdych danych testowych istnieje przynajmniej jedna kolejność zwiedzania wybranych miejscowości spełniająca wszystkie ograniczenia.

Wyjście

W pierwszym i jedynym wierszu standardowego wyjścia należy wypisać jedną liczbę całkowitą, będącą długością najkrótszej trasy z Bitowic do Bajtogrodu, przechodzącej przez wszystkie wybrane przez Bajtazara miejscowości w odpowiedniej kolejności.

Przykład

Dla danych wejściowych:

```
8 15 4
1 2 3
1 3 4
1 4 4
1 6 2
1 7 3
2 3 6
2 4 2
2 5 2
3 4 3
3 6 3
3 8 6
4 5 2
4 8 6
5 7 4
5 8 6
3
2 3
3 4
3 5
```

poprawnym wynikiem jest:

```
19
```

Rysunek i wyjaśnienie do przykładu znajdują się w treści zadania.

Rozwiązanie

Problem przestawiony w zadaniu możemy w naturalny sposób wyrazić w języku grafów — miasta to wierzchołki grafu, a drogi to nieskierowane krawędzie o określonych długościach. Wówczas poszukiwana trasa to najkrótsza ścieżka w grafie łącząca wierzchołek początkowy z końcowym i przebiegająca w określonym porządku przez pozostałe wierzchołki.

Zadanie polegające na wyznaczaniu najkrótszej ścieżki łączącej dwa wierzchołki i przechodzącej przez zadany zbiór wierzchołków jest problemem NP–trudnym, powiązanym z zagadnieniem wyznaczania drogi Hamiltona w grafie. Po dodaniu ograniczeń na kolejność odwiedzania miejscowości przez Bajtazara, zadanie pozostaje NP–trudne — każde znane rozwiązanie tego zadania działa w czasie wykładniczym ze względu na k.

Rozwiązanie zadania można podzielić na dwie niezależne części. W części pierwszej wyznaczamy odległości między wszystkimi parami miejscowości, które Bajtazar chce odwiedzić (włączając w to Bitowice i Bajtogród), czyli pomiędzy wierzchołkami o numerach $\{1,2,\ldots,k+1,n\}$. Druga część rozwiązania zadania polega na wyznaczeniu najkrótszej drogi z Bitowic (wierzchołka o numerze 1) do Bajtogrodu (wierzchołka o numerze n), przechodzącej przez wszystkie wybrane miejscowości (wierzchołki o numerach $\{2,3,\ldots,k+1\}$) w kolejności spełniającej zadane warunki.

Faza 1

W celu wyznaczenia długości najkrótszych ścieżek między wszystkimi parami miast, które Bajtazar chce odwiedzić, można wykonać $k+2$ razy algorytm Dijkstry, zaczynając kolejno z wierzchołków $1,2,\ldots,k+1,n$. Za każdym razem algorytm wykonywany jest na całym grafie wejściowym, zatem czas jego działania to $O(m \cdot \log n)$. Ponieważ algorytm powtarzamy $O(k)$ razy, pierwszą fazę jesteśmy w stanie wykonać w sumarycznym czasie $O(k \cdot m \cdot \log n)$. Po każdym wykonaniu algorytmu Dijkstry zapamiętujemy odległości od wierzchołka startowego do pozostałych wierzchołków ze zbioru $\{1,2,\ldots,k+1,n\}$.

Zamiast korzystać z algorytmu Dijkstry, w fazie pierwszej można zastosować również algorytm Floyda-Warshalla, uzyskując złożoność rzędu $O(n^3)$. Biorąc jednak pod uwagę możliwy rozmiar danych, rozwiązanie to ma istotnie gorszą złożoność.

Faza 2

Druga faza algorytmu polega na wyznaczeniu najkrótszej ścieżki wychodzącej z wierzchołka 1 i kończącej się w wierzchołku n. Ścieżka ta musi odwiedzić wszystkie wierzchołki ze zbioru $\{2,3,\ldots,k+1\}$ i dodatkowo muszą być uwzględnione wymagania dotyczące kolejności odwiedzania tych wierzchołków.

Prostym sposobem wyznaczenia najkrótszej ścieżki jest wygenerowanie wszystkich możliwych ścieżek, a następnie wybranie najkrótszej spośród nich. Rozwiązanie takie sprowadza się do wygenerowania wszystkich permutacji zbioru k miast, które Bajtazar chce odwiedzić, sprawdzenia dla każdej permutacji, czy zachowane są ograniczenia dotyczące kolejności odwiedzania miast, a następnie obliczenia długości ścieżki. Na koniec, spośród wszystkich dozwolonych ścieżek, wybieramy najkrótszą i jej długość zwracamy jako wynik. Algorytm taki może być zaimplementowany w prosty sposób, by działał w czasie $O(k^2 \cdot k!)$, gdyż możemy mieć $O(k^2)$ ograniczeń, które należy sprawdzić dla każdej z wygenerowanych permutacji. Można rozwiązanie to nieco poprawić, osiągając złożoność $O(k \cdot k!)$. W tym celu wykorzystujemy fakt, iż zależności między kolejnością odwiedzania wierzchołków możemy reprezentować dla pojedynczego wierzchołka przy użyciu pojedynczej zmiennej typu całkowitego 32-bitowego (`int` w C/C++, `longint` w Pascalu), dzięki temu, że $k \leqslant 20 < 32$. Dla każdego wierzchołka przechowujemy jedną taką zmienną, której m-ty bit jest ustawiony na 1, o ile rozpatrywany wierzchołek jest zależny od wierzchołka o numerze m. Podczas sprawdzania zgodności permutacji wierzchołków z zadaną kolejnością ich odwiedzania, konstruujemy 32-bitową zmienną odwiedzenia, której m-ty bit jest ustawiony na 1, o ile wierzchołek o numerze m został już odwiedzony. Sprawdzenie zależności dla m-tego wierzchołka sprowadza się w takiej sytuacji do zweryfikowania, czy wszystkie zapalone bity w zmiennej zależności są zapalone w zmiennej odwiedzenia (czyli wykonaniu operacji logicznej AND). Dodatkowo, stosując algorytm generowania permutacji w kolejności wymagającej wykonywania jedynie transpozycji sąsiednich elementów[1], jesteśmy w stanie zaimplementować rozwiązanie tak, by działało w czasie $O(k!)$.

Możliwe jest także inne rozwiązanie, także oparte na pomyśle generowania wszystkich permutacji odwiedzanych wierzchołków. Możemy mianowicie generować stopniowo rozwiązanie (permutację), zatrzymując się w momencie, gdy dochodzimy do miejsca, z którego nie można już kontynuować poprawnej ścieżki. Algorytm ten można zaimplementować tak, by działał w czasie $O(k \cdot p)$, gdzie p to liczba poprawnych permutacji.

W tym celu zaczynamy od pustego ciągu i dokładamy do niego kolejne wierzchołki spośród miast do odwiedzenia — takie, które są zależne jedynie od wierzchołków już znajdujących się w ciągu (czyli powinny być odwiedzone później niż one). W kroku pierwszym możemy umieścić w ciągu tylko wierzchołki niezależne — załóżmy, że wybraliśmy wierzchołek w. W drugim kroku możemy dołączyć wierzchołki niezależne (inne niż w) oraz wierzchołki zależne wyłącznie od wierzchołka w itd.

Powinniśmy jeszcze uzasadnić, że opisana procedura pozwoli nam przejrzeć wszystkie poprawne permutacje. Wystarczy zauważyć, że w każdej poprawnej permutacji, którą chcielibyśmy otrzymać, wierzchołki występują w kolejności zgodnej z zasadą, z jaką są umieszczane w ciągu — przed wierzchołkiem u nie może wystąpić żaden wierzchołek zależny od u. Dodatkowo, w zadaniu jest zagwarantowane, że dla przedstawionych danych istnieje rozwiązanie, więc przynajmniej jedną permutację znajdziemy.

Niestety, poprawnych permutacji może być wiele. W przypadku, gdy mamy niewiele warunków ograniczających kolejność odwiedzania miast (w szczególności, gdy nie ma ich wcale) liczba poprawnych permutacji może być bliska $k!$ i czas działania opisanego algorytmu może wynosić $O(k!)$.

Okazuje się jednak, że w zadaniu można zastosować technikę programowania dynamicznego. Dla każdego podzbioru $A \subseteq \{2,3,\ldots,k+1\}$ już odwiedzonych wierzchołków[2] oraz wierzchołka $v \in A$ (o ile A jest niepusty) odwiedzonego jako ostatni, możemy zapamiętać długość najkrótszej ścieżki z wierzchołka 1 do v odwiedzającej wszystkie wierzchołki A (i tylko przez te, spośród wybranych). Poszukiwana wartość dla zadanych A i v jest równa

$$\mathcal{D}(A,v) = \min(\mathcal{D}(A-\{w\},w) + d(w,v)),$$

gdzie minimum obliczamy po wszystkich wierzchołkach $w \in A$, a $d(w,v)$ to minimalna odległość między wierzchołkami w i v obliczona w pierwszej fazie. Gdy $A = \emptyset$, wówczas dla każdego $v \in \{2,3,\ldots,k+1\}$ przyjmujemy, że

$$\mathcal{D}(\emptyset,v) = 0.$$

---

[1] Ten i inne algorytmy generowania permutacji można znaleźć w [22].

[2] Dowolny podzbiór $A \subseteq \{2,3,\ldots,k+1\}$ możemy reprezentować przy użyciu liczby naturalnej $m \in \{0,1,\ldots,2^k-1\}$ — jeśli i-ty bit liczby m jest ustawiony na 1, oznacza to, że do rozpatrywanego zbioru A należy wierzchołek o numerze $i+2$. Sprawdzanie przynależności elementu do zbioru można wówczas zrealizować za pomocą operacji bitowej AND oraz przesunięcia bitowego.

Po obliczeniu wszystkich wartości $\mathcal{D}$, odpowiedzią na postawione w zadaniu pytanie będzie najmniejsza wartość $\mathcal{D}(A_0, v)$, dla $A_0 = \{2, \ldots, k+1\}$ oraz dowolnego $v \in A_0$, powiększona o odległość między wierzchołkiem v a wierzchołkiem n. Uzyskujemy w ten sposób algorytm działający w czasie $O(k^2 \cdot 2^k)$, gdyż mamy do rozpatrzenia nie więcej niż $k \cdot 2^k$ par złożonych z wierzchołka i podzbioru, a dla każdej pary obliczenia wykonujemy w czasie $O(k)$.

Złożoność pamięciowa rozwiązania dynamicznego

W warunkach zadania określono ograniczenie na pamięć w wysokości 64 MB. Rozwiązanie dynamiczne o złożoności czasowej $O(k^2 \cdot 2^k)$ wymaga zapamiętania $k \cdot 2^k$ rozwiązań — na każde z nich potrzebne są 4 bajty. Oznacza to, że dla maksymalnego zestawu danych potrzeba około 80 MB pamięci ($20 \cdot 2^{20} \cdot 4 \approx 80\,000\,000$). Nie mieści się to w zadanych ograniczeniach, więc musimy zmodyfikować nasze rozwiązanie tak, by zmniejszyć zapotrzebowanie na pamięć. Zauważmy, że przechowywanie przez cały czas wszystkich podrozwiązań nie jest konieczne. W celu obliczenia rozwiązań częściowych dla wszystkich zbiorów o określonej mocy, wystarczy, że będziemy pamiętali rozwiązania częściowe dla zbiorów o mocy o jeden mniejszej. Tak więc na początku, zaczynając od zbioru pustego, generujemy rozwiązania dla zbiorów jednoelementowych, następnie możemy wygenerować rozwiązania dla zbiorów dwuelementowych itd. Najwięcej pamięci potrzebujemy w czasie przetwarzania zbiorów o mocach $k/2$ i $k/2+1$. Stąd zapotrzebowanie na pamięć w całym algorytmie można ograniczyć do $2k \cdot \binom{k}{k/2}$ (dla uproszczenia zakładamy, że $2 \mid k$), co jest znaczną poprawą w stosunku do początkowej wartości $k \cdot 2^k$ i mieści się w zadanych limitach. Podejście takie utrudnia nieco sposób przechowywania wyliczanych wyników — nie możemy sobie pozwolić na użycie tablicy wielkości $k \cdot 2^k$. Dlatego też konieczne jest wprowadzenie dodatkowej listy, do której będą wstawiane analizowane zbiory o kolejnych rosnących mocach. W celu szybkiego wyszukiwania zadanego podzbioru w liście, możemy zastosować tablicę o wielkości 2^k, w której będziemy przechowywali pozycje poszczególnych zbiorów na liście.

Rozwiązanie wzorcowe

Rozwiązanie wzorcowe, którego implementacja jest zawarta w plikach `atr.cpp` i `atr0.pas`, polega na wielokrotnym zastosowaniu algorytmu Dijkstry w fazie 1, a algorytmu dynamicznego w fazie 2. Jego złożoność czasowa to $O(k \cdot m \cdot \log n + k^2 \cdot 2^k)$.

Testy

Programy zawodników zostały przetestowane na następującym zestawie danych testowych (kolumny zawierają wartości n, m, k, g oraz ograniczenie na długość krawędzi w grafie):

Nazwa	n	m	k	g	ogr
atr1a.in	10	9	5	0	10
atr1b.in	10	30	5	3	10
atr1c.in	4	5	0	0	10
atr2a.in	100	99	9	30	100
atr2b.in	100	1 000	9	15	100
atr3a.in	300	600	11	10	200
atr3b.in	300	5 000	12	16	200
atr4a.in	500	499	15	105	500
atr4b.in	500	9 000	16	90	500
atr5a.in	1 000	3 000	20	0	500
atr5b.in	1 000	5 000	20	40	500
atr6a.in	2 000	1 999	12	10	10
atr6b.in	2 000	5 000	12	20	10
atr7a.in	3 000	30 000	14	3	100
atr7b.in	7 000	120 000	14	2	100
atr8a.in	12 000	200 000	16	0	200
atr8b.in	5 000	4 999	16	10	200

Nazwa	**n**	**m**	**k**	**g**	**ogr**
atr9a.in	10 000	50 000	18	105	500
atr9b.in	10 000	200 0000	18	50	500
atr10a.in	20 000	200 000	20	0	1 000
atr10b.in	20 000	5 000	20	50	1 000
atr11a.in	20 000	20 000	20	19	1 000
atr11b.in	20 000	170 000	20	2	1 000

Wszystkie testy poza *atr11a.in* są pseudolosowe, z podanymi wyżej parametrami. Test *atr11a.in* jest ścieżką, po której trzeba chodzić ciągle od jednego końca do drugiego; odpowiedź dla tego przypadku jest bardzo duża — wynosi około 500 000 000. „Złośliwy” jest także test *atr1c.in*, w którym $k = 0$.

Marek Cygan, Jakub Radoszewski
Treść zadania, Opracowanie

Adam Iwanicki
Program

OI, Etap I

Biura

Firma Bajtel jest potentatem na bajtockim rynku telefonów komórkowych. Każdy jej pracownik otrzymał służbowy telefon, w którym ma zapisane numery telefonów niektórych swoich współpracowników (a wszyscy ci współpracownicy mają w swoich telefonach zapisany jego numer). W związku z dynamicznym rozwojem firmy zarząd postanowił przenieść siedzibę firmy do nowych biurowców. W celu polepszenia efektywności pracy zostało postanowione, że każda para pracowników, którzy będą pracować w różnych budynkach, powinna znać (nawzajem) swoje numery telefonów, tzn. powinni oni mieć już zapisane nawzajem swoje numery w służbowych telefonach komórkowych. Równocześnie, zarząd postanowił zająć jak największą liczbę biurowców, aby zapewnić pracownikom komfort pracy. Pomóż zarządowi firmy Bajtel zaplanować liczbę biur i ich wielkości tak, aby spełnić oba te wymagania.

Zadanie

Napisz program, który:

- *wyczyta ze standardowego wejścia opis, czyje numery telefonów mają zapisane w swoich telefonach komórkowych poszczególni pracownicy,*
- *wyznaczy maksymalną liczbę biurowców oraz ich wielkości, spełniające warunki postawione przez zarząd firmy Bajtel,*
- *wypisze wynik na standardowe wyjście.*

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby całkowite: n oraz m ($2 \leqslant n \leqslant 100\,000$, $1 \leqslant m \leqslant 2\,000\,000$), oddzielone pojedynczym odstępem i oznaczające odpowiednio: liczbę pracowników firmy Bajtel i liczbę par współpracowników, którzy mają zapisane nawzajem swoje numery telefonów w swoich telefonach komórkowych. Pracownicy firmy są ponumerowani od 1 do n.

Każdy z kolejnych m wierszy zawiera po jednej parze liczb całkowitych a_i i b_i ($1 \leqslant a_i < b_i \leqslant n$ dla $1 \leqslant i \leqslant m$), oddzielonych pojedynczym odstępem i oznaczających, że pracownicy o numerach a_i i b_i mają zapisane nawzajem swoje numery telefonów w swoich telefonach komórkowych. Każda para liczb oznaczających pracowników pojawi się na wejściu co najwyżej raz.

Wyjście

Pierwszy wiersz wyjścia powinien zawierać jedną liczbę całkowitą: maksymalną liczbę biurowców, które powinna zająć firma Bajtel. Drugi wiersz wyjścia powinien zawierać niemalejący ciąg dodatnich liczb całkowitych, pooddzielanych pojedynczymi odstępami, oznaczających wielkości biurowców (liczby rozlokowanych w nich pracowników). Jeżeli istnieje więcej niż jedno poprawne rozwiązanie, Twój program powinien wypisać dowolne z nich.

Przykład

Dla danych wejściowych:

```
7 16
1 3
1 4
1 5
2 3
3 4
4 5
4 7
4 6
5 6
6 7
2 4
2 7
2 5
3 5
3 7
1 7
```

poprawnym wynikiem jest:

```
3
1 2 4
```

Przykładowy dobry przydział pracowników do biur wygląda następująco: do pierwszego biura pracownik o numerze 4, do drugiego pracownicy 5 i 7, a do trzeciego pracownicy 1, 2, 3 i 6.

Rozwiązanie

Analiza problemu

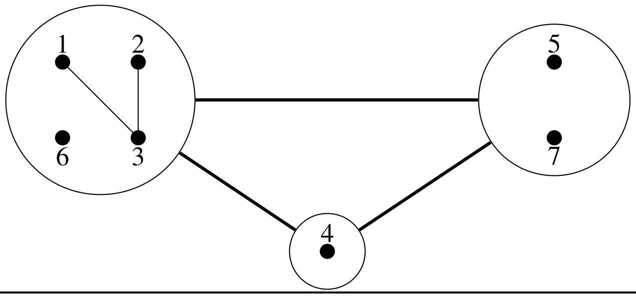
Problem z zadania można opisać za pomocą nieskierowanego grafu $G = (V,E)$. Przyjmijmy, że pracownicy Bajtelu to wierzchołki $V = \{1,2,\ldots,n\}$. Zbiór krawędzi E to zbiór par (u,v) (możemy założyć, że $u < v$, ponieważ graf jest nieskierowany), gdzie u i v oznaczają pracowników posiadających nawzajem swoje numery telefonów.

Przydzielenie pracowników do biur to w interpretacji grafowej podział zbioru wierzchołków V na parami rozłączne, niepuste podzbiory $V_1 \cup \cdots \cup V_k = V$, dla którego:

- dowolne dwa wierzchołki, należące do różnych podzbiorów są połączone krawędzią:

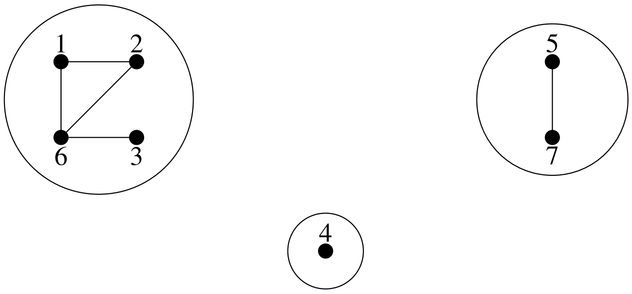
$$\forall_{1 \leqslant i < j \leqslant k} \forall_{u \in V_i} \forall_{v \in V_j}\ (u,v) \in E,$$

- liczba skonstruowanych podzbiorów $V_1, V_2, \ldots, V_k$ jest możliwie największa.



Rys. 1: Struktura grafu G dla przykładu z treści zadania. Większe kółka oznaczają optymalny podział zbioru V (maksymalizujący k), a krawędzie między kółkami oznaczają, że każda para pracowników z połączonych podzbiorów jest w G połączona krawędzią.

Problem podziału grafu na zbiory wierzchołków spełniające podane wyżej kryteria nie brzmi naturalnie i znajomo. Okazuje się jednak, że modyfikując graf G, możemy sprowadzić zadanie do problemu dobrze znanego. Zmieńmy radykalnie zbiór krawędzi w grafie G — usuńmy wszystkie krawędzie ze zbioru E i dodajmy takie pary (u,v), które dotychczas nie były połączone krawędzią. Skonstruowany graf nazwiemy *dopełnieniem grafu G* i będziemy go oznaczać $G' = (V,E')$, gdzie $E' = \{(u,v) : 1 \leqslant u < v \leqslant n\} - E$.



Rys. 2: Graf G' — dopełnienie grafu dla przykładu z treści zadania.

Zauważmy, że jeżeli w zmodyfikowanym grafie G' dwa wierzchołki z V są połączone krawędzią, to muszą oznaczać pracowników pracujących w tym samym biurze, czyli każda spójna składowa grafu G' musi być w całości zawarta w jednym biurze. Z drugiej strony, pracownicy, którym odpowiadają wierzchołki z różnych składowych, mogą być umieszczeni w różnych biurach. Maksymalną liczbę biur k uzyskamy więc, lokując pracowników każdej składowej w odrębnym biurze.

Sprowadziliśmy zatem oryginalne zadanie do problemu znajdowania spójnych składowych w dopełnieniu zadanego grafu — problemu, który ma wiele klasycznych rozwiązań. Są wśród nich przeszukiwanie grafu wszerz (BFS) bądź w głąb (DFS), czy wykorzystanie struktury *Find-Union* (opisy tych metod można znaleźć np. w książce [19]). Jednak ich zastosowanie w naszym zadaniu może być trudne — przeszkodę stanowi fakt, że graf G' może mieć bardzo dużo krawędzi. Ich liczba może wynosić nawet $\frac{n\cdot(n-1)}{2}$, co w przypadku ograniczeń z zadania oznacza około 5 miliardów krawędzi. Poszukiwanie spójnych składowych w grafie G' o tak gigantycznym rozmiarze musimy przeprowadzić bez konstruowania grafu G'. Zanim pokażemy, jak to zrobić, zastanówmy się, jak najlepiej zaimplementować jeden z klasycznych algorytmów.

Rozwiązanie o złożoności czasowej $O(n^2)$

W plikach `bius0.pas` oraz `bius1.cpp` znajdują się rozwiązania zadania, w których spójne składowe grafu G' są wyznaczane algorytmem przeszukiwania grafu w głąb (DFS). W pliku `bius2.cpp` znajduje się rozwiązanie, wykorzystujące do tego celu algorytm *Find-Union*. Ponieważ graf G' może mieć prawie $\frac{n^2}{2}$ krawędzi, to złożoność czasowa zaimplementowanych rozwiązań wynosi $O(n^2)$ (w przypadku wykorzystania algorytmu BFS lub DFS) lub $O(n^2 \log^* n)$ (przy użyciu struktury *Find-Union*).

W załączonych rozwiązaniach nie konstruujemy *explicite* grafu G', gdyż wymagałoby to zbyt wiele pamięci. Dla każdego wierzchołka, na bieżąco, w miarę potrzeby wyznaczamy wychodzące z niego krawędzie. W tym celu wstępnie sortujemy listy sąsiadów wszystkich wierzchołków grafu G. Lista sąsiadów wierzchołka v w grafie G pozwala nam także przejrzeć w czasie $O(n)$ wszystkich sąsiadów wierzchołka v w grafie G' — są to wierzchołki, których brak na liście. Co więcej, taka reprezentacja grafu G' nie wymaga więcej pamięci niż reprezentacja grafu G.

Sortowanie list sąsiadów G można zrealizować w złożoności czasowej $O(m \log m)$ za pomocą jednego z klasycznych algorytmów sortowania, na przykład sortowania przez scalanie. Operację tę można wykonać także w czasie $O(n+m)$ za pomocą sortowania pozycyjnego. W tym celu krawędzie grafu G przedstawiamy jako pary liczb naturalnych z przedziału $[1,n]$. Opisy algorytmów sortowania przez scalanie (ang. *Merge Sort*) i sortowania pozycyjnego (ang. *Radix Sort*) znajdują się, na przykład, w książce [19].

Rozwiązanie wzorcowe

Poszukajmy efektywniejszej metody wyznaczania spójnych składowych grafu G' bez właściwej konstrukcji tego grafu. Zmodyfikujmy w tym celu algorytm *Find-Union* (patrz [19]), który polega na stopniowym grupowaniu wierzchołków w zbiory odpowiadające spójnym składowym. Początkowo tworzymy n zbiorów jednoelementowych — każdy wierzchołek będzie w swoim zbiorze. Następnie przetwarzamy wierzchołki grafu G w dowolnej kolejności. Dla każdego wierzchołka v chcielibyśmy zaktualizować istniejący podział na spójne składowe, uwzględniając wszystkie krawędzie w grafie G' incydentne z tym wierzchołkiem, tzn. łącząc zbiory, pomiędzy którymi przebiegają te krawędzie.

Załóżmy, że przed przetworzeniem wierzchołka v mamy w strukturze *Find-Union* zapisany aktualny podział grafu na składowe $S_1, S_2, \ldots, S_l$ i niech $v \in S_1$. Uwzględnienie krawędzi wychodzących z v może spowodować zmianę podziału G' na spójne składowe — pewne spośród składowych $S_2, \ldots, S_l$ mogą zostać połączone z S_1. Aby stwierdzić, czy krawędzie G' incydentne z v spowodują połączenie składowych S_1 oraz S_i, wystarczy porównać:

- liczbę krawędzi łączących wierzchołek v ze składową S_i w grafie G,
- z liczbą wierzchołków tej składowej $|S_i|$.

Jeśli pierwsza z tych liczb jest mniejsza, to w grafie G' musi istnieć krawędź łącząca wierzchołek v z wierzchołkiem ze zbioru S_i. W ten sposób, korzystając z reprezentacji grafu G, możemy konstruować składowe grafu G' — uwalniając się tym samym od konieczności myślenia o reprezentacji „niewygodnego” grafu G'.

Oznaczmy przez a_i liczbę krawędzi grafu G, łączących wierzchołek v ze spójną składową S_i. Czy potrafimy efektywnie wyznaczyć tę liczbę? Okazuje się, że tak!

- Na początku dla każdej spójnej składowej ustalamy a_i równe 0.
- Dla każdej krawędzi (v,u) wychodzącej z v w grafie G, gdzie $u \in S_j$ oraz $j \neq 1$, zwiększamy wartość a_j o jeden. Znalezienie zbioru S_j, do którego należy wierzchołek u, wykonywane jest za pomocą operacji *Find*.
- Każdą spójną składową S_i ($i \neq 1$), dla której $a_i < |S_i|$, łączymy ze składową S_1. Łączenie zbiorów jest wykonywane za pomocą operacji *Union*.

Po rozważeniu wszystkich wierzchołków v ze zbioru V otrzymujemy podział G' na spójne składowe. Trzeba jeszcze zastanowić się, ile czasu zajmie nam osiągnięcie tego efektu. Niejasne wydaje się, dlaczego zaprezentowany algorytm miałby być szybki: wszak złożoność czasowa przetworzenia jednego wierzchołka v to pesymistycznie $O(n)$ (tyle może być łącznie składowych G'), a w algorytmie wykonywanych jest n kroków. Przyjrzyjmy się zatem dokładniej, ile razy poszczególne operacje zostają wykonane w trakcie działania algorytmu.

- Operacji zerowania zmiennych a_i (pierwsza grupa operacji) wydaje się być zdecydowanie za dużo. Zauważmy jednak, że jeśli zmienna a_i związana ze składową S_i na zakończenie fazy jest równa zero, to na pewno S_i zostanie połączona z S_1 i zniknie przed następną fazą. Operacji zerowania *takich zmiennych* może więc być w czasie trwania całego algorytmu najwyżej $O(n)$, bo tyle jest operacji łączenia składowych. Jeśli natomiast zmienna a_i jest w danej fazie zwiększana, to musiała istnieć krawędź w grafie G (łącząca S_1 z S_i), która to spowodowała. Niezależnie od tego, czy składowa S_i zostanie połączona z S_1 w tej fazie czy nie, takich operacji zerowania a_i może być w całym algorytmie najwyżej m. Razem operacje z tej grupy można więc wykonać w czasie $O(m+n)$.
- Operacje z drugiej grupy są w widoczny sposób związane z istnieniem krawędzi — dla każdej krawędzi grafu G wykonujemy jedno dodawanie i jedną operację *Find*. Razem wymaga to czasu $O(m \log^* n)$.
- Oszacujmy jeszcze koszt operacji z trzeciej grupy. Łączna liczba sprawdzeń warunku $a_i < |S_i|$ jest taka sama, jak łączna liczba zerowań zmiennych a_i, czyli $O(m+n)$. Z kolei liczba wykonań operacji *Union* jest oczywiście rzędu $O(n)$. Cała ta grupa operacji jest więc wykonywana w czasie $O(m + n\log^* n)$.

Ostatecznie pokazaliśmy, że złożoność czasowa zaprezentowanego rozwiązania wynosi $O((n+m)\cdot\log^* n)$. Jego implementacja znajduje się w plikach: `biu.cpp` oraz `biu1.pas`.

Rozwiązanie alternatywne

Spójne składowe G' można także znaleźć, stosując zupełnie odmienne podejście niż poprzednio. W zależności od własności grafu G' zastosujemy do niego klasyczny algorytm znajdowania składowych lub wykażemy, że graf ma postać, przy której efektywne będzie inne podejście.

Podstawą podziału grafów na grupy będzie liczba krawędzi i wielkość składowych. Wielkości te można powiązać następująco:

Obserwacja 1 Jeśli w grafie G' istnieje spójna składowa S zawierająca k wierzchołków, to w grafie G jest co najmniej $k\cdot(n-k)$ krawędzi.

Dowód Poza spójną składową S w grafie G (a zatem i w G') jest dokładnie $n-k$ wierzchołków. Żaden z tych wierzchołków nie może być połączony krawędzią w grafie G' z żadnym wierzchołkiem z S. To oznacza, że w G każdy spośród tych $n-k$ wierzchołków jest połączony z każdym wierzchołkiem S, czyli w grafie G jest co najmniej $k\cdot(n-k)$ krawędzi. ■

Widzimy więc, że jeżeli graf G' zawiera sporą składową, powiedzmy o rozmiarze zbliżonym do $\frac{n}{2}$, to graf G musi mieć prawie $\frac{n^2}{4}$ krawędzi. Stąd, jeżeli m jest małe w stosunku do n^2, to w G' nie może być tak dużych składowych. Zastanówmy się, jak wykorzystać tę zależność do rozwiązania zadania. Zdefiniujmy stałą, którą wykorzystamy przy klasyfikacji grafów,

$$k_0 = \max(k \leqslant \frac{n}{2} : k\cdot(n-k) \leqslant m)$$

(dla uproszczenia zapisu będziemy odtąd zakładać, że $2|n$).

Przypadek 1. Jeżeli $k_0 = \frac{n}{2}$, to $\frac{n^2}{4} = k_0 \cdot (n-k_0) \leqslant m$. To oznacza, że $2m \geqslant \frac{n^2}{2} \geqslant \frac{n(n-1)}{2}$. Ponieważ razem w grafie G oraz w grafie G' jest $\frac{n(n-1)}{2}$ krawędzi, to uzyskana zależność oznacza, że graf G' posiada co najwyżej $\frac{n(n-1)}{2} - m \leqslant 2m - m = m$ krawędzi. W takim przypadku możemy po prostu wyznaczyć graf G', a jego spójne składowe odnaleźć za pomocą jednego ze standardowych algorytmów opisanych na początku opracowania. Stosując przeglądanie DFS lub BFS, można uzyskać złożoność czasową $O(n+m)$.

Przypadek 2. Zastanówmy się, jak wygląda graf G', gdy $k_0 < \frac{n}{2}$. W tym wypadku mamy następującą własność.

Obserwacja 2 Jeżeli $k_0 < \frac{n}{2}$, to w grafie G' musi istnieć spójna składowa o liczności co najmniej $n - k_0$.

Dowód Pokażemy najpierw, że w tym przypadku w grafie G' musi istnieć spójna składowa rozmiaru większego niż $\frac{n}{2}$. Dowód przeprowadzimy przez sprzeczność.

Gdyby wszystkie składowe G' były nie większe niż $\frac{n}{2}$, to z każdego wierzchołka G wychodziłoby co najmniej $\frac{n}{2}$ krawędzi, łączących go z wierzchołkami z pozostałych składowych G'. To oznacza, że graf G miałby co najmniej $n \cdot \frac{n}{2} \cdot \frac{1}{2} = \frac{n^2}{4}$ krawędzi, czyli $\frac{n^2}{4} = \frac{n}{2}\left(n - \frac{n}{2}\right) \leqslant m$. To jest jednak sprzeczne z założeniem, że $k_0 < \frac{n}{2}$.

Wiemy już, że w grafie G' istnieje spójna składowa rozmiaru $l > \frac{n}{2}$. To pozwala nam stwierdzić, że graf G zawiera co najmniej $l \cdot (n-l)$ krawędzi, czyli $m \geqslant l \cdot (n-l)$.

Zauważmy, że wartości $i(n-i)$ wzrastają dla $i = 1,2,\ldots,\frac{n}{2}$. Gdyby więc zachodziło $l < n - k_0$, to z prostego przekształcenia nierówności mielibyśmy $k_0 < n - l \leqslant \frac{n}{2}$ i dalej $k_0 \cdot (n-k_0) < (n-l) \cdot (n-(n-l)) = l(n-l) \leqslant m$. To jest jednak sprzeczne z definicją stałej k_0, więc nierówność $l < n - k_0$ nie może zachodzić. Ostatecznie $l \geqslant n - k_0$ i rzeczywiście w grafie G' istnieje spójna składowa rozmiaru co najmniej $n - k_0$. ■

Pokazaliśmy, że w rozważanym przypadku G' ma dużą składową. Pozostaje jeszcze ją znaleźć. Poniższe spostrzeżenie pozwala nam zidentyfikować sporą grupę wierzchołków tej składowej.

Obserwacja 3 Jeżeli graf G' zawiera spójną składową S o liczności nie mniejszej niż $n - k_0 > \frac{n}{2}$, to należy do niej każdy wierzchołek ze zbioru V, z którego w grafie G wychodzi mniej niż $n - k_0$ krawędzi.

Dowód Z dowolnego wierzchołka, który nie należy do S, wychodzi w G co najmniej $n - k_0$ krawędzi, prowadzących właśnie do wierzchołków składowej S. Skoro tak, to wszystkie wierzchołki G, z których w grafie G wychodzi mniej niż $n - k_0$ krawędzi, muszą należeć do S. ■

Zastanówmy się jeszcze, ile może być w grafie G wierzchołków o stopniu nie mniejszym niż $n - k_0$ (takie wierzchołki mogą, ale nie muszą należeć do S). Oznaczmy przez x liczbę tych wierzchołków. Łączna liczba incydentnych z nimi krawędzi w G to co najmniej $\frac{x(n-k_0)}{2}$, więc także $\frac{x(n-k_0)}{2} \leqslant m$. Gdyby $x \geqslant 2k_0 + 2$, to mielibyśmy zatem $(k_0+1)(n-k_0) \leqslant m$, ale to jest niemożliwe, gdyż z definicji stałej k_0 wynika, że $(k_0+1)(n-k_0) > (k_0+1)(n-(k_0+1)) > m$. Widzimy więc, że $x \leqslant 2k_0 + 1$.

Algorytm. Jesteśmy gotowi do zapisania szkicu algorytmu, opartego na wszystkich wykonanych oszacowaniach:

1. Wyznaczamy stałą $k_0 = \max(k \leqslant \frac{n}{2} : k \cdot (n-k) \leqslant m)$, na przykład przeglądając wszystkie możliwości w złożoności czasowej $O(n)$.
2. Jeżeli $k_0 = \frac{n}{2}$, to do rozwiązania zadania wykorzystujemy klasyczną metodę: konstruujemy graf G' i dzielimy go bezpośrednio na spójne składowe. Złożoność czasowa w tym przypadku to $O(n+m)$.
3. Jeżeli $k_0 < \frac{n}{2}$, to:
   (a) Wszystkie wierzchołki grafu G o stopniu mniejszym niż $n - k_0$ łączymy w jedną spójną składową S grafu G'. Wierzchołki te znajdujemy w czasie $O(n+m)$.
   (b) Dla każdego z pozostałych wierzchołków (jest ich co najwyżej $2k_0 + 1$) znajdujemy listy krawędzi incydentnych z nim w G' — można to zrobić w czasie $O(k_0 n)$ za pomocą algorytmu podobnego do metody znajdowania dopełnienia grafu, opisanej w pierwszym rozdziale.
   (c) Przekształcamy graf G', zastępując wszystkie wierzchołki ze składowej S jednym wierzchołkiem. W tak zmienionym grafie znajdujemy spójne składowe za pomocą klasycznego algorytmu. Złożoność czasowa tego kroku to $O(k_0 n)$, gdyż graf ma co najwyżej $2k_0 + 2$ wierzchołków i co najwyżej $(2k_0+2)n$ krawędzi. Znalezione spójne składowe tego grafu, to (po „rozwinięciu” składowej S) szukane spójne składowe grafu G'.

Jedynym aspektem powyższego algorytmu, jaki może nas martwić, jest występowanie w punktach 3b oraz 3c trudnej do oszacowania złożoności czasowej $O(k_0 n)$. Spróbujmy więc znaleźć górne ograniczenie na wartość wyrażenia $k_0 n$, pamiętając o tym, że $k_0 < \frac{n}{2} < n - k_0$:

$$k_0 n = k_0(k_0 + (n-k_0)) < k_0((n-k_0) + (n-k_0)) = 2 \cdot k_0(n-k_0) \leqslant 2m.$$

Wykonane szacowanie pozwala nam ostatecznie podsumować złożoność zaprezentowanego algorytmu — wynosi ona $O(n+m)$. Implementacje algorytmu znajdują się w plikach `biu2.pas` oraz `biu3.cpp`.

Testy

Rozwiązania zawodników były sprawdzane na zestawie 10 testów. Krawędzie w każdym teście są w losowym porządku.

Nazwa	**n**	**m**	**Opis**
biu1.in	30	306	test poprawnościowy: graf G' składa się z kliki, drzewa wzbogaconego o kilka krawędzi oraz małej liczby losowych krawędzi,
biu2.in	100	3 877	graf G' składa się z dwóch klik i drzewa,
biu3.in	500	96 063	graf G' składa się z klik o rozmiarach będących kolejnymi potęgami 2,
biu4.in	1 000	483 264	graf G' składa się z niewielkich klik oraz losowej części,
biu5.in	5 000	1 962 785	graf G' składa się z kilku klik i drzew oraz losowej części, będącej grafem stosunkowo rzadkim,
biu6.in	10 000	1 944 928	graf G' składa się z 5 klik i 5 drzew oraz losowej części, będącej grafem stosunkowo gęstym,
biu7.in	20 000	1 955 565	graf G' składa się z 5 klik uzupełnionych kilkoma krawędziami oraz losowej części, będącej grafem stosunkowo rzadkim,
biu8.in	40 000	1 832 878	duży test, zawierający jednego pracownika, posiadającego telefony do wszystkich pozostałych (czyli tworzącego jednoosobowe biuro) oraz grupę kilku innych pracowników, którzy mogą zajmować oddzielne biuro,
biu9.in	69 671	1 792 601	graf G' składa się z dwóch niedużych klik z kilkoma dodatkowymi krawędziami oraz losowej części, będącej grafem bardzo rzadkim,
biu10.in	99 328	993 211	graf G' składa się z 3 małych klik oraz losowej, pozostałej części.

Jakub Radoszewski
Treść zadania, Opracowanie

Jakub Radoszewski, Szymon Wąsik
Program

OI, Etap I

Drzewa

Bajtazar ma domek na wsi. Niedawno kupił on n drzew i zlecił swojemu ogrodnikowi posadzenie ich w jednym rzędzie. Gdy ogrodnik posadził drzewa, Bajtazarowi nie spodobała się kolejność, w jakiej zostały one posadzone. Drażni go, że drzewa niskie i wysokie zostały pomieszane ze sobą i całość nie wygląda zbyt estetycznie.

Bajtazar, chcąc wyjaśnić ogrodnikowi, jaki jest jego cel, zdefiniował **współczynnik nieporządku** *rzędu drzew jako: $|h_1 - h_2| + |h_2 - h_3| + \cdots + |h_{n-1} - h_n|$, gdzie $h_1, h_2, \ldots, h_n$ to wysokości kolejnych drzew w rzędzie. Im mniejsza wartość współczynnika nieporządku, tym ładniej wygląda rząd drzew.*

Przesadzanie drzew jest bardzo pracochłonne i kłopotliwe. Dlatego też Bajtazar zlecił ogrodnikowi przesadzenie co najwyżej dwóch drzew (tak, że zostaną one zamienione miejscami). Ogrodnik ma tak wybrać drzewa do przesadzenia, aby współczynnik nieporządku rzędu drzew stał się jak najmniejszy.

Ogrodnik nie jest pewien, czy właściwie wybrał drzewa do przesadzenia, a boi się, że w razie pomyłki straci pracę. Pomóż mu i oblicz, dla każdego drzewa, jaki najmniejszy współczynnik nieporządku może zostać uzyskany poprzez ewentualną zamianę jego pozycji z innym drzewem.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia wysokości drzew w rzędzie,*
- *dla każdego drzewa obliczy najmniejszy współczynnik nieporządku, jaki może być uzyskany, jeżeli rozpatrywane drzewo zostanie zamienione pozycją z pewnym innym lub żadne drzewo nie będzie przesadzane,*
- *wypisze wynik na standardowe wyjście.*

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę całkowitą n ($2 \leqslant n \leqslant 50\,000$). Drugi wiersz wejścia zawiera n liczb całkowitych h_i ($1 \leqslant h_i \leqslant 100\,000\,000$), pooddzielanych pojedynczymi odstępami i oznaczających wysokości kolejnych drzew w rzędzie.

Wyjście

Wyjście powinno zawierać dokładnie n wierszy. Wiersz i-ty powinien zawierać dokładnie jedną liczbę całkowitą — najmniejszy współczynnik nieporządku możliwy do uzyskania przez przesadzenie i-tego drzewa.

Przykład

Dla danych wejściowych:

```
5
7 4 5 2 5
```

poprawnym wynikiem jest:

```
7
7
8
7
7
```

Natomiast dla danych:

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

poprawnym wynikiem jest:

```
4
4
4
4
4
```

W pierwszym przykładzie współczynnik nieporządku równy 7 może być uzyskany poprzez przesadzenie drzew numer 1 i 4, 2 i 5 lub 4 i 5. Tak więc przesadzając każde z wymienionych w poprzednim zdaniu drzew (1, 2, 4 i 5) z odpowiednio wybranym innym drzewem, można uzyskać współczynnik nieporządku 7. Jedynie dla drzewa 3 najlepszy możliwy do uzyskania współczynnik nieporządku jest większy i wynosi 8. W drugim przykładzie przesadzenie dowolnych dwóch drzew może jedynie powiększyć współczynnik nieporządku, więc żadna zamiana nie powinna mieć miejsca i wszystkie współczynniki nieporządku równają się początkowemu współczynnikowi (4).

Rozwiązanie

Najprostsze rozwiązanie — symulacja

Najprostszy sposób rozwiązania zadania to zasymulowanie wszystkich możliwych przesadzeń i obliczenie, dla każdego z nich, współczynnika nieporządku powstałego szpaleru (czyli rzędu) drzew. Ponieważ musimy wówczas rozważyć $n(n-1)$ przesadzanych par i dla każdej z ich wyliczyć w czasie $O(n)$ współczynnik nieporządku, więc złożoność czasowa tego algorytmu wynosi $O(n^3)$. Jego implementacja znajduje się w plikach `drzs0.cpp` i `drzs1.pas`.

Rozwiązanie można łatwo usprawnić, korzystając z faktu, że przesadzenie dwóch drzew ma dość „lokalny” wpływ na współczynnik nieporządku. Jeśli wstępnie policzymy współczynnik nieporządku dla początkowego szpaleru, to po przesadzeniu wybranej pary drzew (powiedzmy zajmujących pozycje i oraz j dla $1 < i < i+1 < j < n$), wystarczy skorygować współczynnik o wartość:

$$\begin{aligned}
& \big(|h_1-h_2|+|h_2-h_3|+\cdots+|h_{n-1}-h_n|\big)+\\
- & \big(|h_1-h_2|+\cdots+|h_{i-1}-h_j|+|h_j-h_{i+1}|+\cdots+\\
+ & |h_{j-1}-h_i|+|h_i-h_{j+1}|+\cdots+|h_{n-1}-h_n|\big)=\\
= & \big(|h_{j-1}-h_i|+|h_i-h_{j+1}|+|h_{i-1}-h_j|+|h_j-h_{i+1}|\big)+\\
- & \big(|h_{j-1}-h_j|+|h_j-h_{j+1}|+|h_{i-1}-h_i|+|h_i-h_{i+1}|\big)
\end{aligned}$$

Jak widać, możemy to zrobić w czasie stałym. Dla przypadków, gdy $i = 1$ lub $i+1 = j$, lub $j = n$ (czyli gdy przesadzamy drzewa skrajne lub sąsiadujące ze sobą), wzory są trochę inne, ale nadal bardzo proste do wyznaczenia.

Złożoność czasowa usprawnionej wersji rozwiązania to $O(n^2)$. Jej implementacja znajduje się w plikach `drzs2.cpp` oraz `drzs3.pas`.

Oba zaprezentowane rozwiązania są stosunkowo proste koncepcyjne, a ich zaprogramowanie nie stwarza zbytnich problemów. Większość zawodników nadesłała tego typu algorytmy, zdobywając za rozwiązania o złożoności $O(n^3)$ około 30 punktów na 100 możliwych, a za rozwiązania o złożoności $O(n^2)$ — od 50 do 70 punktów. Dodatkowe sposoby usprawnienia rozwiązania o złożoności $O(n^2)$ zostały przedstawione — w charakterze ciekawostki — na końcu niniejszego opracowania.

Rozwiązanie wzorcowe

Wprowadzenie

Rozwiązanie wzorcowe zadania *Drzewa* osiąga złożoność czasową zdecydowanie niższą niż rozwiązania symulacyjne — $O(n \log n)$. Jest oparte na całkowicie odmiennym podejściu, którego zrozumienie wymagać będzie od Czytelnika pewnej dozy cierpliwości. Jednak naszym zdaniem zdecydowanie warto się z nim zmierzyć, bo zaprezentowane pomysły są ciekawe i niebanalne. Implementacja tego rozwiązania także jest stosunkowo trudna i żmudna, przede wszystkim z powodu konieczności rozpatrywania wielu szczególnych przypadków. Po tym „zniechęcającym” wstępie, naprawdę szczerze zapraszamy Czytelników do dalszej lektury.

Drzewo *minTree*

W prezentowanym rozwiązaniu będziemy wykorzystywać strukturę danych, w której można przechowywać pary (*klucz*, *wartość*), gdzie *klucz*$\in [1,n]$. Na strukturze tej będziemy wykonywać operacje:

- *insert(k,w)* — wstawianie elementu o kluczu k i wartości w;
- *delete(k)* — usunięcie elementu o kluczu k,
- *min_val(l,p)* — znalezienie minimalnej wartości towarzyszącej elementom o kluczach z przedziału $[l,p] \subseteq [1,n]$.

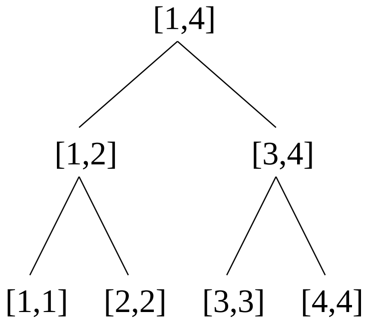
Dodatkowo wiemy, że klucze elementów zapisanych w strukturze nie powtarzają się, a dla prostoty implementacji można założyć, że n jest potęgą dwójki.

Strukturę danych *minTree* można efektywnie zrealizować za pomocą statycznego drzewa poszukiwań binarnych, tzw. *drzewa przedziałowego*. Jest to drzewo binarne, którego wierzchołki są związane z przedziałami zawartymi w $[1,n]$ w następujący sposób:

- korzeń drzewa reprezentuje przedział kluczy $[1,n]$;
- każdy węzeł wewnętrzny v drzewa, który reprezentuje niejednostkowy przedział $[l,r]$ (tzn. $r > l$), ma dwóch synów, którzy reprezentują odpowiednio przedziały kluczy: $[l, \lfloor \frac{l+r}{2} \rfloor]$ oraz $[\lfloor \frac{l+r}{2} \rfloor + 1, r]$;

- drzewo ma n liści, którym odpowiadają przedziały jednostkowe $[1,1],[2,2],\ldots,[n,n]$.

Przedziały reprezentowane przez węzły drzewa nazywamy *przedziałami bazowymi*. Przykład drzewa przedziałowego dla przedziału $[1,4]$ jest pokazany na rys. 1.



Rys. 1: Drzewo przedziałowe, skonstruowane dla przedziału kluczy $[1,4]$.

Dokładniejszy opis drzew przedziałowych można znaleźć na przykład w *Niebieskiej książeczce* z XIII Olimpiady Informatycznej ([13]) w opracowaniu zadania *Tetris 3D*. Tutaj ograniczymy się do przypomnienia ich najważniejszych z naszego punktu widzenia właściwości:

- głębokość drzewa (czyli długość najdłuższej ścieżki od korzenia do liścia) jest rzędu $O(\log n)$,
- liczba węzłów drzewa jest rzędu $O(n)$,
- każdy przedział $[l,r]$ zawarty w przedziale $[1,n]$ można rozłożyć na $O(\log n)$ parami rozłącznych przedziałów bazowych (tzn. każda liczba całkowita w nim zawarta znajdzie się w dokładnie jednym przedziale z rozkładu); przedziały i reprezentujące je wierzchołki można znaleźć w czasie $O(\log n)$.

Za pomocą drzewa przedziałowego możemy zaimplementować efektywnie *minTree*. W tym celu w każdym wierzchołku umieścimy dodatkowe pole *val* przeznaczone na wartość. Zadbamy o to, by w węźle reprezentującym przedział $[l,r]$ w polu *val* znajdowało się minimum wartości związanych z kluczami z przedziału $[l,r]$. W polu *val* liścia reprezentującego przedział $[i,i]$ zapiszemy więc wartość elementu o kluczu i. Jeśli takiego elementu nie ma w drzewie, to w liściu $[i,i]$ możemy wpisać wartość ∞. W polu *val* węzła v reprezentującego przedział $[l,r]$ zapiszemy minimum z wartości przypisanych elementom o kluczach z przedziału $[l,r]$. Zauważmy, że tę wartość możemy wyznaczyć jako minimum z pól *val* synów węzła v.

Operacje na *minTree* wykonamy w następujący sposób:

- Puste drzewo tworzymy, budując kompletną strukturę przedziałów i wypełniając wszystkie pola *val* wartościami ∞. Całą konstrukcję możemy zbudować w czasie $O(n)$. Warto także nadmienić, że drzewo, dzięki regularnej strukturze, można zaimplementować bez użycia wskaźników. Drzewo to „wkładamy” w tablicę $A[1..2n-1]$ poziomami: wówczas korzeń znajduje się w polu o indeksie 1, a synowie wierzchołka zapisanego w polu o indeksie i w polach o indeksach $2i$ oraz $2i+1$.
- Operację $insert(k,w)$ wykonujemy, przechodząc ścieżką od korzenia do liścia $[k,k]$ i zmieniając w nim wartość *val* na w. Wracając od liścia $[k,k]$ do korzenia, aktualizujemy pola *val* we wszystkich węzłach napotkanych po drodze — zauważmy, że są to wszystkie węzły drzewa, reprezentujące przedziały zawierające przedział $[k,k]$. Operację wykonujemy w czasie $O(\log n)$, bo taka jest wysokość drzewa.
- Operację $delete(k)$ można zrealizować, wykonując $insert(k,\infty)$. Koszt czasowy tej operacji to także $O(\log n)$.
- Znalezienie $min\_val(l,r)$ wymaga rozłożenia $[l,r]$ na przedziały bazowe, a następnie wyznaczenia minimum z wartości pól *val* dla węzłów drzewa reprezentujących te przedziały. Zarówno rozkład, jak i minimum, można znaleźć w czasie $O(\log n)$.

Zarys rozwiązania wzorcowego

Rozpoczniemy od wprowadzenia terminologii, która ułatwi nam opis rozwiązania.

W rozwiązaniu będziemy wykorzystywać drzewiaste struktury danych (*minTree*). Aby Czytelnik w każdym momencie opracowania wiedział czy słowo „drzewo” oznacza roślinę z treści zadania, czy strukturę danych, umówmy się, że wszystkie drzewa w ogrodzie Bajtazara to *cyprysy*.

Miejsca w ogrodzie Bajtazara są wstępnie ponumerowane kolejno liczbami $1,2,\ldots,n$, więc i-tym miejscem będziemy nazywać i-te miejsce w szpalerze. Wprowadzimy dodatkowe uporządkowanie. *Rangą* cyprysa nazwiemy jego pozycję w ciągu cyprysów uporządkowanych według wysokości, tzn. rangę 1 przypiszemy najniższemu cyprysowi, rangę 2 — drugiemu w kolejności itd.; najwyższy cyprys dostanie rangę n. Jednakowe cyprysy możemy uporządkować dowolnie — po przyjęciu tego porządku będziemy uważać, że wszystkie cyprysy mają różne wysokości i będziemy rozstrzygać remisy zgodnie z przyjętym porządkiem rang. Przez H_i oznaczymy rangę cyprysa rosnącego na i-tym miejscu.

Domyślnie miejsca będziemy oznaczać według numerów, a cyprysy według rang, tzn. mówiąc i-te miejsce, będziemy mieli na myśli i-te miejsce w szpalerze, a mówiąc i-ty cyprys — cyprys i-ty co do wielkości. Dodatkowo, przez D_i oznaczymy miejsce zajmowane przez i-ty cyprys. To oznacza, że najniższy cyprys stoi w szpalerze na miejscu D_1, drugi — na miejscu D_2 itd.; najwyższy stoi na miejscu D_n.

Przykład 1 Rozważmy pierwszy z przykładów w treści zadania, w którym $n = 5$, a wysokości kolejnych cyprysów w szpalerze to: $7,4,5,2,5$. Rangi H przypisane kolejnym cyprysom w szpalerze to: $5,2,4,1,3$ (przy innym rozstrzygnięciu remisu możliwe jest także przypisanie rang: $5,2,3,1,4$). Ciąg D to: $4,2,5,3,1$ (a w drugim przypadku $4,2,3,5,1$).

Cyprysy będziemy przeglądać w kolejności rosnących wysokości, czyli według rang $p = 1,2,\ldots,n$. To znaczy, że będziemy rozważać kolejno cyprysy stojące na pozycjach: $t = D_1, D_2, \ldots, D_n$. Dla każdego z nich wyznaczymy optymalną zmianę współczynnika nieporządku, jaką można uzyskać, przesadzając go z dowolnym innym (ewentualnie nic nie zmieniając w szpalerze). Przyjrzyjmy się więc teraz p-temu cyprysowi rosnącemu na t-tym miejscu i możliwościom, jakie stwarza jego przesadzenie.

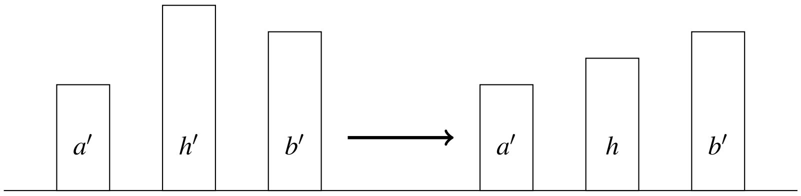
Zamianę cyprysa rosnącego na miejscu t-tym z innym, powiedzmy rosnącym na miejscu t'-tym, podzielimy na dwie operacje:

Faza 1: dokonamy zmian na miejscu t', to znaczy wyrwiemy cyprys z miejsca t' i na jego miejsce posadzimy p-ty cyprys,

Faza 2: dokonamy zmian na miejscu t, to znaczy wyrwiemy cyprys z miejsca t i w powstałej wyrwie posadzimy cyprys z miejsca t'.

Oznaczmy przez a, h oraz b wysokości cyprysów rosnących (początkowo) na miejscach $t-1$, t oraz $t+1$ (możemy bez straty ogólności założyć, że $a \leqslant b$, w przeciwnym razie zamienimy oznaczenia, przyjmując $a = h_{t+1}$ oraz $b = h_{t-1}$). Analogicznie oznaczymy przez a', h' oraz b' wysokości cyprysów z miejsc $t'-1$, t' oraz $t'+1$ (także zakładamy, że $a' \leqslant b'$).

Faza 1



Rys. 2: Efekt posadzenia cyprysa z miejsca t w miejscu t'.

Zmiany zachodzące w miejscu t' to wyrwanie cyprysa rosnącego w tym miejscu (powoduje to zmniejszenie współczynnika nieporządku o $|a'-h'|+|b'-h'|$) i posadzenie w powstałej wyrwie cyprysa o wysokości h (powoduje to wzrost współczynnika nieporządku o $|a'-h|+|b'-h|$). Zmianę współczynnika możemy zapisać jako:

1. $a'+b'-2h-|a'-h'|-|b'-h'| = -2h+(a'+b'-|a'-h'|-|b'-h'|)$, jeżeli $h \leqslant a'$,
2. $b'-a'-|a'-h'|-|b'-h'| = 0+(b'-a'-|a'-h'|-|b'-h'|)$, jeżeli $a' \leqslant h \leqslant b'$,
3. $2h-a'-b'-|a'-h'|-|b'-h'| = 2h+(-a'-b'-|a'-h'|-|b'-h'|)$, jeżeli $h \geqslant b'$.

Zauważmy, że każde z wyrażeń podzieliliśmy na dwie części: *składową cyprysa* (równą $-2h$, 0 lub $2h$) oraz *składową miejsca* (oznaczymy ją w kolejnych przypadkach $f_1(t')$, $f_2(t')$ oraz $f_3(t')$). Otrzymana postać wyrażenia skłania nas do następujących spostrzeżeń:

- z punktu widzenia cyprysa o wysokości h, pozycje t', na których możemy go posadzić, dzielą się na trzy grupy:
  - *doliny*, czyli miejsca, gdzie trafi pomiędzy dwa wyższe cyprysy, tzn. $h \leqslant a'$,
  - *zbocza*, gdzie będzie sąsiadował z wyższym i z niższym cyprysem, czyli $a' \leqslant h \leqslant b'$ oraz
  - *szczyty*, gdzie po obu stronach będzie miał niższe cyprysy, czyli $b' \leqslant h$;

  jeśli jakaś pozycja kwalifikuje się do więcej niż jednej grupy (z powodu równości), to możemy ją przydzielić dowolnie, ale tylko do jednej grupy;

- dla dolin powinniśmy policzyć wartości $f_1(t') - 2h$, dla zboczy — wartości $f_2(t')$, dla szczytów — wartości $f_3(t') + 2h$;
- optymalna pozycja docelowa t' dla rozważanego cyprysa o wysokości h to taka, dla której wartość z poprzedniego punktu jest minimalna (na razie nie martwimy się faktem, że trzeba będzie jeszcze posadzić cyprys wyrwany z pozycji t' — o tym później).

Pozostaje nam zrealizować zakreślony plan. Do przechowywania pozycji wraz ze składowymi miejsca wykorzystamy trzy drzewa *minTree*:

A — będziemy w nim przechowywać pary $(H_{t'}, f_1(t'))$, gdzie t' jest doliną dla rozważanego cyprysa (a $H_{t'}$, przypomnijmy, oznacza rangę cyprysa zajmującego miejsce t');

B — w tym drzewie będziemy przechowywać pary $(H_{t'}, f_2(t'))$, gdzie t' jest zboczem dla rozważanego cyprysa;

C — to drzewo przeznaczymy na pary $(H_{t'}, f_3(t'))$, gdzie t' jest szczytem dla rozważanego cyprysa.

Dzięki zastosowaniu struktur *minTree* będziemy w stanie znaleźć optymalną pozycję dla rozważanego cyprysa, wykonując operację *min_val*$(1,n)$ dla każdego z drzew. Pozostaje tylko wyjaśnić, jak podzielić pozycje na doliny, zbocza i szczyty dla pierwszego rozważanego cyprysa i jak aktualizować ten podział przy przechodzeniu do kolejnych rangą cyprysów.

Teraz wyjaśni się, dlaczego wybraliśmy rozważanie cyprysów według rang. Otóż z punktu widzenia pierwszego, najniższego cyprysa *wszystkie pozycje są dolinami*, więc wszystkie powinny trafić do drzewa A. Następnie, załóżmy, że w trakcie rozważania p-tego cyprysa zajmującego miejsce t mamy pozycje poprawnie rozdzielone pomiędzy drzewa A, B i C. Wówczas prawie wszystkie pozycje są poprawnie sklasyfikowane także z punktu widzenia $(p+1)$-ego cyprysa. Jeśli bowiem na pozycjach sąsiednich względem rozważanej pozycji t' stoją cyprysy o rangach z przedziałów $[1, p-1] \cup [p+1, n]$, to relacja ich wysokości jest taka sama w stosunku do p-tego oraz $(p+1)$-szego cyprysa. Klasyfikacja może ulec zmianie jedynie dla pozycji $t' = t-1$ oraz $t' = t+1$, które sąsiadują z p-tym cyprysem. Jeśli któraś z tych pozycji była doliną dla p-tego cyprysa i p-ty cyprys był niższym z jej sąsiadów, to dla cyprysa $(p+1)$-szego pozycja ta jest zboczem (i musi zostać przeniesiona z drzewa A do drzewa B). Jeśli natomiast pozycja ta była zboczem dla p-tego cyprysa i p-ty cyprys był wyższym z sąsiadów, to dla cyprysa $(p+1)$-szego pozycja ta jest szczytem (i musi zostać przeniesiona z drzewa B do drzewa C). Oczywiście, przenosząc pozycję do nowego drzewa, umieszczamy ją tam z wartością f_i stosowaną w tym drzewie.

Złożoność. Każdą aktualizację stanu drzew: A, B i C potrafimy wykonać w czasie $O(\log n)$. W takim samym czasie znajdujemy również minimum z wartości w tych drzewach. Widzimy więc, że fazę pierwszą dla dowolnego cyprysa potrafimy wykonać w czasie $O(\log n)$, a dla wszystkich — w czasie $O(n \log n)$.

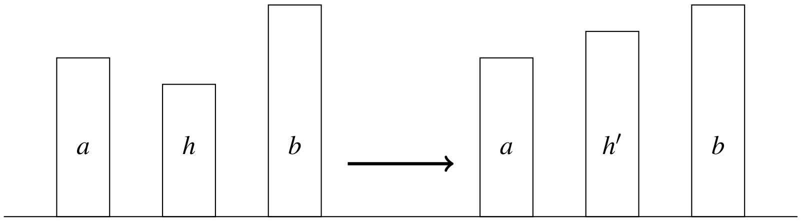
Przypadki szczególne. Przedstawiony opis rozwiązania dla fazy 1 jest dość przejrzysty i elegancki. Jednak zapewne nawet wyjątkowo tolerancyjny Czytelnik już niejednokrotnie odczuł niepokój związany z dość niefrasobliwym traktowaniem przypadków szczególnych — dotychczas właściwie je ignorowaliśmy. W szczególności większość z podanych wzorów nie jest poprawna, jeżeli mamy do czynienia z pozycjami skrajnymi w szpalerze, tzn. $t, t' \in \{1, n\}$, lub sąsiednimi, tzn. $|t - t'| = 1$. Próba zmodyfikowania algorytmu tak, by uwzględnić wszystkie sytuacje, doprowadziłaby do nadmiernego skomplikowania całości i pogrzebania głównych idei w powodzi szczegółów. Proponujemy więc *wyeliminowanie* z powyższego rozwiązania tych przypadków i *rozważenie ich osobno* kosztem niewielkiego, dodatkowego nakładu czasu.

Eliminacja. Przede wszystkim w całym rozwiązaniu nie będziemy brać pod uwagę skrajnych pozycji 1 i n. W ten sposób będziemy mieć gwarancje, że każda rozważana pozycja jest otoczona dwiema sąsiednimi. Kwestię pozycji sąsiednich rozwiążemy, usuwając z puli pozycji (zapisanych w drzewach A, B i C) pozycje sąsiednie z zajmowaną przez aktualnie rozważany cyprys, czyli pozycje $t-1$ oraz $t+1$. Jest to o tyle łatwiejsze, że są to jedyne pozycje, które po zakończeniu rozważania p-tego cyprysa (z miejsca t) przenosimy między drzewami *minTree*. W takim razie nikt nam nie zabrania usunąć ich z drzew *minTree* tuż *przed* rozważaniem p-tego cyprysa i wstawić do odpowiednich drzew *minTree* już *po* jego rozważeniu.

Osobne rozpatrzenie. Ile jest par różnych pozycji takich, z których co najmniej jedna jest skrajna albo są to pozycje sąsiednie? Nie wdając się w dokładne rachunki, możemy śmiało powiedzieć, że jest ich co najwyżej $3n = O(n)$. Ponieważ wpływ zamiany cyprysów z dowolnej takiej pary pozycji na współczynnik nieporządku można wyznaczyć w stałej złożoności czasowej (patrz opis rozwiązania o złożoności czasowej $O(n^2)$), to osobne rozpatrzenie wszystkich przypadków szczególnych może zostać wykonane w dodatkowym czasie liniowym względem n. Taki narzut nie ma więc żadnego wpływu na całkowitą złożoność obliczeniową niniejszego rozwiązania.

Podsumowanie fazy 1. Na zakończenie opisu rozwiązania fazy 1 zauważmy, że zastosowanie drzew *minTree* wydaje się być przesadą — wszak do wykonania wszystkich potrzebnych operacji wystarczyłaby zwykła kolejka priorytetowa! Co więcej, nie widać jeszcze żadnego powodu, by jako klucze elementów odpowiadających pozycjom wybierać wartości H_t, a nie na przykład t. Chcielibyśmy zapewnić Czytelnika, że to wszystko jest celowe i ma ułatwić integrowanie realizacji faz w dalszej części, o czym można się przekonać, kontynuując lekturę.

Faza 2



Rys. 3: Efekt wstawienia cyprysa z miejsca t' w miejscu t.

Przypomnijmy, że nadal rozważamy kwestię znalezienia optymalnego miejsca dla p-tego (według rangi) cyprysa, rosnącego na miejscu t. W fazie 1 rozważyliśmy już koszt wynikający ze zmian przeprowadzonych na jego nowym miejscu. Teraz uwzględnimy zmiany zachodzące na miejscu t. Współczynnik nieporządku zmniejszamy więc o $|a-h|+|h-b|$ (co odpowiada wyrwaniu cyprysa z miejsca t), a następnie powiększamy o $|a-h'|+|h'-b|$ (co odpowiada posadzeniu cyprysa z miejsca t' w powstałej wyrwie). Jak poprzednio, zmianę współczynnika nieporządku możemy podzielić na składowe: miejsca (tym razem t) oraz cyprysa (tym razem wziętego z miejsca t'):

1. $a+b-2h'-|a-h|-|h-b| = -2h'+(a+b-|a-h|-|b-h|)$, jeżeli $h' \leqslant a$,
2. $b-a-|a-h|-|h-b| = 0+(b-a-|a-h|-|b-h|)$, jeżeli $a \leqslant h' \leqslant b$,
3. $2h'-a-b-|a-h|-|h-b| = 2h'+(-a-b-|a-h|-|b-h|)$, jeżeli $h' \geqslant b$.

Oznaczmy składową zależną od miejsca, jak poprzednio, $f_1(t)$, $f_2(t)$ lub $f_3(t)$, a składową zależną od cyprysa odpowiednio $g_1(t')$, $g_2(t')$ i $g_3(t')$. Zauważmy także, że tym razem wybór formuły, którą mamy zastosować, zależy od tego, czy dla sadzonego w miejscu t cyprysa miejsce to jest doliną, zboczem czy szczytem. Podział cyprysów ze względu na to kryterium jest bardzo prosty. Oznaczmy przez p_a rangę cyprysa o wysokości a (z miejsca $t-1$ lub $t+1$), a przez p_b rangę cyprysa o wysokości b (z drugiego miejsca sąsiadującego z miejscem t). Miejsce t jest:

- doliną dla cyprysów o rangach z przedziału $[1, p_a]$,
- zboczem dla cyprysów o rangach z przedziału $[p_a, p_b]$ oraz
- szczytem dla cyprysów o rangach z przedziału $[p_b, n]$.

Aby sprawnie zrealizować opisane wyżej wyznaczanie zmiany współczynnika nieporządku, tym razem, podobnie jak w fazie 1, także zastosujemy trzy *minTree*, które dla odmiany oznaczymy T_1, T_2 oraz T_3. W drzewie T_i będziemy przechowywać *wszystkie* pozycje t' w postaci par $(H_{t'}, g_i(t'))$, gdzie, przypomnijmy, $H_{t'}$ jest rangą cyprysa z miejsca t'. W ten sposób wyboru optymalnego cyprysa zastępującego p-ty cyprys na miejscu t możemy dokonać, wyznaczając minimalną spośród wartości: $f_1(a,b,h)+min\_val_{T1}(1,p_a)$, $f_2(a,b,h)+min\_val_{T2}(p_a,p_b)$ oraz $f_3(a,b,h)+min\_val_{T3}(p_b,n)$. Co równie istotne, drzewa T_1, T_2 i T_3 są takie same dla kolejnych rozważanych cyprysów i nie trzeba ich przebudowywać przy przejściu od p-tego cyprysa do $(p+1)$-szego.

Przypadki szczególne. Jak poprzednio, przy realizacji fazy 1, chcielibyśmy usunąć przypadki szczególne z zasadniczego algorytmu i rozważyć je oddzielnie. Bez problemu możemy pominąć rozważanie cyprysów pochodzących ze skrajnych pozycji $t' \in \{1, n\}$. Większym problemem jest uniknięcie rozważania cyprysów z pozycji sąsiednich względem t, czyli $t' \in \{t-1, t+1\}$. Aby to zrealizować, przed przystąpieniem do rozważania cyprysa z pozycji t usuniemy ze struktur T_1, T_2 i T_3 cyprysy zajmujące newralgiczne pozycje. Wstawimy je z powrotem po wykonaniu odpowiednich zapytań *min_val*. Jak widać, nie skomplikuje to nam zbytnio całości obliczeń. Oczywiście, po wyeliminowaniu przypadków szczególnych, rozważymy je odrębnie i uwzględnimy otrzymane wyniki przy poszukiwaniu optymalnego cyprysa, zastępującego cyprys z miejsca t.

Złożoność. Wstępna budowa wszystkich drzew *minTree* wymaga czasu $O(n)$. Znalezienie wartości *min_val* dla każdego rozważanego cyprysa możemy wykonać w czasie $O(\log n)$. Eliminacja przypadków szczególnych w jednej rundzie wymaga czasu $O(\log n)$, a ich uwzględnienie — łącznego czasu $O(n)$. Całkowity czas realizacji fazy 2 wynosi więc $O(n \log n)$.

Podsumowanie fazy 2. Podobnie jak poprzednio, tym razem także nie wykorzystaliśmy wszystkich właściwości drzew przedziałowych, a potrzeba użycia drzewa T_2 — zdegenerowanej struktury, w które przechowujemy zera dla wszystkich wartości — jest całkiem wątpliwa. Pozostaje mieć nadzieję, że wszystko znajdzie swoje zastosowanie w chwili połączenia fazy pierwszej i drugiej, bo nietrudno zauważyć, że fazy te, traktowane oddzielnie, nie dają nam sensownego rozwiązania problemu.

Połączenie faz 1 i 2

Przyszła wreszcie pora na połączenie rozwiązań dla faz 1 i 2. Aby je przeprowadzić, nakreślmy plan działania całego algorytmu.

Przeglądamy cyprysy w kolejności rosnących rang $p = 1, 2, \ldots, n$ (p-ty cyprys rośnie na miejscu $t = D_p$).

- Dla p-tego cyprysa rozważamy wszystkie możliwości zamiany go z innym cyprysem, rosnącym na miejscu t'. Wpływ takiej zamiany jest zależny od:
  - sytuacji, w jakiej znajdzie się cyprys p-ty w miejscu t' — czy będzie to dla niego dolina, zbocze czy szczyt; oraz
  - sytuacji, w jakiej znajdzie się cyprys z miejsca t' przesadzony w miejsce t — czy będzie to dla niego dolina, zbocze czy szczyt.

  Kombinacja powyższych warunków tworzy nam dziewięć sytuacji — w każdej z nich obowiązuje inna formuła wyznaczania zmiany współczynnika nieporządku.

  Dzielimy więc wszystkie potencjalne pozycje t' dla p-tego cyprysa na dziewięć kategorii, w każdej z nich utrzymujemy właściwą wartość zmiany współczynnika nieporządku.
- Poszukując optymalnej pozycji do zamiany dla p-tego cyprysa, wybieramy optymalną pozycję z dziewięciu optymalnych pozycji znalezionych odrębnie w każdej kategorii.
- Dodatkowo dla cyprysa p-tego rozważamy przypadki szczególne i sprawdzamy, czy nie są lepsze od znalezionych dotychczas.

Najbardziej skomplikowany moment w realizacji przedstawionego planu to podział potencjalnych pozycji t' na dziewięć kategorii (z właściwą wartością funkcji zmiany współczynnika nieporządku) i utrzymywanie poprawności tego podziału przy rozważaniu kolejnych cyprysów według rangi p. Aby go zrealizować, posłużymy się *dziewięcioma* (!) drzewami *minTree*: X_i, gdzie $X \in \{A, B, C\}$ oraz $i \in \{1, 2, 3\}$. Drzewo X_i będzie zawierało dane dotyczące pozycji t' w postaci pary (r, w), gdzie:

- r jest rangą drzewa rosnącego na pozycji t',
- w jest sumą wartości zapisanych dla pozycji t' w drzewach X oraz T_i, czyli wartością zmiany współczynnika nieporządku odpowiednią dla sytuacji, gdy pozycja t' jest dla p-tego cyprysa odpowiednio doliną ($X = A$), zboczem ($X = B$) lub szczytem ($X = C$), oraz gdy cyprys sadzony w miejsce cyprysa p-tego trafia do doliny ($i = 1$), na zbocze ($i = 2$) lub na szczyt ($i = 3$).

Aby sobie to lepiej wyobrazić, przyjrzyjmy się dokładniej kilku przykładowym drzewom *minTree*. W chwili rozważania cyprysa p-tego z pozycji $t = D_p$ o wysokości $h = h_t$:

- A_1 zawiera pozycje t', dla których zachodzi $h \leqslant \min(h_{t'-1}, h_{t'+1})$, a wartość przechowywana w nim jest równa sumie wartości z *minTree* A oraz T_1, czyli $f_1(t') + g_1(t')$; z tego drzewa wybieramy pozycje z przedziału $[1, p_a]$, gdzie $p_a = \min(H_{t-1}, H_{t+1})$, by ostatecznie otrzymać wszystkie pozycje, które są *dolinami* dla cyprysa z pozycji t, zajętymi przez cyprysy, które przesadzone w miejsce t także trafią *w dolinę*;
- A_2 zawiera dokładnie te same pozycje t', co A_1, czyli doliny dla p-tego cyprysa; wartości przechowywane dla pozycji są postaci $f_1(t') + g_2(t')$, co odpowiada sumie wartości z A oraz T_2; rozważając w tym drzewie tylko przedział $[p_a, p_b]$, gdzie $p_a = \min(H_{t-1}, H_{t+1})$ i $p_b = \max(H_{t-1}, H_{t+1})$, mamy wszystkie pozycje, które są *dolinami* dla cyprysa z pozycji t, zajętymi przez cyprysy, które przesadzone w miejsce t trafią *na zbocze*.
- B_3 zawiera pozycje, które są *zboczami* dla p-tego cyprysa, czyli $\min(h_{t'-1}, h_{t'+1}) \leqslant h \leqslant \max(h_{t'-1}, h_{t'+1})$, a wartości w nich zapisane są równe $f_2(t') + g_3(t')$; rozważając w tym drzewie tylko przedział $[p_b, n]$, gdzie $p_b = \max(H_{t-1}, H_{t+1})$, mamy wszystkie pozycje, które są *zboczami* dla cyprysa z pozycji t, zajętymi przez cyprysy, które przesadzone w miejsce t trafią *na szczyt*.

Posiadając opisane wyżej drzewa (nad sposobem ich konstrukcji i aktualizacji zastanowimy się za chwilę), możemy łatwo wyznaczyć optymalną zmianę współczynnika nieporządku wynikającą z zamiany p-tego cyprysa z dowolnym innym. Musimy tylko wyznaczyć pozycję t', dla której wypada minimum z wartości:

- $f_1(t) - 2h + min\_val_{A1}(1, p_a)$,
- $f_2(t) - 2h + min\_val_{A2}(p_a, p_b)$,
- $f_3(t) - 2h + min\_val_{A3}(p_b, n)$,
- $f_1(t) + 0 + min\_val_{B1}(1, p_a)$,
- $f_2(t) + 0 + min\_val_{B2}, (p_a, p_b)$,
- $f_3(t) + 0 + min\_val_{B3}(p_b, n)$,
- $f_1(t) + 2h + min\_val_{C1}(1, p_a)$,
- $f_2(t) + 2h + min\_val_{C2}(p_a, p_b)$,
- $f_3(t) + 2h + min\_val_{C3}(p_b, n)$.

Mimo stosunkowo dużej liczby otrzymanych przypadków (a zatem także niemałej stałej multiplikatywnej) mamy więc logarytmiczną względem n złożoność czasową pojedynczego zapytania.

Powróćmy do problemu stworzenia i aktualizacji wszystkich *minTree*. Na samym początku wszystkie pozycje t' przechowujemy w trzech kopiach, po jednej w A_1, A_2 oraz A_3. Jest to poprawne dzięki obranej przez nas kolejności rozpatrywania cyprysów według rang. Następnie dla kolejnych cyprysów o rangach $p = 1, 2, \ldots, n$ (zajmujących odpowiednio pozycję $t = D_p$):

- usuwamy pozycje $t - 1$ oraz $t + 1$ z tych drzew *minTree*, w których się one aktualnie znajdują,
- wyznaczamy dziewięć wartości *min_val*, z których wyznaczamy optymalną,
- wstawiamy pozycje $t - 1$ oraz $t + 1$ do odpowiednich drzew *minTree* (zgodnie z zasadami opisanymi przy fazie 1).

Realizacja jednego kroku aktualizacji wymaga stałej liczby operacji na drzewach *minTree*, co pokazuje, że łączny koszt czasowy utrzymywania wszystkich *minTree* w trakcie działania algorytmu to $O(n \log n)$.

Zauważmy wreszcie, że na tyle dokładnie zajęliśmy się przypadkami szczególnymi w poszczególnych fazach, że tym razem nie będą one stanowić dla nas większego problemu. Przede wszystkim możemy z wszystkich rozważań usunąć skrajne pozycje: pierwszą oraz n-tą. Problem zamian na sąsiednich pozycjach rozwiązujemy, usuwając z *minTree* pozycje sąsiadujące z rozważaną przed wyznaczeniem minimów i umieszczając je z powrotem tuż przed zakończeniem danego kroku. Wreszcie na samym końcu możemy przypadki brzegowe rozważyć osobno, co wobec małej ich liczby nie powoduje wzrostu złożoności czasowej całego algorytmu.

Podsumowanie i analiza złożoności

Ponieważ opis całego algorytmu był dosyć długi i zawiły, na koniec przypominamy poszczególne jego etapy, przy okazji analizując ich złożoności czasowe:

1. Na początku sortujemy wszystkie cyprysy według wysokości i na podstawie posortowanego ciągu wyznaczamy ciągi D_i oraz H_i (dokładniejszy opis tej procedury pozostawiamy Czytelnikowi jako ćwiczenie). Złożoność czasowa tego kroku to — w przypadku zastosowania efektywnego algorytmu sortowania, na przykład przez scalanie — $O(n \log n)$.
2. Budujemy dziewięć pustych drzew *minTree*. Operacja ta zajmuje nam czas $O(n)$.
3. Wstawiamy po jednej kopii każdej pozycji $t' \in [2, n-1]$ do każdego z drzew A_1, A_2 oraz A_3. Krok ten ma łączną złożoność czasową $O(n \log n)$, ponieważ wstawienie jednego elementu wymaga czasu $O(\log n)$. Realizując go procedurą analogiczną do inicjalizacji drzewa przedziałowego, możemy tę złożoność zredukować do $O(n)$, choć niestety nie zmniejszy to złożoności całego algorytmu, gdyż dominujący jest kolejny krok.
4. Rozważamy kolejno cyprysy według rang (pomijając cyprysy zajmujące pierwszą i ostatnią pozycję w szpalerze). Dla każdego z nich:
   (a) dwie sąsiadujące z nim pozycje $t - 1$ oraz $t + 1$ zostają usunięte ze wszystkich *minTree* (czas $O(\log n)$),
   (b) wyliczamy minimalną wartość z dziewięciu wartości zwróconych przez funkcję *min_val* (czas $O(\log n)$),
   (c) dwie sąsiadujące z cyprysem pozycje ($t - 1$ i $t + 1$) wstawiamy do odpowiednich *minTree* (czas, jak poprzednio, to $O(\log n)$).

   Widzimy, że łączny koszt czasowy tych wszystkich operacji jest rzędu $O(n \log n)$.

5. Rozważenie wszystkich przypadków szczególnych ma łącznie dodatkową złożoność czasową $O(n)$.

Otrzymujemy więc algorytm o całkowitej złożoności czasowej $O(n\log n)$. Jego złożoność pamięciowa jest liniowa względem n, ponieważ wielkość każdego drzewa *minTree* można oszacować przez $O(n)$, a są to największe struktury używane w rozwiązaniu.

Implementacje rozwiązania wzorcowego można znaleźć w plikach `drz.cpp` i `drz0.pas`. Gorąco zachęcamy do przyjrzenia się im, gdyż może to istotnie ułatwić dogłębne zrozumienie istoty całości rozwiązania.

Usprawnienia rozwiązania $O(n^2)$

Na koniec powrócimy do rozwiązania o złożoności $O(n^2)$ i omówimy sposoby jego usprawnienia. Chociaż żaden z nich nie powodował istotnego spadku złożoności rozwiązania (pozostawała rzędu n^2), to jednak pozwalały one zdobyć do 70 punktów na 100. Wszystkie usprawnienia polegały na eliminowaniu sprawdzeń pewnych par przesadzanych drzew.

Pierwszy pomysł. Zauważmy, że zamiana miejscami dwóch drzew rosnących na miejscach, które nazwaliśmy wcześniej zboczami (drzewo rosnące na zboczu sąsiaduje z drzewem od siebie niższym i wyższym), nie powoduje zmniejszenia współczynnika nieporządku. Trochę trudniejszy do pokazania jest fakt, że zamiana miejscami dwóch drzew rosnących w dolinach (pomiędzy dwoma wyższymi drzewami) albo dwóch drzew rosnących na szczytach (pomiędzy dwoma mniejszymi drzewami) nie może poprawić tego współczynnika. Pominięcie takich przypadków może istotnie zredukować liczbę sprawdzeń wykonywanych w algorytmie, zwłaszcza dla testów specyficznej postaci, w których dominuje jeden typ pozycji (doliny albo zbocza, albo szczyty).

Drugi pomysł. Rozwiązanie kwadratowe możemy także usprawnić, dzieląc proces zamiany miejscami drzew z pozycji t i t' na zmiany zachodzące na pozycji t oraz zmiany zachodzące na pozycji t', podobnie jak w rozwiązaniu wzorcowym. Oznaczmy wysokość drzewa z pozycji t przez h, a drzewa z pozycji t' — przez h'. Dodatkowo przez a i b oznaczmy wysokości drzew zajmujących pozycje sąsiadujące z t (jak w rozwiązaniu wzorcowym). Zmiany zachodzące w miejscu t powodują zmniejszenie współczynnika nieporządku o $|a-h|+|h-b|$ i zwiększenie go o $|a-h'|+|h'-b|$. Z nierówności $|a-h'|+|h'-b| \geqslant |a-b|$, którą łatwo pokazać, dostajemy dolne oszacowanie na zmianę współczynnika nieporządku, którą można uzyskać, wstawiając jakiekolwiek drzewo w miejsce t:

$$o_t = |a-b| - |a-h| - |h-b|.$$

Jak można to oszacowanie wykorzystać? Jeżeli znaleźliśmy już zamianę drzewa z pozycji t z innym, która zmniejsza współczynnik nieporządku o x, a dla wszystkich nierozważonych jeszcze pozycji t' zachodzi nierówność $o_t + o_{t'} \geqslant x$, to możemy zaprzestać dalszych poszukiwań kandydata do przesadzenia na pozycję t. Zwróćmy uwagę, że powyższe oszacowania są poprawne jedynie dla zamian drzew rosnących na pozycjach niesąsiednich i nieskrajnych, czyli przypadki szczególne wymagają odrębnego rozpatrzenia.

Zastosowanie powyższego pomysłu powodowało w średnim przypadku kilkukrotne przyspieszenie rozwiązania.

Testy

Rozwiązania zawodników były sprawdzane na zestawie 10 testów. Testy poprawnościowe miały specyficzne struktury, natomiast w pozostałych testach występowała mniej więcej zrównoważona liczba pozycji typu dolina, zbocze i szczyt. Efekt zrównoważenia uzyskano dzięki zamieszczeniu w teście naprzemiennych krótkich ciągów drzew o rosnących wysokościach, poprzeplatanych również krótkimi ciągami drzew o malejących wysokościach. Wysokości drzew w tych ciągach były generowane losowo.

Nazwa	n	Opis
drz1.in	10	test poprawnościowy
drz2.in	20	test poprawnościowy
drz3.in	100	test poprawnościowy
drz4.in	1 000	test eliminujący rozwiązania o złożonościach $O(n^3)$
drz5.in	9 600	test eliminujący rozwiązania $O(n^3)$ i gorsze rozwiązania $O(n^2)$
drz6.in	20 000	test eliminujący większość rozwiązań $O(n^2)$
drz7.in	35 000	duży test wydajnościowy
drz8.in	48 000	duży test wydajnościowy

Nazwa	**n**	**Opis**
drz9.in	50 000	duży test wydajnościowy
drz10.in	50 000	duży test wydajnościowy

Piotr Stańczyk
Treść zadania, Opracowanie

Piotr Stańczyk
Program

Osie symetrii

Jaś — powszechnie doceniany młody matematyk — ma młodszą siostrę Justynę. Jaś bardzo lubi swoją siostrę i chętnie pomaga jej w odrabianiu prac domowych, jednak — jak większość osób o ścisłym umyśle — nie lubi rozwiązywać tych samych zadań wielokrotnie. Na jego nieszczęście Justyna jest bardzo pilną uczennicą, przez co dla pewności prosi Jasia o sprawdzanie tych samych prac domowych wielokrotnie.

Pewnego słonecznego piątku, poprzedzającego długi majowy weekend, nauczycielka matematyki zadała wiele zadań polegających na wyznaczaniu osi symetrii różnych figur geometrycznych. Justyna zapewne spędzi znaczną część wolnego czasu, rozwiązując te zadania. Jaś zaplanował już sobie wyjazd nad morze, ale czuje się w obowiązku pomóc siostrze. Wymyślił więc, że najlepszym rozwiązaniem problemu będzie napisanie programu, który ułatwi sprawdzanie odpowiedzi do rozwiązanych przez Justynę zadań. Ponieważ Jaś jest matematykiem, a nie informatykiem, a Ty jesteś jego najlepszym kolegą, Tobie przypadło napisanie stosownego programu.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opisy wielokątów,*
- *dla każdego wielokąta wyznaczy liczbę osi symetrii,*
- *wypisze wynik na standardowe wyjście.*

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę naturalną t $(1 \leqslant t \leqslant 10)$ *— jest to liczba wielokątów, dla których należy wyznaczyć liczbę osi symetrii. Następnie znajduje się* t *opisów wielokątów. Pierwszy wiersz opisu zawiera jedną liczbę naturalną* n $(3 \leqslant n \leqslant 100\,000)$ *oznaczającą liczbę wierzchołków wielokąta. Każdy z następnych* n *wierszy zawiera dwie liczby całkowite* x *i* y $(-100\,000\,000 \leqslant x, y \leqslant 100\,000\,000)$ *reprezentujące współrzędne kolejnych wierzchołków wielokąta. Wielokąty nie muszą być wypukłe, ale nie mają samoprzecięć — jedynym punktem wspólnym dwóch różnych boków jest ich wspólny koniec i każdy wierzchołek należy do dokładnie dwóch boków. Żadne dwa kolejne boki wielokąta nie są równoległe.*

Wyjście

Program powinien wypisać dokładnie t *wierszy;* k*-ty wiersz powinien zawierać dokładnie jedną liczbę naturalną — liczbę osi symetrii* k*-tego wielokąta.*

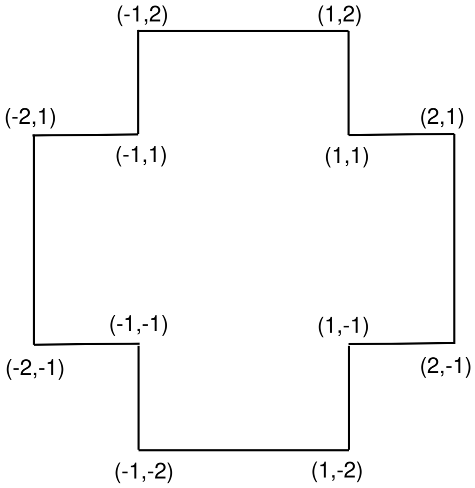
Przykład

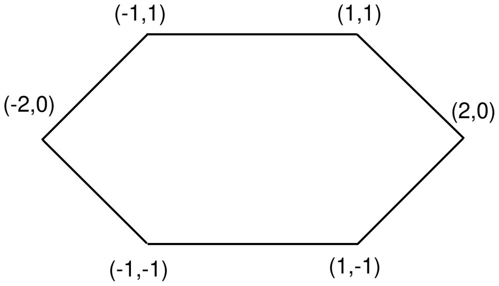
Dla danych wejściowych:

```
2
12
1 -1
2 -1
2 1
1 1
1 2
-1 2
-1 1
-2 1
-2 -1
-1 -1
-1 -2
1 -2
6
-1 1
-2 0
-1 -1
1 -1
2 0
1 1
```

poprawnym wynikiem jest:

```
4
2
```





Rozwiązanie

Wstępne obserwacje

Przystępując do rozwiązywania zadania, należy na wstępie zastanowić się, jakie interesujące nas własności posiadają osie symetrii wielokątów. Własności, których poszukujemy, nie tylko pozwolą na rozwiązanie zadania, ale również mogą pomóc w wymyśleniu efektywnego algorytmu.

Pierwszą ważną obserwacją jest fakt, iż każda oś symetrii wielokąta musi mieć co najmniej jeden punkt wspólny z tym wielokątem. Wynika to bezpośrednio z tego, że wielokąt jest spójny (potocznie mówiąc, można go narysować na kartce papieru bez odrywania ołówka). W związku z tym, dla każdej prostej l nieprzecinającej wielokąta w, znajduje się on dokładnie po jednej stronie prostej l, a co za tym idzie l nie jest osią symetrii w.

Istotnym założeniem, jakie znajdujemy w treści zadania, jest informacja, że rozpatrywane wielokąty są proste (ich obwody nie zawierają samoprzecięć). Założenie to gwarantuje, że każda oś symetrii wielokąta przecina go w dokładnie dwóch różnych punktach. W celu wykazania tego faktu rozpatrzmy wielokąt w oraz pewną jego oś symetrii l. Wiemy już (z poprzednich rozważań), że istnieje co najmniej jeden punkt wspólny obwodu wielokąta w i prostej l. Wybierzmy dowolnie punkt przecięcia wielokąta w z prostą l i nazwijmy go A. Wyobraźmy sobie, że z punktu A wyruszają dwa roboty i poruszają się symetrycznie po obwodzie wielokąta po obu stronach osi l. W końcu roboty spotykają się ponownie w pewnym punkcie — muszą kiedyś się spotkać, gdyż wielokąt jest łamaną zamkniętą; nazwijmy ten punkt B.

Punkt B jest różny od punktu A (inaczej obwód wielokąta zawierałby samoprzecięcie). Co więcej, w punkcie B roboty zakończyły sumaryczne obchodzenie całego wielokąta (w przeciwnym przypadku spotkałyby się w punkcie będącym samoprzecięciem obwodu wielokąta). Oczywiste jest także, że punkt B znajduje się na osi l. Zatem każda rozpatrywana przez nas oś symetrii ma dokładnie dwa różne punkty przecięcia z obwodem wielokąta.

Zastanówmy się, jak mogą wyglądać punkty przecięcia obwodu wielokąta w z osią symetrii l. Bez wątpienia istnieją trzy możliwe sytuacje:

- oś symetrii przechodzi przez dwa wierzchołki wielokąta,
- oś symetrii przecina dwa boki wielokąta,
- oś symetrii przecina jeden bok i przechodzi przez jeden wierzchołek.

W przypadku punktu przecięcia osi symetrii z obwodem wielokąta w wierzchołku, prosta l musi być dwusieczną kąta w tym wierzchołku. Podobnie, jeśli oś symetrii przecina bok wielokąta, to musi być do niego prostopadła. Analizując treść zadania, jesteśmy w stanie powiedzieć coś więcej na temat przecięcia l z bokiem wielokąta. Jedno z założeń mówi bowiem, że żadne dwa kolejne boki nie są do siebie równoległe. Możemy z tego wywnioskować, że każda oś symetrii przecinająca bok wielokąta, musi być jego symetralną (końce rozpatrywanego boku są swoimi obrazami w symetrii osiowej, a zatem znajdują się w takiej samej odległości od osi l).

Z zebranych obserwacji wynika, że przez każdy wierzchołek wielokąta oraz przez każdy jego bok przechodzi co najwyżej jedna oś symetrii. Dzięki temu, problem postawiony w zadaniu możemy sprowadzić do wyznaczenia liczby boków i wierzchołków wielokąta, przez które przechodzi oś symetrii. Uzyskany w ten sposób wynik wystarczy podzielić przez dwa, aby uzyskać poszukiwaną liczbę osi symetrii.

Implementację rozwiązania możemy sobie dodatkowo ułatwić, dzieląc każdy bok na dwie równe części poprzez dodanie na środku sztucznych wierzchołków. W ten sposób możemy skupić się na zliczaniu wierzchołków, przez które przechodzą osie symetrii, nie martwiąc się już więcej bokami wielokąta (symetralne boków stają się dwusiecznymi kątów w dodanych wierzchołkach).

Biorąc pod uwagę wszystkie dotychchzasowe spostrzeżenia, zadanie sprowadza się do sprawdzenia, dla każdego wierzchołka wielokąta, czy przechodzi przez niego oś symetrii. W zależności od tego, jak efektywnie jesteśmy w stanie odpowiadać na takie pytania, uzyskamy rozwiązanie o odpowiedniej złożoności obliczeniowej.

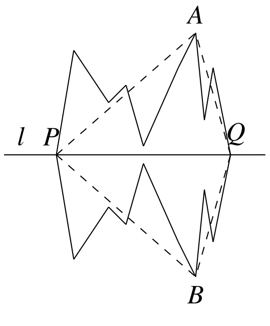
Rozwiązanie zbyt wolne

Dla ustalonego wierzchołka wielokąta można w łatwy sposób sprawdzić w czasie $O(n)$, czy przechodzi przez niego oś symetrii (gdzie n to liczba wierzchołków wielokąta). Ponumerujmy wierzchołki wielokąta (uwzględniając również sztuczne wierzchołki umieszczone na środkach boków) w kolejności występowania na obwodzie, od 0 do $2n-1$. Odwołując się do wierzchołka o numerze k dla $k < 0$ lub $k \geqslant 2n$, będziemy mieli na myśli wierzchołek o numerze $k \bmod 2n$.

Twierdzenie 1 *Przez wierzchołek o numerze k przechodzi oś symetrii l wtedy i tylko wtedy, gdy dla każdego $m \in \{1,2,\ldots n-1\}$, wierzchołek o numerze $k-m$ jest obrazem wierzchołka o numerze $k+m$ w symetrii osiowej względem l.*

Dowód Dowód powyższego faktu jest dość prosty — wystarczy pamiętać o tym, że każda oś symetrii jest dwusieczną kąta w rozpatrywanym wierzchołku oraz że żaden z oryginalnych kątów wielokąta nie jest kątem półpełnym, a wszystkie sztucznie dodane są kątami półpełnymi. Załóżmy, że k jest wierzchołkiem oryginalnym, i rozważmy ponownie dwa roboty wyruszające z k w różne strony po obwodzie wielokąta. Zauważmy, że roboty docierają jednocześnie do wierzchołków $k-1$ i $k+1$ odpowiednio — wierzchołki te są swoimi wzajemnymi obrazami w symetrii względem osi l i są oba wierzchołkami sztucznie dodanymi. Dalej, łatwo zauważyć, że jeśli roboty wyruszają jednocześnie z dwóch wierzchołków $k-m$ oraz $k+m$ tego samego typu (oryginalnych lub sztucznych) będących swoimi wzajemnymi obrazami w symetrii, to docierają do kolejnej pary $k-m-1$ i $k+m+1$, która ma analogiczne własności — są to wierzchołki tego samego typu będące swoimi wzajemnymi obrazami w symetrii względem osi l. Na końcu roboty spotykają się w punkcie $k+n$. To kończy dowód twierdzenia. ■

Załóżmy, że mamy daną prostą l przechodzącą przez dwa różne punkty P i Q. Dla dwóch wierzchołków wielokąta A i B sprawdzenie, czy są one swoimi obrazami w symetrii osiowej względem l, można wykonać, wyznaczając odległości punktów A i B od punktów P i Q. Dokładniej, musi zachodzić $|AP| = |BP|$ oraz $|AQ| = |BQ|$. Prosta l przechodząca przez wierzchołki k i $k+n$ jest osią symetrii rozpatrywanego wielokąta, jeśli powyższa zależność zachodzi dla wszystkich $n-1$ par wierzchołków wielokąta postaci $k-m$ i $k+m$, gdzie $1 \leqslant m < n$.



Powyższy algorytm wymaga przeanalizowania $O(n)$ prostych (jako pary punktów wyznaczających proste wybieramy pary wierzchołków wielokąta k oraz $k+n$). Sprawdzenie każdej z nich zajmuje czas $O(n)$, a zatem cały program działa w czasie $O(n^2)$. Rozwiązanie można nieco poprawić, zauważając, że dla wielu potencjalnych osi symetrii dość szybko znajdowana jest para niesymetrycznych punktów dyskwalifikująca tę prostą — w takiej sytuacji można przerwać sprawdzanie i przejść do kolejnej prostej. Tak zmodyfikowany program działa bardzo szybko w przypadku wielokątów o nieregularnych kształtach.

Rozwiązanie wzorcowe

W poprzednim rozwiązaniu zadania nie uwzględnialiśmy żadnych zależności pomiędzy kolejno analizowanymi prostymi. Okazuje się, że zmieniając nieco podejście, jesteśmy w stanie istotnie poprawić złożoność obliczeniową programu. W tym celu spróbujmy popatrzeć na wielokąt jako na ciąg naprzemiennie występujących wierzchołków i odcinków. Z każdym bokiem wielokąta wiążemy jego długość, natomiast z każdym wierzchołkiem — kąt wewnętrzny wielokąta, występujący pomiędzy przylegającymi do tego wierzchołka bokami. Po ustaleniu wierzchołka początkowego i kierunku możemy zbudować naprzemienny ciąg c, składający się z długości boków oraz miar kątów, występujących kolejno na obwodzie wielokąta. Uzyskujemy w ten sposób jednoznaczną reprezentację wielokąta w postaci ciągu liczb. Musimy jedynie zadbać, by nie pomylić liczb reprezentujących kąty z liczbami reprezentującymi boki — w tym celu miary kątów możemy reprezentować na przykład za pomocą liczb ujemnych.

Twierdzenie 1 sformułowane dla wielokąta reprezentowanego tradycyjnie możemy teraz zapisać dla wielokąta reprezentowanego przez ciąg skonstruowany jak wyżej:

Twierdzenie 2 *Przez dowolny wierzchołek wielokąta reprezentowany przez element* c_i *ciągu* $c = (c_0, c_1, \ldots, c_{2n-1})$ *przechodzi oś symetrii wtedy i tylko wtedy, gdy dla każdego* $m \in \{1, 2, \ldots n-1\}$, $c_{i-m} = c_{i+m}$. *Element* c_i *dla* $i < 0$ *lub* $i \geqslant 2n$ *utożsamiamy przy tym z elementem* $c_{i \bmod 2k}$

Przypatrując się powyższej własności, łatwo spostrzec podobieństwo postawionego problemu do zagadnienia wyszukiwania palindromów w tekście.

Definicja 1 *Palindromem* nazywamy sekwencję liter (liczb), która czytana od przodu oraz od tyłu jest taka sama. Dla przykładu, słowo *anna* jest palindromem o długości 4, natomiast słowo *anannnab* nie jest palindromem. Słowo to jednak zawiera palindrom o długości 5 — *annna*. *Promieniem* palindromu długości l nazywamy liczbę $\lfloor \frac{l}{2} \rfloor$.

Efektywność rozwiązania naszego zadania zależy teraz od tego, jak szybko jesteśmy w stanie wyszukiwać palindromy w zadanym ciągu. Z pomocą przychodzi nam w tym momencie algorytm Manachera, o którym można przeczytać w [16] (Algorytmy i struktury danych). Algorytm ten pozwala, dla każdej pozycji w zadanym ciągu o długości n, wyznaczyć maksymalny promień palindromu o długości nieparzystej o środku na tej pozycji w czasie $O(n)$. Dla przykładowego ciągu *anannnab*, wyznaczone przez algorytm Manachera promienie są następujące: $0, 1, 1, 0, 2, 0, 0, 0$.

Jedyne, co nam pozostaje do zrobienia, to zbudować ciąg d, który stanowi złączenie dwóch kopii ciągu c (w ten sposób każdy wierzchołek wielokąta posiada swojego reprezentanta w ciągu d, który zarówno z lewej, jak i prawej strony ma co najmniej n elementów), a następnie wyznaczyć — z wykorzystaniem algorytmu Manachera — dla każdego elementu ciągu d promień palindromu o środku w tym elemencie, by wreszcie policzyć liczbę r elementów reprezentujących wierzchołki, dla których promienie są nie mniejsze od $n-1$. Odpowiedzią na pytanie postawione w treści zadania jest oczywiście $\frac{r}{2}$.

Testy

Testy wykorzystane w zadaniu zostały wygenerowane przy użyciu specjalnych generatorów:

- $G1$ — generator wielokątów zawierających określony podzbiór osi symetrii: {pionowa, pozioma, ukośna pod kątem 45 stopni, ukośna pod kątem −45 stopni},
- $G2$ — generator wielokątów zawierających oś symetrii nachyloną pod zadanym kątem,
- $G3$ — generator wielokątów o postaci schodkowej. Wielokąty tego typu mają bardzo regularne obwody, co powoduje, że przedstawione w opracowaniu udoskonalone rozwiązanie nieefektywne działa dla nich bardzo długo.

Każdy wygenerowany powyższymi metodami wielokąt został następnie poddany serii modyfikacji — odbić i obrotów, przesunięć o losowy wektor oraz zaburzeń. Ich celem było wyeliminowanie niektórych osi symetrii.

Zestawy testów zostały przygotowane w taki sposób, aby rozwiązania niepoprawne uzyskiwały jak najmniejszą liczbę punktów. Dodatkowo, programy nieoptymalne o pesymistycznej złożoności $O(n^2)$ uzyskują w granicach 40-50 punktów.

Poniższa lista zawiera podstawowe informacje dotyczące testów. Przez t została oznaczona liczba wielokątów, przez m — największa bezwzględna wartość współrzędnych punktów, a przez k — sumaryczna liczba wierzchołków we wszystkich wielokątach.

Nazwa	t	m	k	Opis
osi1.in	4	10	135	Mały test poprawnościowy
osi2.in	10	96	72	6 wielokątów typu $G1$, 1 typu $G2$ i 3 typu $G3$
osi3.in	7	1 423	106	4 wielokąty typu $G1$, 1 typu $G2$ i 2 typu $G3$
osi4.in	6	896 911	82 222	3 wielokąty typu $G1$, 1 typu $G2$ i 2 typu $G3$
osi5.in	5	15 152	29 988	2 wielokąty typu $G1$, 1 typu $G2$ i 2 typu $G3$
osi6.in	4	9 977	37 016	2 wielokąty typu $G1$ i 2 typu $G2$
osi7.in	7	99 381	36 170	3 wielokąty typu $G1$, 2 typu $G2$ i 2 typu $G3$
osi8.in	6	998 863	47 587	3 wielokąty typu $G1$ i 3 typu $G3$
osi9.in	9	954 385	595 692	4 wielokąty typu $G1$, 2 typu $G2$ i 3 typu $G3$
osi10.in	5	9 301 420	442 806	2 wielokąty typu $G1$, 2 typu $G2$ i 1 typu $G3$
osi11.in	8	79 918 099	534 098	2 wielokąty typu $G1$, 3 typu $G2$ i 3 typu $G3$
osi12.in	10	69 659 116	955 740	6 wielokątów typu $G1$, 3 typu $G2$ i 1 typu $G3$

Jakub Radoszewski
Treść zadania

Jakub Pawlewicz, Jakub Radoszewski
Opracowanie, Program

OI, Etap I

Zapytania

Kryptograf Bajtazar pracuje nad złamaniem szyfru ABB (Agencji Bezpieczeństwa Bajtocji). Doszedł już do tego, że przy odszyfrowywaniu wiadomości będzie musiał wielokrotnie odpowiadać na zapytania postaci: „dla danych liczb naturalnych a, b i d, ile jest takich par liczb naturalnych (x,y), że:

- $1 \leqslant x \leqslant a$,
- $1 \leqslant y \leqslant b$,
- *$NWD(x,y) = d$, gdzie $NWD(x,y)$ to największy wspólny dzielnik liczb x i y”.*

Bajtazar chciałby zautomatyzować swoją pracę, więc poprosił Cię o pomoc.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia listę zapytań, na które musi odpowiedzieć Bajtazar,*
- *wyznaczy odpowiedzi na te zapytania,*
- *wypisze wynik na standardowe wyjście.*

Wejście

Pierwszy wiersz wejścia zawiera jedną dodatnią liczbę całkowitą n ($1 \leqslant n \leqslant 50\,000$), oznaczającą liczbę zapytań Bajtazara. Każdy z kolejnych n wierszy zawiera po trzy liczby całkowite a, b i d ($1 \leqslant d \leqslant a,b \leqslant 50\,000$), pooddzielane pojedynczymi odstępami. Każda taka trójka reprezentuje jedno zapytanie.

Wyjście

Twój program powinien wypisać n wierszy. Wiersz i powinien zawierać jedną liczbę całkowitą: odpowiedź na i-te zapytanie z wejścia.

Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
3
2
```

Pary uzyskane w pierwszym zapytaniu to: $(2,2)$, $(2,4)$ i $(4,2)$, a w drugim: $(6,3)$ i $(3,3)$.

Rozwiązanie

Wprowadzenie

Oznaczmy przez $Z_d(a,b)$ odpowiedź na zapytanie (a,b,d), czyli liczbę takich par liczb całkowitych (x,y), dla których $1 \leqslant x \leqslant a$, $1 \leqslant y \leqslant b$ i $\mathrm{NWD}(x,y) = d$. Okazuje się, że stosunkowo łatwo można wyeliminować z zapytań parametr d:

Fakt 1

$$Z_d(a,b) = Z_1\left(\left\lfloor \frac{a}{d} \right\rfloor, \left\lfloor \frac{b}{d} \right\rfloor\right). \tag{1}$$

Dowód Wartość $Z_d(a,b)$ jest równa liczbie elementów zbioru $\{(x,y) : 1 \leqslant x \leqslant a, 1 \leqslant y \leqslant b, \mathrm{NWD}(x,y) = d\}$. Dowolną parę (x,y) z tego zbioru możemy przedstawić w postaci $(x'd, y'd)$, gdzie $\mathrm{NWD}(x',y') = 1$ oraz, co więcej, zachodzą ograniczenia: $1 \leqslant x' \leqslant \lfloor \frac{a}{d} \rfloor$ i $1 \leqslant y' \leqslant \lfloor \frac{b}{d} \rfloor$. Ponieważ przekształcenie (x,y) w (x',y') jest różnowartościowe, to dowodzi to, że $Z_d(a,b) \leqslant Z_1(\lfloor a/d \rfloor, \lfloor b/d \rfloor)$.

Możemy też spojrzeć na sytuację z drugiej strony: z każdej pary (x',y') spełniającej nierówności $1 \leqslant x' \leqslant \lfloor \frac{a}{d} \rfloor$, $1 \leqslant y' \leqslant \lfloor \frac{b}{d} \rfloor$ oraz warunek $\mathrm{NWD}(x',y') = 1$, możemy utworzyć parę $x = x'd$, $y = y'd$ spełniającą $1 \leqslant x \leqslant a$, $1 \leqslant y \leqslant b$ i $\mathrm{NWD}(x,y) = d$. Podobnie jak poprzednio, przekształcenie jest różnowartościowe, więc widzimy, że $Z_1(\lfloor a/d \rfloor, \lfloor b/d \rfloor) \leqslant Z_d(a,b)$. ■

Będziemy od tej pory zakładać, że w każdym zapytaniu parametr d jest równy 1 i dla uproszczenia przyjmiemy zapis:

$$Z(a,b) = Z_1(a,b). \tag{2}$$

Niech dalej $Q \subseteq \mathbb{N} \times \mathbb{N}$ oznacza zbiór wszystkich zapytań, na jakie mamy odpowiedzieć. Rozmiar $|Q|$ tego zbioru oznaczmy zgodnie z treścią zadania przez n. Przez m oznaczmy maksimum liczb, które występują w zapytaniach z Q.

Rozwiązanie wzorcowe

W zadaniu musimy odpowiedzieć na wiele pytań $Z(a,b)$. Sprawdźmy, czy nie da się wykorzystać obliczonych wcześniej odpowiedzi, do wyznaczania odpowiedzi na kolejne pytania. Na przykład, zastanówmy się, czy na podstawie $Z(a,b)$ możemy szybko obliczyć $Z(a,b+1)$ lub $Z(a+1,b)$. Dokładniej, ponieważ $Z(a,b) = Z(b,a)$ i różnica między $Z(a,b+1)$ a $Z(a,b)$ jest dokładnie taka, jak między $Z(b+1,a)$ a $Z(b,a)$, to możemy skoncentrować się jedynie na poszukiwaniu sposobu wyznaczania różnic typu $Z(a,b+1) - Z(a,b)$.

Definicja 1 Przez $C(a,b)$ oznaczać będziemy liczbę takich wartości całkowitych $1 \leqslant x \leqslant a$, dla których $\mathrm{NWD}(x,b) = 1$.

Zauważmy, że $Z(a,b+1) - Z(a,b) = C(a,b+1)$. Efektywny sposób wyznaczania wartości $C(a,b)$ daje z kolei następujący fakt:

Fakt 2 *Niech $b = p^k q$, $b > 1$, gdzie p jest liczbą pierwszą i p nie dzieli q, a k jest liczbą całkowitą dodatnią. Wówczas*

$$C(a,b) = C(a,q) - C\Big(\Big\lfloor \frac{a}{p} \Big\rfloor, q\Big). \tag{3}$$

Dowód Jeżeli liczba naturalna x, $1 \leqslant x \leqslant a$, jest względnie pierwsza z b, to jest ona również względnie pierwsza z q, a zatem $C(a,b) \leqslant C(a,q)$. Ile dokładnie wynosi różnica $C(a,q) - C(a,b)$? Jest ona równa liczbie tych $1 \leqslant x \leqslant a$, które są względnie pierwsze z q, ale dzielą się przez p, czyli liczbie elementów zbioru $\{p \cdot 1, p \cdot 2, \ldots, p \cdot \lfloor \frac{a}{p} \rfloor\}$ względnie pierwszych z q. Ponieważ $\mathrm{NWD}(p,q) = 1$, to $C(a,q) - C(a,b)$ jest równa liczbie elementów zbioru $\{1, 2, \ldots, \lfloor \frac{a}{p} \rfloor\}$ względnie pierwszych z q, czyli $C(\lfloor \frac{a}{p} \rfloor, q)$. ■

Korzystając z Faktu 2, możemy skonstruować rekurencyjny algorytm wyznaczania $C(a,b)$. Dla parametru $b > 1$ stosujemy w nim wzór (3); dla parametru $b = 1$, mamy $C(x,1) = x$. Zastanówmy się, jaką złożoność czasową ma ten algorytm.

Definicja 2 Niech $\omega(b)$ oznacza liczbę różnych dzielników pierwszych b.

W każdym z wywołań rekurencyjnych, zgodnie ze wzorem (3), b zostaje zastąpione przez q, które ma o jeden dzielnik pierwszy mniej ($\omega(q) = \omega(b) - 1$). Na głębokości rekursji równej $\omega(b)$ otrzymamy zatem $2^{\omega(b)}$ składników postaci $\pm C(x,1)$, jako że każdy poziom rekursji jest dwukrotnie liczniejszy od poprzedniego. Ponieważ łączna liczba wywołań rekurencyjnych w algorytmie jest równa $2^0 + 2^1 + \cdots + 2^{\omega(b)} = 2^{\omega(b)+1} - 1$, więc koszt czasowy wyznaczania $C(a,b)$ wynosi $O(2^{\omega(b)})$. Jedyny problem, jaki dotychczas przemilczeliśmy, to sposób znajdowania dzielnika pierwszego liczby b (wraz z krotnością jego występowania w rozkładzie b na czynniki pierwsze). Rozwiązanie tej kwestii stanowi *sito Eratostenesa*.

Sito Eratostenesa

Sito Eratostenesa to powszechnie znany algorytm, który pozwala efektywnie wyznaczyć wszystkie liczby pierwsze w przedziale $[1,m]$. Przedstawimy jedną z jego implementacji.

W algorytmie będziemy wyznaczać tablicę wartości logicznych *pierwsza*$[2..m]$. Na początku wypełniamy ją w całości wartościami *true* (prawda). Następnie analizujemy kolejno liczby $i = 2, \ldots, m$ i dla każdej z nich, „wykreślamy” z tablicy jej wielokrotności. Wykreślanie można rozpocząć od i^2, gdyż, jak łatwo zauważyć, wszystkie mniejsze wielokrotności i musiały już wcześniej zostać wykreślone. Oto pseudokod sita Eratostenesa:

```
1:  for $i := 2$ to $m$ do $pierwsza[i] := true$;
2:  for $i := 2$ to $m$ do
3:    if $pierwsza[i]$ then
4:    begin
5:      $j := i^2$;
6:      while $j \leqslant N$ do
7:      begin
8:        $pierwsza[j] := false$;
9:        $j := j + i$;
10:     end
11:   end
```

Sito Eratostenesa jest algorytmem prostym i krótkim w implementacji, a dodatkowo — efektywnym. Można pokazać, że przedstawiona procedura ma złożoność czasową $O(m \log\log m)$. Dowód tego oszacowania nie jest jednak łatwy, a nam wystarczy, gdy wykażemy, że algorytm działa w czasie $O(m \log m)$ — co jest znacznie prostsze. Zauważmy, że pole j tablicy *pierwsza* odwiedzamy w instrukcji w wierszu 8 co najwyżej tyle razy, ile różnych dzielników pierwszych ma j. Liczbę tych dzielników można ograniczyć przez $O(\log m)$, więc liczbę wszystkich operacji można oszacować przez $O(m \log m)$.

Przedstawiony algorytm można uzupełnić tak, by pozwalał znajdować rozkład na czynniki pierwsze liczb z przedziału $[1,m]$. Wykorzystamy do tego celu dodatkową tablicę *dzielnik*, w której dla każdego elementu zapiszemy jego dzielnik, który znajdziemy jako pierwszy. Na początku algorytmu wypełnimy wszystkie komórki tablicy *dzielnik* wartościami -1, natomiast w momencie modyfikacji tablicy *pierwsza* (wiersz 8) ustawiamy $dzielnik[j] := i$, o ile tylko element $dzielnik[j]$ nie było jeszcze ustawiony na dodatnią wartość. W ten sposób dla każdej liczby złożonej w odpowiedniej komórce tablicy *dzielnik* otrzymamy jej *najmniejszy* dzielnik pierwszy. Zauważmy, że opisana modyfikacja nie powoduje wzrostu złożoności czasowej całego algorytmu.

```
1:  for $i := 2$ to $m$ do $dzielnik[i] := -1$;
2:  for $i := 2$ to $m$ do $pierwsza[i] := true$;
3:  for $i := 2$ to $m$ do
4:    if $pierwsza[i]$ then
5:    begin
6:      $j := i^2$;
7:      while $j \leqslant N$ do
8:      begin
9:        $pierwsza[j] := false$;
10:       if $dzielnik[j] < 0$ then $dzielnik[j] := i$;
11:       $j := j + i$;
12:     end
13:   end
```

Zawartość tablicy *dzielnik* możemy wykorzystać do rozkładu liczby na czynniki pierwsze. Dla liczby j jej pierwszy dzielnik odczytujemy z pola $dzielnik[j]$, po czym przechodzimy do poszukiwania rozkładu liczby $j/dzielnik[j]$ itd. Dla dowolnej liczby spośród $2,\ldots,m$ złożoność czasowa tego procesu to $O(\log m)$.

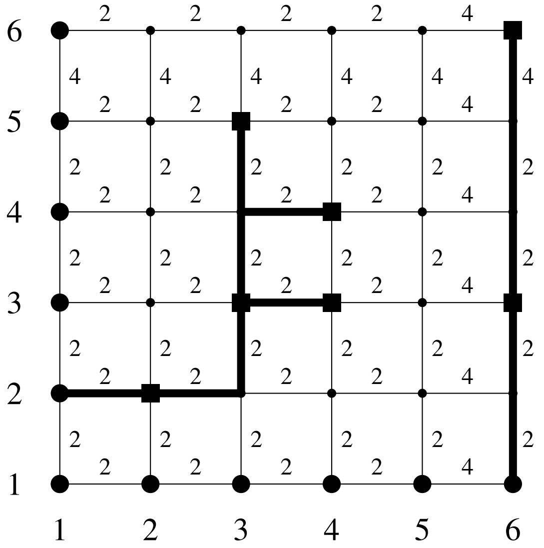
Interpretacje problemu

Potrafimy już efektywnie wyznaczać wartości $C(a,b)$. Spróbujmy zastosować tę umiejętność do znalezienia wartości $Z(a,b)$ dla wszystkich pytań ze zbioru Q. W tym celu przedstawimy dwie interpretacje zagadnienia: geometryczną i grafową — obie przydadzą się nam w dalszym toku opisu.

Krata. Zapytania możemy wyobrazić sobie jako punkty kraty $[1,m] \times [1,m]$ (o całkowitych współrzędnych) na płaszczyźnie. Wiemy, że znając wartość funkcji Z dla punktu $(a, b-1)$, umiemy policzyć tę wartość dla punktu (a,b) w czasie $O(2^{\omega(b)})$. W takim samym czasie umiemy na podstawie $Z(b-1,a)$ wyznaczyć $Z(b,a)$.

Graf. Problem możemy przedstawić także jako graf nieskierowany G, którego wierzchołki odpowiadają punktom kraty, a krawędzie łączą wszystkie pary wierzchołków, które odpowiadają punktom sąsiadującym w kracie (czyli różniącym się na jednej współrzędnej o 1). Krawędziom łączącym wierzchołki $(a, b-1)$ oraz (a,b) przypisujemy wagę $2^{\omega(b)}$. Taką samą wagę przypisujemy krawędziom $((b,a),(b-1,a))$.

Wartościami funkcji Z, które potrafimy wyznaczyć natychmiast, są wartości w punktach ze zbioru $X = \{(x,1) \mid 1 \leqslant x \leqslant m\} \cup \{(1,x) \mid 1 \leqslant x \leqslant m\}$ — dla każdej liczby naturalnej x zachodzi $Z(x,1) = Z(1,x) = x$. W takim razie problem wyznaczenia wszystkich potrzebnych nam wartości funkcji Z możemy sprowadzić do problemu konstrukcji najlżejszego lasu rozpinającego w G, łączącego wszystkie punkty ze zbioru Q z punktami ze zbioru X. Na rys. 1 jest przedstawione najlżejsze drzewo rozpinające dla przykładowego zbioru Q w przykładowym grafie G.



Rys. 1: Grafowa interpretacja problemu dla $m = 6$. Przy krawędziach są zapisane wagi, równe 2 lub 4 (gdyż $\omega(2) = \omega(3) = \omega(4) = \omega(5) = 1$, a $\omega(6) = 2$). Dużymi kółkami oznaczono punkty ze zbioru X, natomiast kwadraty reprezentują punkty zbioru $Q = \{(2,2),(3,3),(3,5),(4,3),(4,4),(6,3),(6,6)\}$. Pogrubionymi kreskami wyróżniono krawędzie najlżejszego drzewa rozpinającego (dowolnie wybranego, bo istnieje ich więcej). Ma ono wagę 26.

Powstaje pytanie, w jaki sposób wyznaczyć taki las. Niestety, graf G ma $\Theta(m^2)$ wierzchołków oraz krawędzi, co nie nastraja optymistycznie (oczywiście nie oznacza to, że problemu nie da się rozwiązać, tylko nie widać sposobu opartego na klasycznych technikach). W rozwiązaniu wzorcowym zadowolimy się więc algorytmem pozwalającym znaleźć potencjalnie trochę gorszy sposób połączenia punktów ze zbioru Q z punktami ze zbioru X.

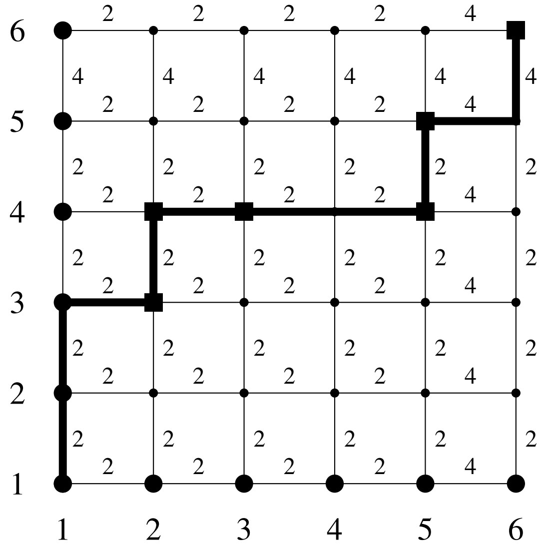
Rozwiązanie części problemu

Skoro zrezygnowaliśmy z konstrukcji optymalnego lasu połączeń, spróbujmy wybrać pewną podgrupę powiązanych ze sobą zapytań, na które będziemy w stanie stosunkowo szybko odpowiedzieć. Wybierzmy ze zbioru Q taki ciąg k punktów $(x_1,y_1),\ldots,(x_k,y_k)$, że ciągi $x_1,\ldots,x_k$ i $y_1,\ldots,y_k$ są monotoniczne. Możemy to zrobić następująco:

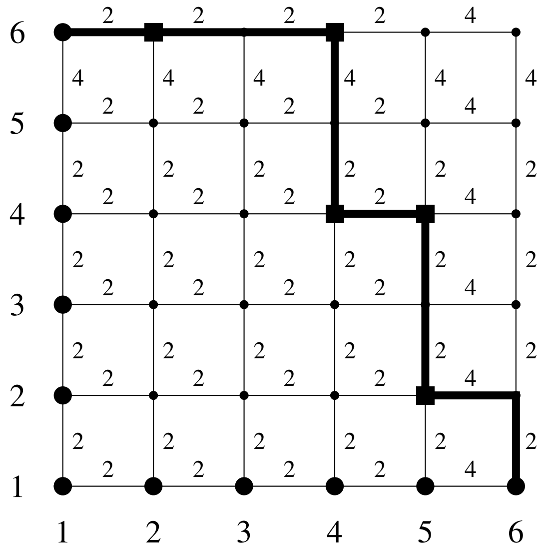
- sortujemy punkty ze zbioru Q niemalejąco po pierwszej współrzędnej (na przykład metodą sortowania przez scalanie w czasie $O(n\log n)$);
- w ciągu drugich współrzędnych znajdujemy najdłuższy podciąg niemalejący i najdłuższy podciąg nierosnący (potrafimy to zrobić w czasie $O(n\log n)$, patrz, na przykład, opracowanie zadania *Egzamin na Prawo Jazdy* w niniejszej książeczce);
- wybieramy punkty, których drugie współrzędne tworzą dłuższy z powyższych podciągów (w czasie $O(n)$).

Wartości Z dla wszystkich punktów $(x_1,y_1),\ldots,(x_k,y_k)$ możemy wyznaczyć, przemieszczając się w grafie po ścieżce długości $2m-2$:

- jeżeli ciąg y_i jest niemalejący, to jest to ścieżka od punktu $(1,1)$ do (m,m) (patrz rys. 2);
- jeżeli zaś ciąg y_i jest nierosnący, to ścieżka prowadzi od punktu $(1,m)$ do $(m,1)$ (patrz rys. 3).



Rys. 2: Ścieżka biegnąca z punktu $(1,1)$ do (m,m) przez punkty $(2,3)$, $(2,4)$, $(3,4)$, $(5,4)$, $(5,5)$, $(6,6)$; w tym przykładzie mamy $m=6$ oraz $k=6$.



Rys. 3: Ścieżka biegnąca z punktu $(1,m)$ do $(m,1)$ przez punkty $(2,6)$, $(4,6)$, $(4,4)$, $(5,4)$, $(5,2)$; w tym przykładzie mamy $m=6$ oraz $k=5$.

W obu przypadkach ścieżka zawiera dokładnie $m-1$ krawędzi poziomych oraz $m-1$ krawędzi pionowych. Suma wag zarówno poziomych, jak i pionowych krawędzi, wyraża się wzorem $\sum_{i=2}^{m} 2^{\omega(i)}$. Okazuje się, że możemy oszacować rząd wielkości tej sumy[1]:

$$\sum_{i=2}^{m} 2^{\omega(i)} = \frac{6m\ln m}{\pi^2} + O(m) = \theta(m\log m). \tag{4}$$

To oznacza, że obliczenie wszystkich wartości $C(a,b)$ odpowiadających krawędziom ścieżki oraz wartości $Z(a,b)$ odpowiadających punktom ścieżki, można wykonać w czasie $O(m\log m)$.

Udało się nam wybrać grupę k zapytań i odpowiedzieć na nie w czasie $O(n\log n + m\log m)$. Możemy więc usunąć je ze zbioru Q i kontynuować takie samo postępowanie, aż do momentu znalezienia odpowiedzi na wszystkie zapytania z Q. Pozostaje pytanie, ile tego typu rund musimy wykonać, aby rozwiązać zadanie.

Ile części ma problem?

Wykażmy poniższy fakt, który pomoże nam znaleźć odpowiedź na nurtujące nas pytanie:

Fakt 3 *Niech dany będzie ciąg liczb naturalnych* $a=(a_1,\ldots,a_{k^2+1})$ *długości* k^2+1. *W ciągu tym istnieje podciąg niemalejący długości* $k+1$ *lub podciąg nierosnący długości* $k+1$.

Dowód Uzasadnienie faktu, wykorzystujące klasyczne twierdzenie Dilwortha, można znaleźć w książce [34]. Tutaj przedstawimy inny dowód, oparty jedynie na prostych sztuczkach kombinatorycznych.

---

[1] Źródło: `http://www.research.att.com/~njas/sequences/A064608`, *The On-Line Encyclopedia of Integer Sequences*, A064608

Zdefiniujmy ciąg $b_1,\ldots,b_{k^2+1}$, gdzie b_i jest długością najdłuższego podciągu niemalejącego ciągu a kończącego się elementem a_i. Następnie rozważmy dwa przypadki:

(1) Jeżeli w ciągu b istnieje wyraz większy niż k, to znaczy, że w ciągu a istnieje podciąg niemalejący długości $k+1$, co kończy dowód w tym przypadku.

(2) Niech wszystkie wyrazy ciągu b będą mniejsze niż $k+1$, czyli w ciągu b występuje najwyżej k różnych wartości. Wówczas, na mocy zasady szufladkowej Dirichleta, wiemy, że istnieje wartość (oznaczmy ją x) występująca w tym ciągu $k+1$ razy. Pokażmy, że elementy ciągu a, którym w ciągu b odpowiadają (czyli występują na tych samych pozycjach) wartości x, tworzą ciąg malejący. Załóżmy, że $b_i = b_j = x$ dla $i < j$. Gdyby $a_i \leqslant a_j$, to moglibyśmy ciąg kończący się na a_i o długości x przedłużyć o a_j. Tym samym $b_j \geqslant x+1$, co jest sprzeczne z założeniem $b_i = b_j$.

■

Z Faktu 3 widać natychmiast, że jeżeli zbiór Q ma n elementów, to długość k skonstruowanego w jednej rundzie algorytmu ciągu zapytań jest nie mniejsza niż $\lceil\sqrt{n}\rceil$. Na pierwszy rzut oka może się więc wydawać, że po $O(\sqrt{n})$ rundach rozwiążemy całe zadanie. Nie jest to jednak oczywiste, ponieważ w kolejnych rundach rozmiar $|Q|$ zmniejsza się i znajdujemy odpowiedzi na mniej niż $\lceil\sqrt{n}\rceil$ pytań. Niemniej jednak pierwsze wrażenie okazuje się słuszne, co możemy udowodnić.

Fakt 4 *Liczba rund jest rzędu* $O(\sqrt{n})$.

Dowód Podzielmy zbiór liczb naturalnych na parami rozłączne przedziały $[i^2+1,(i+1)^2]$, dla $i \geqslant 1$. Rozważmy liczbę naturalną $n \in [i^2+1,(i+1)^2]$. Pokażmy, że jeśli byłaby ona długością ciągu a w naszym problemie, to po co najwyżej 3 rundach algorytmu długość ciągu znajdzie się w przedziale $[(i-1)^2+1,i^2]$. Wynika to stąd, że wartość $n > i^2$ musi na mocy Faktu 3 w jednej rundzie zmaleć co najmniej o i. Po maksymalnie 3 takich redukcjach, długość ciągu osiągnie wartość $n-3i \leqslant (i+1)^2-3i = i^2+2i-1-3i \leqslant i^2$, czyli spadnie do przedziału $[(i-1)^2+1,i^2]$, o ile nie znalazła się tam wcześniej.

Ponieważ wartość i jest równa $\lceil\sqrt{n}\rceil-1$, to po co najwyżej $3(\lceil\sqrt{n}\rceil-1) = O(\sqrt{n})$ rundach cały zbiór Q zostanie przetworzony. ■

Ostatecznie pokazaliśmy, że złożoność czasowa zaproponowanego algorytmu to

$$O\big(\sqrt{n}(n\log n + m\log m)\big).$$

Możemy się jeszcze zastanowić, ile tracimy w porównaniu z metodą opartą na wyznaczaniu najlżejszego lasu rozpinającego w grafie G. Okazuje się, że w pesymistycznym przypadku waga takiego lasu może wynosić

$$\Omega\Big(\sqrt{n}\sum_{i=2}^{m} 2^{\omega(i)}\Big),$$

gdzie — jak już wspominaliśmy — sumę występującą w tym wyrażeniu można oszacować wyrażeniem:

$$\sum_{i=2}^{m} 2^{\omega(i)} = \theta(m\log m).$$

Pesymistyczny przypadek występuje, na przykład, dla punktów Q równomiernie rozłożonych na kracie:

$$Q = \Big\{\Big(\Big\lfloor\frac{im}{\sqrt{n}}\Big\rfloor, \Big\lfloor\frac{jm}{\sqrt{n}}\Big\rfloor\Big) \mid i,j = 1,\ldots,\lfloor\sqrt{n}\rfloor\Big\}.$$

Zatem w najgorszym razie, nawet gdybyśmy pominęli koszt znajdowania najlżejszego lasu rozpinającego w G, to i tak na wszystkie pozostałe obliczenia musielibyśmy poświęcić czas $\Omega(\sqrt{n}m\log m)$. Widzimy więc, że zastępując najlżejsze drzewo rozpinające układem ścieżek o monotonicznie posortowanych współrzędnych punktów, nie pogarszamy złożoności obliczeniowej rozwiązania.

Implementacja opisanego w tym rozdziale algorytmu znajduje się w plikach `zap.cpp` i `zap0.pas`.

Rozwiązanie alternatywne

W rozwiązaniu wzorcowym w istotny sposób korzystaliśmy z tego, że w zadaniu mamy udzielić odpowiedzi na dużą liczbę zapytań. Okazuje się jednak, że istnieje rozwiązanie, które pozwala na efektywne odpowiadanie także na pojedyncze zapytania. Pomysł ten został zaimplementowany przez zdecydowaną większość zawodników, którzy zdobyli za zadanie maksymalną liczbę punktów. Ponieważ w rozwiązaniu tym są wykorzystywane zaawansowane techniki matematyczne, więc prezentujemy je jako drugie w kolejności.

Rozważmy wszystkie pary liczb naturalnych, z których pierwsza należy do zbioru $\{1,\dots,a\}$, a druga — do zbioru $\{1,\dots,b\}$. Tych par jest oczywiście $a\cdot b$. Możemy je podzielić na rozłączne zbiory ze względu na wartość $NWD(a,b)$, czyli:

$$a\cdot b = Z_1(a,b)+Z_2(a,b)+Z_3(a,b)+\dots \tag{5}$$

Zdefiniujmy następujące funkcje, określone dla par nieujemnych liczb rzeczywistych:

- $g(x,y)=\lfloor x\rfloor\cdot\lfloor y\rfloor$,
- $f(x,y)=Z(\lfloor x\rfloor,\lfloor y\rfloor)$.

Na podstawie równości (1) oraz (5) możemy zapisać następującą zależność między funkcjami f oraz g:

$$g(x,y)=\sum_{d\geqslant 1} f\left(\frac{x}{d},\frac{y}{d}\right).$$

Zależność nie wydaje się być przydatna, bo interesują nas wartości funkcji f, a nie g, jednak okazuje się, że możemy z niej wyznaczyć wzór na funkcję f! W tym celu można zastosować *metodę odwracania z użyciem funkcji Möbiusa* (patrz, na przykład, rozdział 4.9 w [30]). W rozważanym przypadku da nam ona następujący wzór:

$$f(x,y)=\sum_{d\geqslant 1}\mu(d)g\left(\frac{x}{d},\frac{y}{d}\right), \tag{6}$$

gdzie μ jest tak zwaną funkcją Möbiusa. Powyższy wzór za chwilę udowodnimy, jednak najpierw musimy poznać lepiej funkcję μ Möbiusa. Ma ona dwie równoważne definicje (dowód ich równoważności można znaleźć także w [30]).

Def. 1: Pierwsza z nich jest definicją typu rekurencyjnego:

$$\sum_{d|m}\mu(d)=\begin{cases}1 & \text{dla } m=1\\ 0 & \text{dla } m>1.\end{cases}$$

Z tej definicji łatwo wnioskujemy, że $\mu(1)=1$, dalej $\mu(2)=0-\mu(1)=-1$, podobnie $\mu(3)=-1$, wreszcie $\mu(4)=0-\mu(2)-\mu(1)=0$ itd. Kilka początkowych wartości funkcji μ prezentujemy w poniższej tabelce:

i	1	2	3	4	5	6	7	8	9	10
$\mu(i)$	1	-1	-1	0	-1	1	-1	0	0	1

Def. 2: Według drugiej definicji $\mu(m)=0$, jeżeli m dzieli się przez p^2 dla pewnej liczby pierwszej p. W przeciwnym przypadku $\mu(m)=(-1)^r$, gdzie r jest liczbą różnych dzielników pierwszych m.

Poznawszy funkcję Möbiusa, możemy udowodnić równość (6):

Fakt 5 *Jeżeli funkcje f oraz g są związane zależnością*

$$g(x,y)=\sum_{d\geqslant 1} f\left(\frac{x}{d},\frac{y}{d}\right)$$

oraz $\sum_{k,d\geqslant 1}|f(x/(kd))|<\infty$, to

$$f(x,y)=\sum_{d\geqslant 1}\mu(d)g\left(\frac{x}{d},\frac{y}{d}\right).$$

Dowód Przyjrzyjmy się następującemu ciągowi przekształceń:

$$\begin{aligned}\sum_{d\geqslant 1}\mu(d)g\left(\frac{x}{d},\frac{y}{d}\right) &= \sum_{d\geqslant 1}\mu(d)\sum_{k\geqslant 1} f\left(\frac{x}{kd},\frac{y}{kd}\right)\\ &= \sum_{m\geqslant 1} f\left(\frac{x}{m},\frac{y}{m}\right)\sum_{d,k\geqslant 1, m=kd}\mu(d)\\ &= \sum_{m\geqslant 1} f\left(\frac{x}{m},\frac{y}{m}\right)\sum_{d|m}\mu(d).\end{aligned}$$

Zauważmy, że na podstawie pierwszej definicji funkcji Möbiusa wewnętrzna suma $\sum_{d|m}\mu(d)$ jest niezerowa (a dokładniej równa 1) tylko dla $m=1$. To pokazuje, że jedynym niezerowym składnikiem całej sumy będzie $f(x/1,y/1)=f(x,y)$, co kończy dowód. ■

Z definicji funkcji f i g oraz udowodnionego faktu otrzymujemy równość, która pomoże nam efektywnie wyznaczać odpowiedzi na zapytania:

$$Z(a,b)=\sum_{d\geqslant 1}\mu(d)\left\lfloor\frac{a}{d}\right\rfloor\cdot\left\lfloor\frac{b}{d}\right\rfloor. \tag{7}$$

Od wzoru do algorytmu

Rozpocznijmy od uproszczenia dopiero co wyprowadzonego wzoru (7). Przede wszystkim zauważmy, że sumowanie możemy ograniczyć do $d \leqslant \max(a,b)$, gdyż dla pozostałych wartości d wyrażenie wewnątrz sumy i tak będzie równe zeru (zamiast $\max(a,b)$ można by użyć nawet $\min(a,b)$, ale nam będzie wygodniej pozostać przy wersji z maksimum). Możemy też bez straty ogólności założyć, że $a \geqslant b$, co pozwala zapisać wzór (7) w postaci:

$$Z(a,b) = \sum_{d=1}^{a} \mu(d) \left\lfloor \frac{a}{d} \right\rfloor \cdot \left\lfloor \frac{b}{d} \right\rfloor. \tag{8}$$

Obliczenia możemy rozpocząć od wyznaczenia wszystkich potrzebnych wartości funkcji Möbiusa $\mu(1),\ldots,\mu(m)$. W tym celu, za pomocą sita Eratostenesa, rozkładamy wszystkie liczby od 1 do m na czynniki pierwsze i wyznaczamy wartość μ z definicji drugiej. Operacja ta ma złożoność czasową $O(m\log m)$. Znając wartości μ, wartość $Z(a,b)$ potrafimy wyliczyć, korzystając ze wzoru (8) w czasie $O(a)$. Nie jest to jednak metoda wystarczająco szybka. Okazuje się, że można ją przyspieszyć, osiągając złożoność czasową odpowiedzi na jedno zapytanie $O(\sqrt{a})$, a całego rozwiązania: $O(n\sqrt{m} + m\log m)$, gdzie drugi składnik wynika z konieczności wyznaczenia potrzebnych wartości funkcji μ. Zobaczmy, jak się to robi.

Na początek obliczmy sumy prefiksowe funkcji μ, czyli wartości

$$F(k) = \sum_{i=1}^{k} \mu(i) \text{ dla } 1 \leqslant k \leqslant m.$$

Potrafimy to zrobić w czasie $O(m)$. Potem możemy w czasie stałym znajdować wartości: $\mu(a) + \mu(a+1) + \cdots + \mu(b) = F(b) - F(a-1)$.

Następnie wyznaczenie sumy ze wzoru (8) podzielmy na dwie części:

$$Z(a,b) \;=\; \Big(\sum_{d=1}^{\lfloor\sqrt{a}\rfloor} \mu(d) \left\lfloor \frac{a}{d} \right\rfloor \cdot \left\lfloor \frac{b}{d} \right\rfloor\Big) + \Big(\sum_{d=\lfloor\sqrt{a}\rfloor+1}^{a} \mu(d) \left\lfloor \frac{a}{d} \right\rfloor \cdot \left\lfloor \frac{b}{d} \right\rfloor\Big). \tag{9}$$

Składniki pierwszej sumy wyznaczymy bezpośrednio, jak w metodzie opisanej na początku tego rozdziału. Natomiast przy obliczeniach dla $d > \lfloor\sqrt{a}\rfloor$ posłużymy się sprytniejszym algorytmem. Zauważmy, że dla $d > \sqrt{a}$, zarówno $\lfloor \frac{a}{d} \rfloor < \sqrt{a}$, jak i $\lfloor \frac{b}{d} \rfloor < \sqrt{b} \leqslant \sqrt{a}$. To oznacza, że istnieje tylko $O(\sqrt{a})$ różnych wartości każdego z tych ilorazów. Zbadajmy najpierw $\lfloor \frac{a}{d} \rfloor$ (z drugim ilorazem postąpimy analogicznie). Dla każdej liczby $e = 1,\ldots,\lfloor\sqrt{a}\rfloor$ znajdźmy przedział całkowitych wartości d, dla których $\lfloor \frac{a}{d} \rfloor = e$:

$$e \leqslant \frac{a}{d} < e+1,$$

czyli

$$de \leqslant a < d(e+1).$$

Stąd $d \leqslant \lfloor \frac{a}{e} \rfloor$ oraz $d > \lfloor \frac{a}{e+1} \rfloor$ i ostatecznie otrzymujemy zależność

$$d \in \left[\left\lfloor \frac{a}{e+1} \right\rfloor + 1, \left\lfloor \frac{a}{e} \right\rfloor\right]. \tag{10}$$

Korzystając z (10), możemy podzielić cały przedział $[\lfloor\sqrt{a}\rfloor + 1,\ldots,a]$ na $O(\sqrt{a})$ parami rozłącznych podprzedziałów, w których wyrażenie $\lfloor \frac{a}{d} \rfloor$ ma wartość stałą. W podobny sposób możemy podzielić ten przedział na $O(\sqrt{a})$ podprzedziałów, w których z kolei wyrażenie $\lfloor \frac{b}{d} \rfloor$ jest stałe. Następnie możemy połączyć oba te podziały w jeden gęstszy podział, tak aby w poszczególnych przedziałach zarówno $\lfloor \frac{a}{d} \rfloor$, jak i $\lfloor \frac{b}{d} \rfloor$, było stałe.

Przykład 1 Niech $a = 18$, $b = 15$. Wówczas $\lfloor\sqrt{a}\rfloor = 4$. W poniższej tabelce są przedstawione wartości $\lfloor \frac{a}{d} \rfloor$ dla $d \in [1,18]$:

d	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$\lfloor \frac{18}{d} \rfloor$	18	9	6	4	3	3	2	2	2	1	1	1	1	1	1	1	1	1

Na podstawie wartości w tabelce możemy dokonać następującego podziału $[\lfloor\sqrt{a}\rfloor + 1, a] = [5,18]$ dla a:

- w przedziale $[5,6]$ zachodzi $\lfloor \frac{a}{d} \rfloor = 3$,
- w przedziale $[7,9]$ zachodzi $\lfloor \frac{a}{d} \rfloor = 2$,
- w przedziale $[10,18]$ zachodzi $\lfloor \frac{a}{d} \rfloor = 1$.

Zauważmy, że podział ten jest identyczny z tym, jaki otrzymalibyśmy za pomocą zależności (10):

- $\lfloor\frac{18}{d}\rfloor = 3$ w przedziale $[\lfloor\frac{18}{4}\rfloor+1, \lfloor\frac{18}{3}\rfloor] = [5,6]$,
- $\lfloor\frac{18}{d}\rfloor = 2$ w przedziale $[\lfloor\frac{18}{3}\rfloor+1, \lfloor\frac{18}{2}\rfloor] = [7,9]$,
- $\lfloor\frac{18}{d}\rfloor = 2$ w przedziale $[\lfloor\frac{18}{2}\rfloor+1, \lfloor\frac{18}{1}\rfloor] = [10,18]$.

Podobnie możemy skonstruować tabelkę dla b:

d	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\lfloor\frac{15}{d}\rfloor$	15	7	5	3	3	2	2	1	1	1	1	1	1	1	1

Widoczny w niej podział przedziału $[5,18]$ dla b to:

- w przedziale $[5,5]$ zachodzi $\lfloor\frac{b}{d}\rfloor = 3$
- w przedziale $[6,7]$ zachodzi $\lfloor\frac{b}{d}\rfloor = 2$,
- w przedziale $[8,15]$ zachodzi $\lfloor\frac{b}{d}\rfloor = 1$.

Połączenie podziałów dla a oraz dla b daje podział:

- w przedziale $[5,5]$ mamy $\lfloor\frac{a}{d}\rfloor = 3$ oraz $\lfloor\frac{b}{d}\rfloor = 3$,
- w przedziale $[6,6]$ mamy $\lfloor\frac{a}{d}\rfloor = 3$ oraz $\lfloor\frac{b}{d}\rfloor = 2$,
- w przedziale $[7,7]$ mamy $\lfloor\frac{a}{d}\rfloor = 2$ oraz $\lfloor\frac{b}{d}\rfloor = 2$,
- w przedziale $[8,9]$ mamy $\lfloor\frac{a}{d}\rfloor = 2$ oraz $\lfloor\frac{b}{d}\rfloor = 1$,
- w przedziale $[10,15]$ mamy $\lfloor\frac{a}{d}\rfloor = 1$ oraz $\lfloor\frac{b}{d}\rfloor = 1$,
- w przedziale $[16,18]$ iloczyn $\lfloor\frac{a}{d}\rfloor \cdot \lfloor\frac{b}{d}\rfloor$ ma wartość 0.

■

Liczba podprzedziałów w gęstszym podziale jest również rzędu $O(\sqrt{a})$, gdyż początki i końce podprzedziałów są w nim wyznaczone przez początki i końce podprzedziałów z osobnych podziałów dla a oraz dla b. Znalezienie podziału na podstawie wzorów (10) dla a i b jest dosyć techniczne, lecz niezbyt skomplikowane i może być wykonane w czasie $O(\sqrt{a})$. Oznaczmy znaleziony podział $[l_1,r_1] \cup \cdots \cup [l_s,r_s]$. Teraz możemy wyliczyć drugą sumę z (9). Ponieważ w każdym podprzedziale $[l_i,r_i]$ wyrażenie $\lfloor\frac{a}{d}\rfloor \cdot \lfloor\frac{b}{d}\rfloor$ ma wartość stałą (możemy przyjąć, na przykład, $\lfloor\frac{a}{l_i}\rfloor \cdot \lfloor\frac{b}{l_i}\rfloor$), więc

$$\sum_{d=\lfloor\sqrt{a}\rfloor+1}^{a} \mu(d)\left\lfloor\frac{a}{d}\right\rfloor \cdot \left\lfloor\frac{b}{d}\right\rfloor = \sum_{i=1}^{s}\sum_{d\in[l_i,r_i]} \mu(d)\left\lfloor\frac{a}{d}\right\rfloor \cdot \left\lfloor\frac{b}{d}\right\rfloor = \sum_{i=1}^{s}(F(r_i)-F(l_i-1)) \cdot \left\lfloor\frac{a}{l_i}\right\rfloor \cdot \left\lfloor\frac{b}{l_i}\right\rfloor.$$

(przypomnijmy, że przez F oznaczyliśmy sumy prefiksowe funkcji Möbiusa). W ten sposób zakończyliśmy konstrukcję algorytmu wyznaczania odpowiedzi na zapytanie $Z(a,b)$ w złożoności czasowej $O(\sqrt{m})$.

Implementacja przedstawionej metody znajduje się w pliku `zap1.cpp`.

Testy

Zadanie było sprawdzane na 15 zestawach danych testowych. Poniżej przedstawiona została tabela z opisami testów, w której n oznacza liczbę zapytań w teście, a parametry m_1 i m_2 wyliczone są według wzorów:

$$\begin{aligned} m_1 &= \max\{\max(a,b) \mid (a,b) \in Q\}, \\ m_2 &= \max\{\min(a,b) \mid (a,b) \in Q\}. \end{aligned}$$

Nazwa	**n**	**m$_1$**	**m$_2$**	**Opis**
zap1.in	10	40	34	test losowy
zap2.in	37	98	63	test losowy
zap3.in	100	200	185	test losowy
zap4.in	12 500	3 125	100	test losowy

Nazwa	**n**	**m_1**	**m_2**	**Opis**
zap5.in	49 999	8 333	8304	test losowy
zap6.in	50 000	50 000	100	test losowy
zap7.in	100	49 842	49 768	test losowy
zap8.in	3124	10 000	9 796	test losowy
zap9.in	12 501	16 666	16 506	test losowy
zap10.in	50 000	50 000	49 953	test losowy
zap11.in	49 729	50 000	49 778	do pokrycia Q potrzeba $\sqrt{n}$ ścieżek
zap12.in	50 000	50 000	50 000	Q jest zbiorem $\{(50\,000, x) : 1 \leqslant x \leqslant 50\,000\}$
zap13.in	49 770	49 980	49 980	liczby występujące w zapytaniach mają dużo dzielników pierwszych
zap14.in	50 000	50 000	50 000	liczby występujące w zapytaniach mają duże największe wspólne dzielniki
zap15.in	50 000	50 000	43 630	liczby występujące w zapytaniach to liczniki i mianowniki pewnych ułamków z ciągu Fareya (więcej o ciągu Fareya można przeczytać np. w [35])

Zawody II stopnia

opracowania zadań

Zbigniew Czech **Piotr Stańczyk**
Treść zadania, Opracowanie **Program**

OI, Etap II, dzień próbny, 06.02.2007

Grzbiety i doliny

Bajtazar lubi wędrować po górach. Podczas swoich wędrówek odwiedza wszystkie okoliczne grzbiety i doliny. Dlatego, aby dobrze zaplanować czas, musi wiedzieć, ile grzbietów oraz dolin znajduje się na obszarze, po którym będzie wędrował. Twoim zadaniem jest pomóc Bajtazarowi.

Bajtazar dostarczył Ci mapę obszaru, który wybrał jako cel swojej kolejnej wyprawy. Mapa ma kształt kwadratu o wymiarach $n \times n$*. Dla każdego pola* (i,j) *tego kwadratu (dla* $i,j \in \{1,\ldots,n\}$*) podana jest jego wysokość* $w_{(i,j)}$*.*

Mówimy, że pola sąsiadują ze sobą, jeżeli mają wspólny bok lub wierzchołek (tzn. pole (i,j) *sąsiaduje z polami* $(i-1,j-1)$*,* $(i-1,j)$*,* $(i-1,j+1)$*,* $(i,j-1)$*,* $(i,j+1)$*,* $(i+1,j-1)$*,* $(i+1,j)$*,* $(i+1,j+1)$*, o ile pola o takich współrzędnych mieszczą się na mapie).*

Mówimy, że zbiór pól S *tworzy grzbiet (dolinę), jeżeli:*

- *wszystkie pola z* S *mają tę samą wysokość,*
- *zbiór* S *stanowi spójny kawałek mapy (z każdego pola ze zbioru* S *można przejść do każdego innego pola z* S*, przechodząc tylko z sąsiedniego pola na sąsiednie i nie wychodząc poza zbiór* S*),*
- *jeśli* $s \in S$ *oraz pole* $s' \notin S$ *sąsiaduje z* s*, to* $w_s > w_{s'}$ *(dla grzbietu) lub* $w_s < w_{s'}$ *(dla doliny).*

W szczególności, w przypadku, gdy cały obszar przedstawiony na mapie ma tę samą wysokość, jest on jednocześnie grzbietem i doliną.

Twoim zadaniem jest wyznaczyć liczbę grzbietów oraz dolin dla krajobrazu opisanego przez otrzymaną mapę.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis mapy,*
- *obliczy liczbę grzbietów i dolin dla krajobrazu opisanego przez tę mapę,*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita n $(2 \leqslant n \leqslant 1\,000)$ *oznaczająca rozmiar mapy. W każdym z kolejnych* n *wierszy znajduje się opis kolejnego wiersza mapy. W* $i+1$*-szym wierszu (dla* $i \in \{1,\ldots,n\}$*) znajduje się* n *liczb całkowitych* $w_{(i,1)},\ldots,w_{(i,n)}$ $(0 \leqslant w_{(i,j)} \leqslant 1\,000\,000\,000)$*, pooddzielanych pojedynczymi odstępami. Są to wysokości kolejnych pól w* i*-tym wierszu mapy.*

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać dwie liczby całkowite oddzielone pojedynczym odstępem — liczbę grzbietów, a następnie liczbę dolin dla krajobrazu opisanego przez mapę.

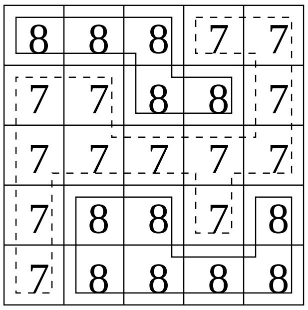
Przykład

Dla danych wejściowych:

```
5
8 8 8 7 7
7 7 8 8 7
7 7 7 7 7
7 8 8 7 8
7 8 8 8 8
```

poprawnym wynikiem jest:

```
2 1
```

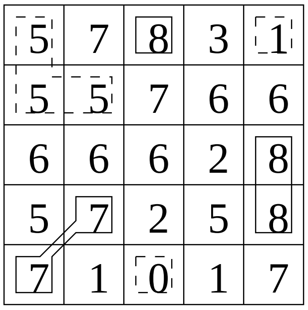


Natomiast dla danych:

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

poprawnym wynikiem jest:

```
3 3
```



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

Na powyższych rysunkach zaznaczone są grzbiety (linia ciągła) i doliny (linia przerywana).

Rozwiązanie

Rozwiązanie wzorcowe

Na wstępie wprowadźmy pojęcie *osiągalności pola*, które będzie pomocne w dalszej analizie.

Definicja 1 *Pole k jest osiągalne z pola p*, jeżeli ma taką samą wysokość co pole p oraz jeżeli do pola k można dojść z pola p, poruszając się tylko pomiędzy sąsiednimi polami o tej samej wysokości.
W szczególności, pole p jest osiągalne z pola p.

W rozwiązaniu wzorcowym liczbę grzbietów oraz dolin wyznaczamy w dwóch niezależnych fazach. Poniżej przedstawiamy sposób wyznaczania liczby dolin; podobnie można wyznaczyć liczbę grzbietów. Zadanie zinterpretujemy w języku grafów. Każde pole mapy Bajtazara potraktujemy jako wierzchołek grafu G. Krawędzie w tym grafie reprezentują możliwe ruchy pomiędzy sąsiednimi polami. Z każdego pola (wierzchołka grafu) nieleżącego na brzegu mapy wychodzi zatem osiem krawędzi. Dolina w tej interpretacji jest maksymalnym, spójnym (jednokawałkowym) zbiorem pól o tej samej wysokości w, otoczonym polami o wysokości większej od w.

Zastanówmy się, w jaki sposób można stwierdzić, czy dane pole k należy do doliny. W tym celu można wyznaczyć wszystkie pola osiągalne z pola k, a następnie sprawdzić, czy wszystkie one sąsiadują z polami o nie mniejszej wysokości. Jeśli którekolwiek pole sąsiaduje z polem o wysokości mniejszej, to pole k (oraz każde inne pole osiągalne z pola k) nie należy do doliny. W przeciwnym razie, pole k oraz wszystkie pola z niego osiągalne należą do jednej, tej samej doliny. Co więcej, nie ma dalszych pól należących do tej doliny, bo gdyby takowe istniały, to byłyby osiągalne z pola k.

Powstaje pytanie, w jaki sposób efektywnie przeprowadzić opisaną wyżej procedurę. Okazuje się, że wystarczy w tym celu zastosować przeszukiwanie grafu G wszerz. Przeszukiwanie rozpoczynamy w wierzchołku k i odwiedzamy wszystkie osiągalne wierzchołki o tej samej wysokości co wierzchołek wyjściowy, odnajdując obszar *potencjalnej doliny* zawierającej k. Jeśli jakakolwiek krawędź wychodząca z wyznaczonego obszaru dochodzi do wierzchołka o mniejszej wysokości od wierzchołka k, to potencjalna dolina nie jest w rzeczywistości doliną i ani k, ani żaden z jej pozostałych wierzchołków nie należy do doliny. W przeciwnym razie potencjalna dolina okazuje się być rzeczywiście doliną. Dla każdego zbioru pól osiągalnych wystarczy wykonać tylko jedno przeszukiwanie — liczba dolin jest równa liczbie wykonanych przeszukiwań, dla których znaleziono rzeczywistą dolinę.

Ponieważ każdy wierzchołek w grafie G ma ograniczony stopień, nie większy niż osiem, więc sumaryczny czas wykonania przeszukiwań jest liniowy względem liczby wierzchołków. Tym samym otrzymujemy algorytm o złożoności $O(n^2)$, zapisany w plikach `grz.cpp` i `grz1.pas`.

Rozwiązanie alternatywne

Zadanie można także rozwiązać, używając struktury danych *Find-Union*[1] zamiast reprezentacji grafowej. Struktura umożliwia efektywny podział pól na rozłączne zbiory pól wzajemnie osiągalnych. W tym celu na początku działania algorytmu każde pole mapy reprezentowane jest jako jednoelementowy zbiór. Następny krok algorytmu polega na przeanalizowaniu wszystkich par sąsiadujących pól oraz połączeniu zbiorów, do których należą pola o tej samej wysokości. Na koniec wystarczy dla każdego uzyskanego zbioru sprawdzić, czy wszystkie pola należące do niego sąsiadują z polami nie niższymi od siebie — jeśli tak, to rozpatrywany zbiór stanowi dolinę. Oczywiście test sprawdzający, czy obszar jest grzbietem, można zrealizować analogicznie. Uzyskany w ten sposób algorytm ma złożoność $O(n^2 \log^* n)$. Rozwiązanie zostało zapisane w pliku `grzs1.cpp`.

---

[1] Struktury *Find-Union* służą do przechowywania elementów pogrupowanych w rozłączne zbiory i efektywnego wykonywania operacji: połączenia zbiorów (*Union*) oraz zidentyfikowania zbioru, do którego należy element (*Find*). Więcej na ich temat można przeczytać w książkach [14] i [19].

Rozwiązanie niepoprawne

Głównym źródłem rozwiązań niepoprawnych było stosowanie rekurencyjnego przeszukiwania grafu, na przykład za pomocą przeszukiwania w głąb. Podejście to może spowodować przepełnienie stosu. Jest ono wysoce prawdopodobne dla testów typu „wąż” (patrz dalej). Rozwiązanie takie zostało zaimplementowane w pliku `grzb1.cpp`.

Testy

Testy do zadania zostały wygenerowane losowo, przy użyciu pięciu niezależnych metod:

- **płaski** — generator testów zawierających płaski obszar o zadanej wielkości oraz wysokości,
- **wąż** — generuje test o zadanej wielkości, umieszcza w nim „węża” (ciąg pól o tej samej wysokości i długości rzędu $O(n^2)$) oraz otacza go polami o losowych wysokościach,
- **losowy** — generuje planszę o zadanej wielkości i losowych wysokościach pól,
- **szachownica** — generuje planszę o zadanej wielkości, zawierającą dwie różne wysokości pól ułożone we wzór szachownicy,
- **losowy obszar** — generuje planszę o zadanej wielkości, wypełniając ją losowymi spójnymi obszarami o zadanej maksymalnej wielkości oraz wysokości.

Poniższa lista zawiera podstawowe informacje dotyczące testów. Przez *n* została oznaczona wielkość mapy, a przez *h* — maksymalna wysokość pól. W ostatniej kolumnie podany został rodzaj generatora użyty do stworzenia testu.

Nazwa	n	h	Opis
grz1a.in	8	2	szachownica
grz1b.in	10	100	wąż
grz2a.in	15	100	płaski
grz2b.in	16	267	szachownica
grz2c.in	13	20	losowy
grz3a.in	20	9852	płaski
grz3b.in	50	29	wąż
grz4.in	100	96 500	losowy
grz5.in	120	9 999	losowy obszar
grz6.in	200	1 000 000	losowy obszar
grz7.in	400	9 000 000	losowy
grz8a.in	800	100 000 000	losowy
grz8b.in	800	1 000 000	losowy obszar
grz9a.in	1 000	854	szachownica
grz9b.in	750	1 000 000 000	losowy
grz10a.in	1 000	50	wąż
grz10b.in	1 000	1 000 000 000	płaski

Marcin Kubica
Treść zadania, Opracowanie

Marek Żylak
Program

OI, Etap II, dzień pierwszy, 07.02.2007

Powódź

Bajtogród, stolica Bajtocji, to malownicze miasto położone w górskiej kotlinie. Niestety ostatnie ulewne deszcze spowodowały powódź — cały Bajtogród znalazł się pod wodą. Król Bajtocji, Bajtazar, zebrał swoich doradców, w tym i Ciebie, aby radzić, jak ratować Bajtogród. Postanowiono sprowadzić pewną liczbę pomp, rozmieścić je na zalanym terenie i za ich pomocą osuszyć Bajtogród. Bajtazar poprosił Cię o określenie minimalnej liczby pomp wystarczającej do osuszenia Bajtogrodu.

Otrzymałeś mapę miasta wraz z kotliną, w której jest położone. Mapa ma kształt prostokąta podzielonego na $m \times n$ jednostkowych kwadratów. Dla każdego kwadratu jednostkowego mapa określa jego wysokość nad poziomem morza oraz czy należy on do Bajtogrodu czy nie. Cały teren przedstawiony na mapie znajduje się pod wodą. Ponadto jest on otoczony dużo wyższymi górami, które uniemożliwiają odpływ wody. Osuszenie terenów, które nie należą do Bajtogrodu, nie jest konieczne.

Każdą z pomp można umieścić na jednym z kwadratów jednostkowych przedstawionych na mapie. Będzie ona wypompowywać wodę aż do osuszenia kwadratu, na którym się znajduje. Osuszenie dowolnego kwadratu pociąga za sobą obniżenie poziomu wody lub osuszenie kwadratów jednostkowych, z których woda jest w stanie spłynąć do kwadratu, na którym ustawiono pompę. Woda może bezpośrednio przepływać tylko między kwadratami, których rzuty na poziomą płaszczyznę mają wspólną krawędź.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis mapy,*
- *wyznaczy minimalną liczbę pomp potrzebnych do osuszenia Bajtogrodu,*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie liczby całkowite m i n, oddzielone pojedynczym odstępem, $1 \leqslant m,n \leqslant 1000$. Kolejnych m wierszy zawiera opis mapy. Wiersz $i+1$-szy opisuje i-ty pas kwadratów jednostkowych na mapie. Zawiera on n liczb całkowitych $x_{i,1}, x_{i,2}, \ldots, x_{i,n}$, pooddzielanych pojedynczymi odstępami, $-1000 \leqslant x_{i,j} \leqslant 1000$, $x_{i,j} \neq 0$. Liczba $x_{i,j}$ opisuje j-ty kwadrat w i-tym wierszu. Grunt w tym kwadracie znajduje się na wysokości $|x_{i,j}|$. Jeżeli $x_{i,j} > 0$, to kwadrat ten znajduje się na terenie Bajtogrodu, a w przeciwnym przypadku znajduje się poza miastem. Teren Bajtogrodu nie musi stanowić spójnej całości, może być podzielony na kilka oddzielonych od siebie części.

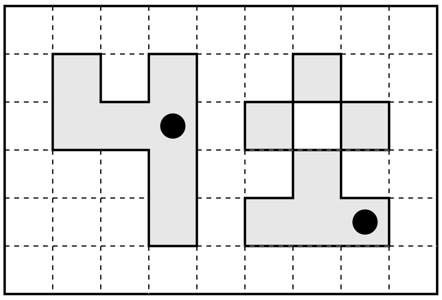
Wyjście

Twój program powinien wypisać na standardowe wyjście jedną liczbę całkowitą — minimalną liczbę pomp potrzebnych do osuszenia Bajtogrodu.

Przykład

Dla danych wejściowych:

```
6 9
-2 -2 -1 -1 -2 -2 -2 -12 -3
-2 1 -1 2 -8 -12 2 -12 -12
-5 3 1 1 -12 4 -6 2 -2
-5 -2 -2 2 -12 -3 4 -3 -1
-5 -6 -2 2 -12 5 6 2 -1
-4 -8 -8 -10 -12 -8 -6 -6 -4
```



poprawnym wynikiem jest:

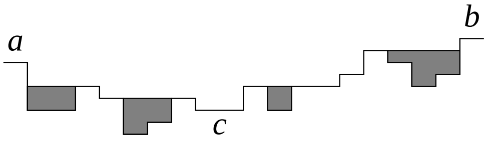
```
2
```

Na rysunku przedstawiono teren Bajtogrodu oraz przykładowe rozmieszczenie pomp.

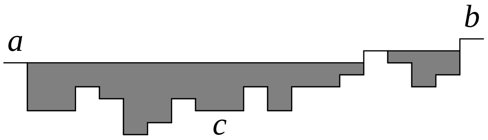
Rozwiązanie

Analiza problemu

Zaczniemy od prostej obserwacji. Niech a, b i c będą takimi kwadratami jednostkowymi, że pompa umieszczona w kwadracie c osusza zarówno kwadrat a, jak i b. Oznaczmy wysokości tych kwadratów, odpowiednio, przez h_a, h_b i h_c. Dodatkowo możemy założyć, że $h_a \leqslant h_b$ (jeśli tak nie jest, to zamieniamy oznaczenia kwadratów a i b). Wówczas pompa umieszczona w kwadracie a osusza kwadrat b.



Rys. 1: Pompa umieszczona w kwadracie c osusza kwadraty a i b.



Rys. 2: Pompa umieszczona w kwadracie a osusza kwadrat b.

Dzieje się tak, gdyż z a do c musi istnieć droga, którą woda spływa z a do c i na której wysokość terenu nie przekracza h_a. Podobnie, z b do c musi istnieć droga, którą woda spływa z b do c i na której wysokość terenu nie przekracza h_b. Tak więc istnieje droga z b do a, na której wysokość terenu nie przekracza h_b i którą woda może spłynąć z kwadratu b do pompy umieszczonej w kwadracie a.

Dzięki tej obserwacji możemy pokazać następujący fakt:

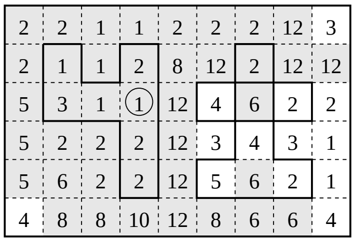
Fakt 1 *Niech a będzie najniżej położonym kwadratem Bajtogrodu — dowolnie wybranym, jeśli istnieje wiele takich kwadratów. Istnieje takie optymalne ustawienie pomp, w którym jedna z pomp znajduje się w kwadracie a.*

Dowód Rozważmy pewne optymalne ustawienie pomp i załóżmy, że w ustawieniu tym w kwadracie a nie stoi żadna pompa. Niech c będzie kwadratem, na którym znajduje się pompa osuszająca kwadrat a. (Jeśli jest kilka takich pomp, to wybieramy dowolną z nich.) Niech b będzie dowolnym kwadratem na terenie Bajtogrodu, osuszanym przez pompę znajdującą się w kwadracie c. Przenieśmy pompę z kwadratu c do kwadratu a — z poprzednio uczynionej obserwacji wynika, że przeniesiona pompa nadal osusza kwadrat b. Powyższe uzasadnienie można zastosować do dowolnie wybranego kwadratu b, z czego wynika, że pompa po przeniesieniu osusza te same kwadraty Bajtogrodu, co poprzednio. W ten sposób uzyskaliśmy optymalne ustawienie pomp, w którym w kwadracie a stoi pompa. ■

Fakt ten pozwala sformułować następujący zachłanny algorytm rozmieszczania pomp:

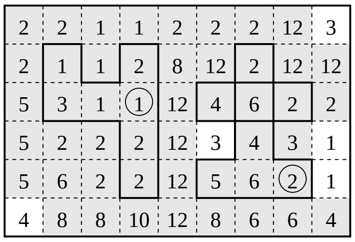
- umieszczamy pompę w najniżej położonym kwadracie Bajtogrodu — dowolnie wybranym, jeśli jest ich wiele,
- znajdujemy wszystkie kwadraty należące do Bajtogrodu, które osusza pompa,
- zmniejszamy problem, usuwając z terenu Bajtogrodu osuszone kwadraty,
- umieszczamy kolejną pompę itd.

Poniższe rysunki ilustrują zastosowanie tego algorytmu dla przykładowych danych z treści zadania.



2	2	1	1	2	2	2	12	3
2	1	1	2	8	12	2	12	12
5	3	1	①	12	4	6	2	2
5	2	2	2	12	3	4	3	1
5	6	2	2	12	5	6	2	1
4	8	8	10	12	8	6	6	4

Pierwsza pompa jest ustawiana na jednym z kwadratów Bajtogrodu na wysokości 1. Osuszony przez nią teren zaznaczono na szaro.



2	2	1	1	2	2	2	12	3
2	1	1	2	8	12	2	12	12
5	3	1	①	12	4	6	2	2
5	2	2	2	12	3	4	3	1
5	6	2	2	12	5	6	②	1
4	8	8	10	12	8	6	6	4

Druga pompa jest ustawiana na wysokości 2. Osusza ona pozostały teren Bajtogrodu.

Rozwiązanie nieoptymalne

Prosta implementacja przedstawionego w poprzednim punkcie algorytmu może polegać na przeglądaniu kwadratów należących do Bajtogrodu w kolejności rosnących wysokości oraz symulowaniu obniżania się poziomu wody po ustawieniu każdej pompy. Do symulowania skutków ustawienia pojedynczej pompy możemy zastosować technikę podobną do algorytmu Dijkstry: kwadraty jednostkowe traktujemy jak wierzchołki grafu, krawędziami łączymy kwadraty sąsiadujące ze sobą — jednak zamiast rozpatrywać wierzchołki w kolejności rosnącej odległości od pompy, rozpatrujemy je w kolejności rosnącej wysokości, na którą opada woda. W algorytmie Dijkstry do przechowywania dostępnych w kolejnym kroku wierzchołków wykorzystywana jest kolejka priorytetowa. W naszym przypadku wysokości, według których wybieramy kwadraty, są liczbami całkowitymi od 1 do 1 000 — można więc do ich zapisania użyć tablicy zbiorów (lub jakichkolwiek kolekcji, np. list) kwadratów.

Pojedyncze przeszukiwanie ma złożoność czasową rzędu $O(m \cdot n)$. Tak więc cały algorytm działa w czasie $O(m \cdot n \cdot p)$, gdzie p jest liczbą pomp, które należy ustawić.

Rozwiązanie optymalne

Opisany w poprzednim punkcie algorytm można usprawnić, łącząc kolejne przeszukiwania w jedną całość. W tym celu zdefiniujemy dla każdej wysokości h dwa zbiory kwadratów:

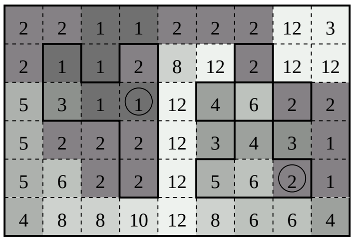
- M_h — zbiór kwadratów, które leżą na terenie Bajtogrodu i są położone na wysokości h (wszystkie zbiory M_h możemy wyznaczyć na początku, przeglądając raz całą mapę) oraz
- L_h — zbiór kwadratów leżących na wysokości h, które sąsiadują z kwadratami, na których woda opadła poniżej poziomu h (początkowo wszystkie zbiory L_h są puste; zapełniamy je w trakcie obliczeń, gdy stwierdzimy, że w sąsiedztwie kwadratu leżącego na wysokości h woda opadła poniżej tego poziomu).

Dla każdego kwadratu pamiętamy także poziom wody w tym kwadracie (początkowo, we wszystkich kwadratach woda jest na poziomie ∞).

W algorytmie przeglądamy wysokości h w rosnącej kolejności, $h = 1, 2, \ldots, 1\,000$, ustawiając w miarę potrzeby pompy i osuszając wszystkie kwadraty na danej wysokości (i być może inne przy okazji). Kwadraty należące do L_h zostały osuszone wcześniej przez pompy postawione niżej, gdyż woda spłynęła z nich do sąsiadujących kwadratów, gdzie jej poziom jest niższy. Możemy więc wyznaczyć wszystkie kwadraty, w których woda opadnie do poziomu h (osuszając je lub nie), przeszukując mapę wszerz (BFS) lub w głąb (DFS), począwszy od kwadratów należących do L_h. Gdy w trakcie przeszukiwania napotkamy na kwadrat sąsiadujący z odwiedzanym kwadratem, ale położony na wysokości $h' > h$, to wstawiamy go do zbioru $L_{h'}$.

Po wykonaniu powyższej procedury sprawdzamy, czy w zbiorze M_h pozostał nieosuszony kwadrat — jeśli tak, to należy w nim ustawić pompę i uzupełnić przeszukiwanie, zaczynając je w tym kwadracie. Postępujemy w ten sposób tak długo, aż wszystkie kwadraty ze zbioru M_h zostaną osuszone.

Dzięki temu, że wysokości h rozpatrujemy w rosnącej kolejności, każdy kwadrat będziemy odwiedzać tylko raz, wyznaczając właściwy poziom, na jaki opadnie w nim woda. W szczególności, kwadraty należące do Bajtogrodu zostaną zawsze całkowicie osuszone, natomiast pozostałe mogą co najmniej częściowo pozostać pod wodą. W rezultacie, złożoność czasowa i pamięciowa takiego algorytmu wynosi $O(m \cdot n)$.



2	2	1	1	2	2	2	12	3
2	1	1	2	8	12	2	12	12
5	3	1	①	12	4	6	2	2
5	2	2	2	12	3	4	3	1
5	6	2	2	12	5	6	②	1
4	8	8	10	12	8	6	6	4

Rys. 3: Kwadraty mapy są odwiedzane w kolejności zgodnej z poziomem, na jaki opada na nich woda (reprezentowanym przez odcienie szarości).

Jeżeli do przeszukiwania mapy zastosujemy przeszukiwanie w głąb, to oba zbiory L_h i M_h możemy trzymać na jednym stosie. Musimy wówczas zadbać, by elementy zbioru L_h znajdowały się na wierzchu stosu, a elementy zbioru M_h na dole stosu. W trakcie przeszukiwania musimy bowiem najpierw wyznaczyć kwadraty, z których woda spłynie za pośrednictwem kwadratów już osuszonych (L_h) i dopiero potem możemy przystąpić do rozstawiania nowych pomp na zalanych jeszcze kwadratach z M_h. Takie właśnie rozwiązanie zaimplementowano w plikach `pow.cpp` i `pow1.pas`.

Rozwiązanie alternatywne

Zadanie można również rozwiązać w odrobinę gorszej złożoności czasowej niż optymalna, używając struktury *Find-Union*[1]. Wyobraźmy sobie proces odwrotny do osuszania — przypuśćmy, że poziom wody stopniowo podnosi się, zalewając coraz większe obszary.

Niech a będzie kwadratem leżącym na terenie Bajtogrodu na wysokości h_a. Przyjrzyjmy się, jak wygląda „jezioro”, które zaleje kwadrat a, gdy poziom wody przekroczy h_a (ale będzie jeszcze niższy niż $h_a + 1$). Gdyby gdziekolwiek na terenie zalanym przez to jezioro znajdowała się pompa, kwadrat a zostałby osuszony. Obserwacja ta prowadzi do następującego algorytmu:

- sortujemy kwadraty według wysokości — możemy tu zastosować sortowanie przez zliczanie, działające w czasie $O(m \cdot n)$;
- następnie przeglądamy kwadraty w kolejności niemalejących wysokości symulując podnoszenie się poziomu wody i śledzimy zasięg jezior przy aktualnym poziomie.

Przeglądając kwadrat a, rozważamy, co się z nim dzieje w momencie, gdy poziom wody przekroczy h_a. Początkowo dla kwadratu a tworzymy osobny element — jezioro jednostkowe, które łączymy z jeziorami odpowiadającymi tym sąsiadom kwadratu a, którzy są położeni nie wyżej niż on. W ten sposób otrzymujemy element reprezentujący jezioro pokrywające kwadrat a. Oczywiście, w miarę rozpatrywania coraz wyżej położonych kwadratów (czyli podnoszenia się poziomu wody), jeziora mogą łączyć się — do śledzenia tego procesu przydaje się nam struktura *Find-Union*.

Dodatkowo, dla każdego jeziora pamiętamy, czy na jego terenie znajduje się jakaś pompa. Po przejrzeniu wszystkich kwadratów znajdujących się na wysokości h, przeglądamy ponownie te z nich, które leżą na terenie Bajtogrodu. Dla każdego takiego kwadratu sprawdzamy, czy na terenie pokrywającego go jeziora znajduje się pompa — jeżeli nie, to ustawiamy pompę w tym kwadracie i odnotowujemy, że w odpowiednim jeziorze jest już pompa. Łącząc dwa jeziora, przyjmujemy, że jeżeli na terenie któregokolwiek z nich znajduje się pompa, to na terenie jeziora powstałego w wyniku połączenia też znajduje się pompa.

Łączna liczba kwadratów to $m \cdot n$, czyli *zamortyzowany*[2] czas jednej operacji na strukturze *Find-Union* wynosi $O(\log^*(m \cdot n))$. Tak więc, łączny czas działania całego algorytmu to $O(m \cdot n \cdot \log^*(m \cdot n))$. W praktyce, złożoność tego algorytmu nie odbiega znacząco od algorytmu wzorcowego. Rozwiązanie to zaimplementowano w pliku `pow2.cpp`.

Ciekawe rozwiązanie prawie poprawne

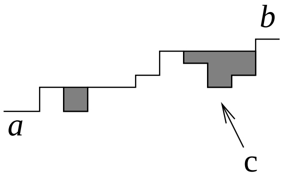
Na koniec prezentujemy jeszcze jedno ciekawe rozwiązanie, bardzo zbliżone do opisanych już rozwiązań nieoptymalnego i optymalnego. Przeglądamy wysokości pól w rosnącej kolejności $h = 1, 2, \ldots, 1000$ i każdorazowo w przypadku napotkania nieosuszonego pola Bajtogrodu stawiamy w nim pompę. Różnica między tym podejściem, a wcześniej zaprezentowanymi, jest taka, że osuszanie symulujemy za pomocą algorytmu DFS, poruszając się wyłącznie „pod górę”. Innymi słowy, jeżeli w pewnym momencie algorytmu znajdujemy się w polu o wysokości h, to przeszukiwanie kontynuujemy w nieosuszonych jeszcze kwadratach sąsiednich o wysokościach nie mniejszych niż h. W ten sposób mamy gwarancję, że zawsze w momencie odwiedzenia pola zostaje ono całkowicie osuszone. To oznacza, że każde pole zostanie odwiedzone w algorytmie dokładnie raz, co daje złożoność czasową opisanego algorytmu $O(m \cdot n)$. Rozwiązanie to zaimplementowano w pliku `powb3.cpp`.

Opisane podejście nie jest niestety poprawne, o czym można się przekonać, analizując chociażby rysunek 1. Faktycznie, czasem może się opłacać odwiedzić pole, nie osuszając go całkowicie, jeżeli jest to pole leżące poza Bajtogrodem. Okazuje się jednak, że w przypadku, gdy *każde pole kotliny należy do Bajtogrodu* (czyli ma on $m \cdot n$ pól), to zaprezentowany algorytm jest poprawny! Uzasadnimy to przez sprowadzenie do sprzeczności. Załóżmy, że kwadrat b jest *najniższym* kwadratem, który został osuszony przez pompę ustawioną w naszym algorytmie w jakimś kwadracie a, ale którego osuszenie zostało „przeoczone” ze względu na ograniczony sposób przeszukiwania kwadratów. Dla uproszczenia dowodu załóżmy, że wszystkie kwadraty w kotlinie mają różne wysokości. Ponieważ pompa w a osusza b, to istnieje ścieżka z a do b, na której wysokości wszystkich pól są nie większe niż wysokość kwadratu b. Rozważana ścieżka musi zawierać pewne obniżenia wysokości, gdyż w przeciwnym przypadku kwadrat b zostałby uznany przez nasz algorytm za osuszony (patrz rys. 4). Niech c oznacza pierwszy kwadrat na ścieżce, od którego biegnie ona już tylko pod górę (kolejne odwiedzane kwadraty mają wysokości nie mniejsze niż poprzednie). Ponieważ przez b oznaczyliśmy najniższe „przeoczone” pole, to kwadrat c musiał zostać uznany za osuszony w naszym algorytmie. Skoro tak, to w momencie, gdy przeszukiwanie dotarło do kwadratu c, osuszając go, musiało ono być kontynuowane aż do dotarcia kwadratu b, co stanowi sprzeczność z założeniem o przeoczeniu b.

---

[1] Struktury *Find-Union* służą do przechowywania elementów pogrupowanych w rozłączne zbiory i efektywnego wykonywania operacji: połączenia zbiorów (*Union*) oraz zidentyfikowania zbioru, do którego należy element (*Find*). Więcej na ich temat można przeczytać w książkach [14] i [19].

[2] O zamortyzowanym sposobie liczenia czasu operacji można przeczytać w tych samych książkach, w których znajduje się opis struktur *Find-Union* — [14], [19].



Rys. 4: Ścieżka z kwadratu *a* do *b* zawierająca obniżenia.

Rodzi się więc pytanie, czy opisany algorytm może się do czegokolwiek przydać w przypadku ogólnym, kiedy niekoniecznie wszystkie kwadraty należą do Bajtogrodu. W takiej sytuacji można zastosować rozwiązanie, będące hybrydą opisanego algorytmu i zaprezentowanego wcześniej rozwiązania nieoptymalnego: jeżeli w trakcie przeszukiwania (symulacji osuszania) natrafiamy na pole Bajtogrodu, to odwiedzamy je tylko wówczas, gdy spowoduje to całkowite jego osuszenie; jeżeli natomiast natrafiamy na pole spoza Bajtogrodu, to odwiedzamy je pod warunkiem, że spowoduje to obniżenie dotychczasowego poziomu wody nad tym polem. Dowód poprawności opisanego podejścia pozostawiamy Czytelnikowi, po szczegóły odsyłając równocześnie do implementacji w pliku `pows5.cpp`. Na koniec wspomnijmy, że mimo pesymistycznej złożoności czasowej $O(n \cdot m \cdot h_{max})$, opisane rozwiązanie świetnie spisuje się dla losowych danych testowych.

Testy

Rozwiązania zawodników były testowane na 57 testach pogrupowanych w 40 zestawów. Nie będziemy ich tu szczegółowo i z osobna opisywać, lecz poprzestaniemy jedynie na zbiorczej charakterystyce:

- testy 1–9 to nieduże testy poprawnościowe, przez które powinny przejść wszystkie rozwiązania poprawne, a nawet niektóre sprytne rozwiązania błędne; większość tych testów zawiera mapy złożone z zaledwie kilku kwadratów, tylko trzy z nich zawierają mapy w kształcie paska o wymiarach 1×500 lub $1 \times 1\,000$,
- testy 10–17 to mniejsze testy losowe, przez które przechodzą wszystkie poprawne rozwiązania (jakich się spodziewano); zawierają one od 25 do 93 000 kwadratów,
- testy 18–23 to duże testy losowe, w których teren zajmowany przez Bajtogród stanowi niewielką część terenu pokazanego na mapie; przez te testy przechodzą wszystkie przedstawione tu rozwiązania; testy te zawierają od 810 000 do 1 000 000 kwadratów,
- grupy testów 24–30 zawierają po dwa testy, jeden test w każdej grupie służy do odróżnienia rozwiązania optymalnego i alternatywnego od rozwiązań wolniejszych, a drugi test to mniejszy test sprawdzający poprawność rozwiązania; przez te testy przechodzi rozwiązanie wzorcowe i alternatywne; większe z testów w grupach zawierają od 324 900 do 1 000 000 kwadratów,
- grupy testów 31–39 zawierają po dwa testy, jeden test w każdej grupie służy do odróżnienia rozwiązania optymalnego i alternatywnego od rozwiązań wolniejszych, a drugi test to duży test losowy, w którym teren zajmowany przez Bajtogród stanowi niewielką część terenu pokazanego na mapie; przez testy te przechodzi rozwiązanie wzorcowe i alternatywne; zawierają one od 384 400 do 1 000 000 kwadratów,
- ostatnia grupa testów (40) to dwa testy maksymalnej wielkości: jeden specyficzny i jeden losowy, dobrane tak, żeby przechodziły przez nie jedynie rozwiązania optymalne i alternatywne.

OI, Etap II, dzień pierwszy, 07.02.2007

Skalniak

Wicehrabia de Bajteaux jest właścicielem znanej na całym świecie kolekcji głazów. Dotychczas utrzymywał ją w piwnicach swojego pałacu, lecz teraz postanowił wyeksponować kolekcję w swoim ogromnym ogrodzie.

Ogrody wicehrabiego mają kształt kwadratu o bokach długości 1 000 000 000 jednostek. Boki kwadratu leżą w kierunkach wschód-zachód i północ-południe. Dla każdego głazu wicehrabia wyznaczył współrzędne punktu, w którym chciałby, żeby został on umiejscowiony (współrzędne te to odległości od południowego i zachodniego boku ogrodu), a następnie przekazał te informacje służącym. Niestety jednak zapomniał im powiedzieć, w jakiej kolejności podał współrzędne poszczególnych punktów. Dokładniej, współrzędne niektórych punktów mógł podać w kolejności odcięta-rzędna, a niektórych rzędna-odcięta. Nieświadomi tego faktu służący rozmieścili głazy, zakładając, że kolejność podawania współrzędnych jest standardowa (odcięta-rzędna).

Wicehrabia postanowił zadbać o bezpieczeństwo swojej kolekcji i ogrodzić ją płotem, tworząc skalniak. Płot ze względów estetycznych powinien mieć kształt prostokąta o bokach równoległych do boków ogrodu. Wicehrabia tak rozplanował uporządkowania współrzędnych punktów, żeby łączna długość potrzebnego płotu była jak najmniejsza (czyli spośród wszystkich uporządkowań par współrzędnych podanych punktów, uporządkowanie wicehrabiego wymaga minimalnej łącznej długości płotu). Przyjmujemy przy tym, że prostokąt może mieć boki zerowej długości.

Aby nie wyszła na jaw omyłkowa realizacja planu wicehrabiego, służący muszą poprzestawiać głazy tak, by długość potrzebnego do ich ogrodzenia płotu była najmniejsza z możliwych. Każdy głaz mogą przestawić tylko tak, by jego nowe położenie miało takie same współrzędne, jak aktualne, tylko w odwrotnej kolejności. Chcą się przy tym jak najmniej napracować, gdyż głazy są ciężkie. Chcieliby więc zminimalizować łączny ciężar przestawianych głazów.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia aktualne rozmieszczenie głazów w ogrodach wicehrabiego oraz ich ciężar,*
- *wyznaczy taki sposób poprzestawiania głazów, który umożliwi ogrodzenie ich jak najkrótszym płotem, a dodatkowo zminimalizuje łączny ciężar przestawianych głazów,*
- *wypisze wynik na standardowe wyjście.*

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę całkowitą n ($2 \leqslant n \leqslant 1\,000\,000$)*, oznaczającą liczbę głazów w kolekcji wicehrabiego. Kolejnych n wierszy zawiera po trzy liczby całkowite* x_i, y_i *oraz* m_i ($0 \leqslant x_i, y_i \leqslant 1\,000\,000\,000$, $1 \leqslant m_i \leqslant 2\,000$)*, pooddzielane pojedynczymi odstępami i oznaczające współrzędne aktualnego położenia i ciężar i-tego głazu. Żadna para nieuporządkowana współrzędnych nie pojawia się na wejściu więcej niż raz.*

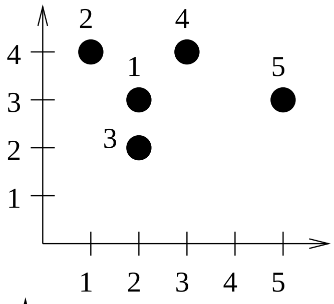
Wyjście

Pierwszy wiersz wyjścia powinien zawierać dwie liczby całkowite, oddzielone pojedynczym odstępem — najmniejszą możliwą do osiągnięcia długość płotu i minimalny ciężar kamieni, jakie trzeba przemieścić, by taki wynik osiągnąć. Drugi wiersz powinien zawierać ciąg n zer i/lub jedynek — i-ty znak powinien być jedynką, jeżeli w optymalnym rozwiązaniu przemieszczamy i-ty kamień, a zerem w przeciwnym przypadku. Jeżeli istnieje wiele poprawnych rozwiązań, Twój program powinien wypisać jedno z nich.

Przykład

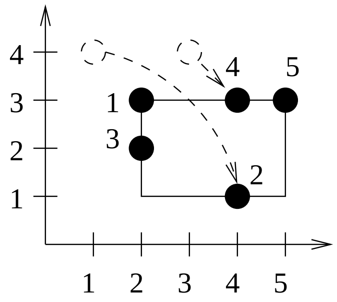
Dla danych wejściowych:

```
5
2 3 400
1 4 100
2 2 655
3 4 100
5 3 277
```



poprawnym wynikiem jest:

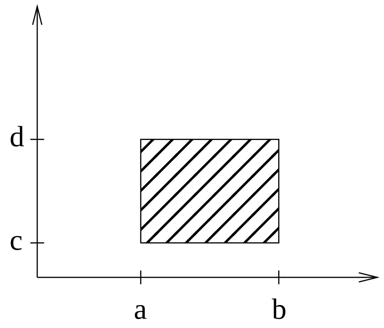
```
10 200
01010
```



Rozwiązanie

Test przydatności prostokąta

Zastanówmy się na początek nad łatwiejszym problemem niż postawiony w treści zadania: jak dla określonego położenia prostokątnego płotu sprawdzić, czy da się poprzestawiać głazy tak, by znalazły się wewnątrz ogrodzenia. Jeżeli takie przestawienie jest możliwe, to chcielibyśmy także umieć wyznaczyć jego minimalny koszt. Położenie płotu będziemy odtąd identyfikować z jego rzutami na każdą z osi, to znaczy czwórką liczb a, b, c, d, takich że prostokąt ogrodzony płotem jest iloczynem kartezjańskim $[a, b] \times [c, d]$.



Parametry opisujące prostokątny płot

Prostokąt $[a, b] \times [c, d]$ nazwiemy *przydatnym*, jeżeli da się w nim umieścić wszystkie głazy (przestawiając w razie konieczności pewne z nich w opisany w treści zadania sposób). Przydatność prostokąta można w prosty sposób sprawdzić, wyznaczając jednocześnie optymalny koszt przeniesienia głazów. Dla każdego głazu sprawdzamy mianowicie, czy istnieje jego położenie, mieszczące się w granicach badanego prostokąta. Jeżeli nie istnieje, to prostokąt nie jest przydatny. W przeciwnym wypadku trzeba wybrać mniej kosztowny sposób umieszczenia głazu w ogrodzeniu — jeżeli głaz bez przestawiania mieści się wewnątrz płotu, to oczywiście pozostawiamy go na miejscu; w przeciwnym razie jesteśmy zmuszeni go przestawić i do całkowitego kosztu operacji doliczyć jego ciężar. Ten prosty, ale bardzo przydatny w dalszej części, algorytm ma oczywiście złożoność czasową $O(n)$.

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

Wykorzystując algorytm z poprzedniego rozdziału możemy rozwiązać zadanie, sprawdzając wszystkie możliwe prostokątne płoty. Oczywiście bierzemy pod uwagę tylko prostokąty przydatne — spośród nich wybieramy prostokąt o najkrótszym obwodzie, a jeśli jest takich wiele, to wymagający najmniejszego kosztu przestawienia głazów. Jak już zauważyliśmy, każde położenie płotu może zostać scharakteryzowane przez cztery liczby a, b, c, d. Niech Z oznacza zbiór wszystkich liczb, które występują w danych jako współrzędne głazów, czyli $Z = \{x_i : 1 \leqslant i \leqslant n\} \cup \{y_i : 1 \leqslant i \leqslant n\}$. Bez straty ogólności możemy ograniczyć nasze rozważania do prostokątów, dla których $a,b,c,d \in Z$ — każdy inny prostokąt można bowiem zmniejszyć, nie zmieniając jego podstawowych własności: przydatności i kosztu przestawiania głazów.

Poczynione spostrzeżenie ogranicza liczbę potencjalnych położeń płotu do $O(n^4)$ ($|Z| \leqslant 2 \cdot n$) i, w połączeniu z liniowym algorytmem sprawdzania przydatności prostokąta, daje rozwiązanie zadania o złożoności czasowej $O(n^5)$. Aby przyspieszyć ten prosty algorytm, przyjrzymy się bliżej zbiorowi Z. Niech *min* oznacza najmniejszą liczbę w Z, a *max* — największą. Zauważmy, że wartość *min* musi wystąpić jako rzędna lub odcięta położenia jednego z głazów — stanowi ona tym samym najlepsze dolne ograniczenie rzutu prostokąta na odpowiednią oś. Analogiczny argument dotyczy wartości *max*. Stąd $min \in \{a,c\}$ oraz $max \in \{b,d\}$, co daje cztery odrębne przypadki do rozważenia. W każdym z nich tylko dwie spośród liczb a,b,c,d pozostają niewiadomymi; rozważając każdy przypadek osobno uzyskujemy zawsze $O(n^2)$ możliwych położeń płotu, co przy liniowym względem n czasie sprawdzania przydatności prostokąta daje rozwiązanie o złożoności czasowej $O(n^3)$. Jest ono zaimplementowane w plikach `skas2.cpp` oraz `skas5.pas`. Rozwiązanie to pozwalało uzyskać na zawodach 30% punktów.

Rozwiązanie o złożoności czasowej $O(n^2)$

Powyższe rozwiązanie można jeszcze przyspieszyć, uzyskując złożoność czasową $O(n^2)$. W tym celu poszukiwanie najlepszego położenia płotu podzielimy na dwie fazy:

1. znajdowanie minimalnej długości obwodu prostokąta przydatnego;
2. wyznaczanie najmniejszego kosztu poprzestawiania wszystkich głazów, tak aby zmieściły się w pewnym prostokącie o minimalnym obwodzie.

Przypomnijmy, że dwie spośród czterech liczb opisujących płot mamy ustalone; dla każdego z czterech przypadków wynikających z tych ustaleń możemy przeanalizować wszystkie wartości (należące do zbioru Z) jednej z niewiadomych liczb. Dla ustalenia uwagi (choć nie bez straty ogólności!) przyjmijmy, że $a = min$, $b = max$ i za c podstawiamy kolejno wszystkie wartości ze zbioru Z. Wśród parametrów a,b,c,d pozostaje nam wówczas jedna niewiadoma — d. Przeglądamy teraz wszystkie głazy i dla każdego z nich wyznaczamy te spośród dwóch możliwych położeń, które są zgodne z przyjętymi wartościami a,b,c, czyli pierwsza współrzędna mieści się w przedziale $[a,b]$, a druga jest nie mniejsza od c (jeżeli żadne położenie głazu nie spełnia tego warunku, to odrzucamy trójkę a,b,c — nie da się jej uzupełnić do prostokąta przydatnego). Następnie spośród wyznaczonych położeń wybieramy to, dla którego uzyskujemy minimalną wartość drugiej współrzędnej głazu — takie położenie wymusza najmniejszy wzrost wartości parametru d, a tym samym długości płotu. Po przeanalizowaniu wszystkich głazów uzyskujemy zatem najmniejszą wartość liczby d, dla której da się umieścić wewnątrz płotu wszystkie głazy, a $2(b-a+d-c)$ jest długością tego płotu.

Rozważając pozostałe trzy przypadki wynikające z ograniczeń $min \in \{a,c\}$ i $max \in \{b,d\}$ i przeglądając w pętli wszystkie możliwe wartości jednego z pozostałych parametrów, wyznaczamy minimalny możliwy do osiągnięcia obwód prostokąta w każdym z tych przypadków. Najmniejsza z tych czterech wartości jest poszukiwaną, optymalną długością płotu. Złożoność czasową tej fazy algorytmu uzyskujemy, przemnażając liczbę sprawdzanych trójek parametrów a,b,c,d przez złożoność czasową analizy wszystkich głazów, czyli wynosi ona $4n \cdot O(n) = O(n^2)$.

Skoro wyznaczyliśmy już minimalną długość płotu, możemy przejść do drugiej fazy rozwiązania. Przeprowadzamy ją w sposób bardzo podobny do poprzedniej. Ponownie rozważamy $4n$ sposoby ustalenia wartości trzech spośród czterech liczb a,b,c,d, ale tym razem — znając optymalny obwód ℓ prostokąta — czwarty parametr wyznaczamy jednoznacznie z równości $2(b-a+d-c) = \ell$ w czasie $O(1)$. Mając wartości wszystkich parametrów a,b,c,d, sprawdzamy w czasie liniowym względem n, czy taki prostokąt jest przydatny i jeżeli tak, to wyznaczamy minimalny koszt przestawienia głazów, tak by znalazły się wewnątrz ogrodzenia.

Złożoność drugiej fazy algorytmu to $4n \cdot O(n) = O(n^2)$. Złożoność całego algorytmu także wynosi $O(n^2)$. Implementacje tego rozwiązania znajdują się w plikach `skas1.cpp` oraz `skas4.pas`. Rozwiązanie to pozwalało uzyskać na zawodach około 50% punktów.

Rozwiązanie wzorcowe

Wprowadzenie

Opisane powyżej rozwiązania wymagały kilku prostych spostrzeżeń, które pozwalały usprawnić pierwsze (proste, lecz czasochłonne) rozwiązanie do algorytmu działającego w czasie $O(n^2)$. Zaprezentowane przy tym techniki poprawiania złożoności czasowej można z powodzeniem wykorzystywać w konstrukcji wielu innych algorytmów. Tymczasem rozwiązanie wzorcowe, o złożoności czasowej $O(n)$, wymaga zupełnie innego podejścia opartego na kilku nietrywialnych do udowodnienia i momentami zaskakujących spostrzeżeniach. Zastosowaną w nim metodę prezentujemy w sposób, w jaki została ona wymyślona. Rozpoczniemy od rozwiązania maksymalnie uproszczonego wariantu problemu. Następnie, poprzez analizę ogólniejszej wersji, dojdziemy do rozwiązania problemu postawionego w treści zadania.

Rysunki w niniejszym rozdziale przedstawiają położenia prostokątnego płotu w dość nietypowy sposób — osie układu współrzędnych są narysowane *równolegle* do siebie. Taki sposób przedstawienia jest konieczny do właściwego zilustrowania zaprezentowanych dowodów.

Rozwiązanie uproszczonej wersji zadania

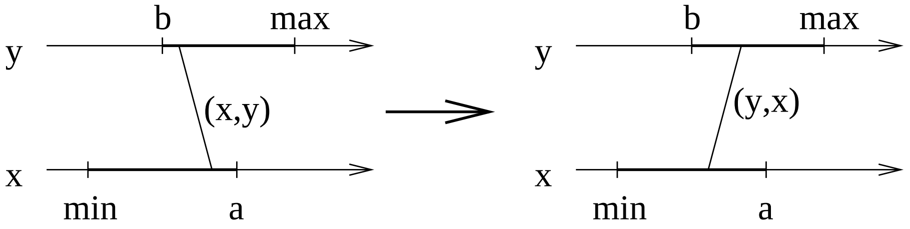
Spróbujmy na początek znaleźć *jakikolwiek* sposób poprzestawiania głazów, w którym minimalizujemy długość płotu koniecznego do ich ogrodzenia i nie zwracamy uwagi na ich ciężar. Okazuje się, że dobrym sposobem jest ustawienie głazów tak, by **każdy głaz miał pierwszą współrzędną nie większą niż drugą.** To oznacza, że dla głazu położonego na pozycji (x,y), jeśli $x > y$, to dokonujemy jego przestawienia, a w przeciwnym przypadku pozostawiamy go na miejscu. Oznaczmy uzyskane w ten sposób ustawienie głazów przez M.

Niech U będzie dowolnym ustawieniem głazów, dla którego długość płotu potrzebnego do ich ogrodzenia jest minimalna. Pokażemy, że dla ustawienia M uzyskujemy taką samą długość płotu. Wykażemy to, przestawiając głazy w ustawieniu U w ten sposób, by nie zwiększyć długości otaczającego go płotu i doprowadzić do jego zrównania z ustawieniem M. Dowód przeprowadzimy odrębnie dla dwu przypadków:

1. w ustawieniu U wartości *max* i *min* występują na różnych osiach,
2. w ustawieniu U wartości *min* i *max* występują na tej samej osi,

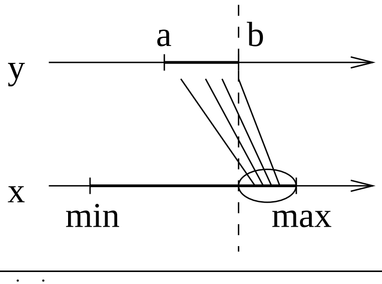
gdzie *min* i *max* są zdefiniowane jak poprzednio.

Rozważmy najpierw pierwszy przypadek. Przypuśćmy, że w ustawieniu U wartość *min* występuje jako odcięta pewnego punktu, a wartość *max* jako rzędna — w przeciwnym przypadku możemy przestawić wszystkie głazy w układzie U. Najkrótszy płot potrzebny do ogrodzenia układu U ma teraz postać $[min,a] \times [b,max]$ dla pewnych wartości parametrów a oraz b. Jeżeli jakiś głaz w U ma współrzędne (x,y), takie że $x > y$, to możemy go przestawić, nie wychodząc poza ogrodzenie. Dzieje się tak dlatego, że skoro przed przestawieniem $x \leqslant a$ oraz $y \geqslant b$, to $y < x \leqslant a$ oraz $x > y \geqslant b$, a zatem przestawiony głaz mieści się w pierwotnym ogrodzeniu dla U. To kończy dowód w pierwszym przypadku.



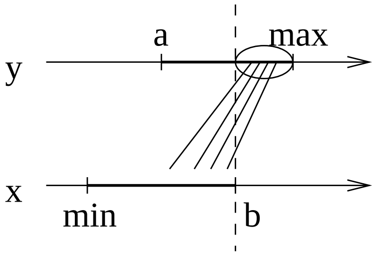
Pierwszy przypadek dowodu

Zajmijmy się teraz drugim przypadkiem. Jeżeli w ustawieniu U współrzędne *min* i *max* znajdują się na tej samej osi (możemy podobnie jak poprzednio doprowadzić do sytuacji, w której jest to oś odciętych), to na osi rzędnych rzut prostokąta stanowi pewien przedział $[a,b]$. Przyjrzyjmy się wszystkim głazom, których odcięta leży w przedziale $[b+1,max]$. Ich rzędne należą oczywiście do przedziału $[a,b]$.



Drugi przypadek dowodu — przed przestawieniem

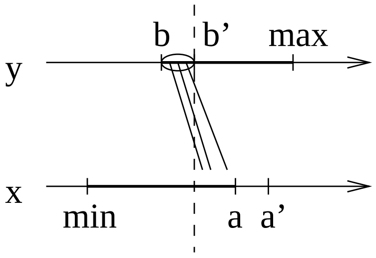
Przestawiając wszystkie te głazy, otrzymujemy ustawienie U', którego rzut na oś odciętych mieści się w przedziale $[min,b]$, a rzut na oś rzędnych — w przedziale $[a,max]$.
Stąd ustawienie U' daje się ogrodzić płotem o długości nie większej niż $(b - min) + (max - a) = (max - min) + (b - a)$, czyli nie większej niż długość płotu otaczającego ustawienie U (z optymalności ustawienia U wynika ponadto, że musi to być płot tej samej długości). Dodatkowo to ustawienie spełnia warunki przypadku pierwszego i możemy zastosować do niego opisaną powyżej procedurę sprowadzania do ustawienia M. To spostrzeżenie kończy uzasadnienie drugiego przypadku i zarazem cały dowód.



Drugi przypadek dowodu — po przestawieniu

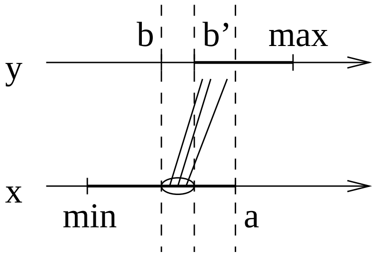
Optymalne prostokąty

Zastanówmy się teraz, ile jest (dla ustalonego zbioru głazów) *różnych ustawień płotu* o minimalnej długości obwodu. Ponownie rozważania podzielimy na dwa przypadki, analogiczne do powyższych. Pokażemy najpierw, że **istnieje dokładnie jedno (z dokładnością do przestawienia wszystkich głazów, co odpowiada zamianie osi miejscami) ustawienie płotu o minimalnej długości obwodu, w którym** min **i** max **są umieszczone na różnych osiach.** Dowód przeprowadzimy przez sprowadzenie do sprzeczności. Załóżmy, że istnieją dwa takie prostokąty: $[min,a] \times [b,max]$ oraz $[min,a'] \times [b',max]$ (odpowiadające im ustawienia oznaczmy U_1 i U_2) i dla ustalenia uwagi przyjmijmy, że $a < a'$. Ponieważ prostokąty mają równe obwody, musi zachodzić $b < b'$. Oczywiście można tak dokonać przestawienia pewnych głazów, by z ustawienia U_1 uzyskać ustawienie U_2. Podczas przekształcenia głazy o rzędnych z przedziału $[b,b'-1]$ (występujące w ustawieniu U_1 i niemieszczące się w ogrodzeniu U_2) muszą zostać przestawione. Aby zmiana konfiguracji mogła się powieść, to odcięte wspomnianych głazów muszą zawierać się w przedziale $[b',max]$. Stąd możemy wnioskować, że $a \geqslant b'$, gdyż w przeciwnym przypadku zbiór możliwych wartości odciętych rozważanych głazów ($[min,a] \cap [b',max]$) byłby pusty (a to jest sprzeczne z założeniem o optymalności U_1).



Pierwszy przypadek — przed przestawieniem

Można stąd wywnioskować, że $[b,b'-1] \subseteq [min,a]$, a zatem przestawiając *tylko* rozważane głazy, uzyskujemy prostokąt $[min,a] \times [b',max]$, czyli lepszy od obu wyjściowych.



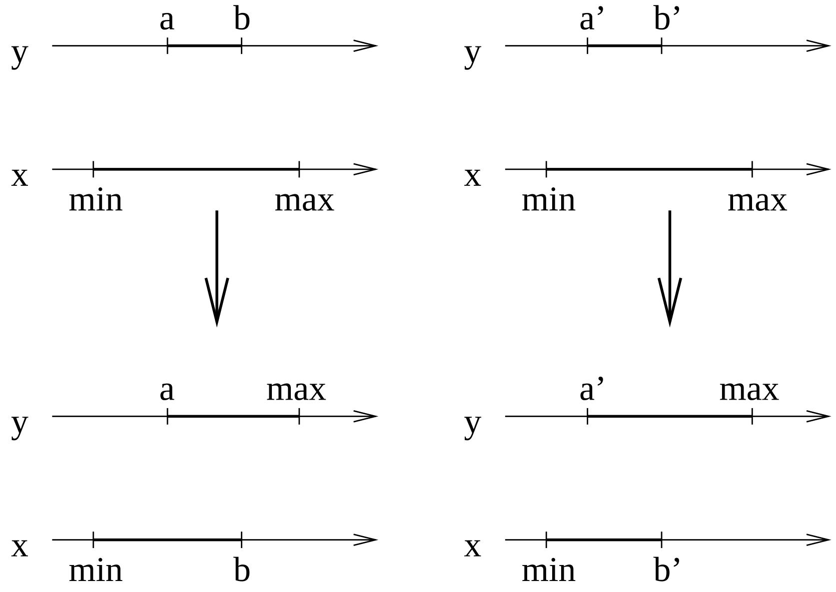
Pierwszy przypadek — po przestawieniu

To daje nam żądaną sprzeczność.

Pokażemy teraz, że **z dokładnością do zamiany osi miejscami istnieje co najwyżej jedno optymalne ustawienie drugiego typu, w którym współrzędne** min **i** max **znajdują się na tej samej osi.** Gdyby istniały dwa takie prostokąty $[min,max] \times [a,b]$ oraz $[min,max] \times [a',b']$, to oba miałyby krótszy bok tej samej długości: $b-a = b'-a'$. Za pomocą opisanej w poprzednim rozdziale transformacji U do U', otrzymalibyśmy z nich dwa istotnie różne ustawienia pierwszego typu — $U_1 = [min,b] \times [a,max]$ oraz $U_2 = [min,b'] \times [a',max]$ — co, jak już udowodniliśmy, nie jest możliwe.

Algorytm

Przedstawione wyżej spostrzeżenia pozwalają skonstruować (zaskakująco proste) rozwiązanie zadania. Na początek przestawiamy każdy głaz, którego pierwsza współrzędna jest większa od drugiej. Dla tak otrzymanego ustawienia



Drugi przypadek — sprowadzenie do pierwszego

M wyznaczamy parametry a oraz b otaczającego je płotu $[min,a] \times [b,max]$. W ten sposób znamy już optymalną długość płotu i pozostaje ustalić, przy jakim ustawieniu osiągamy minimalny, sumaryczny ciężar przenoszonych głazów. Ponieważ mamy jedynie cztery możliwości ustawienia płotu o minimalnej długości obwodu:

- $[min,a] \times [b,max]$,
- $[b,max] \times [min,a]$,
- $[min,max] \times [b,a]$,
- $[b,a] \times [min,max]$,

to wystarczy dla każdej z nich, za pomocą opisanego na początku opracowania zadania testu przydatności prostokąta, sprawdzić, czy da się poprzestawiać głazy tak, by zostały ogrodzone tym płotem i jaki jest minimalny koszt takiego przestawienia. Najmniejszy z uzyskanych w ten sposób wyników jest poszukiwanym przez nas optymalnym sposobem poprzestawiania głazów.

Dzięki bardzo małej (stałej równej 4) liczbie przypadków do rozpatrzenia, rozwiązanie wzorcowe ma złożoność czasową $O(n)$. Jest ono zaimplementowane w plikach `ska.cpp` oraz `ska1.pas`.

Rozwiązania błędne

Warto zastanowić się, czy konieczne jest rozważanie wszystkich czterech przedstawionych wyżej przypadków. Okazuje się jednak, że dla każdego z nich istnieje taki zbiór głazów wraz z przypisanymi ciężarami, dla których optymalne ogrodzenie trzeba skonstruować tak, jak w tym przypadku. W plikach `skab1.cpp`, `skab2.cpp`, `skab3.cpp` oraz `skab4.cpp` znajdują się implementacje rozwiązania wzorcowego, w których pominięto rozważenie jednej z czterech możliwości ustawienia płotu.

Rozwiązania implementowane przez zawodników

Zaledwie kilku zawodników zaimplementowało poprawne rozwiązanie o złożoności czasowej liniowej; wszystkie te rozwiązania były podobne do wzorcowego. Duża grupa zawodników przedstawiła do oceny rozwiązania niepoprawne, w których pominięto niektóre spośród czterech wyżej opisanych przypadków; to niedopatrzenie zazwyczaj nie było przypadkowe, lecz było wynikiem nieudanej próby „zgadnięcia” zachłannej metody rozwiązania zadania. Rozwiązania tego typu uzyskiwały, wskutek grupowania testów, maksymalnie 10% punktów.

Pojawiło się także kilka rozwiązań o złożoności czasowej wykładniczej względem n, w których dla każdego sposobu poprzestawiania głazów (2^n możliwości) wyznaczano długość obwodu wynikowego płotu i koszt tego przestawienia. Łączna złożoność takiego algorytmu to $O(2^n \cdot n)$; jest on zaimplementowany w plikach `skas3.cpp` oraz `skas6.pas`. Rozwiązanie to zdobywało 10% punktów. Istniały także rozwiązania, których złożoności zależały od maksymalnej wartości współrzędnych głazów; zazwyczaj nie pozwalały one zdobyć żadnych punktów.

Zaplanowany sposób punktowania rozwiązań zadania *Skalniak* polegał na faworyzowaniu poprawnych rozwiązań wielomianowych (względem liczby głazów) w stosunku do prawie poprawnych liniowych rozwiązań zachłannych.

Testy

Rozwiązania zawodników były sprawdzane na 10 grupach testów. Prawie wszystkie grupy (poza pierwszą) składają się z 4 testów:

- testy *a* eliminują rozwiązanie `skab1.cpp`,
- testy *b* eliminują rozwiązanie `skab2.cpp`,
- testy *c* eliminują rozwiązanie `skab3.cpp`,
- testy *d* eliminują rozwiązanie `skab4.cpp`.

Wszystkie testy zostały wygenerowane w sposób losowy.

Poniżej zamieszczony jest opis każdej z grup testów. We wszystkich testach każdej grupy (poza pierwszą) wartość parametru *n* jest taka sama.

Nazwa	**n**	**Opis**
ska1a.in	3	bardzo mały test
ska1b.in	15	mały test
ska2(abcd).in	100	grupa losowych testów
ska3(abcd).in	500	grupa losowych testów
ska4(abcd).in	3 000	grupa losowych testów
ska5(abcd).in	10 000	grupa losowych testów
ska6(abcd).in	60 000	grupa losowych testów
ska7(abcd).in	300 000	grupa losowych testów
ska8(abcd).in	600 000	grupa losowych testów
ska9(abcd).in	1 000 000	grupa losowych testów
ska10(abcd).in	1 000 000	grupa losowych testów

Szymon Acedański
Treść zadania, Opracowanie

Anna Niewiarowska
Program

OI, Etap II, dzień drugi, 08.02.2007

Megalopolis

Globalizacja nie ominęła Bajtocji. Nie ominęła również listonosza Bajtazara, niegdyś chodzącego polnymi drogami pomiędzy wioskami, a dziś pędzącego samochodem po autostradach. Jednak to te dawne spacery Bajtazar wspomina dziś z rozrzewnieniem.

Dawniej n bajtockich wiosek, ponumerowanych od 1 do n, było połączonych dwukierunkowymi polnymi drogami, w taki sposób, że z każdej wioski można było dojść do wioski numer 1 (zwanej Bitowicami) na dokładnie jeden sposób; w dodatku droga ta przechodziła jedynie przez wioski o numerach nie większych niż numer wioski początkowej. Ponadto każda polna droga łączyła dwie różne wioski i nie przechodziła przez żadne inne wioski oprócz tych dwóch. Drogi się nie krzyżowały poza wioskami, lecz mogły istnieć tunele bądź wiadukty.

Z biegiem czasu kolejne polne drogi zamieniano na autostrady. Bajtazar dokładnie pamięta, kiedy każda z polnych dróg została zamieniona w autostradę. Dziś w Bajtocji nie można już spotkać ani jednej polnej drogi — wszystkie zostały zastąpione autostradami, które połączyły wioski w Bajtockie Megalopolis.

Bajtazar pamięta swoje wyprawy do wiosek z listami. Za każdym razem wyruszał z Bitowic, idąc z listami do pewnej innej wioski. Teraz prosi Cię, żebyś dla każdej takiej wyprawy (która miała miejsce w określonym momencie i prowadziła z Bitowic do określonej wioski) policzył, przez ile polnych dróg ona prowadziła.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia:*
  - *opis dróg, które łączyły kiedyś bajtockie wioski,*
  - *sekwencję zdarzeń: wypraw Bajtazara i momentów, gdy poszczególne polne drogi były zamieniane w autostrady,*
- *dla każdej wyprawy obliczy, iloma polnymi drogami Bajtazar musiał przejść,*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita n ($1 \leqslant n \leqslant 250\,000$), oznaczająca liczbę wiosek w Bajtocji. W kolejnych $n-1$ wierszach znajdują się opisy dróg. Każdy z nich składa się z dwóch liczb całkowitych a, b ($1 \leqslant a < b \leqslant n$) oddzielonych pojedynczym odstępem. Są to numery wiosek połączonych drogą.

W kolejnym wierszu znajduje się jedna liczba całkowita m ($1 \leqslant m \leqslant 250\,000$), oznaczająca liczbę wypraw odbytych przez Bajtazara. W kolejnych $n+m-1$ liniach znajdują się opisy zdarzeń, w kolejności chronologicznej:

- *Opis postaci* `A` **a** **b** *(dla $a < b$) oznacza, że w danym momencie polną drogę pomiędzy wioskami a oraz b zamieniono na autostradę.*
- *Opis postaci* `W` **a** *oznacza, że Bajtazar odbył wyprawę z Bitowic do wioski numer a.*

Wyjście

Na standardowe wyjście Twój program powinien wypisać dokładnie m liczb całkowitych, po jednej w wierszu, oznaczających liczbę polnych dróg, które pokonał Bajtazar w kolejnych wyprawach.

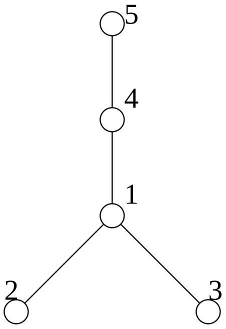
Przykład

Dla danych wejściowych:

```
5
1 2
1 3
1 4
4 5
4
W 5
A 1 4
W 5
A 4 5
W 5
W 2
A 1 2
A 1 3
```

poprawnym wynikiem jest:

```
2
1
0
1
```



Rozwiązanie

Pierwszym krokiem do rozwiązania prawie każdego zadania z Olimpiady Informatycznej jest wypatrzenie problemu algorytmicznego ukrytego za „historyjką”. Zapiszmy zatem zadanie *Megalopolis* tak, by problem stał się lepiej widoczny:

> Drogi i wioski tworzą drzewo ukorzenione, w którym niektóre krawędzie (autostrady) mogą być wyróżnione. Struktura danych musi umożliwiać wykonywanie dla drzewa następujących operacji:
>
> - *zaznacz(e)* — krawędź *e* zostaje wyróżniona,
> - *ile(v)* — zwraca liczbę niewyróżnionych krawędzi na ścieżce od korzenia do wierzchołka *v*.

Ewentualnie nieco inaczej:

> Drzewo ukorzenione reprezentuje istniejące w danym momencie polne drogi. Struktura danych umożliwia wykonywanie następujących operacji:
>
> - *skróć(e)* — operacja polega na zastąpieniu krawędzi *e* i jej obu końców przez jeden wierzchołek w grafie, czyli usunięciu z drzewa drogi zamienionej w autostradę;
> - *wysokość(v)* — zwraca wysokość wierzchołka *v* w drzewie, czyli długość ścieżki prowadzącej z wierzchołka *v* do korzenia.

Proste rozwiązania

Oto kilka prostych sposobów zaimplementowania zaproponowanych struktur danych.

Pierwsza interpretacja. Poniższe procedury są prostą implementacją drzewa z wyróżnionymi krawędziami.

```
function zaznacz(e)
begin
  e.zaznaczona := true;
end

```

```
function ile(v)
begin
  wynik := 0;
  while v nie jest korzeniem do
  begin
    if krawędź(v, ojciec(v)) nie jest zaznaczona then
      wynik := wynik + 1;
    v := ojciec(v);
  return wynik;
end
```

Złożoność operacji *zaznacz* jest stała ($O(1)$), ale wyznaczanie wartości *ile* działa w czasie $O(h)$, gdzie h to wysokość drzewa. Niestety, wysokość ta może wynosić nawet $n-1$.

Druga interpretacja. Tworzymy drzewo oddające aktualny układ polnych dróg. Każdy wierzchołek posiada listę swoich dzieci. Operacja *skróć(e)* polega na utożsamieniu obu końców e, co symulujemy, łącząc listy ich dzieci. Do sprawnego operowania na zbiorach dzieci wierzchołka można użyć struktury *Find-Union* (patrz, na przykład, [19]). Operacja *wysokość(v)* może być zaimplementowana przez przejście od v do korzenia i policzenie krawędzi na ścieżce. Wówczas skracanie krawędzi działa w czasie $O(\log^* n)$, a liczenie wysokości (po uwzględnieniu faktu, że niektóre wierzchołki zastępują kilka innych) w czasie $O(h \cdot \log^* n)$.

W obu przedstawionych strukturach jedna z operacji, niestety bardzo często wykonywana w zadaniu, wymaga czasu proporcjonalnego do wysokości drzewa. Chcąc otrzymać efektywne rozwiązanie problemu, musimy to ulepszyć.

Rozwiązanie wzorcowe

Zastanówmy się, czy do rozwiązania problemu nie przydałaby się jakaś ogólna struktura danych — na przykład słownik. W takim słowniku, wraz z wierzchołkami moglibyśmy przechowywać ich wysokość w drzewie. Dodatkowo chcielibyśmy umieć szybko zmniejszyć o jeden wysokości wszystkich wierzchołków w pewnym poddrzewie (to odpowiada operacji *zaznacz()* czy też *skróć()*). Niestety, popularne implementujące słowników (np. drzewa zrównoważone), nie pozwalają na efektywne wykonanie tej operacji. Przydatna okazuje się jednak struktura danych zwana *drzewem licznikowym* (patrz, na przykład, opracowanie zadania *Koleje* w *Niebieskiej Książeczce* z IX OI, [9]). Jest to słownik z tradycyjnymi operacjami *szukaj(klucz)* i *wstaw(klucz, wartość)*, w którym kluczami są liczby ze zbioru $\{1,\ldots,n\}$, dla uprzednio ustalonego n. Dodatkowo drzewo licznikowe posiada operację *zwiększ_w_przedziale(a,b)*, która zwiększa o jeden wszystkie wartości odpowiadające kluczom *z przedziału* $[a,b]$ w czasie $O(\log n)$.

Jeśli udałoby się nam ponumerować wierzchołki w drzewie z treści zadania tak, żeby dla każdego poddrzewa numery wierzchołków tworzyły przedział, to bylibyśmy na drodze prowadzącej wprost do rozwiązania. Zauważmy, że nie jest to trudne:

Obserwacja 1 Jeśli ponumerujemy wierzchołki drzewa w kolejności *preorder*, *inorder* lub *postorder*, to numery wierzchołków wchodzących w skład dowolnego poddrzewa tworzą przedział.

Korzystając z tego spostrzeżenia, możemy zaimplementować operacje wymagane w pierwszej interpretacji w następujący sposób:

```
function inicjuj()
begin
  Ponumeruj wierzchołki w kolejności preorder:
  { Uruchamiamy procedurę DFS i zapisujemy w każdym wierzchołku: }
  { nr — jego numer }
  { h — jego pierwotną wysokość }
  { od,do — przedział numerów wierzchołków w jego poddrzewie };
  Stwórz drzewo licznikowe o n kluczach, we wszystkich wierzchołkach
  (pole val) wpisując wartość zero;
end

function zaznacz(e)
begin
  zwiększ_w_przedziale(e.od, e.do);
end

function ile(v)
```

```
begin
  return $v.h - szukaj(v)$;
end
```

W powyższym rozwiązaniu operacja *inicjuj* działa w czasie $O(n)$, natomiast *zaznacz(e)* oraz *ile(v)* — w czasie $O(\log n)$. Razem daje to złożoność $O((m+n)\log n)$. Rozwiązanie to zostało zapisane w plikach `meg.cpp` i `meg1.pas`.

Rozwiązanie ***offline***

Opracowanie rozpoczęliśmy od zinterpretowania zadania w języku teorii grafów. Obie podane interpretacje, choć dość naturalne, narzucają nam podobny sposób widzenia problemu. W obu interpretacjach staramy się na bieżąco symulować operacje wykonywane na sieci dróg i udzielać odpowiedzi na pytania o liczbę polnych dróg zaraz po pojawieniu się zapytania w danych. Tymczasem taki pośpiech nie jest konieczny. Możemy rozpocząć od przeczytania wszystkich danych wejściowych, przeprowadzić obliczenia i na końcu wypisać odpowiedzi na wszystkie pytania. Takie podejście, zwane *rozwiązaniem offline*, może okazać się bardziej eleganckie, prostsze w implementacji, a nawet szybsze (sprawdziło się już wcześniej, na przykład w zadaniu *Małpki* z finału X OI, [10]).

Zacznijmy od wprowadzenia dwóch pojęć: za *moment_zapytania* oraz *moment_zbudowania* przyjmiemy odpowiednio numer wiersza danych wejściowych, w której pojawiło się pytanie lub informacja o budowie autostrady.

W naszym rozwiązaniu offline *drzewo dróg* będziemy reprezentować, zapisując przy każdym wierzchołku listę jego dzieci. Algorytm będzie polegał na przejściu drzewa dróg procedurą DFS. Podczas przechodzenia będziemy utrzymywać następującą strukturę danych:

- *historia budowy*, czyli zbiór zawierający *momenty_zbudowania* wszystkich autostrad na ścieżce od korzenia do aktualnie odwiedzanego wierzchołka — informacje te będą zapisane w zbiorze uporządkowanym, na którym będziemy wykonywać operacje *dodaj(wartość)*, *usuń(wartość)* oraz *ile_wcześniejszych_niż(wartość)*; dobrą implementacją takiej struktury jest zrównoważone drzewo binarne — wszystkie trzy operacje można wykonać w czasie logarytmicznym.

Odwiedzając wierzchołek v i chcąc dowiedzieć się, przez ile autostrad jechał Bajtazar do v w momencie t, wystarczy sprawdzić, ile zdarzeń wcześniejszych niż t jest zapisanych w historii budowy. W ten sposób odpowiedzi na pytania będziemy znajdować w kolejności zgodnej z przechodzeniem drzewa dróg w porządku *preorder* i będziemy je zapisywać w tablicy *odpowiedź* — na i-tej pozycji znajdzie się odpowiedź na pytanie, które pojawiło się w i-tym momencie.

```
function czytaj_dane()
begin
  { Dla każdego wierzchołka znajdujemy: }
  {   jego ojca oraz listę jego dzieci, }
  {   listę momentów_zapytań o ten wierzchołek, }
  {   odległość od korzenia w początkowym drzewie dróg, }
  {   moment_zbudowania autostrady od wierzchołka do jego ojca. }
end

function dfs($v$)
begin
  dodaj(moment_zbudowania autostrady ($v$, ojciec($v$)));
  for moment_zapytania in zapytania o $v$ do
    odpowiedź[moment_zapytania] :=
      ile_wcześniejszych_niż(moment_zapytania);
  for $u \in$dzieci($v$) do dfs($u$);
  usuń(moment_zbudowania autostrady ($v$, ojciec($v$)));
end

function rozwiąż()
begin
  czytaj_dane();
  dfs(korzeń_drzewa_dróg);
end
```

Po uruchomieniu funkcji *rozwiąż* w tablicy *odpowiedź* otrzymamy szukane wyniki. Czas działania to $O((m+n)\cdot\log h)$, gdzie n oznacza rozmiar drzewa, $h = O(n)$ jest jego wysokością, a m to liczba zapytań.

Zadanie na deser

Wśród rozwiązań zgłoszonych przez zawodników pojawił się ciekawy pomysł. My w pierwotnym rozwiązaniu wzorcowym użyliśmy drzewa licznikowego. Podobne rozwiązanie można jednak uzyskać bez tej struktury — wykorzystując dwa zbiory uporządkowane (drzewa zrównoważone). Zastanów się, jak to zrobić. Może przydać Ci się przy tym poniższa wskazówka:

Obserwacja 2 Przyjmijmy, że wierzchołki drzewa są ponumerowane w kolejności *preorder*. Niech P_i oznacza przedział numerów odpowiadających wierzchołkom w poddrzewie ukorzenionym w wierzchołku i. Wówczas liczba autostrad na drodze z korzenia do wierzchołka v jest równa liczbie takich i, dla których $P_v \subset P_i$ oraz krawędź $(i, \text{ojciec}(i))$ została już przerobiona na autostradę.

Testy

Do oceny przygotowano zestaw czternastu testów, o następujących parametrach:

Nazwa	n	m	Opis
meg1.in	10	15	mały test, krótkie ścieżki w drzewie
meg2.in	30	50	mały test, dłuższe ścieżki w drzewie
meg3.in	50	2 000	mały test, dużo pytań w porównaniu z liczbą krawędzi
meg4.in	100 000	50 000	duży test, bardzo krótkie ścieżki
meg5.in	20 000	30 000	średni test, ścieżki średniej długości
meg6.in	50 000	65 000	średni test, długie ścieżki
meg7.in	150 000	160 001	duży test, ścieżki średniej długości
meg8.in	200 001	170 001	duży test, ścieżki średniej długości
meg9.in	200 000	200 000	duży test, długie ścieżki
meg10.in	200 000	150 000	duży test, długie ścieżki

Nazwa	**n**	**m**	**Opis**
meg11.in	120 000	250 000	duży test, maksymalna liczba pytań, mniej wiosek
meg12.in	250 000	100 000	duży test, maksymalna liczba wiosek, mniej pytań
meg13.in	250 000	250 000	duży test, maksymalna liczba wiosek i pytań
meg14.in	250 000	250 000	duży test, maksymalna liczba wiosek i pytań

Tomasz Waleń
Treść zadania, Opracowanie

Paweł Parys
Program

OI, Etap II, dzień drugi, 08.02.2007

Tetris Attack

Ostatnimi czasy w Bajtocji bardzo popularną grą stała się łamigłówka „Tetris Attack”. Jej uproszczona wersja ma następującą postać: Gracz otrzymuje do dyspozycji stos, na którym umieszczono $2n$ elementów (jeden na drugim), oznaczonych n różnymi symbolami. Przy tym każdym symbolem są oznaczone dokładnie dwa elementy na stosie. Ruch gracza polega na zamianie dwóch sąsiednich elementów miejscami. Jeśli w wyniku zamiany na stosie sąsiadują ze sobą elementy oznaczone identycznymi symbolami, to w „magiczny” sposób znikają, a elementy znajdujące się powyżej spadają w dół (być może powodując kolejne zniknięcia). Celem gracza jest opróżnienie stosu w jak najmniejszej liczbie ruchów.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis początkowej zawartości stosu,*
- *obliczy rozwiązanie wymagające minimalnej liczby ruchów,*
- *wypisze znalezione rozwiązanie na standardowe wyjście.*

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita n, $1 \leqslant n \leqslant 50\,000$. W kolejnych $2n$ wierszach zapisana jest początkowa zawartość stosu. Wiersz $i+1$-szy zawiera jedną liczbę całkowitą a_i — symbol elementu znajdującego się na wysokości i ($1 \leqslant a_i \leqslant n$). Każdy symbol występuje na stosie **dokładnie** *2 razy. Na początku żadne dwa identyczne symbole nie występują obok siebie. Ponadto dane testowe są tak dobrane, że istnieje rozwiązanie zawierające nie więcej niż 1 000 000 ruchów.*

Wyjście

Na standardowym wyjściu należy wypisać opis rozwiązania, wymagającego minimalnej liczby ruchów. Pierwszy wiersz powinien zawierać jedną liczbę całkowitą m — długość najkrótszego rozwiązania. Kolejne m wierszy powinno zawierać opis rozwiązania, czyli ciąg m liczb całkowitych $p_1, \ldots, p_m$, po jednej w każdym wierszu. Wartość p_i oznacza, że w i-tym ruchu gracz zdecydował o zamianie elementów, znajdujących się na wysokościach p_i oraz $p_i + 1$.

Jeżeli istnieje wiele rozwiązań, to Twój program powinien wypisać dowolne z nich.

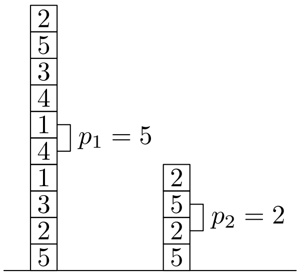
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
2
5
2
```



Natomiast dla danych wejściowych:

```
3
1
```

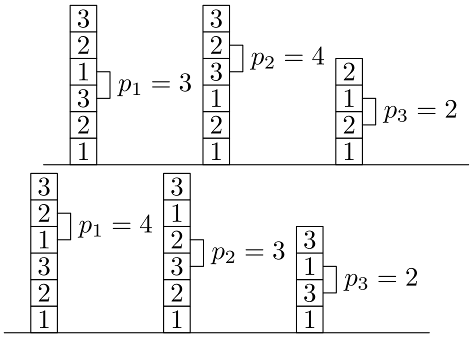
```
2
3
1
2
3
```

poprawnym wynikiem jest:

```
3
3
4
2
```

jak również:

```
3
4
3
2
```



Rozwiązanie

O zadaniu

Nazwa i pomysł zadania pochodzą od gry *Tetris Attack*[1] (gra występuje również pod nazwami *Pokemon Puzzle League* lub *Panel de Pon*). Oczywiście oryginalna gra jest bardziej skomplikowana: klocki są ułożone w dwóch wymiarach, liczba klocków oznaczonych takimi samymi symbolami nie jest ograniczona itd.

Podstawowe pojęcia

Rozpocznijmy od wprowadzenia kilku definicji, które ułatwią nam prezentację rozwiązania. Przyjmiemy konwencję, zgodnie z którą stos będziemy zapisywać od elementu na dole stosu do elementu na szczycie; na przykład ciąg: 1 3 2 1 2 3 oznacza stos, w którym element 1 znajduje się na samym dole, natomiast 3 — na szczycie stosu.

Symbole odróżniające klocki będziemy nazywać *kolorami*, zgodnie z konwencją przyjętą w oryginalnych grach.

Powiemy, że para kolorów (a,b) tworzy *inwersję*, jeśli klocki w kolorach a, b występują na stosie w następującej kolejności:

$$\dots a \dots b \dots a \dots b \dots$$

Przykładowo na stosie:

$$1\ 2\ 3\ 2\ 1\ 3$$

występują inwersje $(1,3)$ oraz $(2,3)$.

Układ na stosie nazwiemy *stabilnym*, jeśli na sąsiednich pozycjach nie występują jednakowe kolory.
W stabilnym układzie o n kolorach możemy mieć od 1 do $n(n-1)/2$ inwersji. Najmniejsza liczba inwersji występuje w układzie:

$$1\ 2\ 3\ \dots\ n-2\ n\ n-1\ n\ n-1\ n-2\ \dots\ 2\ 1$$

natomiast największą liczbę inwersji ma układ:

$$1\ 2\ 3\ \dots\ n-1\ n\ 1\ 2\ 3\ \dots\ n-1\ n$$

Rozwiązanie wzorcowe

Dla rozwiązania zadania kluczowe są następujące spostrzeżenia:

- pojedynczy ruch zmniejsza liczbę inwersji co najwyżej o 1;
- jeśli stos zawiera układ stabilny i nie jest pusty, to zawsze można wykonać ruch, który zmniejszy liczbę inwersji o 1;

---

[1] Więcej informacji o grze można znaleźć na stronie `http://www.tetrisattack.net/`

- wykonując tylko ruchy zmniejszające liczbę inwersji o 1, otrzymamy rozwiązanie optymalne, czyli wymagające minimalnej liczby ruchów.

Pierwsze, a zatem także i trzecie z powyższych stwierdzeń, są oczywiste. Pozostaje wykazać stwierdzenie drugie — poniżej udowodnimy twierdzenie, w którym pokazujemy, jak znajdować ruchy zmniejszające liczbę inwersji. Twierdzenie to stanie się podstawą przedstawionego dalej algorytmu rozwiązania wzorcowego.

Twierdzenie 1 *Niech* $a_1,\ldots,a_{2n}$ *oznacza stos elementów, w którym każdy z* n *kolorów występuję dokładnie 2 razy. Zakładamy, że układ jest stabilny, tzn.* $a_i \neq a_{i+1}$, *dla* $1 \leqslant i < 2n$. *Istnieje ruch, którego wykonanie powoduje zmniejszenie liczby inwersji o 1.*

Dowód W większości wypadków istnieje wiele ruchów o takiej własności, jednak my wyznaczymy ruch, którego wykonanie pozwoli na efektywne rozwiązanie zadania.

Niech j oznacza maksymalny indeks $1 \leqslant j < 2n$ taki, że każdy z elementów $a_1,\ldots,a_j$ ma inny kolor. Oczywiście w ciągu $a_1,\ldots,a_{j+1}$ jakiś kolor musi się już powtórzyć, czyli istnieje indeks $1 \leqslant i \leqslant j$ taki, że $a_i = a_{j+1}$.

Łatwo zauważyć, że kolory a oraz b występujące odpowiednio na pozycjach a_i oraz a_j tworzą inwersję:

kolor:	...	a	...	b	a	...	b	...
indeks:		i		j	$j+1$		$>j+1$	

Możemy ją zlikwidować, zamieniając sąsiednie elementy o indeksach j oraz $j+1$. ■

W rozwiązaniu wzorcowym będziemy kolejno eliminować inwersje, w których występuje pierwszy powtarzający się kolor — jak w dowodzie twierdzenia. Aby efektywnie przeprowadzić tę procedurę, stworzymy dwa stosy, pomiędzy które rozdzielimy zadane elementy:

- D — stos ten będzie zawierał początkowe, zbadane elementy układu, wśród których nie występują powtarzające się kolory — początkowo stos ten jest pusty.

- S — stos ten będzie zawierał resztę elementów tworzących aktualny układ — początkowo stos ten zawiera wszystkie elementy, ułożone w kolejności odwrotnej do wejściowej, tzn. a_1 znajduje się na szczycie stosu S.

W trakcie algorytmu analizujemy kolejno elementy x pobierane ze stosu S. Jeśli kolor elementu x nie występuje jeszcze w D, to odkładamy element x na szczyt stosu D. W przeciwnym przypadku możemy zastosować twierdzenie, wykonując ruch dla x i jego poprzednika, likwidując w ten sposób jedną inwersję. W kolejnych ruchach eliminujemy kolejne inwersje, w których występuje kolor x, przesuwając ten element w głąb stosu D (w rzeczywistości będziemy przekładać elementy ze stosu D do stosu S), aż do momentu, gdy spotka on swoją „parę” i zniknie z układu. Warto zauważyć, że wykonywane przy okazji przesuwania x zamiany są (podobnie jak pierwsza) również zamianami „pierwszego powtarzającego się koloru” i każda z nich powoduje zmniejszenie liczby inwersji o 1. Opisane postępowanie jest realizowane w poniższym algorytmie:

```
$D$ := $\emptyset$;
$S$ := { stos elementów w kolejności $a_{2n}, a_{2n-1}, \ldots, a_1$ };
while $S \neq \emptyset$ do
begin
  $x$ := $S.Pop$; { kolejny element z $S$ }
  if $x \notin D$ then { nowy kolor, więc dodajemy do $D$ }
    $D.Push(x)$;
  else { kolor $x$ już występuje w $D$ }
  begin
    $j$ := $LiczbaElementów(D)$;
    $y$ := $D.Pop$;
    if $x \neq y$ then { jeśli kolory są różne, to wykonujemy zamianę }
    begin
      { print: zamień $(j, j+1)$; }
      $S.Push(y)$; $S.Push(x)$; { odłóż elementy $x$ i $y$ z powrotem na stos $S$ }
    end
  end
end
```

Procedury *Pop* i *Push* powodują odpowiednio pobranie elementu ze szczytu stosu i włożenie elementu na szczyt stosu.

Czas działania powyższego algorytmu wynosi $O(n+k)$, gdzie k to liczba inwersji w ciągu podanym na wejściu — przypomnijmy, że k może wynosić od $O(1)$ do $O(n^2)$. Jest to jednak algorytm o bardzo dobrych parametrach, ponieważ składnika k w złożoności i tak uniknąć się nie da — jest to rozmiar informacji, które trzeba wypisać jako wynik działania algorytmu.

Inne rozwiązania

Istnieje wiele rozwiązań opartych na schemacie:

```
while LiczbaInwersji > 0 do
begin
  wyznacz ruch, który zmniejsza liczbę inwersji o 1;
  wykonaj ten ruch;
  zaktualizuj zawartość stosu (w szczególności usuń znikające elementy);
end
```

W schemacie tym kryją się dwa potencjalne źródła nieefektywności algorytmu. Po pierwsze, czasochłonne może być „naiwne” poszukiwanie ruchu zmniejszającego liczbę inwersji. Po drugie, nieprzemyślane symulowanie aktualnego stanu stosu, w szczególności usuwanie “znikających elementów”, także może powodować wydłużenie obliczeń. Częściowo niedogodności te można zmniejszyć, odpowiednio dobierając kolejne ruchy, na przykład poszukując ich możliwie blisko szczytu stosu. Niestety są to tylko heurystyki i w pesymistycznym przypadku bazujące na nich rozwiązania mają jednak złożoność $O(n^2+k)$.

Testy

Poniższa tabelka przedstawia zestawienie testów użytych do oceny rozwiązań. Parametr w oznacza rozmiar optymalnego rozwiązania, a n — liczbę kolorów w danych wejściowych.

Nazwa	w	n	Opis
tet1.in	78	40	test losowy
tet2.in	406	50	test losowy
tet3.in	6 624	200	test losowy
tet4.in	19 956	1 000	test losowy
tet5.in	135 926	1 200	test losowy
tet6.in	663 589	4 000	test losowy
tet7.in	657 197	4 500	test losowy
tet8.in	998 360	5 000	test losowy
tet9.in	999 986	5 000	test losowy
tet10a.in	999 614	5 000	test losowy
tet10b.in	1 000 000	2 000	test z największą odpowiedzią
tet11a.in	830 998	42 000	test specjalny
tet11b.in	998 991	10 000	test specjalny
tet12.in	592 528	50 000	test specjalny
tet13.in	759 921	50 000	test specjalny
tet14.in	472 199	50 000	test specjalny
tet15a.in	905 241	50 000	test specjalny
tet15b.in	1	50 000	maksymalny test z odpowiedzią 1

Testy specjalne to losowe testy uzupełnione o duży fragment typu $1\ 2\ \ldots\ k\ \ldots\ k\ \ldots\ 2\ 1$ oraz wiele (krótkich) fragmentów typu $1\ 2\ \ldots\ p\ 1\ 2\ \ldots\ p$.

Zawody III stopnia

opracowania zadań

Marcin Kubica
Treść zadania, Opracowanie

Anna Niewiarowska
Program

OI, Etap III, dzień próbny, 27.03.2007

Koleje

Bajtockie Koleje Państwowe (BKP) stanęły przed koniecznością restrukturyzacji i redukcji sieci linii kolejowych. Po jej dokładnym przeanalizowaniu zdecydowano, które stacje kolejowe mają być zlikwidowane, a które mają pozostać. Zdecydowano się także zredukować sieć linii kolejowych. Trzeba jeszcze wybrać, które linie kolejowe mają być zlikwidowane, a które mają pozostać.

Sieć linii kolejowych składa się z odcinków torów łączących stacje kolejowe. Wiadomo, że z każdej stacji kolejowej da się dojechać do każdej innej (potencjalnie odwiedzając stacje pośrednie). Odcinki torów są dwukierunkowe. Między każdą parą stacji może być co najwyżej jeden odcinek torów. Każdy taki odcinek torów charakteryzuje się **kosztem utrzymania** *— dodatnią liczbą całkowitą. Odcinki torów, które mają pozostać, muszą być tak wybrane, aby:*

- *dało się przejechać między każdymi dwiema stacjami, które mają pozostać,*
- *ich sumaryczny koszt utrzymania był mały (może być co najwyżej dwukrotnie większy od najmniejszego kosztu, jaki da się uzyskać, zachowując poprzedni warunek).*

Wszystkie pozostałe odcinki torów zostaną zlikwidowane. Linie kolejowe, które mają pozostać, mogą przebiegać przez stacje, które zostaną zlikwidowane.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis sieci linii kolejowych oraz stacje, które mają pozostać,*
- *wyznaczy, które odcinki torów mają pozostać, a które mają być zlikwidowane,*
- *wypisze na standardowe wyjście, które odcinki torów mają pozostać oraz poda ich sumaryczny koszt utrzymania.*

Wejście

W pierwszym wierszu standardowego wejścia zapisane są dwie dodatnie liczby całkowite n i m, $2 \leqslant n \leqslant 5\ 000$, $1 \leqslant m \leqslant 500\ 000$ ($m \leqslant \frac{n(n-1)}{2}$), oddzielone pojedynczym odstępem. Liczba n to liczba stacji kolejowych, a m — liczba odcinków torów. Stacje kolejowe są ponumerowane od 1 do n. W kolejnych m wierszach są opisane odcinki torów kolejowych, po jednym w wierszu. W każdym z tych wierszy są zapisane trzy dodatnie liczby całkowite a, b i u, $1 \leqslant a,b \leqslant n$, $a \neq b$, $1 \leqslant u \leqslant 100\ 000$. Liczby a i b to numery stacji, które łączy dany odcinek torów, a u to jego koszt utrzymania. W $m+2$ wierszu zapisany jest ciąg liczb całkowitych pooddzielanych pojedynczymi odstępami. Pierwsza z nich to p — liczba stacji, które mają pozostać ($1 \leqslant p \leqslant n$, $p \cdot m \leqslant 15\,000\,000$). Dalej w tym wierszu wymienione są numery tych stacji w kolejności rosnącej.

Wyjście

W pierwszym wierszu standardowego wyjścia program powinien wypisać dwie liczby całkowite c i k oddzielone pojedynczym odstępem, gdzie c jest sumarycznym kosztem utrzymania pozostawionych odcinków, a k — liczbą tych odcinków. W każdym z kolejnych k wierszy powinny znaleźć się dwie liczby a_i oraz b_i oddzielone pojedynczym odstępem — numery stacji połączonych przez pozostawione odcinki torów. Sumaryczny koszt utrzymania odcinków torów może być co najwyżej **dwukrotnie większy** *od najmniejszego kosztu, możliwego do uzyskania.*

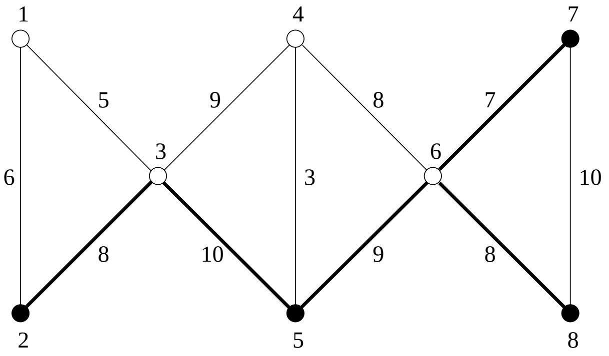
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
42 5
2 3
3 5
5 6
6 7
6 8
```



Rozwiązanie

Analiza problemu

Problem przedstawiony w treści zadania jest znany jako grafowe sformułowanie[1] problemu *drzew Steinera*. Dany jest graf nieskierowany $G = (V,E)$, w którym każdej krawędzi $e \in E$ przypisana jest dodatnia waga w_e, oraz zbiór wierzchołków $V' \subseteq V$. Oznaczmy przez $c(X)$ (dla $X \subseteq E$) sumę wag krawędzi należących do zbioru X. Szukamy takiego drzewa złożonego z krawędzi $T \subseteq E$, że:

- każde dwa wierzchołki ze zbioru V' są połączone pewną ścieżką złożoną z krawędzi należących do zbioru T,
- $c(T)$ jest minimalne.

Niestety, problem ten jest NP-zupełny[2], co oznacza, że nie jest znany algorytm znajdujący takie drzewo w czasie wielomianowym. Na szczęście, w niniejszym zadaniu nie wymagamy, aby znalezione drzewo było optymalne (nawet nie wymagamy, aby to było drzewo)! Dopuszczamy, aby suma wag krawędzi była większa od najmniejszej możliwej — ale nie więcej niż dwukrotnie. Rozwiązanie zadania polega więc na napisaniu algorytmu *aproksymacyjnego*, czyli obliczającego rozwiązanie bliskie optymalnemu. W problemach aproksymacyjnych za miarę przybliżenia przyjmuje się często stosunek wielkości znalezionego rozwiązania do wielkości rozwiązania optymalnego. W tym przypadku stosunek ten nie może przekraczać 2 — takie przybliżenie jest nazywane 2-aproksymacją.

Problem drzew Steinera przypomina problem znajdowania minimalnego drzewa rozpinającego. Jednak ten ostatni nie jest NP-zupełny i potrafimy go rozwiązywać w czasie wielomianowym. Służą do tego na przykład algorytmy zachłanne Kruskala i Prima ([19], p. 24).

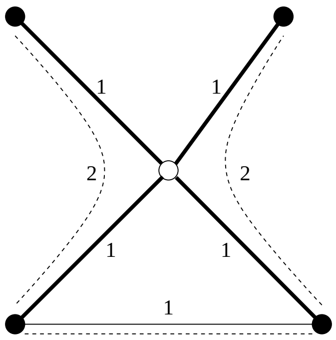
Spróbujemy użyć algorytmu znajdującego minimalne drzewo rozpinające do przybliżenia optymalnego drzewa Steinera. Najpierw jednak musimy zmienić graf, w którym będziemy szukać minimalnego drzewa rozpinającego. Będzie to graf pełny $G' = (V',E')$, gdzie dla każdej pary wierzchołków $u, v \in V'$ wagą krawędzi (u,v) jest długość najkrótszej (tj. najlżejszej) ścieżki w grafie G. W grafie G' znajdujemy minimalne drzewo rozpinające T' — jego krawędzie to ścieżki w oryginalnym grafie G. Niech T będzie zbiorem tych krawędzi z E, które należą do którejkolwiek krawędzi-ścieżki w drzewie T' (w razie potrzeby usuwamy powtórzenia krawędzi T występujących w kilku krawędziach-ścieżkach T'). Graf T łączy wszystkie wierzchołki z V', natomiast nie musi być drzewem — może okazać się, że łącząc ścieżki z T', uzyskamy cykle — jest to jednak dopuszczalne! Zauważmy, że $c(T) \leqslant c(T')$, gdyż w drzewie T' wagi krawędzi powtarzających się na wielu ścieżkach są liczone wielokrotnie, a w drzewie T są liczone tylko raz.

Minimalne drzewo rozpinające T' w grafie G' nie musi odpowiadać drzewu Steinera w grafie G. Wynika to stąd, że w poszukiwaniu tych drzew kierujemy się nieco innymi kryteriami optymalności — koszt drzewa Steinera to zwykła

---

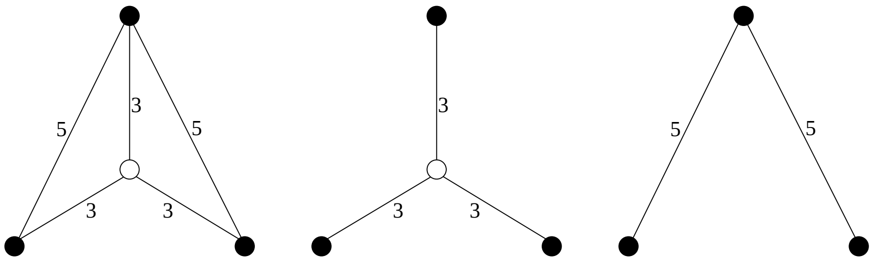
[1] W innym sformułowaniu problemu drzew Steinera mamy dany zbiór punktów na płaszczyźnie i szukamy najkrótszej sieci dróg łączących te punkty. Taką wersję problemu także można przedstawić w postaci grafu, choć może to wymagać wprowadzenia nieskończonego zbioru wierzchołków.

[2] Problemy NP-zupełne są przedstawione np. w [19, 21].



Rys. 1: Drzewo Steinera w grafie G i minimalne drzewo rozpinające w grafie G'.

suma wag krawędzi, a koszt drzewa T' to suma długości ścieżek, wśród których mogą występować powtórzenia krawędzi. Ilustruje to następujący przykład. Łączna waga krawędzi w drzewie Steinera wynosi 9, natomiast minimalne drzewo rozpinające w grafie G' ma wagę 10. Pokażemy, że choć waga minimalnego drzewa rozpinającego w grafie G' może być większa niż waga drzewa Steinera w grafie G, to jednak nie więcej niż dwukrotnie. Oznacza to, że T' będzie wystarczającym dla naszych celów przybliżeniem rozwiązania optymalnego.



Rys. 2: Dany graf G, drzewo Steinera i minimalne drzewo rozpinające w grafie G'.

Lemat 1 (na podstawie [36]) *Niech $G = (V,E)$ będzie danym grafem, w którym każdej krawędzi $e \in E$ przypisana jest dodatnia waga w_e, oraz niech $V' \subseteq V$ będzie danym zbiorem wierzchołków. Niech także G' będzie grafem skonstruowanym jak powyżej. Oznaczmy przez T drzewo Steinera rozpinające w G wierzchołki ze zbioru V', a przez T' minimalne drzewo rozpinające w G'. Wówczas zachodzi nierówność $c(T') \leqslant 2 \cdot c(T)$.*

Dowód Rozważmy eulerowskie obejście drzewa T, tzn. taki cykl, który prowadzi wyłącznie przez krawędzie z T, odwiedza każdy wierzchołek drzewa T i przez każdą krawędź z T przechodzi dokładnie dwukrotnie. Dla każdego wierzchołka $v \in V'$ wybierzmy jedno jego wystąpienie w takim cyklu. Następnie podzielmy ten cykl na ścieżki, przecinając go w wybranych wystąpieniach wierzchołków z V'. W ten sposób otrzymujemy ścieżki $\sigma_1, \sigma_2, \ldots, \sigma_p$. Zauważmy, że:

$$c(\sigma_1) + c(\sigma_2) + \cdots + c(\sigma_p) = 2 \cdot c(T)$$

Zauważmy też, że ścieżki $\sigma_1, \sigma_2, \ldots, \sigma_p$ odpowiadają krawędziom w grafie G' (dokładniej, są nie lżejsze od odpowiadających im krawędzi G'), tworzącym w nim cykl przechodzący przez każdy wierzchołek dokładnie raz (tzw. cykl Hamiltona). Czyli $\sigma_1, \sigma_2, \ldots, \sigma_{p-1}$ odpowiadają ścieżce przechodzącej przez każdy wierzchołek G' dokładnie raz (tzw. ścieżce Hamiltona). Ścieżka to jednak szczególny przypadek drzewa. Tak więc ścieżki $\sigma_1, \sigma_2, \ldots, \sigma_{p-1}$ mają wagę nie mniejszą od wagi pewnego drzewa rozpinającego w G', które z kolei nie może mieć wagi mniejszej niż T' — minimalne drzewo rozpinające dla G'.

Mamy więc ostatecznie:

$$c(T') \leqslant c(\sigma_1) + c(\sigma_2) + \cdots + c(\sigma_{p-1}) \leqslant c(\sigma_1) + c(\sigma_2) + \cdots + c(\sigma_p) = 2 \cdot c(T).$$

■

Rozwiązanie oparte o konstrukcję minimalnego drzewa rozpinającego

Algorytm opisany w poprzednim punkcie jest dość prosty koncepcyjnie, ale jego efektywna implementacja może przysporzyć kłopotów. Najpierw musimy skonstruować graf G', czyli wyznaczyć najkrótsze ścieżki między wszystkimi parami wierzchołków ze zbioru V'. Prostym rozwiązaniem jest zastosowanie algorytmu Floyda-Warshalla dla całego grafu G, a następnie zawężenie wyników do wierzchołków z G'. Jednak za tę prostotę płacimy złożonością czasową rzędu $O(n^3)$. Możemy też p-krotnie zastosować algorytm Dijkstry, wyznaczając odległości od kolejnych wierzchołków z V' do wszystkich pozostałych wierzchołków. Oczywiście, oprócz odległości musimy też być w stanie szybko zrekonstruować najkrótsze ścieżki realizujące te odległości. Zakładając, że użyjemy kolejki priorytetowej o logarytmicznym czasie wykonywania operacji, sumaryczna złożoność czasowa takiego rozwiązania to $O(p \cdot (n+m) \cdot \log n)$. Zważywszy, że $p \cdot m \leqslant 15\,000\,000$, jest to istotnie lepsze rozwiązanie. Jednak w obu przypadkach trzeba zapamiętać długości krawędzi grafu G'. Prosty rachunek wykazuje, że zmieścimy się w zadanym limicie pamięci tylko dla p maksymalnie rzędu $2\,000$.

Oba algorytmy wyznaczania minimalnego drzewa rozpinającego: Kruskala i Prima (zakładając, że w algorytmie Prima także użyjemy kolejki priorytetowej o logarytmicznym czasie wykonywania operacji) działają w czasie $O(p^2 \log p)$. Nie jest to więc dominująca faza obliczeń. Pozostało jeszcze przejrzenie zbioru krawędzi tworzących ścieżki oraz usunięcie powtórzeń. Ponieważ mamy $p-1$ ścieżek, każda nie dłuższa niż $n-1$ krawędzi, to wymaga to czasu rzędu $O(p \cdot n + m)$. W rezultacie otrzymujemy algorytm działający w czasie $O(p \cdot (n+m) \cdot \log n)$, jednak o zbyt dużej złożoności pamięciowej.

Okazuje się, że możemy nie konstruować grafu G' *a priori*. W algorytmie Prima budujemy drzewo rozpinające w sposób zachłanny, zaczynając od pojedynczego wierzchołka i dodając do niego kolejne krawędzie, za każdym razem wybierając najlżejszą krawędź, która nie domyka żadnego cyklu. Zamiast najpierw konstruować graf G', a potem budować rozpinające go drzewo, można „w locie” wyznaczać długości interesujących nas krawędzi-ścieżek w grafie G'. Oznaczmy przez D zbiór wierzchołków grafu G', które (w danym momencie) należą do budowanego drzewa. Konstrukcję drzewa zaczynamy od pojedynczego wierzchołka $v \in V'$, $D = \{v\}$. Następnie zachłannie rozszerzamy budowane drzewo. Szukamy najkrótszej krawędzi w grafie G' łączącej którykolwiek wierzchołek $v \in D$ z dowolnym wierzchołkiem $w \in V' \setminus D$. Ponieważ nie wyznaczyliśmy wcześniej grafu G', musimy w tym momencie znaleźć najkrótszą ścieżkę w grafie G łączącą którykolwiek wierzchołek $v \in D$ z dowolnym wierzchołkiem $w \in V' \setminus D$. Możemy tutaj zastosować algorytm Dijkstry, rozpoczynając przeszukiwanie grafu G równocześnie ze wszystkich wierzchołków ze zbioru D i kończąc je w momencie osiągnięcia pierwszego wierzchołka należącego do $V' \setminus D$.

Stosując w rozwiązaniu kolejkę priorytetową o logarytmicznym koszcie operacji, otrzymujemy taką samą złożoność czasową jak poprzednio, czyli $O(p \cdot (n+m) \cdot \log n)$, lecz złożoność pamięciowa jest dużo mniejsza! Musimy oczywiście pamiętać dany graf G, jednak wszystkie dodatkowe struktury danych, w tym kolejka priorytetowa wierzchołków, mają rozmiar $O(n)$. Tak więc łączna złożoność pamięciowa jest rzędu $O(n+m)$ i pozwala zmieścić się w zadanym limicie pamięciowym.

Rozwiązanie wzorcowe

Rozwiązanie wzorcowe jest podobne do opisanego powyżej zastosowania algorytmu Prima, z wyszukiwaniem „w locie” (za pomocą algorytmu Dijkstry) najkrótszych ścieżek w grafie G. Różnica między tym rozwiązaniem a opisanym poprzednio, polega na uproszczeniu procedury rozbudowy drzewa D o kolejne krawędzie-ścieżki. Zamiast dodawać do drzewa w każdym kroku najkrótszą ścieżkę łączącą pewien wierzchołek z $V' \cap D$ (należący już do budowanego drzewa) z pewnym wierzchołkiem z $V' \setminus D$ (nienależącym jeszcze do budowanego drzewa), dodajemy najkrótszą ścieżkę łączącą pewien wierzchołek z $V \cap D$ (*wierzchołek ten nie musi należeć do* V') z pewnym wierzchołkiem z $V' \setminus D$. Algorytm ten możemy wyrazić w postaci następującego pseudokodu:

```
Wybieramy dowolny $v \in V'$;
$D := \{v\}$;
$R := \emptyset$;
while $V' \not\subseteq D$ do begin
   $\sigma :=$ najkrótsza ścieżka w $G$ łącząca dowolny $v \in D$ z $w \in V' \setminus D$;
   $D := D \cup$ wierzchołki($\sigma$);
   $R := R \cup$ krawędzie($\sigma$);
end
return $R$;
```

Zbiór D zawiera wierzchołki, a zbiór R — krawędzie budowanego drzewa. Na pierwszy rzut oka algorytm ten powinien dawać przynajmniej tak dobrą aproksymację, jak algorytm opisany w poprzednim punkcie. Należy to jednak pokazać formalnie.

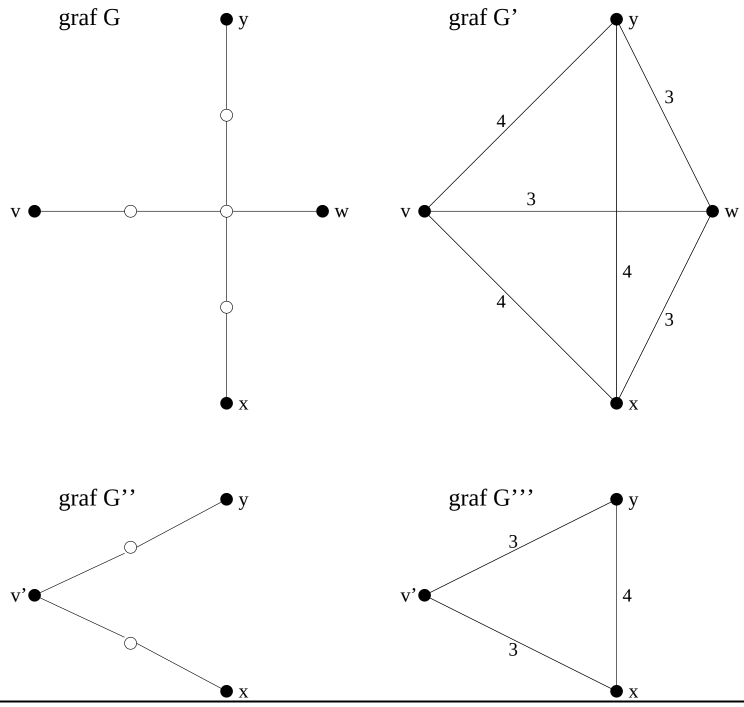
Lemat 2 *Niech* (D,R) *będzie drzewem utworzonym przez algorytm wzorcowy, rozpinającym w danym grafie* G *wierzchołki z* V'. *Niech* $c_{\min}$ *będzie sumą wag krawędzi w minimalnym drzewie rozpinającym w grafie* G'. *Wówczas* $c(R) \leqslant c_{\min}$.

Dowód Dowód przebiega przez indukcję po liczbie wybranych wierzchołków $p = |V'|$.

1. Jeśli $p = 1$, to $V' = \{v\}$ i oba drzewa nie zawierają ani jednej krawędzi. Zatem $c(R) = c_{\min} = 0$.

2. Załóżmy, że $p > 1$. Niech v będzie wierzchołkiem wybranym z V' na początku algorytmu wzorcowego. Niech σ będzie ścieżką wybraną w pierwszym kroku pętli **while** w algorytmie wzorcowym. Łączy ona v z pewnym wierzchołkiem $w \in V' \setminus \{v\}$.

Rozważmy graf $G'' = (V'', E'')$ powstały z G przez sklejenie wszystkich wierzchołków na ścieżce σ; wierzchołek powstały w wyniku sklejenia oznaczmy przez v'. Analogicznie, oznaczmy przez $G''' = (V''', E''')$ graf powstały ze sklejenia w grafie G' wierzchołków v i w w jeden wierzchołek v'. Intuicyjnie, para grafów (G'', G''') odpowiada parze grafów (G, G') po wykonaniu jednego kroku przez każdy z algorytmów. Przykład konstrukcji grafów G'' i G''' dla zadanych grafów G oraz G' jest zilustrowany na rys. 3.



Rys. 3: Przykładowy wygląd grafów G i odpowiadającego mu grafu G' oraz postać grafów G'' i G'''. Dla uproszczenia zakładamy, że krawędzie grafów G oraz G'' mają wagi jednostkowe.

Niech (D', R') będzie drzewem wyznaczonym przez algorytm wzorcowy dla grafu G'' i wybranego na początku wierzchołka v'. Sklejenie wierzchołków leżących na ścieżce σ nie wpływa na dalsze działanie algorytmu wzorcowego, gdyż traktuje on wierzchołki do tej pory skonstruowanego drzewa tak, jakby były sklejone. Stąd, $c(R) = c(\sigma) + c(R')$. Niech dalej $c'_{\min}$ będzie sumą wag krawędzi w minimalnym drzewie rozpinającym w grafie G'''. Z poprawności algorytmu Prima wiemy, że w grafie G' istnieje minimalne drzewo rozpinające, które zawiera krawędź odpowiadającą ścieżce σ. Stąd $c_{\min} = c(\sigma) + c'_{\min}$. Niech wreszcie $H = (G'')'$ będzie grafem skonstruowanym z G'' w ten sam sposób, w jaki z grafu G skonstruowaliśmy G', czyli grafem najkrótszych ścieżek między wierzchołkami wybranymi G''. Można zauważyć, że graf H musi wyglądać dokładnie tak samo jak G''', lecz potencjalnie może mieć krótsze niektóre krawędzie (2 zamiast 3 w przypadku dwóch krawędzi na rys. 3). Oznaczmy przez $c''_{\min}$ wagę minimalnego drzewa rozpinającego H. Wówczas $c''_{\min} \leqslant c'_{\min}$.

Ponieważ $|V'''| < |V'| = p$, to możemy skorzystać z założenia indukcyjnego dla grafów G'' oraz H, otrzymując $c(R') \leqslant c''_{\min}$. Nierówność ta implikuje, że $c(R') \leqslant c'_{\min}$. Stąd ostatecznie

$$c(R) = c(\sigma) + c(R') \leqslant c(\sigma) + c'_{\min} = c_{\min}.$$

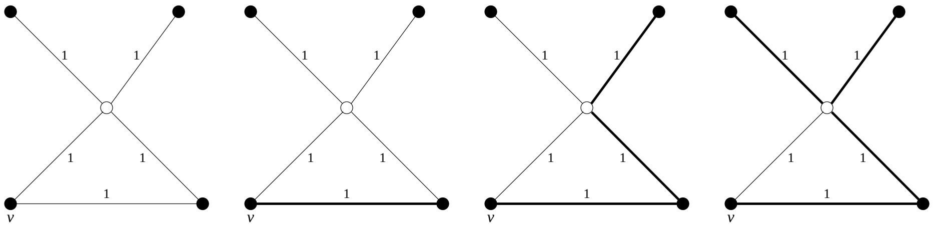
■

Z lematu tego wynika poprawność algorytmu wzorcowego.

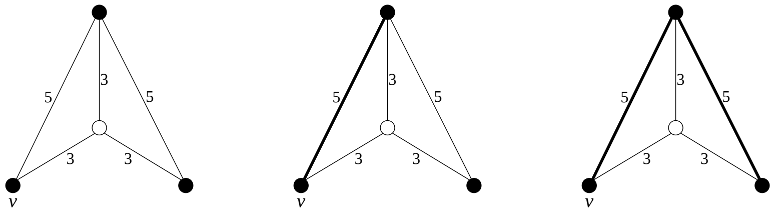
Twierdzenie 3 *Niech (D, R) będzie drzewem utworzonym przez algorytm wzorcowy, rozpinającym w danym grafie G wierzchołki ze zbioru V'. Niech c_S będzie sumą wag krawędzi w drzewie Steinera rozpinającym w grafie G wierzchołki ze zbioru V'. Wówczas $c(R) \leqslant 2 \cdot c_S$.*

Dowód Z Lematu 2 wynika, że $c(R) \leqslant c_{\min}$, gdzie $c_{\min}$ jest sumą wag krawędzi w minimalnym drzewie rozpinającym w grafie G'. Z Lematu 1 wiemy, że $c_{\min} \leqslant 2c_S$. Stąd, $c(R) \leqslant 2 \cdot c_S$. ■

Prześledźmy działanie algorytmu wzorcowego na pokazanych wcześniej dwóch przykładowych grafach. Okazuje się, że algorytm wzorcowy może znaleźć drzewo Steinera, podczas gdy minimalne drzewo rozpinające w grafie G' ma większą wagę. Niestety nie zawsze się tak dzieje. Oto przykład grafu, dla którego algorytm wzorcowy znajduje dokładnie drzewo rozpinające w grafie G', a jego waga jest wyższa od wagi drzewa Steinera.



Rys. 4: Konstrukcja drzewa Steinera o wadze 4 przez algorytm wzorcowy.



Rys. 5: Algorytm wzorcowy nie znajduje drzewa Steinera (waga 9), lecz drzewo o wadze 10.

W wierszu 5 algorytmu wzorcowego stosujemy algorytm Dijkstry do znalezienia w grafie G najkrótszej ścieżki łączącej dowolny $v \in D$ z $w \in V' \setminus D$. Możemy trochę poprawić efektywność tego algorytmu, jeżeli nie będziemy w każdym kroku pętli niezależnie wywoływać algorytmu Dijkstry, lecz wykorzystamy zawartość kolejki priorytetowej z poprzedniej iteracji do wykonania następnej iteracji. W kolejce priorytetowej trzymamy wierzchołki, które nie należą do drzewa, a ich priorytet odpowiada temu, co wiemy o ich odległości od konstruowanego drzewa. Początkowo konstruowane drzewo to pojedynczy wierzchołek $v \in V'$, $D = \{v\}$. Wierzchołki incydentne z nim mają priorytet równy długości łączących je krawędzi, a pozostałe wierzchołki mają priorytet równy ∞. Równocześnie, dla każdego wierzchołka o skończonym priorytecie pamiętamy ścieżkę, takiej długości jak jego priorytet, łączącą go z konstruowanym drzewem.

W kolejnych krokach algorytmu Dijkstry, za każdym razem wyciągamy z kolejki wierzchołek o najmniejszym priorytecie; oznaczmy go przez w. Oczywiście $w \notin D$. Spośród wierzchołków znajdujących się w kolejce, jest to wierzchołek najbliżej położony konstruowanego drzewa. Możliwe są dwa przypadki:

- Jeżeli $w \notin V'$, to korygujemy priorytety wierzchołków incydentnych z w, uwzględniając ścieżki prowadzące przez niego.
- Jeżeli $w \in V' \setminus D$, to spośród wierzchołków z $V' \setminus D$ jest to wierzchołek położony najbliżej konstruowanego drzewa. Rozszerzamy więc konstruowane drzewo o najkrótszą ścieżkę łączącą je z w.

  Po rozszerzeniu drzewa nie musimy ponownie tworzyć kolejki priorytetowej z wierzchołkami spoza drzewa. Wystarczy skorygować istniejącą kolejkę — musimy poprawić priorytety wierzchołków incydentnych z dodawaną ścieżką, uwzględniając fakt, że stała się ona częścią konstruowanego drzewa.

  Dodatkowo, musimy zwrócić uwagę na wierzchołki $v \notin V'$, których priorytet zmienia się już po usunięciu z kolejki priorytetowej. Jeśli po rozszerzeniu drzewa priorytet v (czyli jego odległość od drzewa) spadł poniżej wartości z chwili, gdy był on wyjmowany z kolejki, to wierzchołek v należy ponownie wstawić do kolejki i rozważyć.

Wierzchołki ze zbioru V' są tylko raz wstawiane do kolejki priorytetowej i tylko raz są z niej wyjmowane. Wierzchołki spoza V' mogą być wielokrotnie wstawiane i usuwane z niej — jednak nie więcej niż p razy każdy. Jednak dla większości danych wejściowych, liczba wstawianych i usuwanych wierzchołków jest istotnie mniejsza. Przyjmując, że operacje na kolejce priorytetowej działają w czasie $O(\log n)$, uzyskujemy pesymistyczną złożoność czasową $O(p \cdot (n+m) \cdot \log n)$. Jest to taka sama złożoność jak w przypadku poprzednio opisanego rozwiązania, jednak dla większości danych algorytm wzorcowy działa istotnie szybciej. Algorytm ten został zaimplementowany w programach `kol.cpp` i `kol0.pas`.

Inne rozwiązania

Oprócz przedstawionych wcześniej rozwiązań wielomianowych, można się było spodziewać rozwiązań znajdujących drzewo Steinera, ale działających w czasie wykładniczym. Na przykład, algorytm taki mógłby rozważać wszystkie możliwe zbiory wierzchołków z $V \setminus V'$, które wchodzą w skład drzewa Steinera i dla tak ograniczonego grafu znajdować minimalne drzewo rozpinające (jeśli takowe istnieje). Rozwiązanie takie działa w czasie $O(2^{n-p} \cdot m \log m)$. Nie należy się więc spodziewać, że przejdzie ono jakiekolwiek testy poza małymi testami poprawnościowymi. Zostało ono zaimplementowane w programach `kols1.cpp` i `kols3.pas`.

Testy

Rozwiązania zawodników były sprawdzane na 10 testach. Zostały one podsumowane w poniższej tabelce.

Dużym problemem przy przygotowywaniu testów był fakt, że do właściwej oceny rozwiązań zawodników konieczna jest znajomość wielkości drzewa Steinera. W ogólności jest to problem NP-zupełny. W przypadku testów nr 1, 2, 3 i 5 optymalny wynik można wyznaczyć w rozsądnym czasie za pomocą algorytmu wykładniczego. Dodatkowo, testy nr 3, 4, 6 i 9 mają na tyle prostą strukturę, że optymalny wynik można wyznaczyć ręcznie. W testach nr 7 i 10 mamy $p = 3$. Tak więc problem wyznaczenia drzewa Steinera sprowadza się do znalezienia takiego jego „środka", z którego rozchodzące się trzy najkrótsze ścieżki dają najlepszy wynik. Taki problem można już prosto rozwiązać, korzystając na przykład z algorytmu Floyda-Warshalla. Pozostał test nr 8. Składa się on z niedużego podgrafu losowego, dla którego można wyznaczyć optymalny wynik za pomocą algorytmu wykładniczego, oraz krawędzi i wierzchołków, co do których wiadomo, że nie są częścią drzewa Steinera, gdyż nie łączą one żadnych dwóch wierzchołków z V'.

Nazwa	n	m	koszt	Opis
kol1.in	25	161	32 143	prosty test poprawnościowy, losowy
kol2.in	41	443	220 078	prosty test poprawnościowy, losowy
kol3.in	100	540	774	test średniej wielkości, drzewo Steinera zawiera wszystkie wierzchołki
kol4.in	900	1 334	2 975	test średniej wielkości, drzewo Steinera zawiera jedynie wierzchołki z V'
kol5.in	999	14 800	5 068 199	test średniej wielkości, losowy
kol6.in	3 300	6 000	207 000	test wydajnościowy
kol7.in	3 000	405 181	2 670	test wydajnościowy
kol8.in	5 000	11 008	118 733	test wydajnościowy (maksymalna liczba wierzchołków)
kol9.in	4 841	9 504	50 600	test wydajnościowy
kol10.in	4 000	372 802	3 693	test wydajnościowy

Jakub Radoszewski — **Maciej Jaśkowski**

Treść zadania, Opracowanie — **Program**

OI, Etap III, dzień pierwszy, 28.03.2007

Gazociągi

Koncern GazBit zamierza zdominować rynek gazowy w Bajtocji. Specjaliści umiejscowili już na mapie Bajtocji optymalne położenia punktów wydobycia gazu oraz stacji dystrybucji gazu — pozostało jeszcze tylko przyporządkować stacje do punktów wydobycia. Każda stacja dystrybucji ma być połączona z dokładnie jednym punktem wydobycia i odwrotnie, każdy punkt wydobycia z dokładnie jedną stacją.

GazBit specjalizuje się w budowie gazociągów prowadzących z punktów wydobycia do stacji dystrybucji w kierunku południowym i wschodnim — dokładniej, każdy wybudowany gazociąg ma (patrząc z lotu ptaka) kształt łamanej, której każda kolejna część biegnie w kierunku południowym lub wschodnim i jest prostopadła do poprzedniej. Zarząd koncernu zastanawia się, jak przyporządkować punktom wydobycia gazu stacje dystrybucji tak, by zminimalizować łączną długość koniecznych do wybudowania gazociągów. Przy planowaniu można pominąć problem przecinania się gazociągów — ich kolidujące fragmenty zostaną umieszczone na różnych głębokościach pod ziemią.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia planowane położenia punktów wydobycia i stacji dystrybucji gazu,*
- *wyznaczy takie przyporządkowanie stacji dystrybucji do punktów wydobycia, które pozwala na wybudowanie gazociągów o minimalnej łącznej długości,*
- *wypisze wynik na standardowe wyjście.*

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę całkowitą n ($2 \leqslant n \leqslant 50\,000$)*, oznaczającą liczbę punktów wydobycia (równą liczbie stacji dystrybucji). Kolejne n wierszy zawiera po dwie liczby całkowite* x_i *oraz* y_i ($0 \leqslant x_i, y_i \leqslant 100\,000$ *dla* $1 \leqslant i \leqslant n$)*, oddzielone pojedynczym odstępem i oznaczające współrzędne na mapie punktów wydobycia gazu. Przyjmujemy, że wraz z rosnącą współrzędną x poruszamy się na wschód, a wraz z rosnącą współrzędną y poruszamy się na północ. Następne n wierszy zawiera po dwie liczby całkowite* x'_j *oraz* y'_j ($0 \leqslant x'_j, y'_j \leqslant 100\,000$ *dla* $1 \leqslant j \leqslant n$)*, oddzielone pojedynczym odstępem i oznaczające współrzędne na mapie stacji dystrybucji gazu. Zarówno punkty wydobycia, jak i stacje dystrybucji, numerujemy liczbami naturalnymi od 1 do n w kolejności występowania na wejściu. Żadna para współrzędnych nie powtórzy się w jednym zestawie danych wejściowych. Ponadto dla każdych danych wejściowych istnieje jakieś przyporządkowanie punktom wydobycia stacji dystrybucji, które można zrealizować za pomocą gazociągów idących tylko na południe i wschód.*

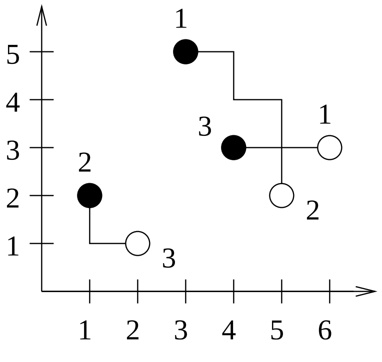
Wyjście

Pierwszy wiersz wyjścia powinien zawierać jedną liczbę całkowitą, oznaczającą minimalną sumaryczną długość wszystkich koniecznych do wybudowania gazociągów. Dalej na wyjściu powinien wystąpić przykładowy opis przyporządkowania stacji punktom wydobycia, który realizuje to minimum. Każdy z kolejnych n wierszy powinien zawierać dwie liczby całkowite, oddzielone pojedynczym odstępem i oznaczające numery punktu wydobycia i stacji dystrybucji, które powinny być połączone gazociągiem. Kolejność wypisywania przyporządkowań może być dowolna. Jeżeli istnieje wiele poprawnych rozwiązań, Twój program powinien wypisać jakiekolwiek z nich.

Przykład

Dla danych wejściowych:

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



poprawnym wynikiem jest:

```
9
2 3
1 2
3 1
```

Rozwiązanie

Kilka oznaczeń

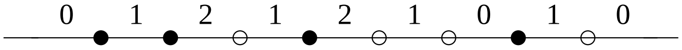
Przez $P = \{p_1, \ldots, p_n\}$ i $S = \{s_1, \ldots, s_n\}$ będziemy oznaczać zbiory punktów na płaszczyźnie, w których znajdują się odpowiednio punkty wydobycia i stacje dystrybucji gazu. Elementy zbioru P będziemy często nazywali po prostu *punktami*, zaś elementy zbioru S — *stacjami*. Wreszcie dowolny sposób połączenia stacji dystrybucji i punktów wydobycia za pomocą gazociągów południowo-wschodnich nazwiemy po prostu *przyporządkowaniem*, a przyporządkowanie nazwiemy *optymalnym*, jeżeli minimalizuje ono sumaryczną długość wykorzystanych gazociągów.

Wstępne spostrzeżenia

Z treści zadania wiemy, że punkt p i stację s można połączyć gazociągiem, jeżeli $s_x - p_x \geqslant 0$ oraz $p_y - s_y \geqslant 0$. Długość potrzebnego gazociągu wyraża się wówczas jako odległość pomiędzy p oraz s w metryce miejskiej, czyli $d(p,s) = (s_x - p_x) + (p_y - s_y)$. Zauważmy, że ta wartość jest niezależna od dokładnego przebiegu gazociągu.

Zanim poszukamy rozwiązania problemu postawionego w treści zadania, spróbujmy rozważyć jego uproszczoną wersję. Ograniczmy się mianowicie do jednowymiarowej wersji problemu, w której punkty wydobycia i stacje dystrybucji znajdują się na jednej (poziomej) prostej i zastanówmy się, jak w takiej sytuacji wygląda poszukiwane optymalne przyporządkowanie. Wymaganie, by gazociągi przebiegały w kierunku południowo-wschodnim redukuje się do wymagania, by dla każdej połączonej pary punkt–stacja o odciętych równych odpowiednio p_x i s_x zachodziła nierówność $p_x \leqslant s_x$.

Niech $[x, x+1]$, dla liczby całkowitej x, będzie odcinkiem jednostkowym na rozważanej prostej. Chcielibyśmy wiedzieć, ile gazociągów może przechodzić przez cały ten odcinek w różnych możliwych przyporządkowaniach (zauważmy, że żaden punkt ani żadna stacja nie znajdują się we wnętrzu tego odcinka). Niech P_x będzie liczbą punktów o odciętych nie większych niż x, a S_x — liczbą stacji spełniających ten warunek. Każda z tych S_x stacji musi zostać połączona z pewnym spośród wybranych P_x punktów. Ponieważ w zadaniu mamy zagwarantowane istnienie jakiegoś poprawnego przyporządkowania, to musi zachodzić $P_x \geqslant S_x$. Każdy z $P_x - S_x$ punktów wydobycia musi zostać zatem połączony z jakąś stacją dystrybucji o odciętej większej bądź równej $x+1$, co oznacza, że w *każdym* przyporządkowaniu przez odcinek $[x, x+1]$ będzie przechodzić dokładnie $P_x - S_x$ gazociągów. Zauważmy, że właśnie pokazaliśmy, że każde przyporządkowanie na prostej wymaga takiej samej łącznej długości gazociągów — jest ona równa sumie wartości $P_x - S_x$ dla wszystkich odcinków $[x, x+1]$. Innymi słowy, każde poprawne przyporządkowanie na prostej jest optymalne!



Przykładowe jednowymiarowe rozmieszczenie punktów i stacji. Dla każdego istotnego odcinka jednostkowego zaznaczony został współczynnik $P_x - S_x$.

Rozwiązanie dla przypadku jednowymiarowego pozwala nam postawić hipotezę, że również w przypadku dwuwymiarowym każde przyporządkowanie charakteryzuje się tą samą sumą długości wykorzystanych gazociągów. Okazuje się, że to stwierdzenie jest prawdziwe i na dodatek istnieje jego prosty i elegancki dowód. Suma długości gazociągów w dowolnym przyporządkowaniu wyraża się jako:

$$L = \sum_{i=1}^{n} d(p_i, s_{\pi(i)}),$$

gdzie π jest permutacją liczb od 1 do n, oznaczającą konkretne przyporządkowanie. Korzystając z definicji d oraz przemienności sumy i różnicy możemy wykonać następujące przekształcenia:

$$L = \sum_{i=1}^{n} \left(\left(s_{\pi(i)}\right)_x - (p_i)_x + (p_i)_y - \left(s_{\pi(i)}\right)_y \right) =$$

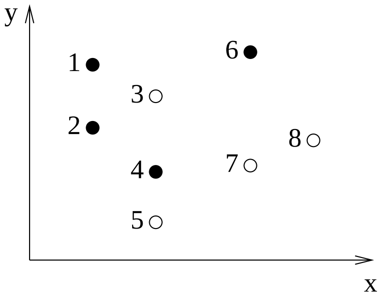
$$\sum_{i=1}^{n} \left(s_{\pi(i)}\right)_x - \sum_{i=1}^{n} (p_i)_x + \sum_{i=1}^{n} (p_i)_y - \sum_{i=1}^{n} \left(s_{\pi(i)}\right)_y =$$

$$\sum_{i=1}^{n}(s_i)_x - \sum_{i=1}^{n}(p_i)_x + \sum_{i=1}^{n}(p_i)_y - \sum_{i=1}^{n}(s_i)_y$$

Ostatnie wyrażenie pokazuje, że wartość L nie zależy od przyporządkowania, a jedynie od współrzędnych punktów i stacji. Stąd każde przyporządkowanie wymaga tej samej łącznej długości gazociągów, a zatem dowolne poprawne przyporządkowanie jest optymalne.

Rozwiązanie wzorcowe

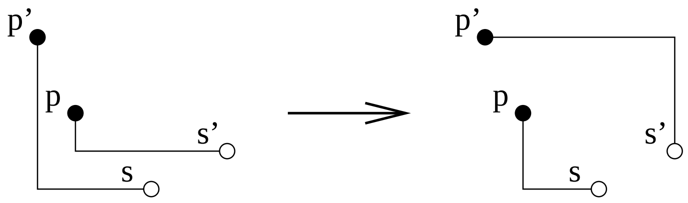
Na podstawie uprzednio poczynionych spostrzeżeń doszliśmy do wniosku, że w celu rozwiązania zadania wystarczy znaleźć dowolne *poprawne* przyporządkowanie. Wykorzystamy do tego celu technikę *zamiatania* płaszczyzny prostą pionową (*miotłą*). Na początku sortujemy wszystkie elementy (punkty i stacje) niemalejąco względem ich odciętych, rozstrzygając remisy na korzyść elementów o większej rzędnej. Następnie rozważamy elementy w otrzymanej kolejności.



Przykładowe rozmieszczenie punktów i stacji na płaszczyźnie z zaznaczoną kolejnością ich rozważania.

Jeżeli rozważany element jest punktem wydobycia, to umieszczamy go w miotle, reprezentowanej przez strukturę danych, utrzymującą punkty w porządku niemalejącym rzędnych. Z kolei przy napotkaniu stacji dystrybucji, przydzielamy jej punkt spośród znajdujących się w miotle. Zauważmy, że dzięki odpowiedniemu kryterium wstępnego sortowania, każdy punkt p znajdujący sie w miotle może zostać połączony z napotkaną stacją s, jeżeli tylko $p_y \geqslant s_y$. Udowodnimy, że jeżeli do połączenia za każdym razem wybierzemy punkt z miotły o najmniejszej rzędnej nie mniejszej niż s_y, to na końcu otrzymamy poprawne przyporządkowanie (przypomnijmy, że w zadaniu mamy zagwarantowane istnienie przynajmniej jednego poprawnego przyporządkowania).

Zastanówmy się, dlaczego postępowanie według takiego zachłannego kryterium daje poprawny wynik. Wystarczy pokazać, że zawsze istnieje poprawne przyporządkowanie, w którym pierwszą rozważaną stację s połączymy z najniżej położonym punktem miotły p, dla którego $p_y \geqslant s_y$ — jeżeli to jest prawdą, to podobne rozumowanie można przeprowadzić dla kolejno rozważanych stacji. Niech R będzie poprawnym przyporządkowaniem, w którym stacja s nie jest połączona z punktem p, lecz z innym punktem p' i niech s' będzie stacją, z którą jest połączony punkt p. Z warunków budowy gazociągów wiemy, że $s'_y \leqslant p_y$, Natomiast z faktu, że algorytm zamiatania wybiera punkt p w sposób zachłanny wynika, że $p_y \leqslant p'_y$ (p' musiał znajdować się w miotle wraz z p w momencie przydzielania punktu do stacji s, a nie został wybrany). Razem nierówności te dają zależność $s'_y \leqslant p'_y$. Na podstawie kryterium wstępnego sortowania elementów wiemy, że $s'_x \geqslant s_x$, z kolei ze sposobu prowadzenia gazociągów wynika, że $s_x \geqslant p'_x$. Łącząc te nierówności, otrzymujemy $s'_x \geqslant p'_x$. Skoro więc $s'_y \leqslant p'_y$ oraz $s'_x \geqslant p'_x$, to punkt p' i stację s' można połączyć gazociągiem. Elementy p oraz s oczywiście także można połączyć. To pozwala nam wykonać w przyporządkowaniu R odpowiednią zamianę połączeń.



Zmiana połączeń między p, p', s i s'.

Otrzymujemy w ten sposób poprawne rozwiązanie, w którym p i s są połączone, co kończy dowód poprawności algorytmu zachłannego.

Ostatnią kwestią, jaką należy rozważyć, jest wybór struktury danych, reprezentującej miotłę. Współrzędne punktów możemy przechowywać w zwykłej tablicy i za każdym razem wybierać (w czasie liniowym względem liczby punktów w miotle, a więc pesymistycznie liniowym względem n) punkt spełniający kryterium zachłanne. Otrzymamy w ten sposób rozwiązanie o złożoności czasowej $O(n^2)$, zaimplementowane w pliku `gazs0.c`. Takie rozwiązanie uzyskiwało 40% punktów możliwych do zdobycia za zadanie. Można także spróbować wykorzystać strukturę w postaci zrównoważonego, binarnego drzewa poszukiwań, w którym w złożoności czasowej $O(\log n)$ można wyznaczyć dla danej wartości punkt o najmniejszej rzędnej nie mniejszej od niej; dzięki temu złożoność czasowa całego rozwiązania wyniesie $O(n \log n)$. Programując w C++, można sięgnąć po STL-owy kontener `set`, w którym dysponujemy odpowiednią metodą `lower_bound()` (implementacja rozwiązania z tą strukturą znajduje się w pliku `gaz.cpp`). Inną prostą w implementacji strukturą, którą można zastosować, jest statyczne drzewo licznikowe (skonstruowane na samym początku dla całego zakresu rzędnych). W strukturze tej w każdym węźle utrzymujemy liczbę punktów zawartych w poddrzewie, którego jest on korzeniem. Za pomocą jednego przejścia od korzenia drzewa do odpowiedniego liścia możemy zarówno wstawić nowy punkt do struktury, jak i wykonać zapytanie o punkt o najmniejszej rzędnej nie mniejszej od zadanej. Na podstawie ograniczeń z zadania możemy przyjąć, że maksymalna możliwa rzędna punktu jest rzędu $O(n)$, co pozwala oszacować złożoność czasową każdej z wyżej wymienionych operacji jako $O(\log n)$. Ponieważ drzewa licznikowe są ogólnie znane, to dokładny opis budowy oraz implementacji operacji na tej wersji struktury miotły pozostawiamy Czytelnikowi jako ćwiczenie. Implementacja rozwiązania z drzewem licznikowym znajduje się w plikach `gaz0.c` i `gaz1.pas`. Każde z rozwiązań wzorcowych ma złożoność pamięciową $O(n)$.

Inne rozwiązania

Zauważmy, że polecenie z zadania można przeformułować na poszukiwanie w grafie dwudzielnym (jedną grupą wierzchołków są punkty, drugą stacje, a krawędź istnieje wtedy, kiedy da się poprowadzić gazociąg) najtańszego doskonałego skojarzenia. Używając najlepszego implementowalnego w praktyce algorytmu, rozwiązującego ten problem (np. metody węgierskiej lub Busackera-Gowena z wykorzystaniem algorytmu Dijkstry — patrz opis rozwiązania zadania Szkoły z XIII Olimpiady Informatycznej), da się skonstruować rozwiązanie o złożoności czasowej $O(n^3)$. Dzięki spostrzeżeniu, że każde przyporządkowanie jest optymalne, można też zamiast najtańszego szukać po prostu dowolnego doskonałego skojarzenia w grafie. W pliku `gazs1.c` znajduje się rozwiązanie tego problemu, wykorzystujące metodę Edmondsa-Karpa (złożoność czasowa $O(n^3)$). Używając szybszych metod, na przykład algorytmu Hopcrofta-Karpa, można by otrzymać rozwiązanie o złożoności czasowej nawet $O(n^2\sqrt{n})$. Widać jednak wyraźnie, że sprowadzając nasz problem do wyznaczania skojarzenia w grafie dwudzielnym nie możemy osiągnąć zbyt efektywnych algorytmów, jako że samych krawędzi w tym grafie jest $O(n^2)$. Stwarza to także problemy w zakresie złożoności pamięciowej: przechowywanie wszystkich krawędzi w pamięci jest niemożliwe do zrealizowania dla dużych danych wejściowych, natomiast wyznaczanie krawędzi online jest z kolei czasochłonne. Rozwiązania tego typu nie uzyskiwały zazwyczaj więcej niż 30% punktów.

Najczęstszy błąd popełniany przez zawodników polegał na zapomnieniu o konieczności użycia typów całkowitych 64-bitowych przy wyznaczaniu sumy długości gazociągów w optymalnym przyporządkowaniu, co powodowało utratę 20% punktów.

Testy

Zadanie było sprawdzane na 10 zestawach testów. Większość testów została wygenerowana w sposób losowy. Dokładniejszą ich charakterystykę przedstawia poniższa tabelka:

Nazwa	n	Opis
gaz1a.in	6	test wygenerowany ręcznie
gaz1b.in	50	test wygenerowany ręcznie
gaz2a.in	100	test losowy gęsty
gaz2b.in	100	test losowy rzadki
gaz3a.in	100	test losowy gęsty
gaz3b.in	100	test losowy rzadki
gaz4a.in	1 000	test losowy gęsty
gaz4b.in	1 000	test losowy rzadki
gaz5a.in	30 000	test losowy gęsty
gaz5b.in	30 000	test losowy rzadki
gaz6a.in	50 000	test losowy gęsty
gaz6b.in	50 000	test losowy rzadki
gaz7a.in	40 000	test losowy gęsty
gaz7b.in	40 000	test losowy rzadki

Nazwa	**n**	**Opis**
gaz8a.in	40 000	test losowy gęsty
gaz8b.in	40 000	test losowy rzadki
gaz9a.in	40 000	test z dużym wynikiem
gaz9b.in	50 000	test losowy rzadki
gaz10a.in	50 000	test z dużym wynikiem
gaz10b.in	50 000	test losowy rzadki
gaz10c.in	50 000	wszystkie punkty leżą na jednej prostej i są posortowane

W testach gęstych wszystkie punkty i stacje znajdują się w jednym niedużym kwadracie, natomiast w testach rzadkich są one rozmieszczone na dużo większym obszarze.

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

OI, Etap III, dzień pierwszy, 28.03.2007

Odważniki

Bajtocki Instytut Fizyki Doświadczalnej przy przeprowadzce do nowego budynku napotkał na nie lada problem logistyczny — kłopotliwe okazało się przeniesienie bogatej kolekcji precyzyjnych odważników.

Instytut ma do dyspozycji pewną liczbę kontenerów, każdy o ograniczonej wytrzymałości. Należy pomieścić w nich jak najwięcej odważników, gdyż pozostałe, niestety, będzie trzeba wyrzucić. Do każdego z kontenerów można włożyć dowolnie wiele odważników, ale nie wolno przekroczyć przy tym jego wytrzymałości. Kontener może również pozostać pusty.

Dowolne dwa odważniki w Instytucie mają ciekawą własność: masa jednego z nich jest zawsze wielokrotnością masy drugiego z nich. W szczególności, oba odważniki mogą mieć równe masy.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia wytrzymałości kontenerów i masy odważników,*
- *wyznaczy maksymalną liczbę odważników, jakie można pomieścić w kontenerach,*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu wejścia znajdują się dwie liczby całkowite n *oraz* m ($1 \leqslant n, m \leqslant 100\,000$)*, oddzielone pojedynczym odstępem i oznaczające odpowiednio: liczbę kontenerów i liczbę odważników. Drugi wiersz wejścia zawiera* n *liczb całkowitych* w_i ($1 \leqslant w_i \leqslant 1\,000\,000\,000$ *dla* $1 \leqslant i \leqslant n$)*, pooddzielanych pojedynczymi odstępami i oznaczających wytrzymałości kontenerów, wyrażone w miligramach. W trzecim wierszu wejścia znajduje się* m *liczb całkowitych* m_j ($1 \leqslant m_j \leqslant 1\,000\,000\,000$ *dla* $1 \leqslant j \leqslant m$)*, pooddzielanych pojedynczymi odstępami i oznaczających masy odważników, również wyrażone w miligramach.*

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać jedną liczbę całkowitą — największą liczbę odważników, jakie można porozmieszczać w kontenerach bez przekroczenia ich wytrzymałości.

Przykład

Dla danych wejściowych:

```
2 4
13 9
4 12 2 4
```

poprawnym wynikiem jest:

```
3
```

W kontenerach można umieścić dowolne trzy z czterech odważników, ale nie można umieścić wszystkich odważników.

Rozwiązanie

Wprowadzenie

Odważniki z Bajtockiego Instytutu Fizyki Doświadczalnej spełniają następującą zależność: gdy weźmiemy dowolne dwa z nich, to *zawsze masa jednego z nich będzie całkowitą wielokrotnością masy drugiego*. Ta prosta własność ma poważne konsekwencje — spróbujmy je odkryć.

Uporządkujmy wszystkie odważniki według mas w kolejności niemalejącej: $m_1 \leqslant m_2 \leqslant \cdots \leqslant m_m$. Wówczas dla dowolnych dwóch sąsiednich odważników zachodzi:

$$k_i \cdot m_i = m_{i+1}, \quad i = 1, 2, \ldots, m-1,$$

gdzie k_i jest liczbą naturalną. Stąd

$$m_i = m_{i+1} \quad \text{albo} \quad 2 \cdot m_i \leqslant m_{i+1}.$$

To oznacza, że jeśli w ciągu mamy k różnych odważników $m_{i_1} < m_{i_2} < \ldots < m_{i_k}$, to

$$2^{j-1} \cdot m_{i_1} \;\leqslant\; m_{i_j},$$

dla $j = 1, 2, \ldots, k$. Ponieważ z ograniczeń zadania wiemy, że masy wszystkich odważników mieszczą się w przedziale $[1, 10^9]$, więc:

$$1 \leqslant m_{i_1} \quad \text{oraz} \quad 2^{k-1} \cdot m_{i_1} \leqslant m_{i_k} \leqslant 10^9.$$

Ponieważ $2^{30} = 1\,073\,741\,824 > 10^9 > 536\,870\,912 = 2^{29}$, więc nawet przyjmując minimalną masę pierwszego odważnika $m_{i_1} = 1$, mamy $k \leqslant 30$, co po prostu oznacza, że może być co najwyżej 30 różnych odważników!

Oznaczmy przez $M_1 < \cdots < M_k$ posortowane rosnąco wszystkie różne masy odważników, a przez $a_1, \ldots, a_k$ — liczby odważników o kolejnych masach. Aby wyznaczyć wartości M_i oraz a_i, wystarczy posortować zadany ciąg $m_1, \ldots, m_m$ i „zliczyć”, ile razy występują w nim poszczególne wartości. Jeżeli zastosujemy dobry, szybki algorytm sortowania, na przykład sortowanie przez scalanie, to wykonamy te obliczenia w złożoności czasowej $O(m \log m)$.

Istnieje jednak bardziej pomysłowy i efektywniejszy sposób. Wyznaczmy przedziały (*kubełki*) ograniczone kolejnymi potęgami dwójki: $[2^j, 2^{j+1} - 1]$ dla $j = 0, 1, \ldots, 29$. Zauważmy, że dwie liczby M_i, M_{i+1} nie mogą należeć do tego samego przedziału, bo jeśli $2^j \leqslant M_i < 2^{j+1}$, to $M_{i+1} \geqslant 2 \cdot M_i \geqslant 2 \cdot 2^j$. Wystarczy zatem podzielić odważniki na kubełki, a następnie wyznaczyć masę i liczbę odważników z każdego kubełka. Zakładając, że z jest największą masą odważnika, mamy $\log z$ kubełków (wiemy, że $\log z \leqslant 30$, ale być może jest to wartość jeszcze mniejsza, jeśli na przykład maksymalna masa odważnika jest istotnie mniejsza od zadanego ograniczenia). Przydział odważnika do kubełka możemy zatem wykonać za pomocą wyszukiwania binarnego w czasie $O(\log \log z)$. Przejrzenie kubełków i zliczenie zawartych w nich odważników wymaga czasu $O(\log z + m)$. Ze względu na mniejsze znaczenie, składnik $O(\log z + m)$ możemy pominąć — widzimy więc, że wartości M_i oraz a_i umiemy wyznaczyć w czasie $O(m \log \log z)$.

Przedstawiony algorytm jest ciekawy wyłącznie z teoretycznego punktu widzenia, ponieważ w praktyce sortowanie przez scalanie działa podobnie szybko.

Wiedząc, z jakimi odważnikami mamy do czynienia, ich optymalne przyporządkowanie do kontenerów możemy wyznaczyć na dwa istotnie różne sposoby.

Rozwiązanie wzorcowe – wersja pierwsza

W dalszej części przez *rozwiązanie* będziemy rozumieć dowolne poprawne przyporządkowanie odważników do kontenerów, natomiast przez *rozwiązanie optymalne* — dowolne rozwiązanie, w którym liczba zapakowanych odważników jest maksymalna.

Pierwsze spostrzeżenie przypomina nam, że interesuje nas tylko liczba zapakowanych odważników, a nie ich sumaryczna masa:

Twierdzenie 1 *Jeśli w rozwiązaniu optymalnym pakujemy do kontenerów K odważników, to istnieje rozwiązanie optymalne, w którym pakujemy do kontenerów K* ***najlżejszych*** *odważników.*

Dowód Rozważmy dowolne rozwiązanie optymalne, w którym użyliśmy odważników $m_{i_1} \leqslant m_{i_2} \leqslant \cdots \leqslant m_{i_K}$, a pozostałe odważniki to $m_{j_1} \leqslant m_{j_2} \leqslant \cdots \leqslant m_{j_L}$, gdzie $L = m - i_K$. Zauważmy, że podmieniając w rozwiązaniu dowolny odważnik na lżejszy, nie zmieniamy poprawności rozwiązania (nie przeciążamy kontenera) ani jego optymalności (nie zmieniamy liczby odważników) — dopóki więc $m_{i_K} > m_{j_1}$, to możemy podmieniać te dwa odważniki (po każdej takiej podmianie może zajść potrzeba ponownego posortowania ciągów). Gdy dojdziemy do $m_{i_K} \leqslant m_{j_1}$, wiemy, że mamy rozwiązanie optymalne złożone z odważników o najmniejszych masach. ■

Wartość K wyznaczymy metodą prób. Wiemy, że rozwiązaniem jest liczba z przedziału $[0, m]$. Będziemy przeszukiwać ten przedział binarnie, testując kolejne, potencjalne wartości K:

- dla ustalonej wartości K wybierzemy zestaw K najlżejszych odważników i sprawdźmy, czy da się je zapakować do kontenerów;
- jeśli tak, to spróbujemy zwiększyć liczbę K;
- jeśli nie, to zmniejszamy liczbę K.

Postępując zgodnie z tym schematem, po $O(\log m)$ testach znajdziemy optymalną wartość K. Pozostaje tylko opisać sposób sprawdzenia, czy K określonych odważników można zapakować do kontenerów. Okazuje się, że problem ten można rozwiązać metodą zachłanną. Dla zestawu odważników $m_1 \leqslant m_2 \leqslant \cdots \leqslant m_K$ wystarczy:

- rozważać odważniki w kolejności *od najcięższych do najlżejszych*: $m_K, m_{K-1}, \ldots, m_1$,

- każdorazowo umieszczać rozważany odważnik *w dowolnym kontenerze, w którym się on jeszcze zmieści.*

W dowodzie poprawności takiego rozwiązania kluczowe jest następujące spostrzeżenie.

Twierdzenie 2 *Rozważmy rozwiązanie R, w którym najcięższy odważnik x ma masę m_K i znajduje się w j-tym kontenerze. Wówczas dla dowolnego $t \in \{1,\dots,n\}$, jeżeli $m_K \leqslant w_t$ (czyli odważnik x mieści się w pustym, t-tym kontenerze), to R możemy przekształcić w rozwiązanie R_t, w którym odważnik x znajdzie się w t-tym kontenerze.*

Dowód Jeżeli łączna masa odważników, jakie w rozwiązaniu R zostały umieszczone w t-tym kontenerze, jest nie większa od m_K, to zamieniając miejscami te odważniki z odważnikiem x otrzymujemy szukane rozwiązanie R_t. Jeżeli w rozwiązaniu R łączna masa odważników w kontenerze t-tym jest większa niż m_K, to pokażemy, że można spośród nich wybrać taki zestaw, którego łączna masa jest równa dokładnie m_K. Wówczas zamieniając miejscami odważnik x i wybrany zestaw, także otrzymamy szukane rozwiązanie R_t.

Rozważmy więc kontener, w którym łączna masa odważników przekracza m_K. Dla wygody oznaczmy masy tych odważników $m'_1 \geqslant m'_2 \geqslant \dots \geqslant m'_s$. Wybierzmy zestaw odważników $m'_1, m'_2, \cdots, m'_j$, dobierając kolejne elementy od najcięższych do najlżejszych, tak długo, jak długo masa zestawu nie przekracza m_K, tzn.

$$m'_1 + \dots + m'_j \leqslant m_K < m'_1 + \dots + m'_{j+1}.$$

Okazuje się, że w ten sposób zawsze otrzymamy zestaw o masie równej dokładnie m_K! Uzasadnimy ten fakt, pokazując przez indukcję, że w każdym kroku konstrukcji zestawu masa każdego z pozostałych odważników dzieli masę, jakiej brakuje już utworzonemu zestawowi do m_K, tzn.

$$m'_k \quad | \quad m_K - (m'_1 + \dots + m'_i), \tag{1}$$

dla $0 \leqslant i \leqslant j$ oraz każdego $k = i+1, i+2, \dots, s$. Z własności tej wynika, że do zestawu zawsze bierzemy odważnik o masie nie większej niż różnica m_K i masy zestawu. Stąd mamy pewność, że w j-tym kroku masa konstruowanego zestawu osiągnie dokładnie m_K.

Przejdźmy do zapowiedzianego dowodu indukcyjnego własności (1).

Baza indukcji. Dla pustego zestawu pozostała do uzupełnienia masa wynosi m_K. Dla każdego z dostępnych odważników $m'_k \leqslant m_K$ oraz (co wynika ze specyficznych właściwości zestawu odważników) $m'_k \mid m_K$.

Krok indukcyjny. Załóżmy, że wybraliśmy już odważniki $m'_1 \geqslant m'_2 \geqslant \dots \geqslant m'_i$ i pozostało nam do wypełnienia $y = m_K - (m'_1 + m'_2 + \dots + m'_i)$ miligramów. Zauważmy, że $m'_{i+1} \mid y$, ponieważ:

- z założenia indukcyjnego $m'_i \mid m_K - (m'_1 + m'_2 + \dots + m'_{i-1}) = y + m'_i$, czyli także $m'_i \mid y$;
- wiemy także, że $m'_{i+1} \mid m'_i$, a więc $m'_{i+1} \mid y$.

Zauważmy wreszcie, że dla każdego $l > i+1$ zachodzi $m'_l \mid m'_{i+1}$, a zatem $m'_l \mid y$, co kończy dowód.

■

Korzystając z udowodnionego twierdzenia, możemy wykazać poprawność algorytmu zachłannego przydziału odważników do kontenerów. Przypomnijmy, że mamy K najlżejszych odważników $m_1, m_2, \dots, m_K$ i sprawdzamy, czy zmieszczą się one w kontenerach, przydzielając je zgodnie z zasadą:

- rozważamy odważniki w kolejności *od najcięższych do najlżejszych*: $m_K, m_{K-1}, \dots, m_2, m_1$;
- każdorazowo umieszczamy rozważany odważnik *w dowolnym kontenerze, w którym się on jeszcze zmieści.*

Jeśli istnieje upakowanie R tych odważników do kontenerów, to możemy je podmienić na dowolne rozwiązanie, w którym najcięższy odważnik jest w dowolnym kontenerze, w którym się mieści — wybierzmy kontener t, do którego przydzieli go algorytm zachłanny. Teraz możemy zmodyfikować problem, zmniejszając wytrzymałość kontenera t o masę m_K oraz usuwając odważnik m_K z zestawu. Ze zredukowanym problemem postępujemy analogicznie — odważnik o największej masie m_{K-1}, który gdzieś w rozwiązaniu R został przydzielony, przesuwamy do kontenera wybranego w rozwiązaniu zachłannym. Kontynuując tę przebudowę wyjściowego rozwiązania, dochodzimy do wniosku, że *jeśli istnieje rozwiązanie dla rozważanych odważników, to istnieje dla nich rozwiązanie zachłanne*. Wniosek przeciwny: *jeśli nie istnieje rozwiązanie, to nie istnieje także rozwiązanie zachłanne*, jest oczywisty.

Przekonawszy się o poprawności przedstawionego algorytmu, możemy zastanowić się nad jego efektywną implementacją. Przede wszystkim zauważmy, że zamiast zastanawiać się nad wyborem i umieszczać dany odważnik w *dowolnym* kontenerze, możemy go włożyć w szczególności do kontenera o *największej wytrzymałości*. Potrzebujemy więc struktury danych do przechowywania kontenerów, dla której będziemy w stanie wykonywać operacje:

- pobranie kontenera o największej wytrzymałości,

- wstawienie kontenera (o zmodyfikowanej wytrzymałości).

Do tego celu wygodnie jest użyć kolejkę priorytetową, czyli na przykład kopiec binarny (opis można znaleźć w [19]), który pozwala każdą z powyższych operacji wykonać w złożoności czasowej $O(\log n)$.

Możemy już zaprezentować kompletny algorytm:

```
Posortuj($m_1, m_2, \ldots, m_m$);
Wyznacz($M_i$,$a_i$);
{ Wyszukiwanie binarne wartości $K$. }
$d := 0$; $g := m$;
while $d < g$ do
begin
   $K := \lfloor \frac{d+g}{2} \rfloor$
   if Zapakuj($K$) then $d := K$;
   else $g := K - 1$;
end
return $d$;
```

```
function Zapakuj($K$);
begin
   { Próbujemy zapakować $K$ najlżejszych odważników. }
   $S := Kopiec(w_1, \ldots, w_n)$;
   for $i := K$ downto 1 do
   begin
      $t$ :=Weź_Największy($S$);
      if $w_t < m_i$ then
         return false;
      Wstaw($S$,$t$,$w_t - m_i$);
   end
   return true;
end
```

Złożoność. Przyjrzyjmy się jeszcze złożoności czasowej powyższego algorytmu. Każde wywołanie funkcji *Zapakuj* jest wykonywane w złożoności czasowej $O(n)$ (koszt konstrukcji n-elementowego kopca binarnego) plus $O(m \log n)$ ($K \leqslant m$ operacji wstawiania/usuwania na kopcu n-elementowym). Funkcję *Zapakuj* wywołujemy $O(\log m)$ razy z racji binarnego wyszukiwania wartości K w przedziale $[0, m]$. Cały algorytm ma zatem złożoność $O((n + m \log n) \log m)$, niezależnie od tego, na ile efektywną metodę wstępnego sortowania odważników wybierzemy. Algorytm ten został zaimplementowany w plikach `odw3.cpp`, `odw8.pas` (własna implementacja kopca) oraz `odw2.cpp` (kopiec z biblioteki STL).

Dalsze usprawnienia

Poniżej przedstawimy, jak zrealizować efektywniej zaprezentowane rozwiązanie. Uzyskana poprawa czasu działania jest jednak praktycznie „niemierzalna”, więc na zawodach obie wersje rozwiązania uzyskiwały taką samą ocenę — maksymalną. Opisane w niniejszej sekcji usprawnienia są więc głównie interesujące z teoretycznego punktu widzenia i można je potraktować jako ciekawostkę.

Na początek skoncentrujmy się na ulepszeniu implementacji funkcji *Zapakuj*. Zauważmy, że w skonstruowanym algorytmie w żadnym miejscu nie skorzystaliśmy z faktu, że liczba różnych mas odważników jest bardzo mała, rzędu $O(\log z)$ (przypomnijmy, że przez z oznaczyliśmy maksymalną masę odważnika M_k). Spróbujmy zastąpić kopiec bardziej efektywną w tym przypadku strukturą. Dla każdej wartości M_i ($i \in \{1, \ldots, k\}$) utwórzmy listę L_i tych kontenerów, których wytrzymałości są zawarte w przedziale $[M_i, M_{i+1} - 1]$ (dla wygody przyjmujemy, że $M_{k+1} = \infty$). Będziemy teraz rozważać odważniki w kolejności niemalejących mas $M_k, \ldots, M_1$. Dla każdego M_i wykonujemy następujące czynności:

1. jeżeli $L_{i+1} \neq \emptyset$, to scalamy listy L_i oraz L_{i+1};
2. dla każdego spośród a_i odważników o masie M_i:
   (a) bierzemy dowolny kontener $w_j \in L_i$ — jeżeli takiego kontenera nie ma, to kończymy działanie z informacją, że odważników nie da się zapakować;
   (b) w przeciwnym razie wstawiamy odważnik do kontenera w_j oraz

(c) zmieniamy wytrzymałość kontenera na $w_j - M_i$ i umieszczamy go na właściwej liście (listę odnajdujemy, na przykład, za pomocą wyszukiwania binarnego).

Zauważmy, że w tym algorytmie wykorzystujemy fakt, że odważnik możemy umieścić w *dowolnym* dostatecznie wytrzymałym kontenerze, a niekoniecznie tylko w *najbardziej wytrzymałym*.

Przeanalizujmy koszty czasowe poszczególnych kroków opisanego algorytmu. Znalezienie za pomocą wyszukiwania binarnego listy, na której powinien zostać umieszczony rozważany kontener, to $O(\log\log z)$, gdyż liczba list jest rzędu $O(\log z)$. Początkowe rozmieszczenie kontenerów na listach można więc wykonać w złożoności czasowej $O(n\log\log z)$, wykorzystując wielokrotnie wyszukiwanie binarne. Połączenie list w punkcie 1. można wykonać w czasie stałym, a w ciągu całego algorytmu takich połączeń wykonamy najwyżej $\log z$, stąd sumaryczny czas tego kroku to $O(\log z)$. Operacje opisane w punktach 2a-2c wykonujemy dla każdego odważnika, czyli łącznie $K \leqslant m$ razy, przy czym operacje opisane w punktach 2a oraz 2b mają złożoność czasową $O(1)$, a operacja z punktu 2c ma złożoność $O(\log\log z)$. Razem daje to złożoność $O(m(1+\log\log z))$. Cała funkcja *Zapakuj* ma więc złożoność równą $O((n+m)\log\log z)$. Ulepszenie wydaje się być nikłe, ale pozbyliśmy się z rozwiązania kopca, który może nie być wszystkim znany. Rozwiązanie wykorzystujące zmodyfikowaną wersję funkcji `Zapakuj` zostało zaimplementowane w plikach `odw4.cpp` oraz `odw9.pas`.

Dalsze usprawnienia możemy uzyskać, zamieniając binarne wyszukiwanie wartości K na wyszukiwanie liniowe! Może się to wydawać w pierwszej chwili dziwne, gdyż zamiana $\log m$ testów na m testów nie wydaje się być dobrym pomysłem na poprawę efektywności. Oszczędność jednak wyniknie z możliwości bazowania na rozmieszczeniu odważników z poprzedniego testu przy konstrukcji rozmieszczenia w kolejnym teście. Aby zauważyć zależność pomiędzy takimi rozmieszczeniami, załóżmy, że mamy posortowane odważniki $m_m \geqslant m_{m-1} \geqslant \cdots \geqslant m_1$ i rozpocznijmy od testu dla $K = m$. Wówczas możliwe są dwie sytuacje.

- Udaje się nam zapakować wszystkie odważniki i możemy stwierdzić, że wynikiem jest $K = m$.
- Pakujemy odważniki $m_m, m_{m-1}, \ldots, m_{i+1}$, a dla m_i nie ma już miejsca. Wówczas przechodzimy do testu dla $K = m-1$.

Istotą ulepszenia jest takie przejście do testu dla $K = m-1$, przy którym korzystamy z upakowania odważników dla $K = m$ — robimy to następująco:

- usuwamy odważnik m_m z kontenera;
- odważniki $m_{m-1}, \ldots, m_{i+1}$ pozostawiamy na dotychczasowych miejscach;
- odważnik m_i umieszczamy w kontenerze, w którym poprzednio był m_m;
- kontynuujemy rozmieszczanie kolejnych odważników: $m_{i-1}, m_{i-2}, \ldots$.

W ten sposób, kosztem modyfikacji wytrzymałości jednego kontenera przechodzimy od jednego testowanego rozmieszczania do kolejnego. Przypomnijmy, że w poprzednim rozwiązaniu konieczne było rozmieszczenie wszystkich odważników od początku. Pseudokod ostatecznej wersji rozwiązania, w której zastosowaliśmy oba ulepszenia, prezentujemy poniżej. Zakładamy przy tym dla uproszczenia, że najcięższy odważnik mieści się w jakimś kontenerze (wszystkie odważniki niemieszczące się w żadnym kontenerze możemy odrzucić na samym początku).

```
Posortuj($m_i$);
Wyznacz($M_i$,$a_i$);
for $i := 1$ to $m$ do umieszczony[$i$] := nigdzie;
$K := m$; { Parametr, który będziemy zmniejszać przy kolejnych porażkach. }
for $i := m$ downto 1 do
begin
  if Da_Się_Umieścić($m_i$) then
    $t$ :=Weź_Kontener_Nie_Mniejszy_Od($m_i$);
  else
  begin
    $t$ := umieszczony[$K$];
    umieszczony[$K$] := nigdzie;
    $w_t = w_t + m_K$;
    $K := K - 1$;
  end
  umieszczony[$i$] := $t$;
  $w_t = w_t - m_i$;
end
return $K$;
```

Dokładna implementacja algorytmu zależy od wyboru struktury, w której będziemy przechowywać kontenery. Zastosujemy do tego celu opisane wcześniej „kubełki” — tablicę $\log z$ list kontenerów o wytrzymałościach z przedziału $[M_i, M_{i+1} - 1]$. Inicjalizacja struktury wymaga czasu $O(n \log\log z)$, a każda z instrukcji dostępu do kontenera (wyszukanie, wstawienie i usunięcie odważnika) jest wykonywana w złożoności $O(\log\log z)$. Daje to łączną złożoność czasową rozwiązania $O((n+m)\log\log z)$ w przypadku zastosowania efektywnego algorytmu wstępnego sortowania mas odważników. Szacowanie złożoności jest w tym przypadku nieco sztuczne, gdyż samo wczytanie wejścia wymaga wykonania około $(n+m)\log z$ operacji (wszak wejście jest w praktyce wczytywane znak po znaku, a długość zapisu dziesiętnego liczby naturalnej z to $\lfloor \log_{10} z \rfloor$!). Implementacje tego rozwiązania znajdują się w plikach `odw5.cpp` oraz `odw10.pas`.

Rozwiązanie wzorcowe – wersja druga

W pierwszej wersji rozwiązania wzorcowego odważniki umieszczaliśmy w kontenerach w kolejności od najcięższych do najlżejszych. Okazuje się, że można również poszukiwać rozwiązania, analizując odważniki począwszy od najlżejszych. Jednak w tym przypadku tak proste kryterium zachłanne, jak w pierwszej wersji rozwiązania, nie jest już poprawne. Umieszczając za każdym razem najlżejszy odważnik w najmniej (lub najbardziej) wytrzymałym z pozostałych kontenerów, nie otrzymamy poprawnego algorytmu (zachęcamy Czytelnika do znalezienia przykładu, który pozwala to wykazać). Musimy więc zastosować nieco subtelniejszą regułę.

W dalszej części dla uproszczenia przyjmiemy założenie, że najlżejszy odważnik w zestawie ma masę 1 ($M_1 = 1$). Zauważmy, że bez straty ogólności możemy przekształcić rozważany przypadek w spełniający to ograniczenie. Jeżeli w danym zestawie mamy $M_1 > 1$, to możemy zarówno masy wszystkich odważników, jak i wytrzymałości kontenerów podzielić (całkowitoliczbowo) przez M_1. Dzielenie w przypadku odważników będzie dokładne (i tak masy wszystkich odważników są podzielne przez M_1). W przypadku kontenera, jeżeli reszta z dzielenia jego wytrzymałości przez M_1 jest niezerowa, to i tak tej reszty nie dałoby się wykorzystać przy pakowaniu, więc możemy o nią zmniejszyć kontener.

Przy założeniu, że $M_1 = 1$, wprowadźmy następującą definicję.

Definicja 1 *Rozkładem* liczby naturalnej x przy układzie mas odważników $M_1, \ldots, M_k$ nazywamy taki ciąg $r_1, \ldots, r_k$, że $0 \leqslant r_i < \frac{M_{i+1}}{M_i}$ dla $i = 1, \ldots, k-1$ oraz $x = \sum_{i=1}^{n} r_i M_i$.

Przykład 1 Przyjrzyjmy się rozkładowi liczb z przykładu w treści zadania. Mamy tam do czynienia z ciągiem mas $M = (2, 4, 12)$, co po podzieleniu przez $M_1 = 2$ daje $M' = (1, 2, 6)$. Wytrzymałości kontenerów równe 13 i 9 zostają przekształcone do wartości 6 i 4. Rozkładami kilku przykładowych liczb przy tym układzie odważników są:

- dla 5 ciąg $1, 2, 0$,
- dla 7 ciąg $1, 0, 1$,
- dla 21 ciąg $1, 1, 3$.

■

Wprowadzone pojęcie rozkładu jest analogiczne do zapisu liczby w określonym systemie pozycyjnym (np. binarnym, dziesiątkowym), gdyż układ $M_1, \ldots, M_k$ jest uogólnieniem układów postaci $b^0, b^1, \ldots, b^{k-1}$ dla $b \geqslant 2$ całkowitego. Analogicznie jak w przypadku układów pozycyjnych, można pokazać, że dla liczby x istnieje dokładnie jeden rozkład. Co więcej, można go skonstruować „tradycyjnie”: pomniejszając wartość x o M_k aż do momentu, kiedy $x < M_k$, następnie pomniejszając wartość x o M_{k-1} itd. Udowodnienie opisanych własności rozkładów pozostawiamy Czytelnikowi jako łatwe ćwiczenie.

Podstawą drugiej wersji rozwiązania wzorcowego jest fakt, że możemy bez zmniejszenia wyniku zamienić każdy kontener na zestaw kontenerów o wytrzymałościach odpowiadających jego rozkładowi w układzie $M_1, \ldots, M_k$. Spostrzeżenie to formalizujemy następująco:

Twierdzenie 3 *Niech dany będzie kontener o wytrzymałości w, do którego zapakowaliśmy odważniki o masach $\mu_1 \leqslant \mu_2 \leqslant \ldots \leqslant \mu_s$. Niech dalej $r_1, \ldots, r_k$ będzie rozkładem w. Wówczas te same odważniki można zapakować do zestawu kontenerów $(r_1 \times M_1, \ldots, r_k \times M_k)$, czyli złożonego z: r_1 kontenerów o wytrzymałości M_1, …, r_k kontenerów o wytrzymałości M_k.*

Dowód Pokażemy, jak rozmieścić odważniki $\mu_1, \mu_2, \ldots, \mu_s$ w zestawie kontenerów $(r_1 \times M_1, \ldots, r_k \times M_k)$.

Rozdrobnieniem układu kontenerów R nazwiemy układ kontenerów powstały przez podzielenie niektórych kontenerów z R na mniejsze. Sformułujmy dwa przydatne dalej spostrzeżenia:

- Jeśli zestaw odważników da się rozmieścić w rozdrobnieniu układu kontenerów R, to da się także rozmieścić w układzie kontenerów R. Stąd wynika, że w trakcie rozmieszczania odważników w układzie kontenerów $(r_1 \times M_1, \ldots, r_k \times M_k)$ możemy rozdrabniać ten układ — jeśli ostatecznie rozmieścimy w nim odważniki, to daje się je także rozmieścić w układzie początkowym.

- Jeśli $r_1,\dots,r_j$ jest rozkładem liczby w w układzie $M_1,\dots,M_j$, to $\sum_{i=1}^{j-1} r_i \cdot M_i \;< M_j$. Tę zależność można wywnioskować, na przykład, z zachłannej metody konstrukcji rozkładu liczby względem $M_1,\dots,M_j$.

Odważniki będziemy rozmieszczać w kontenerach w kolejności od najcięższego do najlżejszego: $\mu_s, \mu_{s-1}, \dots, \mu_1$. W każdym kroku będziemy modyfikować początkowy zestaw kontenerów tak, by po umieszczeniu odważnika μ_i o masie M_j mieć zestaw pustych kontenerów $(r_1 \times M_1, \dots, r_j \times M_j)$, gdzie $r_1,\dots,r_j$ jest rozkładem $w - (\mu_s + \dots + \mu_i)$ względem $M_1,\dots,M_j$.

Stan zerowy Na początku rzeczywiście mamy układ kontenerów $(r_1 \times M_1, \dots, r_k \times M_k)$, gdzie $r_1,\dots,r_k$ jest rozkładem w względem $M_1,\dots,M_k$.

Kolejny krok Niech M_ℓ będzie masą kolejnego rozważanego odważnika μ_t ($t = s, s-1, \dots, 1$). Jeśli $\ell < j$, to rozdrobnijmy wszystkie kontenery o wytrzymałości większej niż M_ℓ na kontenery o wytrzymałości M_ℓ — uzyskujemy nowy zestaw kontenerów $(r_1 \times M_1, \dots, r_\ell \times M_\ell)$, gdzie — jak łatwo zauważyć — $r_1,\dots,r_\ell$ jest rozkładem $w - (\mu_s + \dots + \mu_{t+1})$ względem $M_1 \dots, M_\ell$. Ponieważ μ_t mieści się w tym zestawie, więc $M_\ell \leqslant \sum_{p=1}^{\ell} r_p \cdot M_p$. To z kolei oznacza, że $r_\ell \geqslant 1$, więc w zestawie mamy kontener, w którym zmieści się μ_t. Umieszczamy w nim ten odważnik i usuwamy z zestawu zapełniony kontener (czyli zmniejszamy o jeden r_ℓ).

Przed przejściem do rozważania kolejnego odważnika mamy więc zachowany niezmiennik: zestaw pustych kontenerów to układ $(r_1 \times M_1 \dots, r_\ell \times M_\ell)$, gdzie $r_1,\dots,r_\ell$ jest rozkładem względem $M_1,\dots,M_\ell$ pozostałej wytrzymałości zestawu $w - (\mu_s + \dots \mu_t)$.

Stan końcowy Niezmiennik, który zachowujemy w trakcie rozmieszczania odważników, gwarantuje, że uda się nam umieścić wszystkie odważniki w rozdrobnieniu układu początkowego, a więc można je także umieścić w układzie początkowym, co należało pokazać.

■

Na mocy pokazanego twierdzenia, zamiast rozważać pojemniki o wytrzymałościach w_i, możemy dla każdego w_i znaleźć rozkład i rozważać problem upakowania odważników do kontenerów o wytrzymałościach należących do zbioru $\{M_1,\dots,M_k\}$. Wynikowy zestaw kontenerów może być teraz bardzo duży, niemniej jednak rodzajów kontenerów będzie tylko $O(\log z)$. Okazuje się, że dla tak otrzymanego zestawu kontenerów poprawne jest rozwiązanie zachłanne, w którym umieszczamy odważniki w kolejności od najlżejszych do najcięższych, każdorazowo wkładając dany odważnik do kontenera o najmniejszej możliwej wytrzymałości. Po umieszczeniu odważnika w kontenerze, możemy — na mocy Twierdzenia 3 — pozostałą wytrzymałość kontenera rozłożyć na kontenery o rozmiarach należących do zbioru $\{M_1,\dots,M_k\}$, dzięki czemu w żadnym momencie nie będziemy operować na innych wytrzymałościach kontenerów niż wymienione.

Zanim udowodnimy poprawność tego rozwiązania, zapiszmy je w pseudokodzie:

```
Posortuj($m_i$);
Wyznacz($M_i$,$a_i$);
{ Konstruujemy rozdrobnienie początkowego zestawu kontenerów }
{ na kontenery o wytrzymałościach $M_1,\dots,M_k$. }
for $i := 1$ to $k$ do $kontenery[i] := 0$;
for $i := 1$ to $n$ do Rozłóż($w_i$);
{ Rozmieszczamy odważniki od najlżejszych do najcięższych. }
$przetworzone := 0$;
for $i := 1$ to $k$ do { Rozważamy odważniki o masie $M_i$. }
  for $j := 1$ to $a_i$ do
  begin
    $gdzie := i$;
    while $gdzie \leqslant k$ and $kontenery[gdzie] = 0$ do
      $gdzie := gdzie + 1$;
    if $gdzie > k$ then
      { Nie da się umieścić odważnika — kończymy. }
      return $przetworzone$;
    else
      $przetworzone := przetworzone + 1$;
    $kontenery[gdzie] := kontenery[gdzie] - 1$;
    Rozłóż($M_{gdzie} - M_i$);
  end
{ Wszystkie odważniki udało się rozmieścić. }
return $m$;
```

```
procedure Rozłóż($x$);
begin
  { Rozkład wykonujemy za pomocą algorytmu zachłannego. }
  for $i := k$ downto 1 do
  begin
    $kontenery[i] := kontenery[i] + \lfloor x/M_i \rfloor$;
    $x := x - M_i \cdot \lfloor x/M_i \rfloor$;
  end
end
```

Złożoność. Rozkład w_i możemy znaleźć w złożoności czasowej $O(\log z)$ za pomocą opisanego powyżej algorytmu zachłannego, co daje koszt rozdrobnienia kontenerów równy $O(n \log z)$. Złożoność czasowa głównej pętli powyższego algorytmu to, jak łatwo widać, $O(m \log z)$, gdyż dla każdego odważnika co najwyżej raz przeglądamy listę wszystkich wytrzymałości kontenerów oraz rozdrabniamy co najwyżej jeden kontener. Stąd cały algorytm ma złożoność $O((n+m)\log z)$, jeżeli założyć, że sortowanie odważników wykonujemy bardziej efektywną metodą. Algorytm jest więc podobnie efektywny jak pierwsze rozwiązanie wzorcowe.

Poprawność. Zastanówmy się na koniec nad poprawnością tego algorytmu. Z Twierdzenia 1 wynika, że możemy umieszczać odważniki, począwszy od najlżejszych. Pozostaje wyjaśnić, dlaczego możemy rozważany odważnik za każdym razem umieszczać w najmniejszym z pozostałych kontenerów.

Porównajmy w tym celu działanie naszego algorytmu A i pewnego algorytmu optymalnego O, który także przydziela odważniki do układu rozdrobnionych kontenerów. Oznaczmy przez x masę najmniejszego odważnika, który został różnie rozmieszczony w tych dwóch rozwiązaniach. Powiedzmy, że:

- w algorytmie A włożyliśmy x do kontenera K_1,
- w algorytmie O włożyliśmy x do kontenera K_2 (o wytrzymałości $w_{K2} > w_{K1}$; ponadto zachodzi $w_{K1} \mid w_{K2}$).

Po umieszczeniu w kontenerze odważnika x w algorytmie A rozkładamy kontener o wytrzymałości $w_{K1} - x$, a w algorytmie O rozkładamy kontener o wytrzymałości $w_{K2} - x = (w_{K2} - w_{K1}) + (w_{K1} - x)$. Porównajmy zestawy kontenerów w obu rozwiązaniach po przeprowadzonej operacji.

- W algorytmie A mamy pusty cały kontener o wytrzymałości w_{K2} i kontenery $L_1, \dots, L_t$ powstałe z rozdrobnienia kontenera o wytrzymałości $w_{K1} - x$.
- W algorytmie O pozostał pusty kontener o wytrzymałości w_{K1}, a rozdrobniliśmy kontener o wytrzymałości $(w_{K2} - w_{K1}) + (w_{K1} - x)$. Ponieważ pierwszy składnik tej sumy jest podzielny przez w_{K1}, to oznacza, że w wyniku rozkładu dostaliśmy kontenery $N_1, \dots, N_s$ oraz $L_1, \dots, L_t$, gdzie kontenery z pierwszej grupy mają wytrzymałości podzielne przez w_{K1}, a kontenery z drugiej grupy są identyczne jak otrzymane z rozkładu K_1 w algorytmie A.

Przedstawione porównanie wykazuje, że układ kontenerów w algorytmie optymalnym jest rozdrobnieniem układu kontenerów z algorytmu A, stąd każde rozmieszczenie uzyskane za pomocą algorytmu optymalnego zostanie także znalezione za pomocą algorytmu A, co dowodzi optymalności algorytmu A.

Implementacje przedstawionego w tym rozdziale rozwiązania wzorcowego znajdują się w plikach `odw.cpp` i `odw7.pas`.

Testy

Zadanie było sprawdzane na 10 zestawach testów. W poniższej tabelce n oznacza liczbę kontenerów, m to liczba odważników, a K to liczba odważników rozmieszczonych w kontenerach w optymalnym rozwiązaniu.

W opisach testów przyjęto następujące nazewnictwo:

- *test poprawnościowy:* mały test, w którym do pewnego kontenera trzeba włożyć zarówno duże, jak i małe odważniki;
- *test małych odważników:* mały test, w którym jest więcej małych odważników; odważniki trzeba dobrze rozdzielić między dwa kontenery, tak by zmieściły się najcięższe z nich;
- *test potęg dwójki:* duży losowy test, w którym masy odważników są potęgami dwójki;

- *test grupowania:* test losowy, w którym liczba kontenerów jest zbliżona do połowy liczby odważników, a suma wytrzymałości kontenerów jest równa dokładnie sumie mas odważników; aby uzyskać optymalne rozmieszczenie, trzeba umieszczać w kontenerach zarówno małe, jak i duże odważniki; w testach „grupowania 2” masy odważników są potęgami 2, natomiast w testach „grupowania 3-2” są one elementami ciągu $1,3,6,18,36,\ldots$;
- *test całkowitego rozmieszczenia:* duży test losowy, w którym masy odważników są potęgami dwójki; w kontenerach daje się rozmieścić wszystkie odważniki;
- *test jednostkowych mas:* duży test losowy, w którym połowa odważników ma masy 1, a masy pozostałych są potęgami 2 („jednostkowe masy 2”) albo elementami ciągu $1,3,6,18,36,\ldots$ („jednostkowe masy 3-2”); w rozwiązaniu optymalnym do każdego kontenera trzeba włożyć odważnik o masie równej 1.

Nazwa	**n**	**m**	**K**	**Opis**
odw1a.in	3	9	8	test poprawnościowy
odw1b.in	4	6	5	test poprawnościowy
odw1c.in	5	4	0	test z zerowym wynikiem
odw1d.in	2	8	7	test małych odważników
odw2a.in	3	9	8	test poprawnościowy
odw2b.in	3	7	6	test poprawnościowy
odw2c.in	2	5	3	test na przypadek brzegowy
odw2d.in	3	8	7	test małych odważników
odw3a.in	100 000	100 000	96 636	test potęg dwójki
odw3b.in	50 047	100 000	99 999	test grupowania 3-2
odw4a.in	100 000	100 000	96 663	test potęg dwójki
odw4b.in	50 117	100 000	99 999	test grupowania 2
odw4c.in	50 000	100 000	100 000	test jednostkowych mas 3-2
odw5a.in	100 000	100 000	96 634	test potęg dwójki
odw5b.in	49 843	100 000	99 999	test grupowania 3-2
odw6a.in	100 000	100 000	96 693	test potęg dwójki
odw6b.in	49 986	100 000	99 999	test grupowania 2
odw7a.in	100 000	100 000	96 699	test potęg dwójki
odw7b.in	50 064	100 000	99 999	test grupowania 2
odw8a.in	100 000	100 000	96 646	test potęg dwójki
odw8b.in	49 982	100 000	99 999	test grupowania 2
odw9a.in	100 000	100 000	96 696	test potęg dwójki
odw9b.in	50 001	100 000	99 999	test grupowania 3-2
odw9c.in	100 000	100 000	100 000	test całkowitego rozmieszczenia
odw9d.in	50 000	100 000	100 000	test jednostkowych mas 2
odw10a.in	100 000	100 000	96 726	test potęg dwójki
odw10b.in	50 004	100 000	99 999	test grupowania 3-2
odw10c.in	100 000	100 000	100 000	test całkowitego rozmieszczenia
odw10d.in	50 000	100 000	100 000	test jednostkowych mas 2

Rozwiązania zawodników o wykładniczej złożoności czasowej względem n oraz m przechodziły pierwsze dwa testy. Proste rozwiązania zachłanne z błędnym kryterium (na przykład rozwiązanie umieszczające odważniki od najlżejszych w najmniejszych czy też największych możliwych kontenerach) nie przechodziły żadnych testów. Wszystkie wersje rozwiązania wzorcowego przechodziły wszystkie testy.

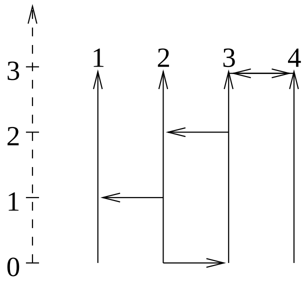
Jakub Radoszewski
Treść zadania, Opracowanie

Michał Brzozowski
Program

OI, Etap III, dzień drugi, 29.03.2007

Egzamin na prawo jazdy

Bajtocki egzamin na prawo jazdy odbywa się na placu, na którym znajduje się n prostych równoległych jednokierunkowych ulic, skierowanych z południa na północ. Każda z ulic ma długość m metrów, wszystkie ulice zaczynają się i kończą na tej samej wysokości. Są one ponumerowane od 1 do n, w kolejności z zachodu na wschód. Na placu znajduje się też p prostopadłych do nich jednokierunkowych ulic, skierowanych ze wschodu na zachód lub z zachodu na wschód i łączących pewne sąsiednie ulice biegnące z południa na północ. Może istnieć dowolnie wiele ulic biegnących ze wschodu na zachód lub z zachodu na wschód, łączących daną parę sąsiednich ulic biegnących z południa na północ. Możliwa jest też sytuacja, w której pewne dwie ulice, jedna biegnąca ze wschodu na zachód, a druga z zachodu na wschód, pokrywają się, tworząc ulicę dwukierunkową.



Przykładowy plac egzaminacyjny ($n = 4$, $m = 3$, $p = 5$).

W trakcie egzaminu egzaminator wybiera ulicę biegnącą z południa na północ, na początku której rozpocznie się egzamin oraz ulicę (również biegnącą z południa na północ), na końcu której egzamin ma się zakończyć. Zadaniem egzaminowanego jest przejechać — oczywiście zgodnie z kierunkami ulic jednokierunkowych — z miejsca, gdzie egzamin się zaczyna, do miejsca, gdzie się kończy.

Żeby uniknąć sytuacji, w której nie istniałaby droga z punktu początkowego egzaminu do punktu końcowego, egzaminatorzy zawsze jako ulicę startową wybierają jedną z takich ulic, z których początku da się dojechać do końca **dowolnej** *innej ulicy biegnącej z południa na północ.*

Praca egzaminatorów jest bardzo monotonna, gdyż ciągle rozpoczynają egzaminy na początku tych samych ulic. Dyrekcja postanowiła wybudować nowy plac, na podstawie istniejących już planów. Obliczono, że funduszy starczy na dodanie nie więcej niż k ulic biegnących ze wschodu na zachód lub z zachodu na wschód. Jednak należy tak dobrać te ulice, by przybyło jak najwięcej potencjalnych punktów początkowych egzaminu (na istniejącym planie mogą, ale nie muszą istnieć ulice będące punktami początkowymi). Dobudowane ulice muszą łączyć pewne pary sąsiednich ulic biegnących z południa na północ.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis planu placu egzaminacyjnego oraz liczbę k,*
- *wyznaczy maksymalną liczbę potencjalnych punktów początkowych egzaminu, jakie mogą się pojawić po dodaniu co najwyżej k ulic biegnących ze wschodu na zachód lub z zachodu na wschód,*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu wejścia znajdują się cztery liczby całkowite n, m, p oraz k ($2 \leqslant n \leqslant 100\,000$, $1 \leqslant m,k \leqslant 100\,000$, $0 \leqslant p \leqslant 100\,000$), pooddzielane pojedynczymi odstępami i oznaczające odpowiednio: liczbę ulic biegnących z południa na północ, długość każdej z tych ulic, liczbę już istniejących ulic biegnących ze wschodu na zachód lub z zachodu na wschód oraz maksymalną liczbę ulic, jakie można dobudować. Ulice biegnące z południa na północ są ponumerowane od 1 do n, w kolejności z zachodu na wschód.

Kolejnych p wierszy zawiera po trzy liczby całkowite n_i, m_i oraz d_i ($1 \leqslant n_i < n$, $0 \leqslant m_i \leqslant m$, $d_i \in \{0,1\}$), pooddzielane pojedynczymi odstępami i opisujące i-tą ulicę biegnącą z zachodu na wschód (dla $d_i = 0$) bądź ze wschodu na zachód (dla $d_i = 1$). Ulica ta łączy ulice biegnące z południa na północ o numerach n_i i $n_i + 1$ oraz łączy się z nimi w punktach odległych o m_i metrów od ich początków.

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać jedną liczbę całkowitą, równą maksymalnej liczbie nowych punktów początkowych egzaminu, jakie mogą pojawić się na placu po dobudowaniu co najwyżej k ulic biegnących ze wschodu na zachód lub z zachodu na wschód. Dobudowane ulice **nie muszą** *łączyć się z ulicami biegnącymi z południa na północ w punktach oddalonych o całkowitą liczbę metrów od ich początków. Dobudowane ulice biegnące ze wschodu na zachód mogą pokrywać się z ulicami biegnącymi z zachodu na wschód i odwrotnie. W ten sposób ulice jednokierunkowe zmieniają się w dwukierunkowe.*

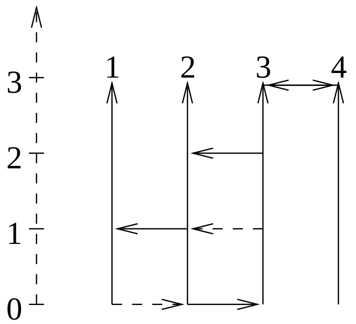
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
2
```



Nowymi punktami początkowymi egzaminu mogą być, na przykład, początki ulic nr 1 i 3.

Rozwiązanie

Rozwiązanie wzorcowe

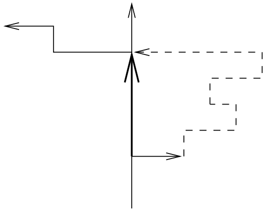
Analiza problemu

Wprowadźmy kilka pojęć, które pozwolą nam czytelniej zaprezentować rozwiązanie. Ulice prowadzące w kierunku południe-północ będziemy nadal nazywać *ulicami*, natomiast ulice prowadzące z zachodu na wschód i ze wschodu na zachód nazwiemy odpowiednio *łącznikami wschodnimi* oraz *łącznikami zachodnimi*. Powiemy, że łącznik jest *na wysokości* k, jeśli łączy punkty dwu sąsiednich ulic położone k jednostek od początku obu ulic. *Trasą* nazwiemy połączenie początku pewnej ulicy i z końcem pewnej ulicy j poprowadzone zgodnie z kierunkami ulic i łączników. Powiemy także, że ulica i-ta *jest połączona* z ulicą j-tą, jeśli istnieje trasa łącząca ulicę i-tą z ulicą j-tą. W takiej sytuacji będziemy również mówić, że z ulicy i-tej *można dojechać* do ulicy j-tej.

Przystąpmy teraz do opisu rozwiązania. Rozpoczniemy od kilku podstawowych spostrzeżeń, które pozwolą nam uprościć problem.

Po pierwsze zauważmy, że egzamin może rozpoczynać się na początku ulicy o numerze i wtedy i tylko wtedy, gdy jest ona połączona z ulicą pierwszą oraz z ulicą n-tą. Jeśli bowiem z ulicy i-tej (dla $i \in \{1,\ldots,n\}$) da się dojechać do ulicy pierwszej, to przerywając podróż odpowiednio wcześniej, możemy zakończyć ją na dowolnej spośród ulic $1,2,\ldots,i-1$; podobnie wygląda sytuacja z ulicami $i+1,\ldots,n-1,n$, które napotykamy, jadąc od ulicy i-tej do n-tej.

Drugie spostrzeżenie pozwala nam uprościć rozważane trasy. Zauważmy, że jadąc z ulicy i-tej do pierwszej, nie trzeba wykorzystywać łączników wschodnich. Gdyby egzaminowany przejechał takim łącznikiem, na przykład poprowadzonym pomiędzy ulicą j-tą a $(j+1)$-szą na wysokości k, to i tak w pewnym momencie musiałby wrócić do ulicy j-tej. Co więcej, znalazłby się wówczas na wysokości $k' > k$, czyli na północ od punktu, w którym poprzednio opuścił ulicę j-tą (gdyż nie ma ulic prowadzących z północy na południe). To oznacza, że mógł przejechać pomiędzy punktami o wysokości k i k' bezpośrednio ulicą j-tą, skracając całą trasę. Analogicznie rozumując, można pokazać, że w drodze z ulicy i-tej do n-tej egzaminowany nie musi wykorzystywać żadnych łączników zachodnich.



Rys. 1: Ilustracja spostrzeżenia drugiego: na drodze z ulicy j-tej do pierwszej nie opłaca się skręcać na wschód.

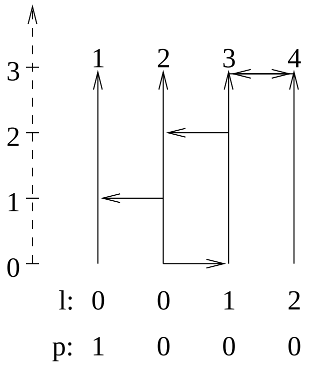
Poczynione spostrzeżenia pozwalają nam podzielić wyjściowy problem na dwa symetryczne podproblemy. Dla ulicy i ($1 \leqslant i \leqslant n$) wyznaczymy dwie wartości: l_i oraz p_i. Pierwsza z nich, to minimalna liczba łączników zachodnich, które trzeba dobudować, żeby połączyć ulicę i-tą z ulicą pierwszą. Analogicznie p_i definiujemy jako minimalną liczbę łączników wschodnich, których dobudowanie pozwala przejechać z ulicy i-tej do ulicy n-tej.

Pozostańmy jeszcze chwilę przy ulicy i-tej. Zauważmy, że dobudowując l_i łączników zachodnich, dzięki którym pojawia się możliwość dojechania z ulicy i-tej do pierwszej, tworzymy także połączenie *każdej* spośród ulic $1,\ldots,i-1,i$ z ulicą pierwszą. Faktycznie, zaczynając egzamin na początku dowolnej z tych ulic, można jechać prosto na północ aż do momentu przecięcia trasy łączącej ulicę i-tą z ulicą pierwszą, i w tym właśnie punkcie włączyć się do niej. Podobne spostrzeżenie można także poczynić dla tras prowadzących na wschód. Dalej zauważmy, że $l_1 \leqslant l_2 \leqslant \cdots \leqslant l_n$ — jeżeli dobudowanie l_i (dla $i \in \{2,\ldots,n\}$) łączników zachodnich wystarcza, aby dojechać z ulicy i-tej do pierwszej, to z pewnością tyle samo łączników wystarczy, by dojechać tam z ulicy $(i-1)$-szej. Podobnie możemy uzasadnić, że $p_1 \geqslant p_2 \geqslant \cdots \geqslant p_n$.

Ograniczenia zadania pozwalają nam dobudować jedynie k łączników. Ustalmy, że najwyżej L z nich to będą łączniki zachodnie. Niech x będzie największym numerem ulicy, dla której $l_x \leqslant L$ — budując co najwyżej $l_x \leqslant L$ łączników zachodnich, można połączyć ulice $1,\ldots,x$ z ulicą pierwszą. Co więcej, nie istnieje sposób, by dobudowując tylko L łączników, stworzyć połączenie ulicy $(x+1)$-szej (i kolejnych) z pierwszą! Po zbudowaniu l_x łączników zachodnich mamy jeszcze do wykorzystania $P = k - l_x$ łączników wschodnich. Zdefiniujmy dla P, analogicznie jak dla L, indeks $y \in \{1,\ldots,n\}$ jako najmniejszy indeks, dla którego $p_y \leqslant P$. Po dobudowaniu wszystkich k łączników łączących ulicę x-tą z pierwszą oraz ulicę y-tą z n-tą, uzyskujemy możliwość dojazdu do ulicy pierwszej z ulic $1,\ldots,x$ oraz do ulicy n-tej z ulic $y,\ldots,n$. To oznacza, że z ulic $y,\ldots,x$ (i tylko tych) da się dojechać zarówno do ulicy pierwszej, jak i n-tej. Początki tych ulic to wszystkie możliwe punkty startowe egzaminu. Ponieważ naszym zadaniem jest wyznaczenie liczby *nowych* punktów startowych — takich, które powstały dopiero po rozbudowie placu — zatem trzeba jeszcze rozpoznać, które spośród znalezionych punktów były dobrymi punktami startowymi także przed rozbudową. Jest to jednak bardzo proste: początek ulicy i-tej był dobrym punktem startowym egzaminu przed rozbudową wtedy i tylko wtedy, gdy $l_i = p_i = 0$.

Znamy już ogólną ideę rozwiązania. Teraz trzeba uzupełnić szczegóły — wyznaczyć wartości l_i, p_i, L, P oraz związane z nimi indeksy x i y. Zajmijmy się najpierw parametrami L i P. W ich przypadku nie będziemy stosować żadnych wyszukanych metod — po prostu przetestujemy wszystkie możliwe wartości parametru L od 1 do k. Zauważmy, że mając wyznaczone wszystkie wartości $l_1,\ldots,l_n$ oraz $p_1,\ldots,p_n$, indeks x możemy odnaleźć w czasie $O(\log n)$, stosując proste wyszukiwanie binarne wartości L w ciągu $l_1,\ldots,l_n$. Wartość l_x pozwala nam określić wartość $P = k - l_x$, a kolejne wyszukiwanie binarne w ciągu uporządkowanym $p_1,\ldots,p_n$ pozwala odnaleźć indeks y — także w czasie $O(\log n)$. Ponieważ $L \in \{0,\ldots,k\}$, to otrzymujemy złożoność czasową tej fazy $O(k \log n)$, czyli akceptowalną z punktu widzenia ograniczeń z zadania.

Zanim przejdziemy do wyznaczania wartości l_i oraz p_i, przedstawimy działanie opisanego algorytmu na przykładzie z treści zadania. Na rysunku 2 podane zostały wartości l_i oraz p_i dla $i = 1,2,3,4$.



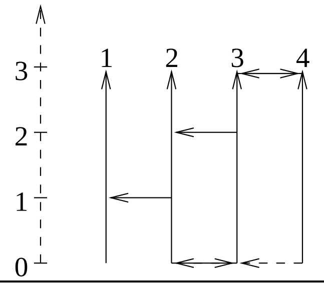
Rys. 2: Wartości l_i oraz p_i dla przykładu z treści zadania.

Przeanalizujmy wszystkie możliwe wartości parametru L i odpowiadające im wartości parametrów P, x oraz y (pamiętając o tym, że w przykładzie $k = 2$):

- $L = 0$, stąd $x = 2$, więc $l_x = 0$ i dalej $P = 2 - 0 = 2$ oraz $y = 1$; to daje łącznie 2 możliwe punkty startowe.
- $L = 1$, stąd $x = 3$, więc $l_x = 1$ i dalej $P = 2 - 1 = 1$ oraz $y = 1$; to daje łącznie 3 możliwe punkty startowe.
- $L = 2$, stąd $x = 4$, więc $l_x = 2$ i dalej $P = 2 - 2 = 0$ oraz $y = 2$; to znów daje łącznie 3 możliwe punkty startowe.

Ponieważ na początku mamy 1 punkt startowy (początek ulicy 2), to w najlepszym przypadku udaje się nam utworzyć 2 nowe punkty startowe egzaminu. Przykład optymalnego sposobu dobudowania łączników dla przypadku $L = P = 1$ jest zilustrowany na rysunku w treści zadania, natomiast dla przypadku $L = 2$, $P = 0$ — na rysunku 3.

Na koniec warto wspomnieć, że rozważany etap rozwiązania można wykonać jeszcze szybciej — w czasie $O(n+k)$. Wystarczy zauważyć, że w miarę wzrostu parametru L, indeks x nie może maleć, gdyż ciąg l_i jest niemalejący. Podobnie,

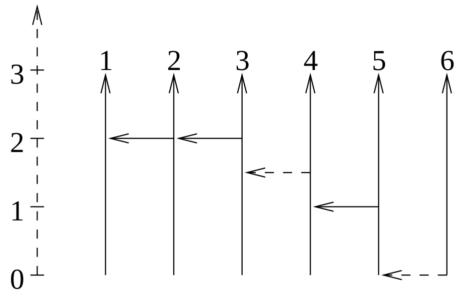


Rys. 3: Przykład optymalnej rozbudowy placu przez dobudowanie dwu łączników zachodnich (jeden z nich pokrywa się z już istniejącym łącznikiem wschodnim).

w miarę zmniejszania się wartości parametru P, indeks y nie może maleć, gdyż ciąg p_i jest nierosnący. Rozpoczynając obliczenia od $L = 0$, można poszukiwanie x rozpocząć od $x = 1$ i sprawdzać kolejne wartości, dopóki nie otrzymamy $l_x \leqslant L < l_{x+1}$. Znaleziona wartość l_x pozwala nam wyznaczyć $P = k - l_x$. Przystępujemy teraz do poszukiwania y — zaczynamy od $y = 1$ i zwiększamy tę wartość, dopóki nie otrzymamy $p_{y-1} > P \geqslant p_y$. Dostajemy czwórkę parametrów: $L = 0$, P, x i y, dla której wyznaczamy rozwiązanie. Przechodząc do $L = 1$, poszukiwanie x i y kontynuujemy od poprzednich wartości tych parametrów. Tak samo postępujemy po kolejnych zwiększeniach L. W ten sposób w trakcie działania algorytmu każdy z parametrów x oraz y zostanie zwiększony o jeden łącznie $O(n)$ razy, co oznacza, że omawiana faza rozwiązania działa w zapowiadanej złożoności czasowej $O(k+n)$. Opisane usprawnienie, ze względu na czas konieczny do wykonania pozostałych faz, nie powoduje niestety istotnego wzrostu efektywności rozwiązania.

Wyznaczanie l_i oraz p_i

Skupimy się na ulicy i-tej i wyznaczeniu wartości l_i, dla pewnego $1 \leqslant i \leqslant n$ — obliczenie p_i wykonamy analogicznie. Zauważmy, że optymalna trasa łącząca tę ulicę z ulicą pierwszą zawiera dokładnie $i-1$ łączników zachodnich. Jeżeli l_i spośród tych łączników to nowo dobudowane, to $i-1-l_i$ łączników musiało być na placu przed rozbudową. „Stare” łączniki tworzą swoisty „ciąg niemalejący”, który formalnie nazwiemy *ciągiem zachodnio-północnym* — jest to ciąg łączników zachodnich uporządkowanych tak, że każdy kolejny jest położony na zachód od poprzedniego i na nie mniejszej wysokości niż poprzedni. Łatwo spostrzec, że dowolny ciąg zachodnio-północny złożony z c łączników można rozbudować, dodając $i-c-1$ łączników zachodnich, tworząc trasę z ulicy i-tej do pierwszej. Trasę taką stworzymy najmniejszym kosztem, jeśli rozbudujemy najdłuższy ciąg zachodnio-północny, rozpoczynający się od tej ulicy — l_i wynosi wówczas $i-c-1$, gdzie c jest długością tego ciągu.

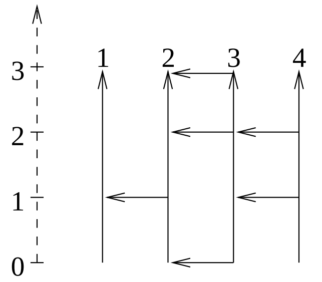


Rys. 4: Ciąg zachodnio-północny. Na rysunku $i = 6$, liczba dobudowanych łączników to $l_i = 2$, a najdłuższy ciąg zachodnio-północny ma długość $i - l_i + 1 = 3$.

Pozostaje już tylko wyznaczyć długość najdłuższego ciągu zachodnio-północnego dla rozważanej ulicy. Spróbujmy przekształcić ten problem do poszukiwania najdłuższego podciągu niemalejącego w ciągu liczbowym. Rozważmy ciąg liczb, jaki otrzymamy, wypisując wysokości wszystkich łączników zachodnich, poczynając od łączników wychodzących z ulicy i-tej i poruszając się w kierunku zachodnim. Łączniki wychodzące z tej samej ulicy wypisujemy w kolejności malejących wysokości, czyli w kierunku z północy na południe. Rysunek 5 obrazuje ten sposób konstrukcji ciągu dla przykładowego placu egzaminacyjnego.

Podciągi niemalejące skonstruowanego ciągu liczb odpowiadają wzajemnie jednoznacznie ciągom zachodnio-północnym — kolejnym liczbom w podciągu odpowiadają bowiem łączniki położone na północ (gdyż podciąg jest niemalejący) i *ściśle* na zachód od poprzednich (ostrą nierówność uzyskujemy dzięki uporządkowaniu łączników wychodzących z tej samej ulicy w kolejności od północy na południe).

Istnieje wiele algorytmów wyznaczania długości najdłuższego podciągu niemalejącego zadanego ciągu — działają one w ogólnym przypadku w czasie $O(n \log n)$ dla ciągu długości n. Stąd wyznaczanie tych wartości dla każdego indeksu $i \in \{1, \dots, n\}$ *osobno*, skończyłoby się przekroczeniem dopuszczalnych limitów czasu. Zauważmy jednak, że dowolny



Rys. 5: Ciągiem liczbowym, skonstruowanym dla powyższego przykładu placu egzaminacyjnego ($i = 4$) jest: $2,1,3,2,0,1$.

podciąg niemalejący danego ciągu jest podciągiem nierosnącym ciągu odwróconego. To z pozoru banalne spostrzeżenie pozwala efektywnie rozwiązać problem:

- dla kolejnych wartości indeksu $i = 1,2,\ldots,n$ możemy stopniowo przedłużać ciąg, dołączając do niego łączniki zachodnie wychodzące z ulicy i-tej oraz
- możemy dynamicznie aktualizować długość najdłuższego podciągu nierosnącego wydłużonego prefiksu, otrzymując wartość l_i.

Najdłuższy podciąg nierosnący

W niniejszym rozdziale skupimy się na ostatnio postawionym problemie i nie będziemy się już odwoływać do pojęć z treści zadania: zakładamy, że mamy ciąg liczbowy $a_1,\ldots,a_n$ i poszukujemy długości najdłuższych podciągów nierosnących każdego prefiksu tego ciągu — *prefiksem* długości k ciągu $a_1,\ldots,a_n$ nazywamy ciąg $a_1,\ldots,a_k$, który będziemy oznaczać $a[1..k]$.

Kluczowe informacje będziemy zapisywać w tablicy $t[1..n]$, aktualizując jej zawartość w miarę analizowania kolejnych elementów ciągu a, czyli rozważania coraz dłuższych prefiksów $a[1..k]$. Element $t[i]$ zdefiniujemy jako wartość *największego* elementu w ciągu $a[1..k]$, który jest ostatnim elementem pewnego podciągu nierosnącego długości i w prefiksie $a[1..k]$ (czyli spośród wszystkich podciągów nierosnących długości i ciągu $a[1..k]$ wybieramy ten, który kończy się największym elementem i element ten ustalamy jako wartość $t[i]$). Jeżeli taki element nie istnieje (gdyż nie ma żadnego podciągu o długości i w prefiksie $a[1..k]$), to przyjmujemy, że $t[i] = -\infty$. Przy takiej definicji tablicy t widzimy, że po każdym kroku jej konstrukcji (rozważeniu $a[1..k]$) długość najdłuższego podciągu nierosnącego $a[1..k]$ będzie po prostu równa największemu indeksowi i, dla którego $t[i] > -\infty$.

Zanim opiszemy algorytm obliczania tablicy t, przeanalizujmy przykład, który pozwoli nam zaobserwować jej kilka interesujących własności. Rozważmy mianowicie ciąg $a = 7,4,3,5,5,1,6,1$ długości 8 i zastanówmy się, jak powinna wyglądać tablica t, zbudowana dla kolejnych jego prefiksów:

- Wyjściowa tablica (dla pustego prefiksu ciągu a) wygląda następująco:

-∞	-∞	-∞	-∞	-∞	-∞	-∞	-∞

- W prefiksie $a[1..1]$ mamy już jeden podciągu rosnący (7) o długości jeden, którego ostatni element jest równy 7. Tablica t po pierwszym kroku powinna więc mieć postać:

7	-∞	-∞	-∞	-∞	-∞	-∞	-∞

- Wydłużamy prefiks do $a[1..2]$. Dołożony element $a_2 = 4$ nie jest lepszym niż poprzedni ciągiem jednoelementowym, ale pozwala przedłużyć ciąg (7) do $(7,4)$ i ustalić $t[2] = 4$:

7	4	-∞	-∞	-∞	-∞	-∞	-∞

- Wydłużamy prefiks o kolejny element $a_3 = 3$. Jest on, podobnie jak poprzedni, mniejszy od wszystkich dotychczas rozważonych:

7	4	3	-∞	-∞	-∞	-∞	-∞

- Kolejny element to $a_4 = 5$. Zauważmy, że za jego pomocą nie można poprawić ani $t[1]$ (gdyż $a_4 < t[1]$), ani $t[4]$ (największym elementem, jakim może się kończyć 3-elementowy ciąg nierosnący, jest $t[3] = 3$, czyli nie można tego ciągu wydłużyć za pomocą 5), ani $t[3]$ (z tego samego powodu, co poprzednio). Ponieważ jednak $t[1] \geqslant a_4$ oraz $a_4 > t[2]$, to widzimy, że w prefiksie istnieje podciąg $(7,5)$ i jest on lepszy niż poprzedni ciąg dwuelementowy $(7,4)$, gdyż kończy się większą wartością — możemy poprawić wartość $t[2]$:

7	5	3	-∞	-∞	-∞	-∞	-∞

- Wydłużamy prefiks do $a[1..5]$ i ponownie natrafiamy na element równy 5. Wykorzystując go, możemy tym razem poprawić $t[3]$, jako że w poprzednim kroku udało nam się zwiększyć $t[2]$ — mamy w ten sposób podciąg $(7,5,5)$; zauważmy, że żadnych innych pozycji w tablicy t nie możemy w tym kroku poprawić:

7	5	5	-∞	-∞	-∞	-∞	-∞

- Następny element $a_6 = 1$, co pozwala wreszcie zmodyfikować wartość $t[4]$, gdyż pojawił się podciąg $(7,5,5,1)$. Pozostałe elementy t z oczywistych względów pozostają bez zmian:

7	5	5	1	-∞	-∞	-∞	-∞

- Dochodzi kolejny element $a_7 = 6$. Tak dużego elementu nie można dołączyć do znalezionych dotychczas ciągów o niemalejących o 2, 3 oraz 4 elementach — możliwe natomiast jest dołączenie go do ciągu (7) i taki ciąg jest lepszy niż najlepszy dotychczas ciąg dwuelementowy $(7,5)$, gdyż kończy się większym elementem; to pozwala (już po raz trzeci) zmodyfikować $t[2]$:

7	6	5	1	-∞	-∞	-∞	-∞

- Został nam jeszcze ostatni element ciągu $a_8 = 1$. Jest on nie większy od elementów $t[1], t[2], t[3], t[4]$, więc nie nadaje się do poprawienia tych wartości; można jednak dołączyć go do znalezionego wcześniej ciągu czteroelementowego kończącego się elementem 1. Stąd ostateczna postać tablicy t to:

7	6	5	1	1	-∞	-∞	-∞

Zawartość tablicy t wyznaczona dla kolejnych prefiksów ciągu a pozwala stwierdzić, że długości najdłuższych podciągów nierosnących w tych prefiksach, to: 1, 2, 3, 3, 3, 4, 4, 5. W pierwszej chwili można by przypuszczać, że znaleźliśmy nie tylko długość, ale także sam ciąg — najdłuższy podciąg nierosnący całego ciągu. Nie jest to prawdą! Zapisany w tablicy t ciąg $7,6,5,1,1$ *nie jest podciągiem* ciągu a! Jako proste, ale wartościowe, ćwiczenie polecamy Czytelnikowi zastanowienie się nad tym zjawiskiem.

Jesteśmy już gotowi, by sformułować i udowodnić kilka własności tablicy t, które prowadzą do prostego i efektywnego algorytmu jej konstrukcji:

1. Elementy zapisane w tablicy t po każdym kroku algorytmu są ustawione w kolejności nierosnącej. Aby się o tym przekonać, wystarczy zauważyć, że jeśli w rozważanym prefiksie $a[1..k]$ mamy podciąg długości i zakończony elementem $t[i]$, to w $a[1..k]$ mamy także podciąg niemalejący długości $i-1$ zakończony $t[i]$ — wystarczy z poprzedniego ciągu usunąć pierwszy element. Stąd $t[i-1] \geqslant t[i]$, o ile $t[i] > -\infty$. Jeżeli zaś $t[i] = -\infty$, to nierówność $t[i-1] \geqslant t[i]$ oczywiście zachodzi.

2. W każdym kroku algorytmu zmianie ulega *dokładnie jeden* element tablicy t. Aby to pokazać, rozważmy krok algorytmu, w którym rozszerzamy prefiks o a_k. Z poprzedniego spostrzeżenia wiemy, że istnieje taki indeks w tablicy t, powiedzmy j, że elementy $t[1], \ldots, t[j]$ są nie mniejsze od a_k, a elementy $t[j+1], \ldots, t[n]$ są mniejsze od a_k:
   - wiadomo, że żadnego spośród elementów $t[1], \ldots, t[j]$ nie można poprawić za pomocą a_k, ponieważ mamy już podciągi odpowiedniej długości, kończące się elementami nie mniejszymi (a więc lepszymi) niż a_k;
   - wartość $t[j]$ mówi nam, że w prefiksie $a[1..k-1]$ jest podciąg nierosnący długości j zakończony elementem $t[j]$; ponieważ $t[j] \geqslant a_i$, więc na końcu tego ciągu możemy dołączyć element a_k, otrzymując podciąg długości $j+1$ zakończony elementem $a_k > t[j+1]$; to oznacza, że możemy powiększyć wartość $t[j+1]$;
   - pozostałe elementy $t[j+1], \ldots, t[n]$ są mniejsze od a_k — to oznacza, że elementu a_k nie da się dołączyć na koniec żadnego podciągu długości większej niż j, czyli elementów $t[j+2], t[j+3], \ldots, t[n]$ nie można poprawić za pomocą a_k.

3. Ostatnie spostrzeżenie będzie prostym wnioskiem z dwóch poprzednich: skoro dodanie elementu a_k powoduje zmianę w dokładnie jednym miejscu tablicy t i miejsce to potrafimy dokładnie określić — jest to pierwszy element tablicy t mniejszy od a_k — to do wyznaczenia jego pozycji można użyć wyszukiwania binarnego. To pozwala wykonać jeden krok algorytmu w złożoności czasowej $O(\log n)$, a cały algorytm aktualizacji t — w złożoności czasowej $O(n \log n)$.

Przedstawmy teraz pseudokod opisanego algorytmu, w którym w każdym z n kroków (dla $i = 1, \ldots, n$) wypisujemy długość najdłuższego podciągu nierosnącego prefiksu $a[1..k]$ ciągu a:

```
for $j := 1$ to $n$ do
  $t[j] := -\infty$;
$len := 0$; { Długość najdłuższego podciągu nierosnącego. }
for $k := 1$ to $n$ do
begin
  { Wyszukiwanie binarne elementu tablicy $t$, który należy zmienić. }
  { Złożoność czasowa $O(\log n)$. }
  $d := 1$; $g := len + 1$; { dolna i górna granica wyszukiwania }
  while $d < g$ do
  begin
    $s := \lfloor \frac{d+g}{2} \rfloor$;
    if $t[s] \geqslant a_k$ then
```

```
                $d := s+1$;
            else
                $g := s$;
        end
        { $d$ $(= g)$ jest szukaną pozycją w tablicy $t$. }
        $t[d] := a_k$;
        if $d > len$ then
            { Musi zachodzić $d = len+1$, czyli wynik się powiększył. }
            $len := d$;
        Wypisz_Długość_Podciagu($len$);
    end
```

Najbardziej skomplikowaną częścią powyższego algorytmu jest wyszukiwanie binarne pierwszego elementu tablicy t, który jest mniejszy od a_k (wiersze 8–16). Taki element musi zawsze istnieć, gdyż przed każdym krokiem algorytmu zachodzi $a_k > t[len+1] = -\infty$.

Podsumowanie

Uporządkujmy dotychczasowe rozważania i przypomnijmy w skrócie kolejne kroki rozwiązania wzorcowego:

- Dzielimy wszystkie łączniki na zachodnie (Z) i wschodnie (W). Dla pierwszej z tych grup wykonujemy wszystkie opisane niżej kroki. Natomiast łączniki wschodnie odbijamy symetrycznie względem dowolnej ulicy (czyli prostej pionowej), co odpowiada odwróceniu numeracji ulic, od których rozpoczynają się łączniki. Złożoność czasowa tego kroku to $O(p)$.
- Sortujemy łączniki ze zbioru Z w porządku malejącym numerów ulic, od których się rozpoczynają. Łączniki wychodzące z tej samej ulicy sortujemy w kolejności z północy na południe. Złożoność czasowa tego kroku to $O(p\log p)$, jeżeli zastosujemy jeden z efektywnych algorytmów sortowania, np. sortowanie przez scalanie.
- Tworzymy ciąg liczbowy złożony z wysokości wszystkich łączników ze zbioru Z we wspomnianej kolejności. Następnie odwracamy otrzymany ciąg i, za pomocą opisanego algorytmu o złożoności czasowej $O(p\log p)$, dla każdego prefiksu ciągu odwróconego znajdujemy długość najdłuższego podciągu nierosnącego. Na tej podstawie wyznaczamy wartości l_i. Całkowita złożoność czasowa tego kroku to $O(p\log p+n)$.
- Wykonując analogiczne do powyższych obliczenia dla zbioru W, otrzymujemy, w takim samym czasie, wartości p_i.
- Obliczywszy wartości l_i oraz p_i, przeglądamy w $k+1$ fazach wszystkie możliwe wartości parametru L od 0 do k, obliczając dla każdej z nich pozostałe parametry: x, P oraz y.
- Końcowy wynik wyznaczamy jako maksimum po wszystkich fazach z wartości $x-y+1-q_{x,y}$, gdzie $x-y+1$ odpowiada liczbie punktów startowych, które możemy otrzymać po rozbudowie placu, a $q_{x,y}$ jest liczbą starych punktów startowych wśród nich (czyli liczbą indeksów z przedziału $[y,x]$, dla których $l_i = p_i = 0$). Ze względu na wykorzystanie wyszukiwania binarnego, złożoność czasowa tego podpunktu to $O(k\log n)$. Wspomnieliśmy także o szybszym sposobie realizacji tej fazy, w którym przy wyznaczaniu kolejnych wartości parametrów x oraz y korzystamy z wiedzy o ich poprzednich wartościach i nigdy ich nie zmniejszamy. W ten sposób można zredukować złożoność czasową tej fazy do $O(n+k)$.

Całkowita złożoność czasowa powyższego rozwiązania to $O(p\log p+n+k)$. Zostało ono zaimplementowane w plikach `egz.cpp`, `egz2.pas` oraz `egz3.c`.

Inne rozwiązania

Wszystkie przewidziane przez Jury Olimpiady poprawne rozwiązania alternatywne, podobnie zresztą jak wszystkie rozwiązania zawodników, które uzyskały dodatnią punktację, opierają się w mniejszym lub większym stopniu na tych samych pomysłach, co rozwiązanie wzorcowe. Występuje w nich faza wyznaczania wartości analogicznych do l_i i p_i oraz faza, w której na ich podstawie jest obliczany wynik. Każdą z opisanych faz można zaimplementować mniej efektywnie niż zostało to uczynione w rozwiązaniu wzorcowym. Jeżeli chodzi o pierwszą fazę, to istnieje cała gama algorytmów wyznaczania długości najdłuższego podciągu nierosnącego w kwadratowej złożoności czasowej (w naszym przypadku jest to $O(p^2)$), jak choćby naiwna implementacja opisanego algorytmu wyznaczania tablicy t. Również drugą fazę można zrealizować nieoptymalnie, przeglądając wszystkie możliwe początki i końce przedziałów potencjalnych punktów startowych i dla każdej takiej pary (x,y) sprawdzając, czy $p_x + l_y \leqslant k$. Taki sposób implementacji drugiej fazy ma koszt czasowy $O(n^2)$.

Testy

Zadanie było sprawdzane na 10 zestawach danych wejściowych. W poniższej tabelce *n* oznacza liczbę ulic w teście, *m* — długości ulic, *p* — liczbę już wybudowanych łączników zachodnich i wschodnich, a *k* — maksymalną liczbę łączników, które można dobudować. Wreszcie przez *w* oznaczono wynik dla testu, to znaczy maksymalną liczbę nowych punktów startowych, jakie można uzyskać, dobudowując co najwyżej *k* łączników. We wszystkich testach, w których $p > 0$, istniejące łączniki zachodnie i wschodnie zostały rozmieszczone na placu w sposób losowy.

Nazwa	**n**	**m**	**p**	**k**	**w**	**Opis**
egz1a.in	4	3	5	2	2	prosty test poprawnościowy
egz1b.in	100	100	200	100	30	większy test poprawnościowy
egz2a.in	10	1	0	8	0	prosty test bez łączników
egz2b.in	10	1	0	9	1	prosty test bez łączników
egz2c.in	10	10	10	10	9	prosty test, w którym wszystkie parametry wejściowe są równe 10
egz2d.in	100 000	1	0	99 999	1	duży test bez łączników
egz2e.in	100	50	150	100	27	większy test losowy
egz3a.in	100	100	300	100	42	prosty test poprawnościowy
egz3b.in	100	100	400	100	46	prosty test poprawnościowy
egz3c.in	100	100	500	100	50	prosty test poprawnościowy
egz4a.in	500	1 000	500	300	0	średniej wielkości test z wynikiem zerowym
egz4b.in	500	1 000	100	800	323	średniej wielkości test z wynikiem niezerowym
egz5.in	1 000	50 000	100 000	1 000	740	średniej wielkości test
egz6.in	10 000	50 000	50 000	15 000	5 536	duży test
egz7.in	20 000	50 000	20 000	30 000	10 353	duży test
egz8a.in	50 000	100 000	100 000	18 000	0	duży test, wynik zerowy
egz8b.in	50 000	100 000	100 000	50 000	637	duży test, wynik niezerowy
egz9.in	100 000	10 000	100 000	99 999	622	duży test
egz10.in	100 000	80 000	100 000	100 000	618	test (prawie) maksymalnej wielkości

Rozwiązania wolniejsze przechodziły na zawodach pewne spośród testów 1-5, w zależności od tego, które fazy rozwiązania wzorcowego były zaimplementowane nieoptymalnie.

Jakub Radoszewski
Treść zadania, Opracowanie

Paweł Parys
Program

OI, Etap III, dzień drugi, 29.03.2007

Klocki

Agatka dostała na urodziny komplet klocków. Klocki mają kształt sześcianów i są wszystkie tej samej wielkości. Na każdym z klocków jest napisana jedna dodatnia liczba całkowita. Agatce klocki bardzo się spodobały i natychmiast ustawiła z nich wszystkich jedną wysoką wieżę.

Mama powiedziała Agatce, że celem zabawy klockami jest ustawienie wieży, w której jak najwięcej klocków znajdzie się na swoich miejscach. Klocek, na którym jest napisana liczba i, jest na swoim miejscu, jeżeli znajduje się w wieży na wysokości i (klocek na samym dole wieży jest na wysokości 1, klocek stojący na nim jest na wysokości 2 itd.). Agatka postanowiła ostrożnie pousuwać z wieży pewne klocki (starając się, żeby wieża się nie przewróciła), tak aby jak najwięcej klocków znalazło się w rezultacie na swoich miejscach. Doradź Agatce, które klocki najlepiej usunąć.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis wieży, jaką na początku ustawiła Agatka,*
- *wyznaczy, które klocki Agatka ma usunąć,*
- *wypisze wynik na standardowe wyjście.*

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę całkowitą n ($1 \leqslant n \leqslant 100\,000$), oznaczającą początkową wysokość wieży klocków. Drugi wiersz wejścia zawiera n dodatnich liczb całkowitych: $a_1, a_2, \ldots, a_n$ ($1 \leqslant a_i \leqslant 1\,000\,000$), pooddzielanych pojedynczymi odstępami i oznaczających liczby napisane na klockach. Liczby te są podane w kolejności od klocka położonego najniżej do klocka położonego najwyżej.

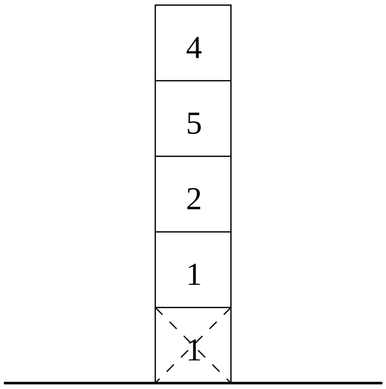
Wyjście

Twój program powinien wypisać w pierwszym wierszu liczbę klocków, które należy usunąć z wieży, aby zmaksymalizować liczbę klocków, które znajdą się na swoich miejscach. Drugi wiersz powinien zawierać numery usuwanych klocków (pooddzielane pojedynczymi odstępami). Klocki są ponumerowane kolejnymi liczbami od 1 do n w porządku od najniżej do najwyżej położonego klocka w początkowej wieży. Jeżeli istnieje więcej niż jedno rozwiązanie, Twój program powinien wypisać dowolne z nich.

Przykład

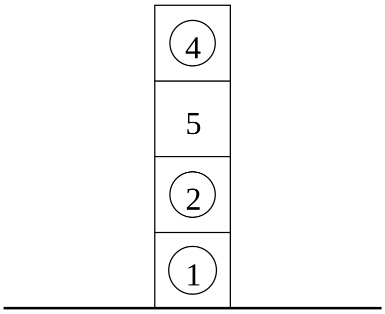
Dla danych wejściowych:

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



poprawnym wynikiem jest:

```
1
1
```



Rozwiązanie

Programowanie dynamiczne

Zadanie można rozwiązać na wiele sposobów z wykorzystaniem techniki programowania dynamicznego. Otrzymane rozwiązania mają w większości złożoność czasową $O(n^3)$ lub, te lepsze, $O(n^2)$. My także rozpoczniemy od rozwiązania o złożoności $O(n^2)$, a następnie je przyspieszymy.

W dalszej części *rozwiązaniem* będziemy nazywać początkową wieżę z usuniętymi dowolnymi klockami. *Rozwiązaniem optymalnym* nazwiemy rozwiązanie, w którym najwięcej klocków jest na swoich miejscach. Okazuje się, że poszukując rozwiązania optymalnego, możemy brać pod uwagę tylko takie rozwiązania, w których ostatni klocek (na szczycie wieży) jest na swoim miejscu. Każde inne rozwiązanie (także optymalne) można bowiem „poprawić”, usuwając z niego wszystkie zbędne klocki z góry wieży, które nie są na swoim miejscu i nie mają wpływu na jakość rozwiązania. *Rozwiązaniem i-optymalnym* nazwiemy rozwiązanie optymalne wśród rozwiązań, w których na szczycie wieży jest klocek i-ty i jest on na swoim miejscu. Przez t_i (dla $1 \leqslant i \leqslant n$) oznaczymy liczbę klocków znajdujących się na swoich miejscach w rozwiązaniu i-optymalnym. Oczywiście wartością poszukiwaną w zadaniu jest maksimum z wartości t_i.

Pozostaje zatem problem, jak wyliczyć wartości t_i. Rozważmy klocek o numerze i dla pewnego $i \in \{1,\ldots,n\}$. Jeśli $a_i > i$, to nie istnieje rozwiązanie i-optymalne, gdyż a_i już na początku jest poniżej „swojej” docelowej pozycji, a usuwanie klocków tylko tę sytuację pogarsza. Przyjmujemy wówczas, że $t_i = 0$. Jeśli $a_i \leqslant i$, to rozwiązanie i-optymalne istnieje. Poszukując dokładnej wartości t_i zauważmy, że możliwe są sytuacje:

- $t_i = 1$, gdy w rozwiązaniu i-optymalnym tylko klocek i-ty jest na swoim miejscu;
- $t_i = 1 + t_j$, gdy w rozwiązaniu i-optymalnym klocek j-ty jest najwyższym spośród klocków znajdujących się pod klockiem i-tym, który znalazł się na swoim miejscu.

Wartość t_i obliczamy zatem ze wzoru:

$$t_i = \max(1, \max\{t_j + 1 : \text{dla niektórych } 1 \leqslant j < i\}),$$

gdzie wewnętrzne maksimum jest wybierane po tych wartościach $j < i$, dla których klocek j-ty może być na swoim miejscu w rozwiązaniu, w którym na szczycie pozostaje klocek i-ty, także zajmujący swoje miejsce. Spróbujmy dokładniej określić warunki, jakie musi spełniać j:

$$\begin{aligned} j &< i, & (1)\\ a_j &< a_i, & (2)\\ j - a_j &\leqslant i - a_i. & (3) \end{aligned}$$

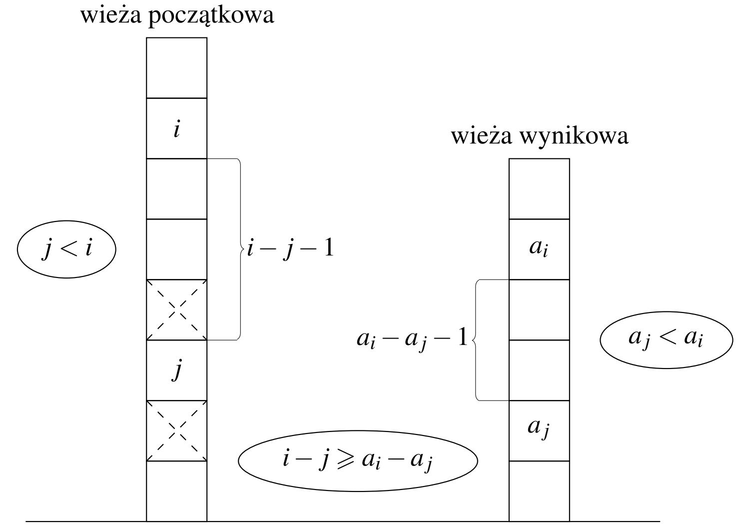
Pierwszy warunek jest oczywisty — klocek j-ty musi znajdować się poniżej klocka i-tego w wieży początkowej. Drugi warunek to analogiczny fakt dla wieży wynikowej — klocek j-ty, który trafi na pozycję a_j, musi znajdować się poniżej klocka i-tego, który trafi na pozycję a_i. Skąd jednak bierze się warunek 3? Zapisawszy go inaczej: $i - j + 1 \geqslant a_i - a_j + 1$, widzimy, iż oznacza on, że liczba klocków pomiędzy klockiem j-tym a i-tym w wieży wynikowej nie może być większa niż w początkowej (Agatka może wyłącznie usuwać klocki).

Na rys. 1 jest przedstawiona ilustracja warunków (1)–(3). Widać z niego także, że warunki te są wystarczające, by przy wyznaczaniu t_i skorzystać z wartości $t_j + 1$.

Możemy teraz zapisać procedurę wyznaczania t_i dla $1 \leqslant i \leqslant n$:

```
for $i := 1$ to $n$ do
begin
  if $a_i > i$ then
    $t_i := 0$
  else
```

```
 6: begin
 7:   $t_i := 1$;
 8:   for $j := 1$ to $i - 1$ do
 9:     if $(a_j < a_i)$ and $(j - a_j \leqslant i - a_i)$ then
10:       { Oczywiście zachodzi warunek 1: $j < i$. }
11:       $t_i = \max(t_i, t_j + 1)$;
12: end
13: end
```



Rys. 1: Ilustracja nierówności (1)–(3).

Liczbę klocków znajdujących się na swoim miejscu w rozwiązaniu optymalnym otrzymujemy, wyliczając t_k, dla którego $t_k = \max(t_1, \ldots, t_n)$. Zadanie *Klocki* polega jednak na skonstruowaniu listy klocków, które należy usunąć, aby to rozwiązanie uzyskać. W tym celu uzupełnimy powyższy algorytm. Dla każdego i $(1 \leqslant i \leqslant n)$ zapamiętamy wartość p_i, oznaczającą indeks j, dla którego ostatecznie $t_i = t_j + 1$ (czyli indeks znaleziony w pętli 8–11). Znając k oraz p_i dla wszystkich $i \in \{1, \ldots, n\}$, będziemy mogli odtworzyć sposób usuwania klocków, zgodnie z poniższym pseudokodem:

```
$i := k$;
$a_0 := 0$; { żeby dać poprawną odpowiedź w przypadku, gdy $t_i = 1$ }
while $t_i > 0$ do
begin
  $j := p_i$;
  Wybierz dowolnie $(i - j) - (a_i - a_j)$ klocków spośród $j + 1, \ldots, i - 1$;
  Usuń wybrane klocki z wieży;
  $i := j$;
end
```

O poprawności zaproponowanej powyżej konstrukcji wieży można się przekonać, powracając do rys. 1 oraz uważnie analizując postępowanie w przypadku, gdy $t_i = 1$ (usuniętych zostaje $i - a_i$ klocków).

Złożoność całego rozwiązania to czas $O(n^2)$, potrzebny na wyznaczenie wartości t_i, oraz czas $O(n)$, konieczny do odtworzenia wyniku, czyli razem — zgodnie z zapowiedzią — $O(n^2)$. Rozwiązanie to zostało zaimplementowane w pliku `klos1.cpp`. Na zawodach można było za nie zdobyć (podobnie jak za inne rozwiązania o złożoności czasowej kwadratowej względem n) niecałe 50% punktów.

Próba usprawnienia poprzedniego rozwiązania

Najwięcej czasu w przedstawionym rozwiązaniu zajmuje wyznaczanie wartości t_i. Gdybyśmy efektywniej wyliczali $\max(t_j + 1 : j < i \text{ oraz } a_j < a_i \text{ oraz } j - a_j \leqslant i - a_i)$ dla zadanego i, to moglibyśmy istotnie przyspieszyć nasze rozwiązanie. Spróbujmy w tym celu zastosować odpowiednią strukturę danych — taką, w której będziemy mogli:

- przechowywać pary postaci (*element*, *wartość*) oraz
- efektywnie odpowiadać na pytania, jaka jest maksymalna *wartość* dla *elementów*, zawartych w wybranym podzbiorze zbioru wszystkich elementów.

Efektywność poszukiwanej struktury danych zależy znacząco od tego, jakiego typu elementy zamierzamy przechowywać i o jakiego typu zbiory elementów planujemy pytać. W interesującym nas przypadku elementy to *krotki liczb* (inaczej *wektory*), a zbiory specyfikujemy, podając dla każdej współrzędnej krotki zakres wartości (przedział). Dokładniej:

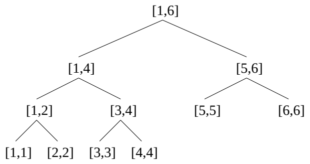
- j-ty klocek możemy reprezentować jako wektor $(j, a_j, j-a_j)$ o wartości t_j+1,
- wartość t_i otrzymujemy jako odpowiedź na pytanie o maksimum z wartości elementów, których:
  - pierwsza współrzędna należy do przedziału $[1,i)$,
  - druga — do przedziału $[1,a_i)$,
  - trzecia — do przedziału $[0,i-a_i]$ (jeżeli $j-a_j<0$, to i tak $t_j=0$).

W jaki sposób można zrealizować opisaną wyżej strukturę danych? Rozważmy bardziej ogólny przypadek — załóżmy, że elementy to k-elementowe krotki (wektory k-wymiarowe). Pokażemy, że wówczas wstawienie elementu do struktury lub znalezienie odpowiedzi na pytanie o maksimum można wykonać w czasie $O(\log^k n)$.

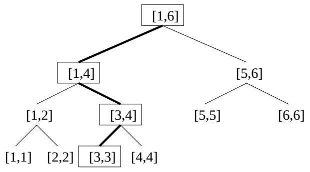
Jeżeli $k=1$, to mamy do czynienia z elementami-liczbami z przypisanymi wartościami liczbowymi, a zapytania dotyczą maksimum wartości przypisanych elementom z pewnych przedziałów. Dobrą strukturą dla tego problemu jest odpowiednio wzbogacone zrównoważone drzewo poszukiwań binarnych (na przykład AVL). Dla każdego elementu mamy w drzewie jeden liść, w którym jest zapisany element i jego wartość. Natomiast w każdym z węzłów wewnętrznych znajduje się zbiorcza informacja o poddrzewie zakorzenionym w tym węźle:

- maksimum z wartości przypisanych liściom z jego poddrzewa oraz
- granice przedziału, w którym zawierają się wszystkie elementy zapisane w jego liściach.

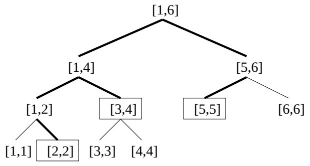
Omawiane drzewo najłatwiej zaimplementować, jeżeli z góry znamy wszystkie możliwe elementy (na przykład są to liczby naturalne z niedużego przedziału). W takim przypadku możemy zarezerwować po jednym liściu na każdy potencjalny element i na samym początku zbudować dobrze wyważone drzewo (tzw. drzewo statyczne). W trakcie działania algorytmu pozostanie nam jedynie „uaktywnianie" elementów, zamiast ich fizycznego wstawiania do drzewa, oraz aktualizacja informacji w wierzchołkach wewnętrznych. Strukturę taką nazywamy *drzewem licznikowym* lub *drzewem przedziałowym*. Przykład dla elementów będących liczbami od 1 do 6 jest przedstawiony na rys. 2. Wstawienie nowego elementu odpowiada przejściu w drzewie od odpowiadającego mu liścia do korzenia i aktualizacji informacji we wszystkich węzłach na tej ścieżce (patrz rys. 3). Natomiast odpowiedź na pytanie o maksimum wartości elementów z danego przedziału znajdujemy, rozkładając ten przedział na *przedziały bazowe* (czyli przedziały związane z węzłami drzewa) i wyznaczając maksimum z wartości w węzłach im odpowiadających (patrz rys. 4). Dokładniejsze omówienie implementacji podobnej struktury można znaleźć na przykład w opisie rozwiązania zadania *Tetris 3D* w książeczce XIII Olimpiady Informatycznej.



Rys. 2: Przykładowe drzewo statyczne dla elementów $1,2,3,4,5,6$.



Rys. 3: „Wstawienie" elementu 3 do drzewa z poprzedniego rysunku.

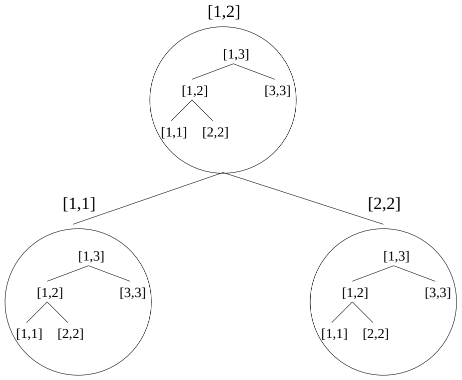


Rys. 4: Rozkład przedziału $[2,5]$ na przedziały bazowe.

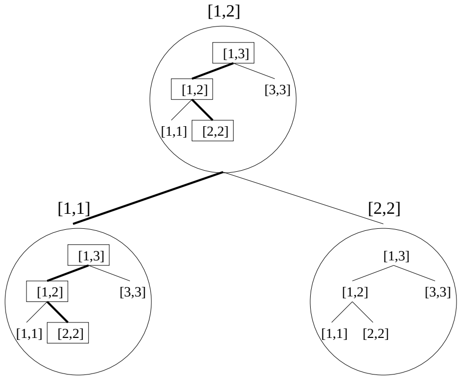
Dla bardziej złożonych elementów, gdy $k > 1$, możemy zbudować strukturę rekurencyjną — drzewo zrównoważone, w którego wierzchołkach znajdują się struktury wymiaru $k-1$. Drzewo to jest skonstruowane według pierwszych współrzędnych elementów i w każdym wierzchołku zawiera:

- granice przedziału dla pierwszej współrzędnej elementów wyznaczone analogicznie, jak w przypadku $k = 1$;
- strukturę wymiaru $k-1$, zawierającą wszystkie elementy, których pierwsza współrzędna należy do przedziału związanego z danym wierzchołkiem.

Podobnie, jak w przypadku jednowymiarowym, i tym razem całą strukturę można zbudować na początku. Przykład struktury dla $k = 2$ i dla elementów, których pierwsza współrzędna należy do przedziału $[1,2]$, a druga do przedziału $[1,3]$, obrazuje rys. 5. Wstawienie elementu do struktury k-wymiarowej polega na wstawieniu go (rekurencyjnie) do wszystkich struktur $(k-1)$-wymiarowych, znajdujących się na ścieżce od odpowiedniego liścia drzewa do jego korzenia — patrz rys. 6. Proste rozumowanie indukcyjne pozwala wykazać, że złożoność czasowa operacji wstawienia elementu do struktury wynosi $O(\log^k n)$. W tym samym czasie możemy znaleźć maksimum wartości dla zbioru elementów, określonego poprzez zadanie przedziału dla każdej współrzędnej. Pytanie to sprowadzamy bowiem do $O(\log n)$ zapytań dla struktur $(k-1)$-wymiarowych (patrz rys. 7).



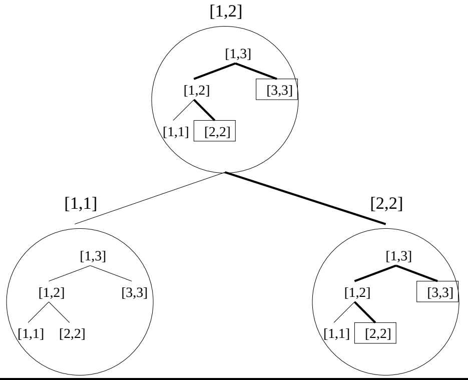
Rys. 5: Przykład drzewa drzew dla elementów ze zbioru $[1,2] \times [1,3]$.



Rys. 6: Wstawienie elementu $(1,2)$ do struktury z poprzedniego rysunku.

Powróćmy do problemu z zadania. Na początku rozdziału przedstawiliśmy klocki jako krotki 3-wymiarowe. Zastosowanie dla nich opisanej powyżej struktury pozwala więc rozwiązać zadanie w czasie $O(n\log^3 n)$. Proste spostrzeżenie pozwala zredukować ten czas do $O(n\log^2 n)$. Wystarczy zauważyć, że klocki rozważamy w kolejności rosnących pierwszych współrzędnych i, gdy pytamy o klocki, dla których pierwsza współrzędna należy do przedziału $[1,i)$, to pytamy o wszystkie dotychczas przetworzone klocki. To oznacza, że pierwszą współrzędną możemy pominąć i rozważać wszystkie przeanalizowane dotychczas klocki, dla których jedna współrzędna (początkowo druga) należy do przedziału $[1,a_i)$, a druga (początkowo trzecia) — do przedziału $[0,i-a_i]$.

Pomimo kolejnego usprawnienia nie będzie chyba dla nikogo niespodzianką, że żaden z zawodników, podobnie jak i autor programu wzorcowego, nie pokusił się o implementację opisanego tu rozwiązania. Problem tkwi w tym, że praktycznie nie ma szans, by implementacja struktury w wersji statycznej zmieściła się w zadanych ograniczeniach pamięciowych. Natomiast zaprogramowanie dynamicznie aktualizowanego drzewa drzew zrównoważonych (czyli struktury dla $k = 2$) jest praktycznie niemożliwe w trakcie pięciogodzinnej sesji (nawet samodzielna implementacja zwykłego drzewa AVL nie jest ani łatwa, ani przyjemna). Dodatkowo rozwiązanie to charakteryzuje się dużą stałą w złożoności czasowej, co uniemożliwiłoby zapewne zmieszczenie się w limicie czasowym. Z powyższych powodów zamieszczony opis struktury danych ma jedynie charakter szkicu. Natomiast zainteresowanym zawodnikom niewątpliwie



Rys. 7: Węzły analizowane w trakcie zapytania o maksimum wartości elementów, których pierwsza współrzędna należy do przedziału $[2,2]$, a druga do przedziału $[2,3]$. Pytamy się zatem o maksimum wartości elementów należących do prostokąta $[2,2] \times [2,3]$.

warto polecić dokładniejszą analizę oraz implementację tego rozwiązania — można ją zrealizować szczególnie elegancko w funkcyjnym języku programowania, na przykład Ocamlu.

Rozwiązanie wzorcowe

Pod koniec poprzedniego rozdziału zredukowaliśmy złożoność naszego problemu, dzięki temu, że jeden z warunków (1)–(3) w naturalny sposób wynikał z porządku, w jakim rozważaliśmy klocki. Udało nam się w ten sposób wyeliminować sprawdzanie warunku (1). Zauważmy, że zmieniając porządek przeglądania klocków na kolejność według wartości a_i albo $i - a_i$, moglibyśmy zamiast warunku (1) wyeliminować równie dobrze warunek (2) albo (3). Spostrzeżenie to daje nam pewną swobodę wyboru, która bardzo nam się przyda, gdyż chcielibyśmy zredukować rozmiar struktury danych do przechowywania klocków o jeszcze jeden stopień, osiągając $k = 1$. Wówczas pozostanie nam zaimplementowanie zwykłego drzewa licznikowego.

W pliku `klob3.cpp` znajduje się *niepoprawne* rozwiązanie, w którym przy wyznaczaniu wartości t_i pominięty został warunek (3). *Nie dostaniemy* także dobrego rozwiązania, gdy pominiemy warunek (2). Okazuje się jednak, że możemy zrezygnować z warunku (1)! Jeśli bowiem przy wyznaczaniu wartości t_i ograniczymy się do j, dla których zachodzą warunki (2)–(3), czyli $a_j < a_i$ oraz $j - a_j \leqslant i - a_i$, to automatycznie mamy zagwarantowaną własność $j < i$, która wynika z dwu pozostałych. Możemy więc wybrać porządek rozpatrywania klocków według wartości a_i i zbudować drzewo licznikowe dla elementów $j - a_j$, dla których będziemy pamiętać wartości t_j. W ten sposób podczas rozważania klocka i-tego:

- warunek (2), czyli ograniczenie się do klocków, dla których $a_j < a_i$, wyniknie z kolejności rozpatrywania klocków,
- warunek (3) spełnimy, wyszukując w strukturze elementy z przedziału $[0, i - a_i]$,
- warunek (1) wyniknie sam z pozostałych dwu.

Tym samym otrzymaliśmy eleganckie rozwiązanie o złożoności czasowej $O(n \log n)$, które zostało zaimplementowane w pliku `klob8.cpp` i … jest rozwiązaniem błędnym! Rozwiązanie to pozwalało zdobyć na zawodach nieco ponad połowę punktów.

Zastanówmy się, gdzie tkwi błąd. Otóż zapomnieliśmy, że nierówności z warunków (1) oraz (2) muszą być *ostre*. Dla warunku (1) nie musimy się tym przejmować — i tak każdy klocek ma inny numer w początkowej wieży, więc nierówności automatycznie są ostre. Jednak rozważając klocki uporządkowane według wartości a_i, musimy jakoś rozstrzygać remisy. Gdybyśmy się nimi nie przejmowali, to dla $j < i$ oraz $a_j = a_i$, licząc t_i, moglibyśmy wziąć pod uwagę $t_j + 1$. Jest to błąd, bo przecież nie możemy jednocześnie w rozwiązaniu mieć obu klocków: i-tego oraz j-tego, zajmujących w wieży wynikowej tę samą pozycję $a_i = a_j$. Najłatwiejszym sposobem wybrnięcia z tej kłopotliwej sytuacji jest uważne posortowanie klocków według wartości a_i — w taki sposób, by klocki z takimi samymi numerami były rozważane w kolejności *od góry do dołu*. Wówczas, jeżeli $j < i$ oraz $a_j = a_i$, to najpierw rozważymy klocek i-ty, a rozpatrując go, nie będziemy jeszcze mieli w drzewie klocka j-tego. Przy liczeniu t_i nie uwzględnimy więc $t_j + 1$, czyli unikniemy poprzedniego błędu. W ten sposób otrzymujemy poprawny sposób wyznaczania wartości t_i, zaimplementowany w plikach `klo.cpp` oraz `klo2.pas`.

Pozostaje jeszcze jeden szczegół, o którym wypada przypomnieć — poszukiwanym rozwiązaniem nie jest t_i, lecz numery klocków, które należy usunąć. Aby je wyznaczyć, posłużymy się, podobnie jak w rozwiązaniu o złożoności kwadratowej względem n, wartościami p_i. To oznacza, że w węzłach drzewa licznikowego, oprócz wartości maksymalnych będziemy zapisywać, skąd te maksima pochodzą (z których liści drzewa się wywodzą). Znając wartości p_i, możemy wygenerować rozwiązanie identycznie, jak w poprzednim algorytmie.

Nietrudno zauważyć, że wymaganie w odpowiedzi numerów klocków do usunięcia zamiast wartości t_i nie ma wpływu na trudność problemu. Można by więc spytać, dlaczego autorom zadania nie wystarcza, by program wyznaczał optymalną liczbę klocków pozostających na swoim miejscu po przebudowie. Odpowiedź na to pytanie jest prosta: znajomość samego wyniku nie pomogłaby Agatce w pousuwaniu z wieży odpowiednich klocków!

Inne spojrzenie na rozwiązanie wzorcowe

Problem wyznaczania wartości t_i możemy sformułować także trochę inaczej. Przedstawmy ponownie każdy klocek jako trójwymiarową krotkę o współrzędnych $(i, a_i, i-a_i)$. Licząc t_i, bierzemy pod uwagę wartość t_j+1, jeżeli krotka odpowiadająca j-temu klockowi ma pierwsze dwie współrzędne mniejsze od krotki odpowiadającej klockowi i-temu, a trzecią — nie większą. Wynika to wprost z warunków (1)–(3). To oznacza, że klocki, które w skonstruowanym przez nas rozwiązaniu optymalnym znajdą się na swoich miejscach, będą tworzyły ciąg punktów z przestrzeni trójwymiarowej:

- rosnący ze względu na pierwszą i drugą współrzędną oraz
- niemalejący ze względu na trzecią współrzędną.

Podobnie jak poprzednio, możemy zauważyć, że ciąg właściwie uporządkowany ze względu na drugą i trzecią współrzędną, musi być rosnący ze względu na pierwszą współrzędną. To oznacza, że wystarczy reprezentować klocki jako krotki dwuwymiarowe postaci $(a_i, i-a_i)$, a konstruowany ciąg będzie rosnący względem pierwszej współrzędnej i zarazem niemalejący względem drugiej (taki ciąg nazwiemy *rosnąco-niemalejącym*).

W jaki sposób taki ciąg wyznaczyć? Można, na przykład, ustawić wszystkie krotki w ciąg S, sortując je niemalejąco względem drugiej współrzędnej $(i-a_i)$, a ewentualne remisy rozstrzygając na korzyść krotek o mniejszej pierwszej współrzędnej a_i. Okazuje się, że wybierając podciąg ciągu S rosnący ze względu na pierwszą współrzędną (a_i), dostajemy rosnąco-niemalejący ciąg krotek, jeśli weźmiemy pod uwagę obie współrzędne $(a_i, i-a_i)$. Zachodzi także stwierdzenie odwrotne — każdy rosnąco-niemalejący ciąg krotek odpowiada podciągowi rosnącemu ze względu na pierwszą współrzędną w ciągu S. W ten sposób sprowadziliśmy rozważany problem do wyznaczania (długości) najdłuższego podciągu rosnącego ciągu liczbowego. Jest to klasyczne zadanie, które można wykonać w złożoności czasowej $O(n \log n)$ — na przykład za pomocą drzewa przedziałowego. Istnieje także inny algorytm, niewykorzystujący złożonych struktur danych, a oparty jedynie na wyszukiwaniu binarnym (jego opis można znaleźć w opracowaniu zadania *Egzamin na prawo jazdy* w niniejszej książeczce). Duża grupa zawodników, którzy rozwiązali niniejsze zadanie, uzyskując maksymalną liczbę punktów, wykorzystała metodę podobną do powyższej.

Na koniec warto jeszcze chwilę zastanowić się, czy przy konstrukcji ciągu S wybór współrzędnej, według której sortujemy, jest istotny. Przypomnijmy, że chodzi nam o znalezienie podciągu właściwie uporządkowanego względem obu współrzędnych:

- dla drugiej $(i-a_i)$ porządek niemalejący gwarantujemy sobie, sortując elementy,
- dla pierwszej (a_i) porządek rosnący uzyskujemy, wybierając podciąg rosnący.

Czy coś stoi na przeszkodzie, by porządek dla pierwszej współrzędnej zapewnić w trakcie sortowania (budowy ciągu S), a dla drugiej — w trakcie wybierania podciągu (tym razem niemalejącego) z ciągu S? Owszem, jest pewien drobny problem, na który natknęliśmy się już wcześniej. Powiedzmy, że posortowaliśmy punkty niemalejąco względem pierwszej współrzędnej (a_i) i szukamy podciągu niemalejącego względem drugiej $(i-a_i)$, a w danych trafiły się nam dwa klocki, dla których $a_i = a_j$ oraz $j < i$. Wówczas może się zdarzyć, że do ciągu wynikowego weźmiemy oba te klocki (bo $j - a_j < i - a_i$), a przecież nie mogą one równocześnie być na swoim miejscu w rozwiązaniu wynikowym. Wcześniej opisany wybór porządku sortowania powoduje natomiast, że możemy poszukiwać podciągu *rosnącego*, a nie *niemalejącego* i problem klocków o równych wartościach a_i po prostu nie występuje.

Inne rozwiązania

Jeszcze inne rozwiązanie poprawne

Na początku opracowania stwierdziliśmy, że istnieje wiele poprawnych rozwiązań opartych na technice programowania dynamicznego. Aby nie być gołosłownym, zaprezentujemy krótko jeszcze jedno podejście.

Rozwiązanie nazwiemy (h,c)-rozwiązaniem, jeżeli wieża wynikowa ma h klocków, z których dokładnie c znajduje się na swoim miejscu. Zdefiniujmy $T_{h,c}$ jako minimalną wysokość dowolnego fragmentu wieży początkowej, z którego można, usuwając klocki, uzyskać (h,c)-rozwiązanie. Dla $h > 0$ oraz $c > 0$ mamy następujący wzór:

$$T_{h,c} = \min(T_{h-1,c} + 1, \min\{i : a_i = h \text{ oraz } T_{h-1,c-1} < i\}),$$

przy czym, jeżeli dla pewnych h oraz c nie istnieje (h,c)-rozwiązanie, to $T_{h,c} = \infty$. Aby przekonać się o poprawności podanego wzoru, przeanalizujmy konstrukcję (h,c)-rozwiązania. Chcąc skonstruować wieżę o wysokości h i c klockach zajmujących swoje pozycje:

- możemy wziąć niższą o jeden klocek wieżę już zawierającą c klocków na swoich miejscach ($(h-1,c)$-rozwiązanie) i na górę dostawić najniższy z pozostałych po jej budowie klocków (stąd we wzorze wyrażenie $T_{h-1,c}+1$) albo
- jeśli $a_i = h$ oraz z klocków o numerach $\{1,2,\ldots,i-1\}$ da się zbudować $(h-1,c-1)$-rozwiązanie (czyli $T_{h-1,c-1} < i$), to możemy dołożyć na jego szczycie pasujący tam klocek i-ty.

W drugim z przypadków, numer i dostawianego klocka decyduje o wartości $T_{h,c}$, a zatem chcemy wybrać najniższy tego typu klocek (stąd minimum po i we wzorze).

Uzupełniając podany wyżej wzór dla przypadków skrajnych (kiedy $h=0$, $c=0$ lub $T_{h-1,c}=n$), otrzymujemy metodę wyznaczenia całej tablicy T. Maksymalna wartość c, dla której istnieje wartość h, taka że $T_{h,c} \neq \infty$, jest wówczas poszukiwanym rozwiązaniem — maksymalną liczbą klocków, które mogą znaleźć się na swoich miejscach w wieży wynikowej. W najprostszej implementacji rozwiązanie to ma złożoność czasową $O(n^3)$ (trzeba wyliczyć wartości $O(n^2)$ komórek tablicy, a wyznaczenie każdej z nich wymaga czasu $O(n)$). Sprytniejsze przeanalizowanie wszystkich klocków, dla których $a_i = h$ (przy ustalonej wartości parametru h), pozwala usprawnić powyższe rozwiązanie i wykonać obliczenia w złożoności czasowej $O(n^2)$. Rozwiązanie można jeszcze bardziej przyspieszyć, otrzymując złożoność czasową $O(n\log n)$. Duży stopień skomplikowania tej metody spowodował jednak, iż zdecydowaliśmy się pominąć jej opis — zainteresowani mogą jednak prześledzić jej implementację w pliku `klo5.cpp`.

Rozwiązania błędne

W pliku `klob4.cpp` znajduje się rozwiązanie zachłanne, w którym budujemy wieżę, wybierając w każdym kroku klocek o najmniejszym numerze i, który możemy umieścić na swoim miejscu w wieży wynikowej. To rozwiązanie przechodziło zaledwie jeden test. Rozwiązanie `klob6.cpp`, także niepoprawne, działa podobnie, jednak jako kolejny do umieszczenia na swoim miejscu wybiera klocek o najmniejszej wartości a_i. Takie rozwiązanie nie pozwalało zdobyć na zawodach żadnych punktów.

Testy

Zadanie było sprawdzane na 11 zestawach danych wejściowych. Wszystkie testy, z wyjątkiem testów z grup b, to testy generowane w pewnym stopniu losowo. Wykorzystane były dwie metody losowej generacji danych:

- losowe rozmieszczenie liczb na klockach — takie testy w poniższej tabelce nazywamy po prostu „losowymi”,
- rozmieszczenie na klockach liczb wzrastających stopniowo, tzn. liczb losowych położonych niedaleko pewnej liniowej funkcji rosnącej (zaletą takich testów jest istnienie rozwiązania, w którym duża liczba klocków znajduje się na swoich miejscach) — testy te nazwaliśmy „podłużnymi”.

W poniższym zestawieniu przez n oznaczono rozmiar testu, a przez w wynik (liczbę klocków, które mogą znaleźć się na swoich miejscach).

Nazwa	n	w	Opis
klo1.in	100	9	test losowy
klo2.in	500	23	test losowy
klo3.in	2 000	43	test losowy
klo4.in	7 000	2 430	test podłużny
klo5.in	8 000	87	test losowy
klo6.in	50 000	218	test losowy
klo7.in	100 000	41 019	test podłużny
klo8.in	100 000	2 206	test podłużny
klo9a.in	100 000	486	test podłużny
klo9b.in	100 000	100 000	test od razu ułożony
klo10a.in	100 000	24 303	test podłużny
klo10b.in	100 000	0	test złożony tylko z dużych liczb
klo11.in	100 000	4 975	test podłużny

Wojciech Rytter, Andrzej Walat **Szymon Acedański**
Treść zadania, Opracowanie Program

OI, Etap III, dzień drugi, 29.03.2007

Waga czwórkowa

Smok Bajtazar zamierza zorganizować ucztę, na którą chce zaprosić wielu gości. Dla uświetnienia uczty, Bajtazar chce każdemu z gości podarować pewną ilość złota. Przy tym każdy z gości powinien dostać tyle samo złota, aby nikt nie czuł się pokrzywdzony.

Smok będzie odważał złoto dla kolejnych gości wagą szalkową. Ma on do dyspozycji odważniki, których wagi w gramach są potęgami czwórki. Odważników każdego rodzaju ma dowolnie wiele. Bajtazar będzie zawsze kładł złoto na lewej szalce wagi, a odważniki na prawej lub na obu szalkach. Bajtazar przy każdym ważeniu chce użyć najmniejszej możliwej liczby odważników. Ponadto, aby gościom się nie nudziło, chce on każdemu z nich odważyć złoto w inny sposób (tzn. używając innych odważników lub inaczej je rozdzielając pomiędzy szalki).

Ponieważ smok nie umie za dobrze liczyć, polecił ci napisanie programu, który obliczy, ilu maksymalnie gości może zaprosić, czyli wyznaczy maksymalną liczbę sposobów, na jakie można odważyć n gramów złota, używając minimalnej liczby odważników. Jeśli dobrze się sprawisz, smok obsypie Cię złotem!

Zadanie

Napisz program który:

- *wczyta ze standardowego wejścia, ile złota (w gramach) Bajtazar zamierza dać każdemu z gości,*
- *obliczy, na ile sposobów można odważyć taką ilość złota za pomocą minimalnej liczby odważników,*
- *wypisze resztę z dzielenia wyniku przez 10^9 na standardowe wyjście.*

Wejście

W pierwszym i jedynym wierszu standardowego wejścia zapisania jest jedna liczba całkowita n, $1 \leqslant n < 10^{1000}$. Jest to ilość złota (w gramach), jaką Bajtazar chce dać każdemu z gości.

Wyjście

Na standardowe wyjście należy wypisać jedną liczbę całkowitą — resztę z dzielenia przez 10^9 maksymalnej liczby gości, których Bajtazar może zaprosić (czyli maksymalnej liczby sposobów, na jakie można odważyć n gramów złota, używając minimalnej możliwej liczby odważników).

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`166`	`3`

Do odważenia 166 gramów potrzeba użyć 7 odważników. Ważenie można wykonać na następujące sposoby:

1. *na lewej szalce złoto, na prawej odważniki 64, 64, 16, 16, 4, 1, 1;*
2. *na lewej szalce złoto i odważniki 64, 16, 16, na prawej odważniki 256, 4, 1, 1;*
3. *na lewej szalce złoto i odważniki 64, 16, 4, 4, 1, 1, na prawej odważnik 256.*

i

Rozwiązanie

Algorytm korzysta istotnie z reprezentacji czwórkowej liczby n. Niech

$$n = n_0 \cdot 4^k + n_1 \cdot 4^{k-1} + \cdots + n_k \cdot 4^0;$$

wówczas

$$n = [n_0, n_1, n_2, \ldots, n_k]_4$$

oznacza reprezentację czwórkową liczby n. Tradycyjnie przyjmujemy, że $n_0, n_1, \ldots, n_k$ są *cyframi czwórkowymi* i pochodzą ze zbioru $\{0,1,2,3\}$. Możemy jednak tę reprezentację uogólnić na system z cyframi ujemnymi $\{-3,-2,-1,0,1,2,3\}$. Nawiązując do treści zadania, możemy powiedzieć, że cyfra ujemna $-s$ oznacza umieszczenie s odważników o odpowiedniej wadze na lewej szalce — tej samej, co ważony przedmiot.

Przykład 1
$166 = [2,2,1,2]_4$, ponieważ $166 = 2\cdot 4^3 + 2\cdot 4^2 + 1\cdot 4^1 + 2\cdot 4^0$.
Z drugiej strony
$166 = [1,-1,-2,1,2]_4$, ponieważ $166 = 1\cdot 4^4 - 1\cdot 4^3 - 2\cdot 4^2 + 1\cdot 4^1 + 2\cdot 4^0$.

Traktując reprezentację czwórkową $[n_0, n_1, \ldots, n_k]$ jako sposób zważenia sztabki o wadze n, widzimy, że liczba użytych odważników to suma wartości bezwzględnych cyfr w reprezentacji. Minimalną wartość tej sumy oznaczymy przez $M(n)$. *Minimalna reprezentacja* to taka, w której suma cyfr wynosi $M(n)$.

W zadaniu chodzi o wyznaczenie liczby, oznaczanej tutaj przez $ILE(n)$, minimalnych reprezentacji liczby n.

Przykład 2 Dla rozważanego przykładu zachodzi:
$M(166) = 7$ (suma minimalna cyfr wynosi 7) oraz $ILE(166) = 3$ (mamy 3 minimalne reprezentacje):

$$166 = [2,2,1,2]_4 = [1,-1,-1,-2,-2]_4 = [1,-1,-2,1,2]_4.$$

Obserwacja 1 *Zauważmy, że w minimalnej reprezentacji nigdy nie wystąpią cyfry o wartości bezwzględnej większej niż* 2. *Jest oczywiste, że użycie czterech takich samych odważników jest nieoszczędne — zawsze można je zastąpić jednym większym odważnikiem. Natomiast trzy odważniki o wadze* 4^i *zawsze można zastąpić jednym odważnikiem o wadze* 4^{i+1} *oraz jednym odważnikiem o wadze* 4^i *postawionym na przeciwnej szalce.*

Przyjmijmy, że $[n_0, n_1, \ldots, n_k]_4$ jest *tradycyjną reprezentacją czwórkową* n, tzn. $n_i \in \{0,1,2,3\}$ dla $i = 0,1,\ldots,k$. Zdefiniujmy także $n^{<i>} = [n_0, n_1, \ldots, n_{i-1}]_4$ — jest to wartość liczby, której reprezentacją jest prefiks ciągu $[n_0, n_1, n_2, \ldots, n_k]$ długości i (złożony z i najbardziej znaczących cyfr). Dodatkowo wprowadźmy oznaczenia:

- $x_i = M(n^{<i>})$;
- $y_i = M(n^{<i>} + 1)$;
- $X_i = ILE(n^{<i>})$;
- $Y_i = ILE(n^{<i>} + 1)$.

Łatwo zauważyć, że poszukiwanym wynikiem jest liczba X_{k+1}. Skonstruujemy algorytm, który będzie wyznaczał kolejno wszystkie czwórki x_i, y_i, X_i, Y_i dla $i = 1, \ldots, k+1$.

Pierwszy sukces — wyznaczenie liczb x_i, y_i

Na początku policzymy tylko sumę cyfr w minimalnej reprezentacji, czyli minimalną liczbę odważników. Kluczem do sukcesu jest wykorzystanie obu wartości — x_i oraz y_i — do wyznaczenia kolejnej pary x_{i+1} oraz y_{i+1}. Zauważmy, że $n^{<i+1>} = n^{<i>} \cdot 4 + n_i$. W zależności od wartości n_i poszukiwane liczby x_{i+1} oraz y_{i+1}, wyznaczamy następująco:

Oblicz(x_{i+1}, y_{i+1})

$$\begin{array}{lcl}
n_i = 0 & \Longrightarrow & (x_{i+1}, y_{i+1}) = (x_i, x_i + 1) \\
n_i = 1 & \Longrightarrow & (x_{i+1}, y_{i+1}) = (x_i + 1, \min(x_i + 2, y_i + 2)) \\
n_i = 2 & \Longrightarrow & (x_{i+1}, y_{i+1}) = (\min(x_i + 2, y_i + 2), y_i + 1) \\
n_i = 3 & \Longrightarrow & (x_{i+1}, y_{i+1}) = (y_i + 1, y_i)
\end{array}$$

Uzasadnijmy, że nasza procedura jest poprawna. Rozważmy w tym celu wyznaczenie wartości x_{i+1} we wszystkich czterech przypadkach. Wartości y_{i+1} to wartości x_{i+1} z „sąsiedniego” przypadku, więc uzasadnienia poprawności obliczeń dla nich pominiemy.

Przypadek 1: $n_i = 0$**.** Jeśli do dotychczas rozpatrywanej wartości $n^{<i>}$ dopisujemy na końcu zero, to optymalny sposób jej zważenia polega na zastąpieniu wszystkich odważników czterokrotnie cięższymi. Gdyby bowiem istniał lepszy dobór $x_{i+1} < x_i$ odważników, wówczas także liczbę $n^{<i>}$ dałoby się odważyć przy pomocy x_{i+1} odważników. Wśród odważników tworzących minimalną reprezentację $n^{<i+1>}$ nie ma bowiem odważnika o wadze jeden i wszystkie odważniki tworzące tę reprezentację możemy zastąpić czterokrotnie lżejszymi.

Przypadek 2: $n_i = 1$. Aby zważyć sztabkę o wadze $n^{<i+1>}$, możemy zmienić wszystkie odważniki używane w minimalnej reprezentacji $n^{<i>}$ na czterokrotnie cięższe i dołożyć odważnik o wadze jednostkowej na prawej szalce, przeciwnej niż sztabka. Ważymy w ten sposób, używając $x_i + 1$ odważników. Dlaczego nie można lepiej? W optymalnym zestawie odważników dla sztabki o wadze $n^{<i+1>}$ musi być dokładnie jeden odważnik o wadze 1 (w optymalnym zestawie nie używamy bowiem więcej niż dwóch odważników tego samego typu i tylko zestawy z jednym odważnikiem jednostkowym pozwalają nam odważyć wartości nieparzyste). Wyrzucając go z zestawu, dostajemy zestaw o wadze $4 \cdot n^{<i>}$ złożony z $x_{i+1} - 1$ odważników. Możemy je zastąpić czterokrotnie lżejszymi, otrzymując rozwiązanie dla $n^{<i>}$, a zatem $x_{i+1} - 1 \leqslant x_i$, z optymalności x_i mamy również nierówność przeciwną $x_{i+1} - 1 \geqslant x_i$, czyli $x_{i+1} - 1 = x_i$.

Przypadek 3: $n_i = 2$. Jeśli do $n^{<i>}$ dopisujemy na końcu dwójkę, to nową sztabkę $n^{<i+1>}$ możemy zważyć:

- w zestawie x_i odważników służącym do zważenia $n^{<i>}$ zamieniając wszystkie odważniki na czterokrotnie cięższe i dokładając dwa o wadze jeden lub
- biorąc optymalny zestaw y_i odważników dla $n^{<i>} + 1$, zamieniając w nim wszystkie odważniki na czterokrotne cięższe i dokładając dwa o wadze jeden na tę samą szalkę, co sztabka.

W pierwszym przypadku użyjemy $x_i + 2$ odważniki, w drugim przypadku wykorzystamy $y_i + 2$ odważniki. Za x_{i+1} przyjmujemy lepszą z tych możliwości. Dlaczego inny sposób nie może być lepszy? Załóżmy, że uda nam się zważyć $n^{<i+1>}$ lepiej, używając $x_{i+1} < \min(x_i + 2, y_i + 2)$ odważników. Ponieważ wartość $n^{<i+1>}$ daje resztę 2 z dzielenia przez 4, więc w optymalnym zestawie odważników muszą być dwa odważniki o wadze 1 i muszą stać na tej samej szalce. Usuwając je, zmieniamy odważoną liczbę o -2 (jeśli stały na przeciwnej szalce niż sztabka) albo o 2 (jeśli stały na tej samej szalce, co sztabka). W uzyskanym zestawie wszystkie odważniki możemy zastąpić czterokrotnie lżejszymi. Taka operacja daje nam zestaw o wadze $n^{<i>}$ lub $n^{<i>} + 1$ złożony z $x_{i+1} - 2$ odważników, co pozwalałoby zważyć $n^{<i>}$ lub $n^{<i>} + 1$ lepiej niż optymalnie!

Przypadek 4: $n_i = 3$. Aby ustawić odważniki o łącznej wadze $n^{<i+1>}$, możemy skomponować zestaw dla wagi $n^{<i>} + 1$, następnie zamienić wszystkie odważniki na czterokrotnie cięższe i potem na szalce ze sztabką położyć odważnik jednostkowy. Dostaniemy w ten sposób zestaw złożony z $y_i + 1$ odważników. Czy jest on optymalny? Załóżmy, że nie jest i że można sztabkę o wadze $n^{<i+1>}$ zważyć, używając $x_{i+1} < y_i + 1$ odważników. Wśród nich musi być jeden odważnik jednostkowy (bo sztabka ma nieparzysty ciężar) — usuńmy go. Dostaliśmy w ten sposób zestaw bez odważników jednostkowych, gotowy do odważenia sztabki o wadze równej (w zależności od tego, z której szalki usunęliśmy odważnik):

- $n^{<i+1>} + 1 = n^{<i>} \cdot 4 + 4$, w którym możemy zamienić wszystkie odważniki na czterokrotnie lżejsze, uzyskując zestaw do odważenia sztabki $n^{<i>} + 1$ przy pomocy mniej niż y_i odważników, co jest niemożliwe;
- $n^{<i+1>} - 1 = n^{<i>} \cdot 4 + 2$, co jest niemożliwe, bo w zestawie nie ma już odważników jednostkowych, więc można odważyć nim tylko wielokrotności liczby 4.

Rozważenie wszystkich przypadków pozwoliło nam upewnić się, że mamy szybką i prostą metodę wyznaczania wartości x_i oraz y_i.

Ostateczny sukces — wyznaczenie liczb X_i, Y_i

Przypomnijmy, że X_i oznacza liczbę optymalnych sposobów zważenia sztabki o wadze $n^{<i>}$, a Y_i — liczbę optymalnych sposobów dla sztabki o wadze $n^{<i>} + 1$. Ponadto X_{k+1} jest poszukiwanym wynikiem. Do wyznaczenia X_i oraz Y_i wykorzystamy następujące zależności:

Oblicz(X_{i+1}, Y_{i+1})

$$\begin{array}{rcl}
n_i = 0 & \Longrightarrow & (X_{i+1}, Y_{i+1}) = (X_i, X_i) \\
n_i = 1, x_i < y_i & \Longrightarrow & (X_{i+1}, Y_{i+1}) = (X_i, X_i) \\
n_i = 1, x_i = y_i & \Longrightarrow & (X_{i+1}, Y_{i+1}) = (X_i, X_i + Y_i) \\
n_i = 1, x_i > y_i & \Longrightarrow & (X_{i+1}, Y_{i+1}) = (X_i, Y_i) \\
n_i = 2, x_i < y_i & \Longrightarrow & (X_{i+1}, Y_{i+1}) = (X_i, Y_i) \\
n_i = 2, x_i = y_i & \Longrightarrow & (X_{i+1}, Y_{i+1}) = (X_i + Y_i, Y_i) \\
n_i = 2, x_i > y_i & \Longrightarrow & (X_{i+1}, Y_{i+1}) = (Y_i, Y_i) \\
n_i = 3 & \Longrightarrow & (X_{i+1}, Y_{i+1}) = (Y_i, Y_i)
\end{array}$$

Zastanówmy się, czy zaproponowana metoda jest poprawna. W uzasadnieniu — podobnie jak poprzednio — ograniczymy się do wartości X_i. Przypomnijmy, że w poprzednim rozdziale pokazaliśmy, jak rozwiązania optymalne dla $n^{<i>}$ przekształcać w rozwiązania optymalne dla $n^{<i+1>}$ i *vice versa*.

Przypadek 1: $n_i = 0$. Każde rozwiązanie optymalne dla $n^{<i>}$ możemy przekształcić w inne rozwiązanie optymalne dla $n^{<i+1>}$ — wystarczy zastąpić wszystkie odważniki czterokrotnie cięższymi. To dowodzi, że $X_i \leqslant X_{i+1}$. Także każde rozwiązanie optymalne dla $n^{<i+1>}$ możemy przekształcić w inne rozwiązanie optymalne dla $n^{<i>}$ — wystarczy zastąpić wszystkie odważniki czterokrotnie lżejszymi. Stąd $X_{i+1} \leqslant X_i$.

Przypadek 2: $n_i = 1$. Jak poprzednio, każde rozwiązanie optymalne dla $n^{<i>}$ przekształcamy w jedno rozwiązanie optymalne dla $n^{<i+1>}$ i *vice versa*. Ponownie więc mamy równość $X_i = X_{i+1}$.

Przypadek 3: $n_i = 2$. Tym razem może być różnie. Tworząc rozwiązanie optymalne dla $n^{<i+1>}$, wybieramy lepszą z dwóch możliwości. Jeśli $x_i < y_i$, to przekształcamy rozwiązanie optymalne dla $n^{<i>}$ i łatwo zauważyć, że jest to przekształcenie różnowartościowe w obie strony (a więc w tym przypadku $X_i = X_{i+1}$). Jeśli $x_i > y_i$, to przekształcamy rozwiązanie optymalne dla $n^{<i>} + 1$ (i analogicznie jak poprzednio mamy $Y_i = X_{i+1}$). Jeśli $x_i = y_i$, to każde rozwiązanie dla $n^{<i>}$ *oraz* każde rozwiązanie dla $n^{<i>} + 1$ można przekształcić w optymalne rozwiązanie dla $n^{<i+1>}$. Co ważne, łatwo zauważyć, że rozwiązania te są różne. To dowodzi, że w tym przypadku $X_{i+1} \geqslant X_i + Y_i$. Ponieważ przekształcenie odwrotne pozwala nam z każdego rozwiązania optymalnego dla $n^{<i+1>}$ otrzymać rozwiązanie optymalne dla $n^{<i>}$ *albo* rozwiązanie optymalne dla $n^{<i>} + 1$, więc także $X_{i+1} \leqslant X_i + Y_i$.

Przypadek 4: $n_i = 3$. W tym przypadku nie mamy wyboru, podobnie jak w dwóch pierwszych przypadkach. Rozwiązanie optymalne dla $n^{<i+1>}$ otrzymujemy z rozwiązania optymalnego dla $n^{<i>} + 1$ i *vice versa*. Ponadto oba przekształcenia są różnowartościowe, więc $X_{i+1} = Y_i$.

Algorytm

Przeprowadzone rozważania pozwalają nam skonstruować bardzo prosty algorytm rozwiązania zadania:

```
$x_0$ := 0; $y_0$ := 1; $X_0$ := 1; $Y_0$ := 1;
for $i$ := 0 to $k$-1 do
begin
  Oblicz($x_{i+1}$,$y_{i+1}$);
  Oblicz($X_{i+1}$,$Y_{i+1}$);
end
return $X_{k+1}$;
```

Jeżeli pominąć koszt implementacji własnej arytmetyki dużych liczb, to powyższe rozwiązanie ma złożoność czasową $O(k) = O(\log n)$ — taka jest liczba wykonywanych operacji porównań i dodawań. Jeśli uwzględnimy, że wszystkie te operacje są wykonywane na liczbach długości $O(\log n)$, to złożoność algorytmu wzrasta do $O(\log^2 n)$. Dodatkowo, sama zamiana danych na czwórkowy układ pozycyjny zajmuje czas $O(\log^2 n)$, co jest składnikiem decydującym o złożoności czasowej programu.

Implementacja rozwiązania wzorcowego znajduje się w plikach `wag.c`, `wag1.pas` i `wag2.cpp`.

Rozwiązania nieoptymalne

Istnieje szeroka gama rozwiązań nieoptymalnych; pokrótce wymienimy najważniejsze z nich.

Zauważmy, że ustaliwszy łączną masę odważników znajdujących się na lewej szalce, można łatwo obliczyć, jaką masę odważników trzeba ustawić na szalce prawej. Z kolei daną masę odważników m można zawsze ustawić na dokładnie jeden sposób, zakładając konieczność użycia minimalnej liczby odważników — liczbę tę można wyznaczyć jako sumę cyfr w rozwinięciu czwórkowym m (dowód tego prostego faktu pozostawiamy Czytelnikowi). Istnieje kilka różnych rozwiązań, które na wszystkie sensowne sposoby (czyli takie, które nie wykorzystują odważników istotnie cięższych niż n) próbują wybierać masę m odważników na lewej szalce i sprawdzać, czy otrzymany układ wykorzystuje nie więcej odważników niż dotychczasowe. Przykładowe implementacje znajdują się w plikach `wags1.c` (złożoność czasowa $O(5^{\log_4 n})$) i `wags2.c` (złożoność $O(n)$).

Istnieją także szybsze rozwiązania nieoptymalne, które częściowo opierają się na obserwacjach z rozwiązania wzorcowego. W rozwiązaniu zaimplementowanym w pliku `wags3.cpp` wykonujemy operację, w której próbujemy dopełniać aktualną wartość n do podzielnej przez 4 na dwa sposoby:

a) umieszczamy $(n \bmod 4)$ odważników jednostkowych na prawej szalce,

b) umieszczamy $(4 - n \bmod 4)$ odważników jednostkowych na lewej szalce.

Dla każdego z przypadków następuje następnie wywołanie rekurencyjne, w którym konstruujemy rozwiązanie dla $\frac{n}{4}$ w przypadku a), natomiast w przypadku b) — dla $\frac{n}{4} + 1$. A zatem w każdej fazie n zostaje podzielone mniej więcej przez 4, skąd można wywnioskować, że liczba faz będzie rzędu $O(\log_4 n)$. Ponieważ każda faza powoduje dwukrotne zwiększenie liczby wywołań rekurencyjnych, to złożoność czasowa całego algorytmu to (na mocy własności logarytmów):

$$O(2^{\log_4 n}) = O(2^{\frac{\log_2 n}{\log_2 4}}) = O(2^{0.5 \log_2 n}) = O(\sqrt{n}).$$

Wreszcie w pliku `wags4.cpp` jest opisane rozwiązanie o pesymistycznie takiej samej złożoności czasowej, ale w praktyce zachowujące się znacznie lepiej. Zauważamy mianowicie, że w przypadku, kiedy $(n \bmod 4) \in \{0,1,3\}$, wiemy dokładnie, na którą szalkę należy stawiać odważniki jednostkowe, żeby wynikowa liczba odważników była najmniejsza możliwa: dla wartości 0 i 1 będzie to szalka prawa, natomiast dla 3 — lewa. Dowód tego faktu wynika wprost z dowodu poprawności rozwiązania wzorcowego, lecz można go także wywnioskować prostszymi metodami. To pokazuje, że w przypadku reszt $0,1,3$ wystarczy jedno wywołanie rekurencyjne powyższego algorytmu, a jedynie dla reszty 2 musimy rozpatrzyć dwie możliwości. Opisane spostrzeżenie pozwalało na przejście praktycznie wszystkich testów dla $n \leqslant 10^{19}$, czyli w zakresie liczb całkowitych 64-bitowych.

Testy

Rozwiązania zawodników były sprawdzane na 12 zestawach testów, z których połowa była testami poprawnościowymi, odróżniającymi od siebie gorsze rozwiązania nieoptymalne, a połowa — wydajnościowymi, które przechodziło jedynie rozwiązanie wzorcowe. W poniższej tabelce n oznacza ilość złota do odważenia (lub liczbę cyfr liczby n, w przypadku gdy n jest bardzo duże), a w — resztę z dzielenia przez 10^9 wynikowej liczby sposobów odważenia masy n na szalce. W przypadku testów poprawnościowych wartości parametru w są dokładne (a nie tylko modulo 10^9).

Nazwa	n	w	Opis
wag1a.in	6 582	4	mały test poprawnościowy
wag1b.in	6 566	5	mały test poprawnościowy
wag1c.in	1	1	mały test poprawnościowy
wag1d.in	4	1	mały test poprawnościowy
wag2a.in	42 394	7	mały test poprawnościowy
wag2b.in	42 395	5	mały test poprawnościowy
wag3a.in	42 406	8	średni test poprawnościowy
wag3b.in	681 382	11	średni test poprawnościowy
wag4a.in	9 857 418	10	duży test poprawnościowy
wag4b.in	9 869 466	9	duży test poprawnościowy
wag5a.in	17-cyfrowe	63	(bardzo) duży test poprawnościowy
wag5b.in	17-cyfrowe	48	(bardzo) duży test poprawnościowy
wag6a.in	1 000-cyfrowe	927 630 336	test wydajnościowy
wag6b.in	1 000-cyfrowe	22 068 224	test wydajnościowy
wag7a.in	1 000-cyfrowe	750 522 880	test wydajnościowy
wag7b.in	1 000-cyfrowe	263 792 640	test wydajnościowy
wag8a.in	1 000-cyfrowe	667 363 840	test wydajnościowy
wag8b.in	1 000-cyfrowe	252 758 528	test wydajnościowy
wag9a.in	1 000-cyfrowe	324 093 440	test wydajnościowy
wag9b.in	1 000-cyfrowe	78 606 848	test wydajnościowy
wag10a.in	998-cyfrowe	928 779 529	test wydajnościowy
wag10b.in	998-cyfrowe	771 686 912	test wydajnościowy
wag11a.in	9 854 718	1	duży test poprawnościowy
wag11b.in	9 864 966	2	duży test poprawnościowy
wag12a.in	1 000-cyfrowe	20 610 560	test wydajnościowy
wag12b.in	1 000-cyfrowe	523 009 536	test wydajnościowy

Testy były generowane w sposób losowy, z dodatkowym założeniem, żeby w większości zestawów wartości parametru n w obu testach były zbliżone do siebie. Dodatkowo, sposób generowania uwzględniał to, żeby wyniki dla testów (wartości parametru w) były duże.

Rozwiązania o złożoności $\Omega(n)$ przechodziły odpowiednio pierwsze 2 i 3 testy. Rozwiązanie o złożoności czasowej $O(\sqrt{n})$, ale bez optymalizacji, przechodziło wszystkie testy poprawnościowe poza bardzo dużymi, natomiast rozwiązanie z optymalizacjami — wszystkie testy poprawnościowe. 50% testów przechodziła także implementacja rozwiązania wzorcowego nieużywająca arytmetyki dużych liczb.

XVIII Międzynarodowa Olimpiada Informatyczna

Mérida, Meksyk 2006

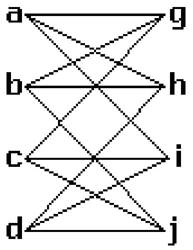
IOI 2006, dzień pierwszy, 15.08.2006

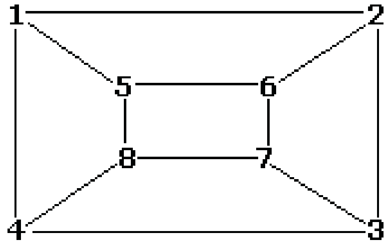
Zakazany podgraf

Dwa nieskierowane grafy G *i* H *są izomorficzne wtedy i tylko wtedy, gdy:*

- *mają taką samą liczbę wierzchołków oraz*
- *istnieje wzajemnie jednoznaczne przyporządkowanie wierzchołków* H *wierzchołkom* G*, takie że krawędź między dowolnymi dwoma różnymi wierzchołkami* G *istnieje wtedy i tylko wtedy, gdy istnieje krawędź między odpowiednimi wierzchołkami* H*.*

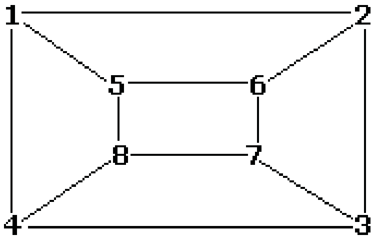
Dwa grafy przedstawione poniżej są izomorficzne, mimo, że wyglądają całkiem inaczej.

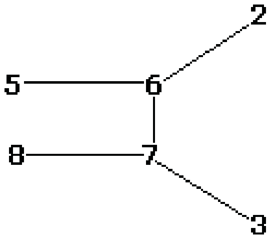




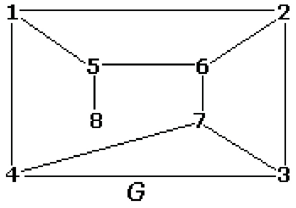
Jednym ze wzajemnie jednoznacznych przyporządkowań, które dowodzą, że te grafy są izomorficzne, jest: $\{a-1, b-6, c-8, d-3, g-5, h-2, i-4, j-7\}$*. Mogą też istnieć inne takie przyporządkowania.*

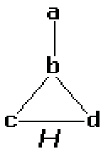
Podgraf grafu G *to graf, którego zbiór krawędzi i wierzchołków jest podzbiorem zbioru krawędzi i wierzchołków grafu* G*. Zauważ, ze* G *jest sam swoim podgrafem. Poniżej pokazano przykład grafu i jednego z jego podgrafów.*

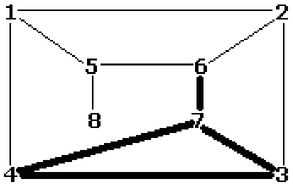




Powiemy, że graf G *zawiera inny graf* H *wtedy i tylko wtedy, gdy istnieje co najmniej jeden graf* H' *będący podgrafem* G*, taki że* H' *jest izomorficzny z* H*. Poniżej pokazano pewien graf* G *i graf* H*, taki że* G *zawiera* H*.*







Zadanie

Mając dane dwa nieskierowane grafy G *i* H*, wyznacz podgraf* G' *grafu* G*, taki że:*

- *liczby wierzchołków* G *i* G' *są takie same oraz*
- G' *nie zawiera* H*.*

Oczywiście może być wiele podgrafów G' *o tych właściwościach. Znajdź ten z nich, który ma możliwie najwięcej krawędzi.*

Algorytm podstawowy

Prawdopodobnie najbardziej podstawową strategią rozwiązywania tego problemu jest rozważać krawędzie G *w porządku ich występowania w pliku wejściowym i kolejno dodawać je do grafu* G'*, za każdym razem sprawdzając, czy* G' *zawiera* H*, czy nie. Poprawna implementacja tego algorytmu zachłannego przyniesie Ci trochę punktów. Wiedz jednak, że istnieją znacznie lepsze strategie.*

Ograniczenia

$3 \leqslant m \leqslant 4$ — liczba wierzchołków H,
$3 \leqslant n \leqslant 1\,000$ — liczba wierzchołków G.

Wejście

Otrzymasz 10 plików o nazwach od `forbidden1.in` *do* `forbidden10.in`*. Każdy z nich będzie zawierał następujące dane.*

`forbiddenK.in`	**OPIS**
`3 5`	**WIERSZ** *1*: *Zawiera dwie oddzielone odstępem liczby, m i n.*

`0 1 0`
`1 0 1`
`0 1 0`
	NASTĘPNE *m* **WIERSZY:** *Każdy wiersz zawiera m pooddzielanych pojedynczymi odstępami liczb całkowitych i reprezentuje jeden wierzchołek grafu H. Wierzchołki H wymienione są w kolejności 1,…,m. i–ta liczba w j–tym wierszu jest jedynką, gdy wierzchołki i oraz j są połączone krawędzią, a zerem w przeciwnym wypadku.*

`0 1 0 0 0`
`1 0 1 0 0`
`0 1 0 1 0`
`0 0 1 0 1`
`0 0 0 1 0`
	NASTĘPNE *n* **WIERSZY:** *Każdy wiersz zawiera n pooddzielanych pojedynczymi odstępami liczb całkowitych i reprezentuje jeden wierzchołek grafu G. Wierzchołki G wymienione są w kolejności 1,…,n. i–ta liczba w j–tym wierszu jest jedynką, gdy wierzchołki i oraz j są połączone krawędzią, a zerem w przeciwnym przypadku.*

Zauważ, że poza pierwszym wierszem, dane wejściowe są macierzami sąsiedztwa grafów H i G.

Wyjście

Masz dostarczyć 10 plików, po jednym dla każdego pliku wejściowego. Każdy z tych plików musi zawierać następujące dane:

`forbiddenK.out`	**OPIS**
`#FILE forbidden K`	**WIERSZ** *1*: *Nagłówek pliku. Nagłówek ten musi zawierać frazę:* `#FILE forbidden K`*, gdzie K to liczba z przedziału od 1 do 10 — numer odpowiedniego pliku wejściowego.*
`5`	**WIERSZ** *2*: *Zawiera jedną liczbę całkowitą: n.*

`0 1 0 0 0`
`1 0 0 0 0`
`0 0 0 0 0`
`0 0 0 0 0`
`0 0 0 0 0`
	NASTĘPNE *n* **WIERSZY:** *Każdy wiersz zawiera n pooddzielanych pojedynczymi odstępami liczb całkowitych i reprezentuje jeden wierzchołek grafu G′. Wierzchołki G′ powinny być wymienione w kolejności 1,…,n. i–ta liczba w j–tym wierszu jest jedynką, gdy wierzchołki i oraz j są połączone krawędzią, a zerem w przeciwnym przypadku.*

Poza pierwszymi dwoma wierszami, dane wejściowe są macierzami sąsiedztwa grafu G'. Zauważ, że może istnieć wiele poprawnych wyników. Wynik z powyższej tabeli jest poprawny, ale nieoptymalny.

Sposób oceniania

Twoja punktacja będzie zależeć od liczby krawędzi wypisanego grafu G' w następujący sposób. Niezerową liczbę punktów za test otrzymasz wtedy, gdy wynik będzie zgodny ze specyfikacją. W takim wypadku liczbę punktów obliczymy następująco. Niech E_y będzie liczbą krawędzi w wypisanym przez Ciebie grafie G'. Niech E_b będzie liczbą krawędzi grafu G' utworzonego przez algorytm podstawowy. Niech E_m będzie maksymalną liczbą krawędzi w grafach G' wypisanych przez wszystkich zawodników. Za taki test otrzymasz punkty wg. wzoru:

$$\begin{cases} 30\frac{E_y}{E_b} & \text{jeśli } E_y \leqslant E_b, \\ 30 + 70\frac{E_y - E_b}{E_m - E_b} & \text{jeśli } E_y > E_b. \end{cases}$$

Piotr Chrząstowski–Wachtel, Krzysztof Stencel, Tomasz Waleń
Tłumaczenie

IOI 2006, dzień pierwszy, 15.08.2006

Rozszyfrowywanie pisma Majów

Rozszyfrowywanie pisma Majów okazało się trudniejsze, niż przypuszczano. Po blisko dwóch wiekach wiemy dziś o nim niewiele. Pewien postęp zanotowano w ciągu ostatnich 30 lat.

Pismo Majów składa się z małych obrazków nazywanych glifami. Glify reprezentują głoski. Słowa Majów składają się z zestawów takich glifów dowolnie uporządkowanych.

Jednym z głównych problemów jest ustalenie porządku odczytu. Pisarze bowiem często zmieniali porządek glifów w ramach słowa, zgodnie z własnym odczuciem estetyki. W efekcie, nawet jeśli wiemy jak poszczególne glify wymawiano, nie znamy brzmienia zapisanych słów.

Archeologowie poszukują pewnego szczególnego słowa W*. Znają zestaw jego glifów, ale nie znają wszystkich porządków, w jakich można je zapisywać. Dowiedziawszy się o IOI 2006, poprosili Cię o pomoc. Dostarczą Ci zestaw* g *glifów, z których składa się słowo* W *oraz ciąg* S *wszystkich glifów (w oryginalnym porządku) badanego tekstu. Pomóż im zliczyć wszystkie potencjalne wystąpienia słowa* W *w* S*.*

Zadanie

Napisz program, który dla zadanych glifów słowa W *oraz ciągu* S*, wyznaczy liczbę wszystkich potencjalnie możliwych wystąpień* W *w ciągu* S*, to znaczy wszystkich podsłów ciągu* S *złożonych z kolejnych* g *glifów, które są permutacją glifów słowa* W*.*

Ograniczenia

$1 \leqslant g \leqslant 3\,000$ *— liczba glifów w słowie* W*,*
$g \leqslant |S| \leqslant 3\,000\,000$*, gdzie* $|S|$ *jest liczbą glifów w ciągu* S*.*

Wejście

Twój program powinien przeczytać dane z pliku `writing.in`*.*

`writing.in`	**OPIS**

`4 11`
`cAda`
`AbrAcadAbRa`
	WIERSZ *1: Zawiera 2 liczby całkowite oddzielone odstępem oznaczające kolejno* g *oraz* $\|S\|$*.*
	WIERSZ *2***:** *Zawiera* g *kolejnych znaków reprezentujących glify wchodzące w skład* W*. Dozwolone znaki, to* $'a' - 'z'$ *oraz* $'A' - 'Z'$*; wielkie i małe litery traktowane są jako różne.*
	WIERSZ *3***:** *Zawiera* $\|S\|$ *kolejnych glifów badanego napisu. Dozwolone znaki to* $'a' - 'z'$ *oraz* $'A' - 'Z'$*; wielkie i małe litery traktowane są jako różne.*

Wyjście

Twój program powinien zapisać następujące dane w pliku `writing.out`*.*

`writing.out`	**OPIS**
`2`	**WIERSZ** *1***:** *ma zawierać liczbę wystąpień permutacji słowa* W *w ciągu* S*.*

Sposób oceniania

50 punktów można dostać za zestaw testów, w których $g \leqslant 10$*.*

Ważna informacja dla pascalowców

Typ string ma przez domniemanie we FreePascalu 255 znaków. Jeśli chcesz korzystać z dłuższych napisów, powinieneś tuż po nagłówku programu umieścić dyrektywę kompilatora `{$H+}`*.*

IOI 2006, dzień pierwszy, 15.08.2006

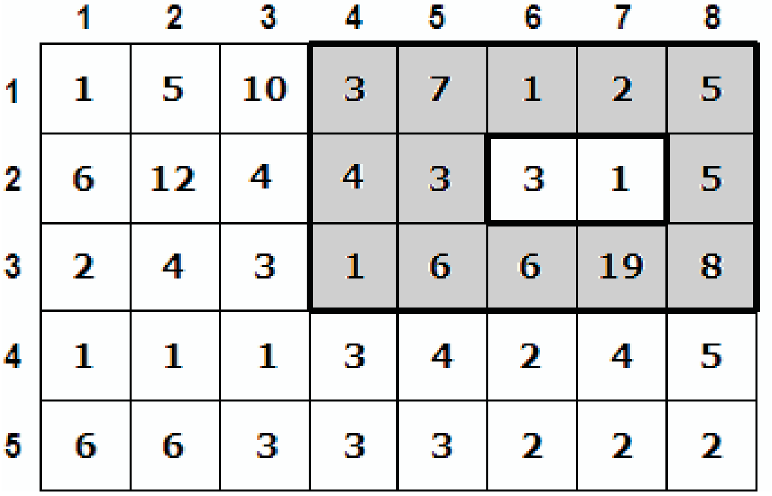
Piramida

Kapitan Żbik po wygraniu wielkiej bitwy z układem, postanowił zbudować piramidę, która uświetni to wielkie zwycięstwo, ale również będzie grobem dla dzielnych milicjantów poległych na polu chwały. Piramida ma zostać zbudowana na polu bitewnym. Jej podstawa będzie prostokątem o a kolumnach i b wierszach. We wnętrzu piramidy, na dolnym poziomie zostanie umieszczona prostokątna komora o c kolumnach i d wierszach. Znajdą się w niej prochy poległych milicjantów.

Pole bitwy jest siatką składającą się z m kolumn i n wierszy. Kapitańscy mierniczy zbadali pole bitwy i zmierzyli wysokość każdego kwadratu.

Piramida i komora muszą zostać zbudowane w ten sposób, że będą całkowicie pokrywać kwadraty siatki a ich boki będą równoległe do boków pola bitwy.

Wysokość kwadratów komory pozostaje bez zmian, natomiast pozostałe kwadraty podstawy piramidy zostaną wyrównane przez przeniesienie ziemi z wyższych kwadratów do niższych. Końcowa wysokość podstawy będzie średnią wysokością jej kwadratów (pomijając kwadraty komory). Architekci mogą dowolnie wybrać położenie komory wewnątrz piramidy, jednak mury otaczające komorę powinny mieć co najmniej 1 kwadrat grubości.



	1	2	3	4	5	6	7	8
1	1	5	10	3	7	1	2	5
2	6	12	4	4	3	3	1	5
3	2	4	3	1	6	6	19	8
4	1	1	1	3	4	2	4	5
5	6	6	3	3	3	2	2	2

Pomóż architektom wybrać najlepsze możliwe położenie piramidy i komory w jej wnętrzu — czyli takie, dla którego końcowa wysokość podstawy będzie najwyższa z możliwych.

Ilustracja przedstawia przykładowe pole bitwy; liczby wewnątrz kwadratów oznaczają ich wysokość. Szare kwadraty przedstawiają podstawę piramidy, natomiast białe kwadraty wewnątrz szarego obszaru reprezentują położenie komory. Na rysunku obok widać optymalne położenie podstawy piramidy i komory.

Zadanie

Napisz program, który dla zadanych: rozmiarów pola bitwy, piramidy i komory oraz wysokości każdego kwadratu, znajdzie położenie piramidy i komory, dla którego końcowa wysokość podstawy będzie maksymalna.

Ograniczenia

$3 \leqslant m \leqslant 1\,000$,
$3 \leqslant n \leqslant 1\,000$,
$3 \leqslant a \leqslant m$,
$3 \leqslant b \leqslant n$,
$1 \leqslant c \leqslant a-2$,
$1 \leqslant d \leqslant b-2$.
Wszystkie wysokości są liczbami całkowitymi z przedziału od 1 do 100.

Wejście

Twój program powinien czytać dane wejściowe z pliku `pyramid.in`.

`pyramid.in`	**OPIS**

`8 5 5 3 2 1`
`1 5 10 3 7 1 2 5`
`6 12 4 4 3 3 1 5`
`2 4 3 1 6 6 19 8`
`1 1 1 3 4 2 4 5`
`6 6 3 3 3 2 2 2`
	WIERSZ *1*: *Zawiera sześć liczb całkowitych pooddzielanych pojedynczymi odstępami, odpowiednio: m, n, a, b, c, d.*
	KOLEJNE n **WIERSZY:** *Każdy wiersz zawiera m liczb całkowitych, pooddzielanych pojedynczymi odstępami, które reprezentują wysokości jednego rzędu siatki. Pierwszy z tych wierszy reprezentuje górny rząd (o numerze 1). Ostatni reprezentuje dolny rząd (o numerze n). W każdym z tych wierszy m liczb reprezentuje wysokości kwadratów w kolumnach tego wiersza, rozpoczynając od kolumny numer 1.*

Wyjście

Twój program powinien zapisać następujące dane do pliku `pyramid.out`.

`pyramid.out`	**OPIS**
`4 1` `6 2`	**WIERSZ** *1*: *Powinien zawierać 2 liczby całkowite oddzielone odstępem, reprezentujące lewy górny róg podstawy piramidy. Pierwsza liczba to kolumna, druga wiersz.* **WIERSZ** *2*: *Powinien zawierać 2 liczby całkowite oddzielone odstępem, reprezentujące lewy górny róg komory. Pierwsza z liczb to kolumna, druga wiersz.*

Uwaga

Jeśli jest wiele optymalnych ułożeń, wypisz dowolne z nich.

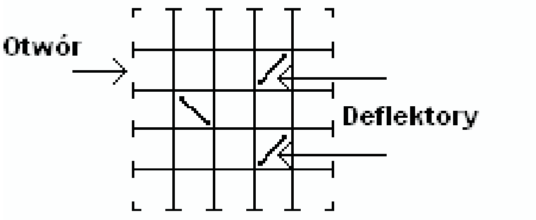
Sposób oceniania

30 punktów możesz otrzymać za zestaw testów spełniających następujące wymagania: $3 \leqslant m \leqslant 10$, $3 \leqslant n \leqslant 10$.

IOI 2006, dzień drugi, 17.08.2006

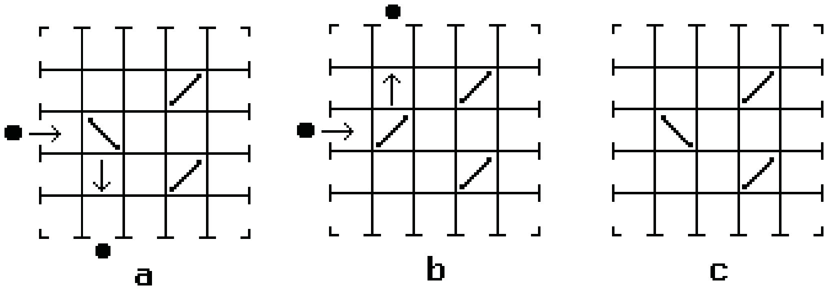
Gra w czarną skrzynkę

Rekwizytem do tej gry jest czarna kwadratowa skrzynka leżąca poziomo na stole. Każda z jej czterech ścian ma n otworów (razem jest więc 4n otworów), do których można wrzucić piłkę. Wrzucona do wnętrza piłka w końcu wyleci z jednego z 4n otworów; może to być nawet ten sam otwór, do którego ją wrzucono.



Wnętrze czarnej skrzynki można przedstawić jako siatkę $n \times n$. Otwory na jej bokach są początkami i końcami jej rzędów i kolumn. Każdy z jednostkowych kwadratów jest albo pusty, albo wypełniony deflektorem. Deflektor to urządzenie, które zmienia kierunek poruszania się piłki o 90 stopni. Spójrz na pokazany obok przykład dla skrzynki 5×5.

Piłka wrzucona do jednego z otworów porusza się po linii prostej do chwili uderzenia w deflektor lub wypadnięcia ze skrzynki. Gdy piłka uderzy w deflektor, zmienia kierunek swojego ruchu zgodnie z prawami odbicia, a deflektor zmienia swoje ustawienie na przeciwne (zmiana o 90 stopni). Poniżej pokazano przykład działania deflektora.



- *a) Piłka wrzucona do otworu trafia na deflektor i zmienia kierunek ruchu.*
- *b) Po wrzuceniu pierwszej piłki, deflektor zmienił swoje ustawienie. Kolejna piłka wrzucona do tego samego otworu uderza w ten deflektor i odbija się w przeciwnym kierunku.*
- *c) Deflektor zmienia swoje położenie przy każdym uderzeniu.*

Deflektor uderzony piłką wydaje dźwięk. Liczba uderzeń piłki w deflektory może być ustalona na podstawie liczby tych dźwięków. Można wykazać, że piłka zawsze wypadnie ze skrzynki. Skrzynka ma guzik, który przywraca początkowe ustawienie wszystkich deflektorów, oraz drugi guzik, który odwraca ustawienia wszystkich deflektorów.

Zadanie

Otrzymasz interfejs do komunikacji z 15 skrzynkami poprzez bibliotekę Pascala/C/C++. Twoim zadaniem jest jak najdokładniej ustalić zawartość wnętrza każdej z tych skrzynek i przedstawić plik z jego opisem. Otrzymasz także możliwość testowania na skrzynkach zdefiniowanych przez Ciebie.

Ograniczenia

$1 \leqslant n \leqslant 30$.

Wyjście

Twoim zadaniem jest przedstawić do oceny plik z następującymi danymi dla każdej z 15 skrzynek.

`BlackboxK.out`	**OPIS**

`#FILE blackbox K`
`.....`
`.../.`
`.\...`
`.../.`
`.?.?.`
	WIERSZ *1***:** *Nagłówek pliku, który musi zawierać następującą frazę:*
	`#FILE blackbox K`
	gdzie K (z przedziału 1 . . . 15) odpowiada opisywanej skrzynce.
	n **WIERSZY:** *Każdy z nich jest opisem jednego rzędu skrzynki (od rzędu najwyższego do najniższego). Każdy z tych wierszy ma zawierać dokładnie n znaków. Każdy znak jest opisem zawartości jednej kolumny (kolumny są opisywane od lewej do prawej):*
	• `'.'` *(kropka) oznacza, że kwadrat jest pusty.*
	• `'/'` *oznacza, że kwadrat zawiera deflektor o początkowym położeniu* `'/'`
	• `'\'` *oznacza, że kwadrat zawiera deflektor o początkowym położeniu* `'\'`
	• `'?'` *oznacza, że nie byłeś w stanie ustalić zawartości tego kwadratu.*

Biblioteka

Otrzymasz bibliotekę, która udostępnia następujące funkcje:

FUNKCJA	**OPIS**

PASCAL
`function Initialize(box: integer): integer;`
C/C++
`int Initialize(int box);`
	Inicjuje bibliotekę. Należy ją wywołać jeden raz na początku programu. Jej wynikiem jest n — liczba otworów na każdym boku skrzynki. Parametr box musi być liczbą całkowitą z przedziału od 1 do 15 (wtedy wskazuje numer badanej skrzynki) lub zerem (co oznacza, że będziesz badał skrzynkę stworzoną przez siebie).

PASCAL
`function throwBall(holeIn, sideIn: integer;`
`   var holeOut, sideOut: integer): longint;`
C
`int throwBall(int holeIn, int sideIn,`
`       int *holeOut, int *sideOut);`
C++
`int throwBall(int holeIn, int sideIn,`
`       int &holeOut, int &sideOut);`
	Powoduje wrzucenie piłki do skrzynki poprzez otwór nr `holeIn` *w ścianie* `sideIn`*. Ściany są ponumerowane następująco: 1 — góra, 2 — prawa, 3 — dół, 4 — lewa. Otwory są ponumerowane od 1 do n na każdej ścianie, od lewej do prawej albo od góry do dołu. W parametrach wyjściowych* `holeOut` *i* `sideOut` *otrzymasz numer otworu i ściany, skąd wypadnie piłka. Wynikiem funkcji throwBall jest liczba dźwięków wydanych przez uderzone deflektory.*

PASCAL
`procedure ResetBox;`
C/C++
`void ResetBox();`
	Przywraca pierwotne ustawienie każdego deflektora w skrzynce.

PASCAL
`procedure ToggleDeflectors;`
C/C++
`void ToggleDeflectors();`
	Zmienia na przeciwne ustawienie każdego deflektora w skrzynce.

PASCAL
`procedure Finalize;`
C/C++
`void Finalize();`
	Łagodnie kończy interakcję ze skrzynką. Należy ją wywołać na końcu programu.

Aby Twój program mógł współpracować z biblioteką, wykonaj następujące czynności:

- **FreePascal:** *W katalogu zadania znajdziesz pliki* `pbblib.o` *i* `pbblib.ppu`*. Aby móc z nich korzystać, umieść w programie następującą frazę:*

  ```
  uses pbblib;
  ```

  Plik `pblackbox.pas` *jest przykładem, jak korzystać z tej biblioteki.*

- **C:** *W katalogu zadania znajdziesz pliki* `cbblib.o` *i* `cbblib.h`. *Aby móc z nich korzystać, umieść w programie następującą frazę:*

  ```
  #include "cbblib.h"
  ```

  Plik `cblackbox.c` *jest przykładem, jak korzystać z tej biblioteki. Aby skompilować swój program, użyj następującego polecenia:*

  ```
  gcc -o yourprogram cbblib.o yourprogram.c
  ```

- **C++:** *W katalogu zadania znajdziesz pliki* `cppbblib.o` *i* `cppbblib.h`. *Aby móc z nich korzystać, umieść w programie następującą frazę:*

  ```
  #include "cppbblib.h"
  ```

  Plik `cppblackbox.cpp` *jest przykładem, jak korzystać z tej biblioteki. Aby skompilować swój program, użyj następującego polecenia:*

  ```
  g++ -o yourprogram cppbblib.o yourprogram.cpp
  ```

UWAGA: *W każdej chwili może działać tylko jeden program korzystający z biblioteki.*

Przykładowa interakcja

Oto przykładowa interakcja dla skrzynki z poprzedniego rysunku:

WYWOŁANIE	**WYNIKI ZWRÓCONE PRZEZ FUNKCJĘ**
`Initialize(0);`	*Zakładamy, że użyto skrzynki z poprzedniego rysunku. Wynikiem jest 5, co oznacza, że n = 5.*

PASCAL
`throwBall(3,4,holeOut,sideOut);`
C
`throwBall(3,4,&holeOut,&sideOut);`
C++
`throwBall(3,4,holeOut,sideOut);`
	Wrzucamy piłkę do otworu nr 3 (trzeci od góry) na ścianie lewej. Wynikiem wywołania jest 1, co oznacza, że piłka uderzyła jeden raz w deflektor. Po zakończeniu wywołania zmienna `holeOut` *będzie równa 2 a zmienna* `sideOut` *będzie równa 3. Oznacza to, że piłka wypadła z otworu 2 (drugiego od lewej) na dolnej ścianie skrzynki.*

Eksperymentowanie

Jeśli wywołasz funkcję `Initialize` *z argumentem 0, biblioteka odczyta opis skrzynki z pliku* `blackbox.in`. *Dzięki temu możesz poeksperymentować z biblioteką. Oto opis formatu pliku* `blackbox.in`.

`blackbox.in`	**OPIS**
`5`	**WIERSZ** *1***:** *Zawiera n — liczbę otworów w jednej ścianie.*
`3`	**WIERSZ** *2***:** *Zawiera liczbę całkowitą wskazującą liczbę deflektorów w skrzynce.*

`2 3 \`
`4 2 /`
`4 4 /`
	NASTĘPNE WIERSZE: *Każdy z deflektorów jest opisany przez jeden wiersz. Każdy wiersz zawiera dwie liczby całkowite oddzielone pojedynczym odstępem, które wskazują odpowiednio kolumnę i rząd deflektora, oraz znak oddzielony pojedynczym odstępem od drugiej liczby. Ten znak określa położenie początkowe deflektora. Tym znakiem jest* `'/'` *albo* `'\'`.

UWAGA: *Powyższy plik* `blackbox.in` *opisuje skrzynkę z rysunku na górze pierwszej strony.*

Komunikaty o błędach

Jeśli pojawią się jakieś anomalie, biblioteka wypisze na standardowe wyjście komunikat o błędzie. W poniższej tabeli wymieniono możliwe komunikaty o błędach.

KOMUNIKAT	**ZNACZENIE**
`ERR 1 More than one app`	*Tylko jeden program może w danej chwili komunikować się ze skrzynkami. Zamknij wszystkie programy i w każdej chwili miej uruchomiony tylko jeden.*
`ERR 2 Invalid box`	*Podany przez Ciebie numer skrzynki leży poza przedziałem [0,15].*
`ERR 3 Invalid deflector`	*Plik* `blackbox.in` *ma deflektor na niepoprawnej pozycji.*
`ERR 4 Invalid symbol`	*Plik* `blackbox.in` *zawiera niedozwolony znak.*
`ERR 5 Invalid size`	*Plik* `blackbox.in` *zawiera błędny rozmiar skrzynki.*
`ERR 6 Invalid input hole`	*Podany numer ściany lub otworu jest niepoprawny.*
`ERR 7 ALARM`	*Zawołaj kogoś z obsługi technicznej.*

Sposób oceniania

Dla każdej skrzynki musisz wygenerować plik tekstowy z jak najdokładniejszym opisem jej wnętrza. Dla każdej skrzynki:

- *Jeśli w zgłoszonym przez Ciebie pliku będzie co najmniej jeden znak* `'.'`, `'/'` *albo* `'\'` *na niewłaściwej pozycji, dostaniesz zero punktów za ten test.*
- *Niech* B_m *będzie maksymalną liczbą odgadniętych pozycji pośród wszystkich poprawnych zgłoszeń. Niech* B_y *będzie liczbą pozycji poprawnie odgadniętych przez Ciebie. Procent punktów otrzymanych przez Ciebie za ten test będzie wówczas wynosić:*

$$100\frac{B_y}{B_m}$$

UWAGA: *Rozwiązanie wzorcowe tego zadania jest w stanie dla dowolnej skrzynki poprawnie ustalić 100% początkowych pozycji w czasie mniejszym niż 8 minut.*

Piotr Chrząstowski–Wachtel, Krzysztof Stencel, Tomasz Waleń
Tłumaczenie

IOI 2006, dzień drugi, 17.08.2006

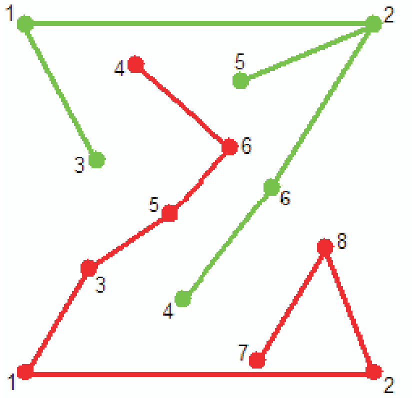
Łączenie punktów

„Łączenie punktów” to gra dla jednej osoby. Aby w nią zagrać, wybieramy dwie liczby całkowite większe niż 2; nazwijmy je g i r. Następnie rysujemy 4 punkty tworzące rogi kwadratu — dwa górne zaznaczamy na zielono, dwa dolne na czerwono. W końcu dodajemy zielone i czerwone punkty wewnątrz kwadratu, przy czym żadne 3 punkty (wliczając to cztery początkowe) nie mogą leżeć na jednej prostej. Tę czynność wykonujemy aż do momentu, gdy łączna liczba zielonych punktów osiągnie g, a czerwonych r.

Gdy plansza jest już gotowa, możemy zacząć łączenie punktów. Każde dwa punkty mogą zostać połączone odcinkiem, jeśli:

- *dwa łączone punkty są tego samego koloru i*
- *odcinek je łączący nie przecina żadnego wcześniej narysowanego odcinka (natomiast odcinki mogą stykać się końcami).*

Dwa punkty u i v należą do jednej składowej, jeśli można przejść z punktu u do v po narysowanych odcinkach. Wygrywasz grę, jeśli uda ci się połączyć wszystkie zielone punkty w jedną składową za pomocą $g-1$ odcinków, a wszystkie czerwone punkty w drugą składową za pomocą $r-1$ odcinków. Można udowodnić, że jeśli punkty wybrano wg powyższych zasad, to wygranie gry jest zawsze możliwe.



Otrzymasz kwadratową planszę, na której umieszczono g zielonych i r czerwonych punktów, reprezentowanych przez pary liczb całkowitych (x_i, y_i). Będziemy numerować zielone punkty liczbami od 1 do g. Górny lewy punkt ma współrzędne $(0, s)$ i numer 1, górny prawy ma współrzędne (s, s) i numer 2, zaś punkty wewnątrz kwadratu mają numery od 3 do g. Czerwone punkty będziemy numerować liczbami od 1 do r, dolny lewy punkt ma współrzędne $(0, 0)$ i numer 1, dolny prawy punkt ma współrzędne $(s, 0)$ i numer 2, zaś punkty wewnątrz kwadratu mają numery od 3 do r.

Rysunek przedstawia przykładową grę. Wszystkie zielone punkty zostały połączone, tworząc jedną składową. Również wszystkie czerwone punkty po połączeniu tworzą jedną składową.

Jak łatwo zaobserwować, żadne trzy punkty nie są współliniowe i żadne dwa odcinki nie przecinają się.

Zadanie

Napisz program, który dla zadanych g współrzędnych zielonych punktów, i r współrzędnych czerwonych punktów obliczy, jak narysować $(g-1)$ zielonych odcinków i $(r-1)$ czerwonych, tak by wszystkie zielone punkty znalazły się w jednej składowej, wszystkie czerwone w drugiej, oraz żadne dwa odcinki nie przecinały się.

Ograniczenia

$3 \leqslant g \leqslant 50\,000$ — *liczba zielonych punktów,*
$3 \leqslant r \leqslant 50\,000$ — *liczba czerwonych punktów,*
$0 \leqslant s \leqslant 200\,000\,000$.

Wejście

Twój program powinien czytać dane z pliku `points.in`*.*

`points.in`	**OPIS**

`6`
`0 1000`
`1000 1000`
`203 601`
`449 212`
`620 837`
`708 537`
`8`
`0 0`
`1000 0`
`185 300`
`314 888`
`416 458`
`614 622`
`683 95`
`838 400`
	WIERSZ 1**:** *Zawiera liczbę całkowitą* g*.*
	NASTĘPNE g **WIERSZY:** *Każdy wiersz zawiera dwie liczby całkowite oddzielone pojedynczym odstępem, które reprezentują współrzędne* x_i *i* y_i *każdego z* g *zielonych punktów, poczynając od* 1 *do* g*.*
	WIERSZ $g+2$**:** *Zawiera liczbę całkowitą* r*.*
	NASTĘPNE r **WIERSZY:** *Każdy wiersz zawiera dwie liczby całkowite oddzielone pojedynczym odstępem, które reprezentują współrzędne* x_i *i* y_i *każdego z* r *czerwonych punktów, poczynając od* 1 *do* r*.*

Wyjście

Twój program powinien zapisać następujące dane do pliku `points.out`*.*

`points.out`	**OPIS**

`1 3 g`
`3 1 r`
`3 5 r`
`4 6 r`
`6 5 r`
`4 6 g`
`1 2 g`
`1 2 r`
`5 2 g`
`2 6 g`
`7 8 r`
`8 2 r`
	Plik wynikowy powinien zawierać $(g-1)+(r-1)$ *wierszy, po jednym dla każdej narysowanej linii łączącej punkty.*
	Każdy wiersz powinien zawierać 3 *wartości pooddzielane pojedynczymi odstępami: dwie liczby całkowite i znak. Dwie liczby całkowite reprezentują numery punktów, które mają zostać połączone. Znak musi mieć wartość* `g`*, jeśli punkty są zielone, lub* `r`*, jeśli są czerwone.*
	
	Kolejność odcinków w pliku nie ma znaczenia, jak również nie jest istotna kolejność końców odcinka.

Sposób oceniania

Możesz dostać do 35 *punktów za zestaw testów spełniających ograniczenia:* $3 \leqslant g \leqslant 20$ *oraz* $3 \leqslant r \leqslant 20$

Piotr Chrząstowski–Wachtel, Krzysztof Stencel, Tomasz Waleń
Tłumaczenie

IOI 2006, dzień drugi, 17.08.2006

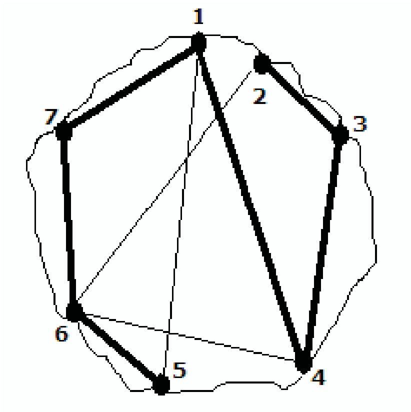
Dolina Meksyku

Miasto Meksyk leży w pięknej dolinie znanej jako Dolina Meksyku, która wieki temu była jeziorem. Na początku XIV w. azteccy przywódcy religijni postanowili zasypać środek jeziora, aby zbudować tam kapitol dla całego imperium. Dziś jezioro jest całkowicie zasypane.

Zanim pojawili się Aztekowie, naokoło jeziora funkcjonowało c miast. Niektóre z nich miały porozumienia handlowe. Fregaty handlowe kursowały między tymi miastami, które takie porozumienia miały.

Przywódcy miast w końcu doszli do porozumienia, w ramach którego zamierzyli uporządkować szlaki handlowe. Opracowali trasę, która łączyła wszystkie miasta wokół jeziora. Trasa ta miała następujące właściwości:

- *trasa zaczynała się w jednym z miast i po kolei wiodła przez wszystkie miasta, docierając w końcu do innego miasta, niż jej początek,*
- *każde miasto pojawiało się na trasie tylko raz,*
- *każda para kolejno odwiedzanych miast miała porozumienie handlowe,*
- *każda para kolejno odwiedzanych miast była połączona szlakiem handlowym w kształcie odcinka,*
- *aby zapobiec zderzeniom, żadne dwa odcinki trasy nie przecinały się.*



Na rysunku widzimy jezioro i miasta wokół niego. Cięciwy reprezentują porozumienia handlowe. Pogrubione cięciwy określają trasę zaczynającą się w mieście 2, a kończącą w mieście 5.
Nie byłoby właściwe zacząć trasę np. w mieście 2, i przedłużyć ją kolejno do 6 — 5 — 1, gdyż trasa taka przecięłaby siebie samą.
Miasta są ponumerowane od 1 do c, poczynając od miasta na górze. Numery zwiększają się w kierunku wskazówek zegara.

Zadanie

Napisz program, który dla zadanej liczby c miast oraz listy porozumień handlowych między nimi, skonstruuje trasę spełniającą powyższe wymagania.

Ograniczenia

$3 \leqslant c \leqslant 1\,000$ — liczba miast wokół jeziora.

Wejście

Twój program powinien przeczytać dane z pliku `mexico.in`.

`mexico.in`	**OPIS**
7 9 1 4 5 1 1 7 5 6 2 3 3 4 2 6 4 6 6 7	**WIERSZ** *1*: *zawiera liczbę całkowitą c* **WIERSZ** *2*: *zawiera liczbę porozumień handlowych* **KOLEJNE WIERSZE:** *reprezentują porozumienia. Każdy taki wiersz zawiera dwie liczby całkowite oddzielone odstępem, reprezentujące miasta, które mają porozumienie. Tych wierszy jest tyle, ile jest porozumień handlowych.*

Wyjście

Twój program powinien wypisać w pliku `mexico.out` *następujące dane.*

`mexico.out`	**OPIS**

`2`
`3`
`4`
`1`
`7`
`6`
`5`
	Jeśli można utworzyć żądaną trasę, należy wypisać c *wierszy. W kolejnych wierszach powinny się znaleźć numery kolejnych miast tworzących trasę w porządku, w którym powinny być odwiedzane. Jeśli utworzenie takiej trasy jest niemożliwe, należy umieścić w jedynym wierszu liczbę* -1.

UWAGA: *Jeśli istnieje więcej niż jedna trasa spełniająca warunki zadania, możesz wypisać dowolną z nich.*

Sposób oceniania

Możesz dostać do 40 punktów za zestaw testów spełniających następujące ograniczenia:
$3 \leqslant c \leqslant 20$.

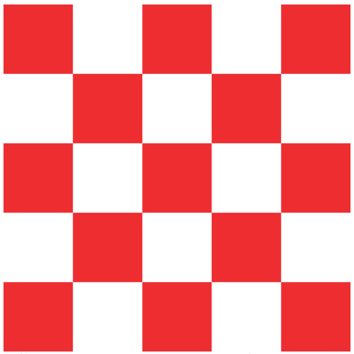
XIX Międzynarodowa Olimpiada Informatyczna

Zagrzeb, Chorwacja 2007

IOI 2007, dzień pierwszy, 17.08.2007

Obcy

Mirko jest znanym fanem znaków na zbożu, czyli geometrycznych znaków utworzonych ze sprasowanych łanów zbóż. Takie znaki przypisywane są Obcym z innego układu. Pewnego letniego wieczora wpadł na pomysł utworzenia własnych znaków na polu swojej babci. Jako wielki patriota Mirko postanowił utworzyć kształt znany z herbu Chorwacji, to jest szachownicę 5×5 *z 13 czerwonymi polami i 12 białymi.*

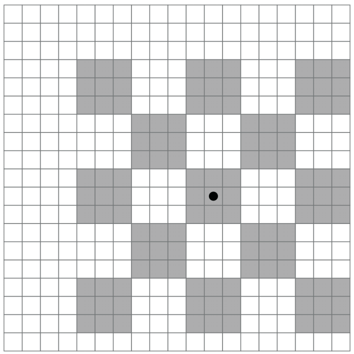


Szachownica z herbu Chorwacji

Pole babci jest podzielone na $N \times N$ *jednostkowych kwadratów. Kwadrat w lewym dolnym rogu pola ma współrzędne* $(1,1)$*, a kwadrat w prawym górnym rogu —* (N,N)*.*

Mirko zdecydował, że będzie prasował zboże należące do czerwonych kwadratów szachownicy, a białe kwadraty pozostawi nietknięte.

Mirko wybrał **nieparzystą liczbę całkowitą** $M \geqslant 3$ *i utworzył wzór, w taki sposób, że każdy kwadrat szachownicy składa się z* $M \times M$ *jednostkowych kwadratów pola oraz cała szachownica całkowicie mieści się wewnątrz pola.*



Przykładowe pole z wzorem utworzonym przez Mirko dla $N = 19$ *i* $M = 3$*.*
Kwadraty sprasowanego zboża są zrobione na szaro. Środek wzoru ma współrzędne $(12,9)$ *i jest zaznaczony czarną kropką.*

Podczas gdy Mirko smacznie spał, wzór przykuł uwagę Prawdziwych Obcych, zwanych PO. Obcy unoszą się gdzieś wysoko w swoim statku kosmicznym i mogą badać wzór Mirka za pomocą specjalnego Urządzenia*. To Urządzenie może jedynie* **stwierdzić, czy konkretny jednostkowy kwadrat pola został sprasowany, czy nie**.

Obcy znaleźli **jeden sprasowany kwadrat jednostkowy** *i teraz chcą znaleźć* **środek wzoru** *(czyli środkowy kwadrat jednostkowy), tak by móc w pełni podziwiać piękno tego artystycznego tworu. Niestety* **nie znają wartości** M *opisującej rozmiar pojedynczego pola szachownicy.*

Zadanie

Napisz program, który dla zadanego N $(15 \leqslant N \leqslant 2\,000\,000\,000)$ *czyli rozmiaru pola oraz współrzędnych* (X_0,Y_0) *sprasowanego kwadratu, znajdzie za pomocą Urządzenia środek wzoru utworzonego przez Mirko. Urządzenie jest bardzo delikatne i możesz go użyć co najwyżej 300 razy w jednym teście.*

Interakcja

To jest zadanie interaktywne. Twój program powinien wysyłać komendy do Urządzenia przez standardowe wyjście i wczytywać odpowiedź ze standardowego wejścia.

- *Na początku program powinien wczytać ze standardowego wejścia trzy liczby całkowite N, X_0 i Y_0 oddzielone pojedynczym odstępem. Liczba N to rozmiar babcinego pola, a (X_0,Y_0) to współrzędne jednego ze sprasowanych kwadratów jednostkowych.*
- *Aby sprawdzić stan kwadratu (X,Y) należy wypisać wiersz* `examine X Y` *na standardowym wyjściu. Jeśli współrzędne (X,Y) nie znajdują się wewnątrz pola (to jest jeden z warunków $1 \leqslant X \leqslant N$, $1 \leqslant Y \leqslant N$ nie jest spełniony), lub gdy liczba odwołań do Urządzenia przekroczy 300, twój program otrzyma 0 punktów za ten test.*
- *Odpowiedź Urządzenia to wiersz zawierający jedno słowo* `true`*, jeśli kwadrat (X,Y) został sprasowany, lub* `false` *w przeciwnym przypadku.*
- *Gdy twój program znajdzie środek wzoru, program powinien wypisać na standardowy wyjście wiersz* `solution` *X_C Y_C, gdzie (X_C,Y_C) są współrzędnymi środkowego kwadratu jednostkowego wzoru. Wykonanie twojego programu po tej komendzie zostanie natychmiast zakończone.*

Pamiętaj, że twój program musi wykonywać operację **flush na standardowym wyjściu** *po każdej komendzie dla Urządzenia. Na stronie konkursu będą dostępne przykłady, jak to zrobić.*

Przykłady programów

Przykłady programów we wszystkich trzech językach programowania są dostępne na stronie „Tasks" systemu konkursowego. Jedynym celem przykładów jest pokazanie prawidłowej komunikacji z Urządzeniem. Nie są to poprawne rozwiązania i nie otrzymują pełnej punktacji.

Ocena rozwiązań

W testach wartych w sumie 40 punktów, rozmiar M każdego z kwadratów Mirko nie przekracza 100. Każdy test ma unikalną poprawną odpowiedź, niezależną od pytań zadawanych przez twój program.

Przykład

W następującym przykładzie komendy są podane w lewej kolumnie, wiersz po wierszu. Odpowiedź Urządzenia jest podana w drugiej kolumnie w odpowiednim wierszu.

Wyjście (polecenia)	**Wejście (odpowiedzi)**
	`19 7 4`
`examine 11 2`	`true`
`examine 2 5`	`false`
`examine 9 14`	`false`
`examine 18 3`	`true`
`solution 12 9`	

Testowanie

Podczas konkursu są trzy sposoby testowania twojego rozwiązania.

- *Pierwszy sposób to ręczna symulacja (przez ciebie) Urządzenia, które komunikuje się z twoim programem.*
- *Drugi sposób to napisanie programu, który symuluje Urządzenie. W celu połączenia twojego rozwiązania z Urządzeniem, które zaprogramowałeś, możesz użyć narzędzia o nazwie „connect", które można pobrać z systemu konkursowego. Aby to uczynić, należy wydać z konsoli polecenie, takie jak:*

  ```
  ./connect ./solution ./device
  ```

  (zastępując słowa „solution" i „device" nazwami twoich programów).
  Dodatkowe argumenty polecenia „connect" będą przekazane do programu „device".
- *Trzeci sposób to wykorzystanie funkcji TEST systemu konkursowego do automatycznego uruchomienia twojego rozwiązania na przygotowanych testach. W testach (używanych w tej opcji) rozmiar pola N jest ograniczony do 100.*

Test powinien zawierać trzy wiersze:

- *pierwszy wiersz zawiera rozmiar pola N i rozmiar M kwadratu szachownicy;*
- *drugi wiersz zawiera współrzędne X_0 i Y_0 pewnego jednostkowego kwadratu, który został sprasowany; te współrzędne zostaną przekazane do twojego programu;*

- *trzeci wiersz zawiera współrzędne* X_C *i* Y_C *środka wzoru.*

System konkursowy przekaże ci szczegółowy zapis wykonania, łącznie z informacjami o błędach, jeśli:

- N *nie spełnia zadanych ograniczeń,*
- M *nie jest nieparzystą liczbą całkowitą większą lub równą 3,*
- *wzór nie mieści się w polu,*
- *kwadrat* (X_0, Y_0) *nie jest sprasowany.*

Poniżej podany jest przykład poprawnego pliku wejściowego dla funkcji TEST. Przykład odpowiada rysunkowi z pierwszej strony.

```
19 3
7 4
12 9
```

IOI 2007, dzień pierwszy, 17.08.2007

Powódź

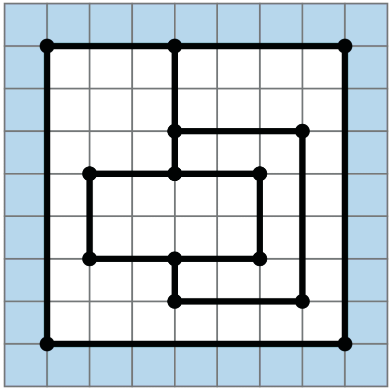
W roku 1964 Zagrzeb dotknęła katastrofalna powódź. Kiedy woda uderzyła w miejskie mury, wiele budynków zostało kompletnie zniszczonych. W tym zadaniu, mając uproszczony plan miasta przed powodzią, należy stwierdzić, które mury nie zostały zniszczone przez wodę.

Plan miasta składa się z N punktów zaznaczonych na siatce danej w kartezjańskim układzie współrzędnych i W murów. **Każdy mur łączy parę punktów i nie przechodzi przez żaden inny punkt.** *Plan miasta ma następujące dodatkowe własności:*

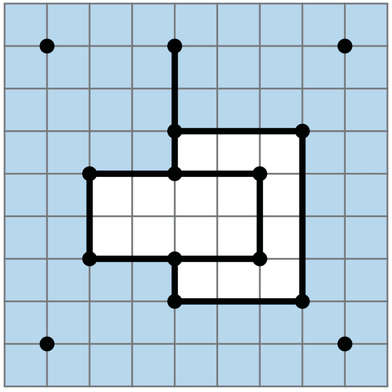
- *Żadne dwa mury nie przecinają się i nie nachodzą na siebie, ale mogą mieć wspólne końce.*
- *Każdy mur jest równoległy do osi poziomej lub do osi pionowej układu współrzędnych.*

Początkowo cały obszar zaznaczony na planie jest suchy. W chwili zero woda zalewa momentalnie obszar nieograniczony murami (przestrzeń na zewnątrz murów). Dokładnie po jednej godzinie każdy mur z wodą po jednej stronie i powietrzem po drugiej rozpada się pod wpływem ciśnienia wody. Wówczas woda zalewa nowy obszar nieograniczony stojącymi jeszcze murami. Teraz znowu mogą pojawić się mury z wodą po jednej i powietrzem po drugiej stronie. Po kolejnej godzinie takie mury także ulegają zniszczeniu i woda zalewa nowe obszary. Ten proces powtarza się, aż cały obszar na planie zostanie zalany.

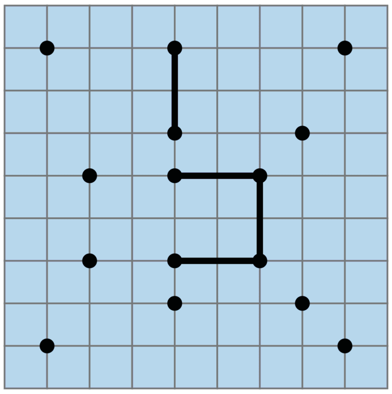
Przykładowy proces zalewania miasta został przedstawiony na rysunkach poniżej.



Stan w chwili zero. Zacienione kwadraciki odpowiadają zalanemu obszarowi, natomiast białe kwadraty reprezentują miejsca suche (powietrze).



Stan po godzinie.



Stan po dwóch godzinach. Woda zalała cały obszar i 4 mury nie zostały zniszczone.

Zadanie

Napisz program, który mając dane współrzędne N punktów i opisy W murów łączących te punkty, określi, które mury pozostaną niezniszczone po powodzi.

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę całkowitą N ($2 \leqslant N \leqslant 100\,000$) — liczbę punktów na planie.

Każdy z następnych N wierszy zawiera dwie liczby całkowite X i Y ($0 \leqslant X, Y \leqslant 1\,000\,000$) — współrzędne jednego punktu. Punkty są ponumerowane od 1 do N w kolejności, w jakiej pojawiają się na wejściu. Żadne dwa punkty nie mają takich samych współrzędnych.

Następny wiersz zawiera jedną liczbę całkowitą W ($1 \leqslant W \leqslant 2N$) — liczbę murów.

Każdy z następnych W wierszy zawiera dwie różne liczby całkowite A i B ($1 \leqslant A \leqslant N$, $1 \leqslant B \leqslant N$), oznaczające, że przed powodzią istniał mur łączący punkty A i B. Mury są ponumerowane od 1 do W, w kolejności ich podania na wejściu.

Wyjście

W pierwszym wierszu wyjścia powinna zostać wypisana jedna liczba całkowita K — liczba niezniszczonych murów po powodzi. Następne K wierszy powinno zawierać numery niezniszczonych murów, po jednym w wierszu. Numery te mogą zostać wypisane w dowolnej kolejności.

Ocena rozwiązań

W testach o łącznej wartości 40 punktów, liczba wszystkich współrzędnych nie przekracza 500.
W tych samych testach i dodatkowych testach za 15 punktów, liczba punktów na planie nie przekracza 500.

Informacja zwrotna po zgłoszeniu rozwiązań

Podczas zawodów możesz wybrać do 10 zgłoszeń dla tego zadania, które zostaną ocenione (tak szybko, jak to tylko możliwe) na części rzeczywistych danych wejściowych. Po ocenieniu zgłoszenia w systemie konkursowym będzie dostępne zbiorcze podsumowanie wyników oceny.

Przykład

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

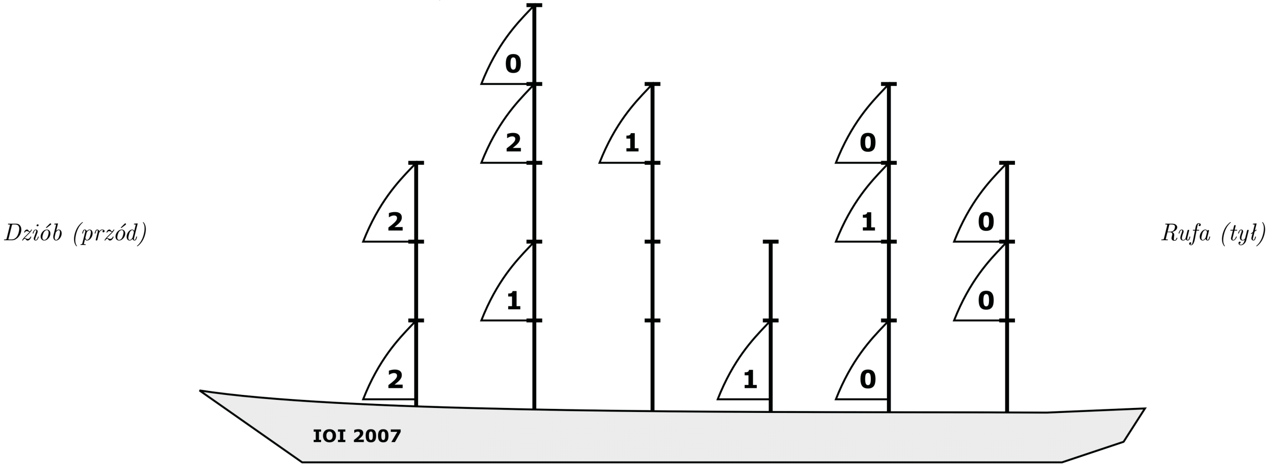
Ten przykład odpowiada rysunkowi z treści zadania.

IOI 2007, dzień pierwszy, 17.08.2007

Żagle

Młodzież z informatycznego klubu żeglarskiego „Piraci” buduje żaglowiec. Żaglowiec ma N masztów, każdy podzielony na segmenty długości 1 — wysokość masztu jest równa liczbie segmentów, na jakie jest on podzielony. Na każdym z masztów zawieszono pewną liczbę żagli, przy czym każdy żagiel zajmuje dokładnie jeden segment. Żagle zawieszone na danym maszcie mogą zajmować dowolne segmenty, przy czym każdy segment może być zajęty przez co najwyżej jeden żagiel.

Różne rozmieszczenia żagli dają różny ciąg, gdy wiatr wieje w żagle. Żagle znajdujące się z przodu, przed innymi żaglami na tej samej wysokości, dostają mniej wiatru i słabiej ciągną. Dla każdego żagla definiujemy jego **zasłonięcie** *jako łączną liczbę żagli, które znajdują się* **z tyłu za** *nim i* **na tej samej wysokości**. *Zwróć uwagę, że określenia „przed” i „za” odnoszą się do ustawienia żaglowca; na poniższym rysunku „przed” oznacza na lewo, a „za” oznacza na prawo.* **Łączne zasłonięcie** *układu żagli to suma zasłonięć poszczególnych żagli.*



Ten żaglowiec ma 6 masztów o wysokościach 3, 5, 4, 2, 4 i 3, patrząc od dziobu żaglowca (po lewej) do rufy. Pokazany układ żagli ma łączne zasłonięcie 10. Zasłonięcia poszczególnych żagli są podane na rysunku wewnątrz żagli.

Zadanie

Napisz program, który na podstawie wysokości masztów oraz liczby żagli na każdym z N masztów, wyznaczy **najmniejsze** *możliwe łączne zasłonięcie.*

Wejście

Pierwszy wiersz wejścia zawiera liczbę całkowitą N $(2 \leqslant N \leqslant 100\,000)$ — liczbę masztów.
Każdy z kolejnych N wierszy zawiera dwie liczby całkowite H i K $(1 \leqslant H \leqslant 100\,000,\ 1 \leqslant K \leqslant H)$ — wysokość odpowiedniego masztu i liczbę zawieszonych na nim żagli. Maszty są podane w kolejności od dziobu do rufy żaglowca.

Wyjście

Na wyjściu powinna znaleźć się jedna liczba całkowita: najmniejsze możliwe łączne zasłonięcie.

Uwaga: *Do obliczenia i wypisania wyniku użyj 64–bitowych liczb całkowitych (*`long long` *w C/C++,* `int64` *w Pascalu).*

Ocena rozwiązań

Testy warte łącznie 25 punktów będą spełniać dodatkowe ograniczenie: w każdym z tych testów liczba możliwych rozmieszczeń żagli nie przekroczy $1\,000\,000$.

Przykład

Wejście

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

Wyjście

```
10
```

Ten przykład odpowiada rysunkowi z poprzedniej strony.

Krzysztof Diks, Paweł Idziak, Przemysława Kanarek, Marcin Kubica, Tomasz Waleń
Tłumaczenie

IOI 2007, dzień drugi, 19.08.2007

Górnicy

Grupa górników pracuje w **dwóch** *kopalniach. Górnicy jako ludzie ciężkiej pracy potrzebują jedzenia, aby wydajnie pracować. Za każdym razem, gdy do kopalni przychodzi dostawa posiłków, jej górnicy produkują pewną ilość węgla. Posiłki dostarczane do kopalń są jednego z trzech rodzajów — składają się z: mięsa, ryb albo chleba. Wszystkie posiłki w jednej dostawie są tego samego rodzaju.*

Górnicy bardzo cenią sobie urozmaicone posiłki i pracują wydajniej, gdy ich dieta jest różnorodna. Dokładniej, ich wydajność **zależy od bieżącej dostawy i dwóch poprzednich** *(lub mniejszej liczby dostaw, jeśli nie było ich jeszcze tyle) w następujący sposób:*

- *jeśli wszystkie dostawy zawierały posiłki tego samego typu, to górnicy produkują jedną jednostkę węgla,*
- *jeśli w dostawach były posiłki dwóch różnych rodzajów, to górnicy produkują dwie jednostki węgla,*
- *jeśli w dostawach były posiłki trzech rodzajów, to górnicy produkują trzy jednostki węgla.*

Znamy z góry rodzaje posiłków w dostawach oraz kolejność, w jakiej mają być wysyłane. Możemy wpłynąć na efektywność produkcji kopalń poprzez podjęcie decyzji, którą dostawę skierować do której kopalni. Dostaw nie można dzielić — każda musi trafić w całości do jednej kopalni. Kopalnie nie muszą otrzymać tej samej liczby dostaw (dopuszczalne jest nawet przesłanie wszystkich dostaw do jednej kopalni).

Zadanie

Twój program otrzyma listę z rodzajami posiłków w kolejnych dostawach. Napisz program, który wyznaczy **maksymalną sumaryczną ilość węgla**, *jaką można wyprodukować (razem, w obu kopalniach), decydując odpowiednio, które dostawy skierować do pierwszej kopalni, a które do drugiej.*

Wejście

Pierwszy wiersz zawiera jedną liczbę całkowitą N ($1 \leqslant N \leqslant 100\,000$) — liczbę dostaw żywności.
Drugi wiersz wejścia zawiera napis składający się z N znaków, oznaczających rodzaje posiłków w kolejnych dostawach. Każdy ze znaków napisu jest jedną z trzech liter `'M'` *(mięso),* `'F'` *(ryby),* `'B'` *(chleb).*

Wyjście

Na standardowym wyjściu należy wypisać jedną liczbę całkowitą, która oznacza maksymalną liczbę jednostek węgla, jakie można wyprodukować.

Ocenianie

Dla testów wartych w sumie 45 punktów, liczba dostaw żywności N nie przekracza 20.

Szczegółowa informacja o ocenie przy zgłaszaniu rozwiązań

Podczas konkursu możesz wybrać do 10 zgłoszeń w tym zadaniu do dokładnej oceny (która będzie wykonana najszybciej jak to możliwe). Ocena będzie odbywać się na podstawie części prawdziwych danych. Po wykonaniu oceny, będziesz mógł odczytać wyniki sprawdzenia w systemie konkursowym.

Przykłady

Wejście

```
6
MBMFFB
```

Wyjście

```
12
```

Wejście

```
16
MMBMBBBBMMMMMBMB
```

Wyjście

```
29
```

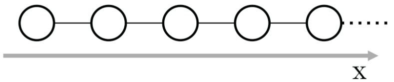
W przykładzie znajdującym się w lewej kolumnie, po skierowaniu dostaw do kopalń w następujący sposób: kopalnia 1, kopalnia 1, kopalnia 2, kopalnia 2, kopalnia 1, kopalnia 2, kopalnie po kolejnych dostawach wyprodukują odpowiednio: 1, 2, 1, 2, 3 i 3 jednostki węgla (dokładnie w tej kolejności), co da razem 12 jednostek. Istnieją także inne sposoby osiągnięcia tej wielkości produkcji węgla.

Krzysztof Diks, Paweł Idziak, Przemysława Kanarek, Marcin Kubica, Tomasz Waleń
Tłumaczenie

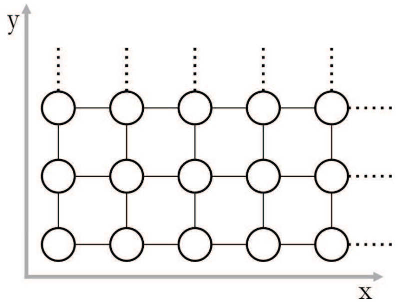
IOI 2007, dzień drugi, 19.08.2007

Pary

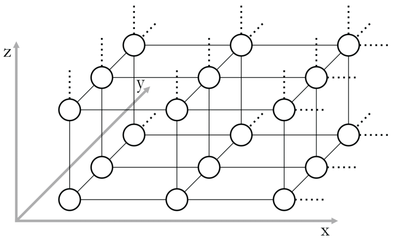
Mirek i Sławek bawią się pluszowymi zwierzątkami. Na początku wybierają jedną z trzech plansz przedstawionych na rysunku poniżej. Każda plansza składa się z pól (zaznaczonych na rysunku kółkami) umieszczonych na planie jedno–, dwu– lub trójwymiarowej siatki.



Plansza 1



Plansza 2



Plansza 3

Następnie Mirek umieszcza na polach planszy N **zwierzątek**.

Odległość *pomiędzy dwoma polami planszy jest mierzona najmniejszą liczbą ruchów potrzebnych do przemieszczenia się zwierzątka z jednego z tych pól na drugie. W jednym ruchu zwierzątko może przesunąć się na jedno z sąsiednich pól (połączonych na planszy krawędziami).*

Dwa zwierzątka słyszą się, jeśli znajdują się na polach odległych **co najwyżej o** *D. Zadanie Sławka polega na policzeniu, ile jest par zwierzątek mogących się usłyszeć nawzajem.*

Zadanie

Napisz program, który przy zadanym typie planszy, rozmieszczeniu zwierzątek na tej planszy i liczbie D, obliczy poszukiwaną liczbę par.

Wejście

Pierwszy wiersz zawiera cztery liczby całkowite w następującej kolejności:

- *typ planszy B ($1 \leqslant B \leqslant 3$),*
- *liczbę zwierzątek N ($1 \leqslant N \leqslant 100\,000$),*
- *największą odległość D, z jakiej zwierzątka mogą słyszeć się nawzajem ($1 \leqslant D \leqslant 100\,000\,000$),*
- *rozmiar planszy M (największa współrzędna dopuszczalna na wejściu):*
  - *dla $B = 1$, M wyniesie co najwyżej $75\,000\,000$,*
  - *dla $B = 2$, M wyniesie co najwyżej $75\,000$,*
  - *dla $B = 3$, M wyniesie co najwyżej 75.*

Każdy z następnych N wierszy zawiera B liczb całkowitych pooddzielanych pojedynczymi znakami odstępu — współrzędne jednego zwierzątka. Każda współrzędna będzie z zakresu od 1 do M (włącznie).
Na jednym polu może znajdować się więcej niż jedno zwierzątko.

Wyjście

Wyjście powinno zawierać jedną liczbę całkowitą — liczbę par zwierzątek mogących usłyszeć się nawzajem.
Uwaga: *do obliczeń i wypisania wyniku wykorzystaj 64–bitowy typ integer (*`long long` *w C/C++,* `int64` *w Pascalu).*

Ocenianie rozwiązań

W testach wartych w sumie 30 punktów, liczba zwierzątek N wyniesie co najwyżej $1\,000$.
Ponadto, dla każdego z trzech typów planszy, rozwiązanie, które da poprawne odpowiedzi dla wszystkich testów dla plansz tego typu, otrzyma co najmniej 30 punktów.

Przykłady

Wejście

```
1 6 5 100
25
50
50
10
20
23
```

Wyjście

```
4
```

Wejście

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

Wyjście

```
8
```

Wejście

```
3 8 10 20
10 10 10
10 10 20
10 20 10
10 20 20
20 10 10
20 10 20
20 20 10
20 20 20
```

Wyjście

```
12
```

Wyjaśnienia do przykładu z lewej strony:
Przyjmijmy, że zwierzątka są ponumerowane od 1 do 6 w porządku zgodnym z tym na wejściu. Poszukiwane cztery pary zwierzątek to:

- *1 – 5 (odległość 5),*
- *1 – 6 (odległość 2),*
- *2 – 3 (odległość 0),*
- *5 – 6 (odległość 3).*

Wyjaśnienia do środkowego przykładu:
Osiem poszukiwanych par to:

- *1 – 2 (odległość 2),*
- *1 – 4 (odległość 4),*
- *1 – 5 (odległość 3),*
- *2 – 3 (odległość 3),*
- *2 – 4 (odległość 4),*
- *3 – 4 (odległość 3),*
- *3 – 5 (odległość 4),*
- *4 – 5 (odległość 3).*

IOI 2007, dzień drugi, 19.08.2007

Trening

Jacek i Placek ciężko trenują, przygotowując się do wyścigu kolarskiego tandemów, odbywającego się co roku w Chorwacji. Muszą wybrać trasę, na której będą trenować.

W ich kraju jest N miast i M dróg. Każda droga łączy dwa miasta i jest dwukierunkowa. Dokładnie $N-1$ z tych dróg jest **asfaltowych***, natomiast pozostałe drogi to drogi gruntowe. Szczęśliwie, sieć dróg została zaprojektowana w taki sposób, że każdą parę miast łączy trasa złożona wyłącznie z asfaltowych dróg. Inaczej mówiąc, N miast i $N-1$* **asfaltowych dróg tworzy drzewo**.

Dodatkowo, z każdego miasta wychodzi **co najwyżej** *10* **dróg**.

Trasa treningowa zaczyna się w pewnym mieście, biegnie pewnymi drogami i kończy się w tym samym mieście, w którym się zaczęła. Jacek i Placek lubią zwiedzać nowe miejsca, więc postanowili, że trasa treningowa **nie może prowadzić przez to samo miasto** *ani* **tę samą drogę dwa razy**. *Trasa treningowa może zaczynać się w dowolnym mieście i nie musi prowadzić przez wszystkie miasta.*

W trakcie pedałowania można być z przodu lub z tyłu. Pedałujący z tyłu ma przyjemniej, gdyż ten z przodu osłania go od wiatru. Dlatego też Jacek i Placek postanowili zamieniać się miejscami w każdym mieście. Aby każdy z nich włożył taki sam wysiłek w trening, muszą wybrać trasę prowadzącą przez **parzystą liczbę dróg**.

Ludzie z „Układu" postanowili **zablokować** *niektóre z dróg gruntowych, tak aby Jacek i Placek* **nie mogli** *wyznaczyć trasy treningowej spełniającej powyższe warunki. Zablokowanie każdej drogi gruntowej wiąże się z określonym* **kosztem** *(***dodatnią** *liczbą całkowitą). Dróg asfaltowych nie da się zablokować.*

Zadanie

Napisz program, który na podstawie opisu sieci dróg łączących miasta wyznaczy najmniejszy łączny koszt, potrzebny do zablokowania takich dróg, żeby **nie istniała żadna trasa treningowa** *spełniająca podane powyżej warunki.*

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby całkowite N i M ($2 \leqslant N \leqslant 1\,000$, $N-1 \leqslant M \leqslant 5\,000$) — liczbę miast i łączną liczbę dróg.

Każdy z kolejnych M wierszy zawiera po trzy liczby całkowite A, B i C ($1 \leqslant A \leqslant N$, $1 \leqslant B \leqslant N$, $0 \leqslant C \leqslant 10\,000$), opisujące jedną drogę. Liczby A i B są różne i reprezentują miasta, które łączy bezpośrednio dana droga. Jeśli $C = 0$, to droga jest asfaltowa; w przeciwnym przypadku droga jest drogą gruntową, a C jest kosztem jej zablokowania.

Z każdego miasta wychodzi co najwyżej 10 dróg. Każdą parę miast łączy co najwyżej jedna bezpośrednia droga.

Wyjście

Wyjście powinno zawierać jedną liczbę całkowitą — minimalny łączny koszt opisany w treści zadania.

Ocena rozwiązań

W testach wartych łącznie 30 punktów, asfaltowe drogi będą tworzyć prostą trasę (tzn. z żadnego miasta nie będą wychodzić trzy lub więcej asfaltowe drogi).

Szczegółowa informacja o ocenie przy zgłaszaniu rozwiązań

W trakcie zawodów możesz wybrać do 10 zgłoszonych rozwiązań tego zadania do oceny (jak tylko to będzie możliwe) na części oficjalnych testów. Po dokonaniu oceny, podsumowanie wyników będzie dostępne na stronie systemu konkursowego.

Przykłady

Wejście

```
5 8
2 1 0
3 2 0
4 3 0
5 4 0
1 3 2
3 5 2
2 4 5
2 5 1
```

Wyjście

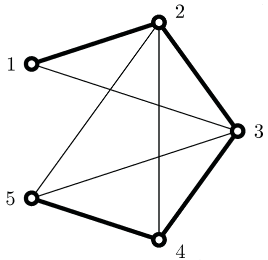
```
5
```

Wejście

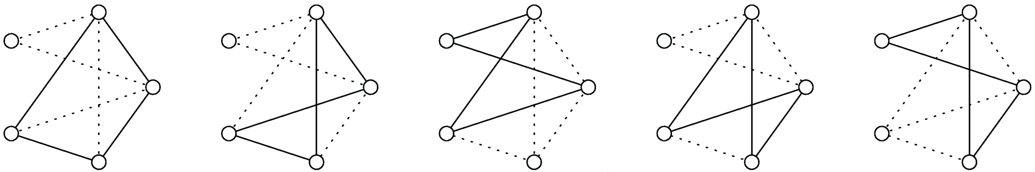
```
9 14
1 2 0
1 3 0
2 3 14
2 6 15
3 4 0
3 5 0
3 6 12
3 7 13
4 6 10
5 6 0
5 7 0
5 8 0
6 9 11
8 9 0
```

Wyjście

```
48
```



Układ dróg i miast w pierwszym przykładzie. Drogi asfaltowe zaznaczono grubą linią.



Jacek i Placek mają pięć możliwych tras treningowych. Gdyby zablokować krawędzie $1-3$*,* $3-5$ *i* $2-5$*, to Jacek i Placek nie mogliby skorzystać z żadnej z tych tras. Koszt zablokowania tych krawędzi wynosi* 5*. Możliwe jest też zablokowanie tylko dwóch krawędzi,* $2-4$ *i* $2-5$*, ale to oznaczałoby wyższy koszt:* 6*.*

XIII Bałtycka Olimpiada Informatyczna,

Güstrow, Niemcy 2007

BOI 2007, dzień pierwszy, 25.04.2007

Dźwięk ciszy

W nagraniach cyfrowych dźwięk jest opisywany przez sekwencję liczb, reprezentujących ciśnienie powietrza, mierzone w krótkich (ale stałych) odstępach czasu. Każda wartość z sekwencji nazywana jest próbką.

Ważnym krokiem w przetwarzaniu dźwięku jest podział nagrania na kawałki, zawierające dźwięk (nieciszę), pooddzielany ciszą. Aby uniknąć podziału nagrania na zbyt mało lub zbyt dużo kawałków, cisza jest często definiowana jako sekwencja m próbek, w których różnica pomiędzy najmniejszą i największą wartością nie przekracza pewnego progu c.

Napisz program, który dla danych wartości parametrów m oraz c wykryje ciszę w zadanej sekwencji n próbek.

Wejście

Dane wejściowe należy wczytać z pliku o nazwie `sound.in`. *Pierwszy wiersz wejścia zawiera trzy liczby całkowite: n ($1 \leqslant n \leqslant 1\,000\,000$) – liczbę próbek w nagraniu, m ($1 \leqslant m \leqslant 10\,000$) – wymaganą długość ciszy oraz c ($0 \leqslant c \leqslant 10\,000$) – maksymalny dopuszczalny poziom szumu w ciszy.*

W drugim wierszu zapisano n liczb całkowitych a_i ($0 \leqslant a_i \leqslant 1\,000\,000$ dla $1 \leqslant i \leqslant n$), pooddzielanych pojedynczymi odstępami i oznaczających kolejne próbki nagrania.

Wyjście

Dane wyjściowe należy zapisać do pliku tekstowego o nazwie `sound.out`. *Plik powinien zawierać wszystkie wartości i takie, że $max(a[i \ldots i+m-1]) - min(a[i \ldots i+m-1]) \leqslant c$. Wartości powinny być wypisane w porządku rosnącym, każda w oddzielnym wierszu.*

Jeśli w wyjściowym nagraniu nie ma żadnej ciszy, należy zapisać słowo `NONE` *w pierwszym i jedynym wierszu pliku wyjściowego.*

Przykład

Dla pliku wejściowego `sound.in`*:*

```
7 2 0
0 1 1 2 3 2 2
```

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

```
2
6
```

BOI 2007, dzień pierwszy, 25.04.2007

Sortowanie rankingu

Masz dane wyniki punktowe, które uzyskali zawodnicy podczas konkursu. Twoim zadaniem jest przygotowanie listy rankingowej zawodników, posortowanej malejąco względem punktów.

Struktura danych, która przechowuje listę zawodników, obsługuje jedynie jedną operację, która przesuwa zawodnika z pozycji i *na pozycję* j *(nie zmieniając przy tym porządku pozostałych zawodników). Jeśli* $i > j$*, to pozycje zawodników na miejscach od* j *do* $i-1$ *zwiększają się o* 1*, natomiast jeżeli* $i < j$*, to pozycje zawodników na miejscach od* $i+1$ *do* j *zmniejszają się o* 1*.*

Taka operacja wymaga wykonania i *kroków (aby zlokalizować zawodnika, który ma być przesunięty), a następnie* j *kroków (aby zlokalizować miejsce docelowe), stąd całkowity koszt przesunięcia zawodnika z pozycji* i *do* j *wynosi* $i+j$*. Pozycje zawodników są ponumerowane, rozpoczynając od* 1*.*

Wyznacz sekwencję operacji o najmniejszym sumarycznym koszcie, która utworzy żądaną listę rankingową.

Wejście

Dane wejściowe należy wczytać z pliku tekstowego o nazwie `sorting.in`*. Pierwszy wiersz wejścia zawiera liczbę całkowitą* n *(*$2 \leqslant n \leqslant 1\,000$*), oznaczającą liczbę zawodników. Każdy z kolejnych* n *wierszy zawiera jedną liczbę całkowitą nieujemną* s_i *(*$0 \leqslant s_i \leqslant 1\,000\,000$*) — są to wyniki punktowe zawodników w początkowym uporządkowaniu. Możesz założyć, że wyniki każdych dwóch zawodników są różne.*

Wyjście

Dane wyjściowe należy zapisać do pliku tekstowego o nazwie `sorting.out`*. W pierwszym wierszu wyjścia należy zapisać liczbę kroków, potrzebną do utworzenia listy rankingowej. W kolejnych wierszach należy zapisać operacje, w wyniku których powstanie docelowa lista, w kolejności w jakiej mają być one wykonywane. Każda operacja powinna być opisana w jednym wierszu, zawierającym dwie liczby całkowite* i *oraz* j*, które oznaczają, że zawodnik z pozycji* i *ma być przesunięty na pozycję* j*. Liczby* i *oraz* j *powinny być oddzielone pojedynczym odstępem.*

Przykład

Dla pliku wejściowego `sorting.in`*:*

```
5
20
30
5
15
10
```

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

```
2
2 1
3 5
```

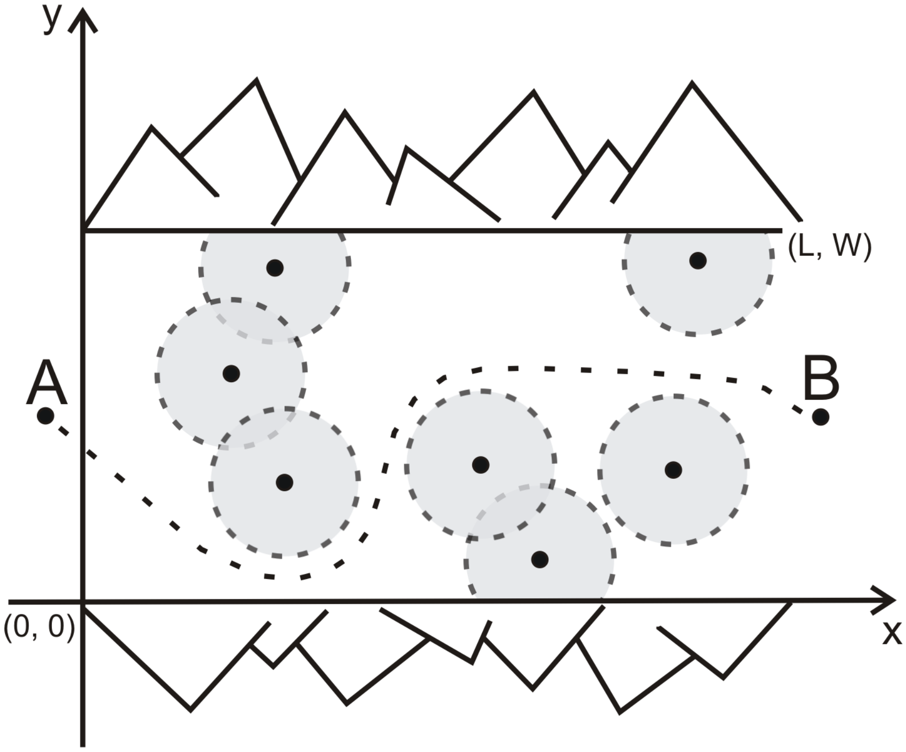
Ocenianie

W 30% testów spełniony jest warunek $n \leqslant 10$*.*

BOI 2007, dzień pierwszy, 25.04.2007

Ucieczka

Grupa jeńców wojennych planuje ucieczkę z więzienia. Opracowali już szczegółowy plan ucieczki, po zakończeniu której mają nadzieję na znalezienie schronienia w pobliskiej wiosce. Jedyny problem, który muszą rozwiązać, to przedostanie się przez kanion, leżący pomiędzy wioską (oznaczoną na rysunku literą B) i więzieniem (oznaczonym literą A). Kanion jest cały czas strzeżony przez żołnierzy. Na szczęście dla uciekinierów żołnierze są bardzo leniwi, i przez większość czasu przebywają na swoich stanowiskach. Zasięg wzroku każdego żołnierza jest ograniczony do dokładnie 100 metrów. Dla danego rozmieszczenia żołnierzy powiemy, że można bezpiecznie przejść przez kanion, jeśli w każdym momencie ucieczki utrzymuje się odległość do najbliższego żołnierza **ostro większą** *od 100 metrów.*



Napisz program, który na podstawie szerokości i długości kanionu oraz współrzędnych wszystkich żołnierzy w kanionie (zakładamy, że żołnierze nie zmieniają swoich położeń) zdecyduje, czy uciekinierzy mogą przedostać się przez kanion niezauważeni. Jeśli jest to niemożliwe, to uciekinierzy (którzy są zagorzałymi pacyfistami) chcieliby znać minimalną liczbę żołnierzy, których muszą wyeliminować, żeby bezpiecznie przedostać się przez kanion. Żołnierz może zostać wyeliminowany nawet, jeżeli jest w polu widzenia innego żołnierza.

Wejście

Dane wejściowe należy wczytać z pliku tekstowego o nazwie `escape.in`*. Pierwszy wiersz wejścia zawiera trzy liczby całkowite L, W oraz N, oznaczające długość i szerokość kanionu oraz liczbę żołnierzy. Każdy z kolejnych N wierszy zawiera dwie liczby całkowite X_i oraz Y_i — współrzędne i–tego żołnierza w kanionie ($0 \leqslant X_i \leqslant L$, $0 \leqslant Y_i \leqslant W$). Współrzędne są podane w metrach; południowo-zachodni róg kanionu ma współrzędne $(0,0)$, a północno-wschodni róg — współrzędne (L,W) (patrz rysunek).*

Ucieczka przez kanion może rozpocząć się w punkcie o współrzędnych $(0,y_s)$ dla dowolnego $0 \leqslant y_s \leqslant W$ i zakończyć w punkcie o współrzędnych (L,y_e) dla dowolnego $0 \leqslant y_e \leqslant W$. Wartości y_s i y_e nie muszą przy tym być liczbami całkowitymi.

Wyjście

Wynik należy zapisać w pliku tekstowym o nazwie `escape.out`*. W pierwszym i jedynym wierszu wyjścia należy wpisać minimalną liczbę żołnierzy, których trzeba wyeliminować, aby uciekinierzy mogli bezpiecznie przejść przez kanion. Jeśli uciekinierzy mogą przejść przez kanion bez żadnych zabójstw, to należy wpisać 0 (zero).*

Przykład

Dla pliku wejściowego `escape.in`*:*

```
130 340 5
10 50
130 130
70 170
0 180
60 260
```

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

```
1
```

Ograniczenia

$1 \leqslant W \leqslant 50\,000 \quad 1 \leqslant L \leqslant 50\,000 \quad 1 \leqslant N \leqslant 250$

Ocenianie

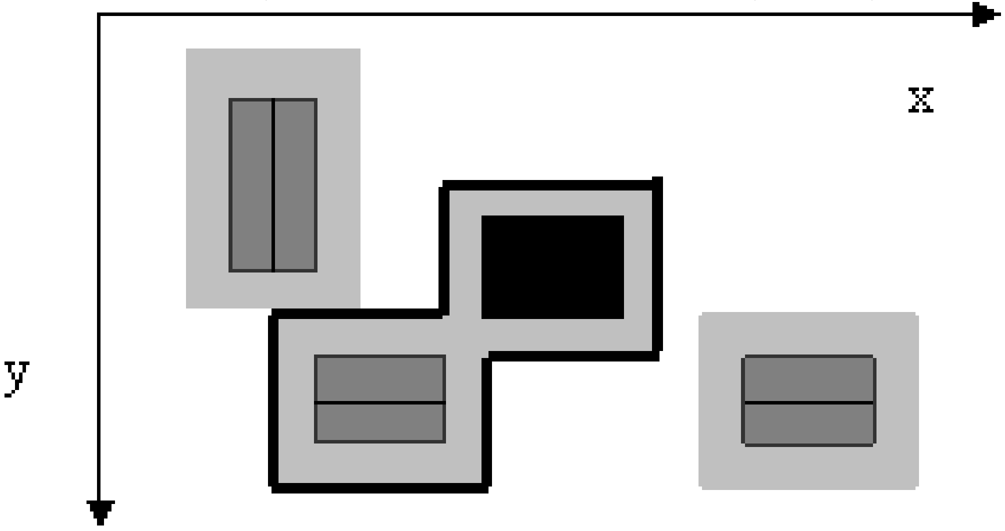
Twój program otrzyma częściowe punkty za zadanie, jeżeli będzie w stanie jedynie odpowiadać na pytanie, czy jeńcy mogą przejść przez kanion bez eliminowania jakichkolwiek żołnierzy. Testy do zadania są połączone w grupy. Jeżeli dla każdego testu w danej grupie Twój program poprawnie stwierdzi, czy eliminacja żołnierzy jest konieczna (0 oznacza, że jeńcy mogą przejść przez kanion niezauważeni, a dowolna liczba całkowita > 0 oznacza, że eliminacja pewnej liczby żołnierzy jest konieczna), to uzyskasz 30% punktów przynależnych za tę grupę. Za daną grupę otrzymasz 100% punktów, jeśli Twój program dla każdego testu z tej grupy poprawnie stwierdzi, ilu minimalnie żołnierzy muszą wyeliminować jeńcy, aby ich ucieczka mogła być bezpieczna.

BOI 2007, dzień drugi, 26.04.2007

Budowanie płotu

Czesiek to ma w życiu szczęście! Właśnie wygrał na loterii ogromną posiadłość. Oprócz głównej rezydencji, w której Czesiek zamierza odtąd zamieszkać, posiadłość zawiera pewną liczbę innych wspaniałych budynków. Niestety posiadłości brakuje płotu, który ochraniałby teren przed wścibskimi przechodniami, przez co Czesiek czuje się bardzo niespokojny. Zdecydował się więc wybudować płot, niemniej jednak nie stać go na ogrodzenie płotem całej posiadłości. Po chwili namysłu doszedł do wniosku, że wystarczy, jeżeli płot ogrodzi główną rezydencję. Płot nie może być poprowadzony zbyt blisko żadnego z budynków. Ściślej rzecz ujmując, patrząc z lotu ptaka, każdy budynek jest otoczony „zabronionym prostokątem” (bez brzegu), w którym nie może się znaleźć żadna część płotu. Boki wszystkich prostokątów są równoległe do osi x albo osi y. Podobnie każdy fragment płotu musi być równoległy albo do osi x, albo do osi y.

Pomóż Cześkowi obliczyć minimalną długość płotu, którym można ogrodzić główną rezydencję jego posiadłości.



Rys. 1: Główna rezydencja (czarny prostokąt) wraz z trzema innymi budynkami, otoczonymi zabronionymi prostokątami. Gruba czarna linia zaznacza najkrótszy możliwy płot, który ogradza główną rezydencję.

Wejście

Wejście znajduje się w pliku tekstowym `fence.in`. *Pierwszy wiersz pliku wejściowego zawiera liczbę całkowitą dodatnią* m *($1 \leqslant m \leqslant 100$), oznaczającą liczbę budynków w posiadłości. Kolejnych* m *wierszy opisuje zabronione prostokąty, które otaczają poszczególne budynki. Każdy wiersz zawiera cztery liczby całkowite* tx, ty, bx *oraz* by, *pooddzielane pojedynczymi odstępami.* (tx,ty) *to współrzędne lewego górnego wierzchołka prostokąta, a* (bx,by) *— współrzędne prawego dolnego wierzchołka. Wszystkie współrzędne na wejściu spełniają warunki* $0 \leqslant tx < bx \leqslant 10\,000$ *oraz* $0 \leqslant ty < by \leqslant 10\,000$. *Pierwszym prostokątem na wejściu jest zawsze prostokąt zabroniony, otaczający główną rezydencję posiadłości.*

Wyjście

Wyjście powinno zostać zapisane do pliku tekstowego `fence.out`. *W pierwszym i jedynym wierszu wyjścia powinna się znajdować jedna liczba całkowita dodatnia, oznaczająca minimalną długość płotu, który ogrodzi główną rezydencję posiadłości.*

Przykład

Dla pliku wejściowego `fence.in`*:*

```
4
8 4 13 8
2 1 6 7
4 7 9 11
14 7 19 11
```

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

```
32
```

Ocenianie

W 30% testów zachodzi nierówność $m \leqslant 10$.

BOI 2007, dzień drugi, 26.04.2007

Ciąg

Niech dany będzie ciąg $a_1, \ldots, a_n$. Możemy na nim wykonywać operację redukuj(i), która zastępuje wyrazy a_i i a_{i+1} przez jeden wyraz równy $max(a_i, a_{i+1})$, w wyniku czego długość ciągu maleje o jeden. Koszt tej operacji jest równy $max(a_i, a_{i+1})$. Po wykonaniu operacji redukuj $n-1$ razy, otrzymujemy ciąg długości 1. Twoim zadaniem jest wyznaczenie kosztu optymalnego schematu redukcji, czyli ciągu operacji redukuj o minimalnym sumarycznym koszcie, po wykonaniu którego ciąg staje się jednoelementowy.

Wejście

Wejście znajduje się w pliku tekstowym `sequence.in`. *Pierwszy wiersz wejścia zawiera liczbę n ($1 \leqslant n \leqslant 1\,000\,000$), oznaczającą długość ciągu. Kolejne n wierszy zawiera wyrazy ciągu — po jednej liczbie całkowitej a_i w wierszu ($0 \leqslant a_i \leqslant 1\,000\,000\,000$).*

Wyjście

Wyjście powinno zostać zapisane do pliku tekstowego `sequence.out`. *Pierwszy i jedyny wiersz wyjścia powinien zawierać minimalny koszt redukcji ciągu do jednego elementu.*

Przykład

Dla pliku wejściowego `sequence.in`*:*

```
3
1
2
3
```

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

```
5
```

Ocenianie

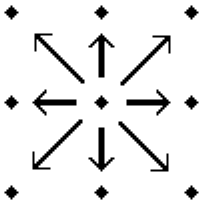
W 30% testów jest spełniony warunek $n \leqslant 500$.
W 50% testów jest spełniony warunek $n \leqslant 20\,000$.

Jakub Radoszewski, Tomasz Waleń
Tłumaczenie

BOI 2007, dzień drugi, 26.04.2007

Połącz punkty!

Rozważmy kratownicę złożoną z $3 \times N$ *punktów. Każdy punkt kratownicy ma maksymalnie osiem punktów sąsiednich (patrz rys. 1).*



Rys. 1: Punkty sąsiednie (oznaczone strzałkami).

Chcielibyśmy policzyć liczbę różnych sposobów połączenia punktów kratownicy w wielokąt, które spełniają następujące warunki:

1. *Wszystkie* $3 \times N$ *punktów kratownicy tworzy zbiór wierzchołków wielokąta.*
2. *Kolejne wierzchołki na obwodzie wielokąta są zawsze sąsiednimi punktami kratownicy.*
3. *Wielokąt jest prosty, tzn. w jego obwodzie nie występują samoprzecięcia.*

Dwa przykładowe wielokąty dla $N = 6$ *są przedstawione na rys. 2.*

Rys. 2: Dwa przykładowe sposoby połączenia punktów dla $N = 6$.

Napisz program, który dla danego N *wyznaczy liczbę możliwych sposobów połączenia punktów w opisany sposób modulo 1 000 000 000.*

Wejście

Wejście znajduje się w pliku tekstowym `points.in`. *Pierwszy i jedyny wiersz wejścia zawiera jedną liczbę całkowitą dodatnią* N *(*$N \leqslant 1\,000\,000\,000$*).*

Wyjście

Wyjście powinno zostać zapisane do pliku tekstowego `points.out`. *Pierwszy i jedyny wiersz wyjścia powinien zawierać resztę z dzielenia przez 1 000 000 000 liczby sposobów połączenia punktów w opisany sposób.*

Przykład

Dla pliku wejściowego `points.in`*:*

```
3
```

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

```
8
```

natomiast dla pliku wejściowego `points.in`*:*

```
4
```

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

```
40
```

Ocenianie

- *W 30% testów spełniona jest nierówność* $N \leqslant 200$.
- *W 70% testów spełniona jest nierówność* $N \leqslant 100\,000$.

XIV Olimpiada Informatyczna Krajów Europy Środkowej,

Brno, Czechy 2007

Tomasz Idziaszek, Jakub Pawlewicz
Tłumaczenie

CEOI 2007, dzień pierwszy, 03.07.2007

Ministerstwo

Pewnego razu, w odległym kraju, rząd powołał do życia Ministerstwo ds. Ograniczania Papierkowej Roboty. Jak zapewne się domyślasz, było to największe ze wszystkich ministerstw. Liczba urzędników pracujących tam była naprawdę ogromna. Jednakowoż struktura ministerstwa była bardzo prosta: minister miał pod sobą co najwyżej trzech podwładnych, każdy z nich również miał co najwyżej trzech podwładnych i tak dalej.

Ostatnie wybory spowodowały zmianę na stanowisku ministra. Fotel objął młody człowiek, pełen zapału i świeżych pomysłów. Postanowił uczynić zadość nazwie swojego ministerstwa, a do roboty wziął się natychmiast. Zauważył, że pewne części hierarchicznej struktury zatrudnienia w ministerstwie wyglądają tak samo, zatem muszą wykonywać tę samą pracę. A gdy dwie takie części robią to samo, jedna jest zbędna i może zostać rozwiązana, a urzędnicy zwolnieni. Twoim zadaniem jest znaleźć liczbę równoważnych części i wykonać trochę papierkowej roboty (tj. wypisać wynik na standardowe wyjście w odpowiednim formacie).

Zadanie

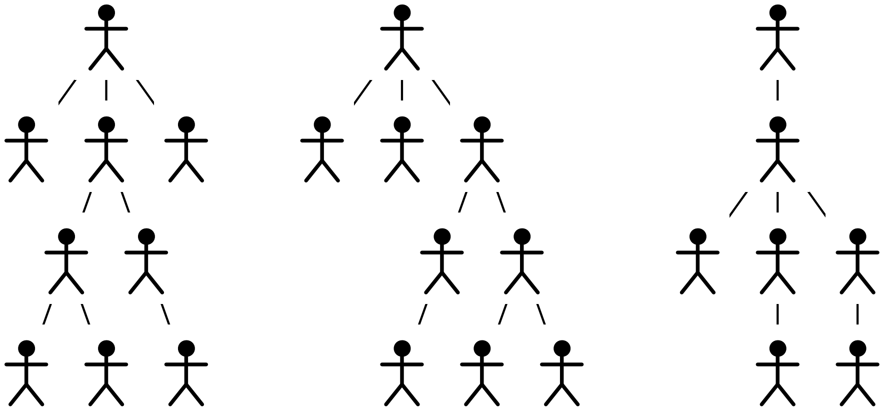
Dana jest struktura organizacyjna ministerstwa. Każdy urzędnik ma dokładnie jednego przełożonego i co najwyżej trzech podwładnych (być może żadnego). Jedynym wyjątkiem jest minister, który nie ma zwierzchnika (ale również ma co najwyżej trzech podwładnych). Nie ma żadnej ustalonej kolejności między podwładnymi danego urzędnika.

Wydział *składa się z urzędnika (nazywanego kierownikiem wydziału), wszystkich jego podwładnych, wszystkich podwładnych tych podwładnych itd. Wyróżniamy dwa specjalne rodzaje wydziałów: całe ministerstwo (na czele którego stoi minister) i jednoosobowe wydziały składające się z pojedynczych urzędników nieposiadających podwładnych.*

Głębokością wydziału *nazywamy długość najdłuższego łańcucha* $x_1,\ldots,x_d$ *urzędników wydziału, takiego że* x_i *jest przełożonym* x_{i+1} *dla każdego* $1 \leqslant i < d$. *Zauważ, że głębokość jednoosobowego wydziału wynosi 1.*

Dwa wydziały A *i* B *mają taką samą strukturę, jeśli każdemu urzędnikowi* x *z wydziału* A *odpowiada dokładnie jeden urzędnik* x' *z wydziału* B *i na odwrót: każdemu urzędnikowi* x' *z* B *odpowiada dokładnie jeden urzędnik* x *z* A. *W szczególności, dla każdych urzędników* x *i* y *musi zachodzić:* x *jest zwierzchnikiem* y *wtedy i tylko wtedy, gdy* x' *(urzędnik odpowiadający* x*-owi) jest zwierzchnikiem* y'*-a (odpowiadający* y*-owi). Zauważ, że jeśli wydziały* A *i* B *mają taką samą strukturę, to kierownikowi wydziału* A *odpowiada kierownik wydziału* B *i oba wydziały mają taką samą głębokość i liczbę urzędników.*

Na poniższym rysunku wydziały A *i* B *mają taką samą strukturę, natomiast struktura wydziału* C *jest różna zarówno od* A*, jak i od* B.



Wydzia A Wydzia B Wydzia C

Twoim zadaniem jest wyznaczenie liczby wydziałów o różnej strukturze dla każdej głębokości. Innymi słowy, masz podać ciąg liczb $n_1,\ldots,n_d$*, taki że* d *jest głębokością całego ministerstwa i dla każdego* i *liczba różnych struktur wydziałów o głębokości* i *wynosi* n_i.

Wejście

Wejście składa się z jednego wiersza, który opisuje strukturę organizacyjną ministerstwa, korzystając z następującej notacji. Każdy wydział jest zakodowany jako $(x_1 \ldots x_k)$ *, gdzie* $0 \leqslant k \leqslant 3$ *jest liczbą podwładnych kierownika wydziału, a* x_i *są kodami ich wydziałów. Jednoosobowy wydział jest zatem zakodowany jako* `()`*. Struktura ministerstwa jest opisana kodem całego ministerstwa.*

Ministerstwo składa się z co najwyżej 1 000 000 urzędników (włącznie z ministrem).

Wyjście

Wyjście powinno składać się z d wierszy, gdzie d jest głębokością ministerstwa (tzn. wydziału kierowanego przez ministra). W wierszu i-tym powinna znajdować się liczba wydziałów o głębokości i mających różne struktury.

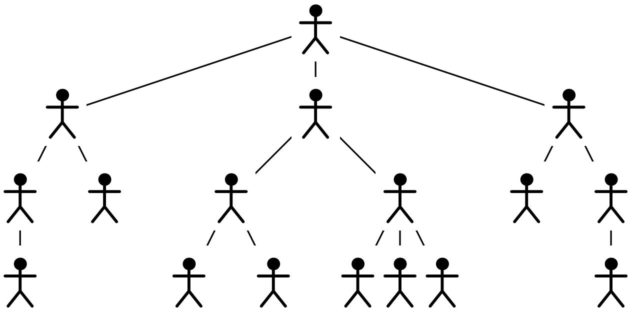
Przykład

Wejście

```
(((())())((()())(()()()))(()(())))
```

Wyjście

```
1
3
2
1
```



Tomasz Idziaszek, Jakub Pawlewicz
Tłumaczenie

CEOI 2007, dzień pierwszy, 03.07.2007

Obrzydliwe obliczenia

Twój przyjaciel Jasiek zapomniał odrobić pracę domową z matematyki, co mocno zdenerwowało jego nauczyciela (zgadnij dlaczego). Kazał on zostać Jaśkowi po szkole i robić nudne zadania. Nauczyciel jest jednak leniwy i nie chce mu się wymyślać ciągle to nowych zadań, zatem dał Jaśkowi jedno wyrażenie f zmiennej x i dużą liczbę wartości dla x (jak zapewne rozumiesz, te są łatwiejsze do wygenerowania). Zadaniem Jaśka jest policzenie $f(x)$ dla podanych wartości x.

Tak naprawdę Jasiek nie musi obliczać dokładnych wartości $f(x)$, jako że mogą być one całkiem duże. Zamiast tego, powinien on wypisać ostatnią cyfrę $f(x)$ (Jasiek podejrzewa, że to dlatego, że jego nauczyciel umie liczyć tylko do 100, ale kto wie...). Jako że tematem ostatniej lekcji były systemy numeryczne w różnych podstawach, wszystkie obliczenia i wyrażenia (w szczególności wyrażenie opisujące f, wartości dla x i ostatnia cyfra $f(x)$) będą podawane w systemie o podstawie B.

Zadanie

Ponieważ wartości dla x są naprawdę duże, a wyrażenie f w żadnym wypadku krótkie, Jasiek poprosił Cię o pomoc. Wie, że jesteś świetnym programistą, zatem jest przekonany, że użyjesz komputera, aby rozwiązać to zadanie. Aby choć trochę Ci pomóc, przepisał wyrażenie f w notacji postfiksowej (opisanej poniżej), gdyż jest świadomy, że komputerom jest łatwiej operować na takich wyrażeniach. Twoim zadaniem jest wyznaczenie ostatniej cyfry $f(x)$ dla danego wyrażenia f i podanych wartości x w systemie o podstawie B.

Notacja postfiksowa

Notacja postfiksowa *(znana też jako* **odwrotna notacja polska***, ONP) jest alternatywnym sposobem zapisu wyrażeń matematycznych. W bardziej rozpowszechnionej notacji infiksowej operator jest umieszczany* **pomiędzy** *operandami, tak jak w wyrażeniach $1+2$, $1+2*3$ czy $(1+2)*3$. Natomiast w notacji postfiksowej, operator jest umieszczany* **za** *swoimi operandami: powyższe wyrażenia będą zapisane jako $1\ 2\ +$, $1\ 2\ 3\ *\ +$ oraz $1\ 2\ +\ 3\ *$.*

Taka notacja wydaje się trudniejsza do czytania przez ludzi (przynajmniej w porównaniu do notacji infiksowej), jednakże jest o wiele łatwiej napisać program, który obliczy wartość wyrażenia w ONP niż taki, który „zrozumie” notację infiksową. Dodatkowo, notacja ONP nie wymaga używania nawiasów, gdyż kolejność operandów i operatorów jednoznacznie opisuje wyrażenie.

System numeryczny o podstawie B

Liczby są zwyczajowo reprezentowane w **systemie dziesiętnym***. W tym systemie ciąg cyfr $d_k d_{k-1} \ldots d_1 d_0$ koduje liczbę $d_k 10^k + d_{k-1} 10^{k-1} + \cdots + d_1 10^1 + d_0 10^0$. Jeśli 10 zastąpimy przez liczbę B, $B \geqslant 2$, i pozwolimy, by d_i były liczbami od 0 do $B-1$, to dostaniemy* **system o podstawie** B*. Wartość B jest* **podstawą** *systemu, a d_i są* **cyframi***. Mówiąc precyzyjniej, w systemie o podstawie B, ciąg $d_k d_{k-1} \ldots d_1 d_0$ koduje liczbę $d_k B^k + d_{k-1} B^{k-1} + \cdots + d_1 B^1 + d_0 B^0$. Oczywiście, jako że mamy tylko dziesięć cyfr w systemie dziesiętnym, zbiór „cyfr” musi być rozszerzony, aby móc reprezentować cyfry $d_i > 9$. Standardowo cyfry dziesiętne zachowują swoje znaczenie, a pozostałe cyfry są reprezentowane przez litery: A oznacza 10, B oznacza 11 itd. Dla przykładu, liczba 29 w systemie dziesiętnym (o podstawie 10) jest zapisywana jako 45 w systemie o podstawie 6 i jako $1D$ w systemie szesnastkowym (o podstawie 16).*

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby B i N oddzielone pojedynczym odstępem. B ($2 \leqslant B \leqslant 36$) jest podstawą systemu numerycznego, natomiast N ($1 \leqslant N \leqslant 100\,000$) oznacza liczbę wartości dla x.

Drugi wiersz zawiera opis wyrażenia f w notacji postfiksowej, składa się on z ciągu elementów oddzielonych pojedynczymi odstępami. Każdy element może być:

- *Ciągiem cyfr i wielkich liter. Ciąg taki opisuje liczbę zapisaną w systemie o podstawie B. Możesz założyć, że liczba ta nie przekracza $2\,000\,000\,000$.*
- *Małą literą* `x`*. Ten znak powinien być zastąpiony podczas obliczeń odpowiednią wartością.*
- *Operatorem dodawania* `+`*.*
- *Operatorem odejmowania* `-`*.*
- *Operatorem mnożenia* `*`*.*

Każdy z kolejnych N wierszy zawiera ciąg cyfr i wielkich liter, który koduje kolejną wartość dla x zapisaną w systemie o podstawie B. Możesz założyć, że żadna z tych wartości nie przekracza $2\,000\,000\,000$. Możesz również założyć, że dane wejściowe są poprawne, tzn. zawartość drugiego wiersza jest poprawnym wyrażeniem zapisanym w notacji postfiksowej i każdy ciąg cyfr i wielkich liter jest poprawną liczbą całkowitą zapisaną w systemie o podstawie B. Ponadto możesz założyć, że długość drugiego wiersza nie przekracza $100\,000$ znaków.

Wyjście

Wyjście powinno składać się z dokładnie N wierszy. W i-tym wierszu powinien znajdować się dokładnie jeden znak (cyfra lub wielka litera), oznaczający ostatnią cyfrę $f(x)$ (w systemie o podstawie B) dla x danego w $(i+2)$-im wierszu wejścia. Możesz założyć, że $f(x)$ będzie nieujemne dla wszystkich x podanych na wejściu.

Przykład

Wejście

```
15 4
2 x * 123A +
1
2
3
4
```

Wyjście

```
C
E
1
3
```

Tomasz Idziaszek, Jakub Pawlewicz
Tłumaczenie

CEOI 2007, dzień pierwszy, 03.07.2007

Podarty żagiel

Podczas niedawnego dwudniowego sztormu, zespół żeglarzy rozbił się na bezludnej wyspie na Morzu Śródziemnym. Stracili również łączność, więc nie mogli prosić o pomoc i jedyną szansą przetrwania była ucieczka na tratwie. Do tego niestety potrzebowali zalet żagla. Na szczęście ocalał główny żagiel z rozbitego statku, ale jest on podarty na wiele kawałków. W związku z tym żeglarze muszą zszyć ze sobą ocalałe kawałki, aby złożyć oryginalny żagiel.

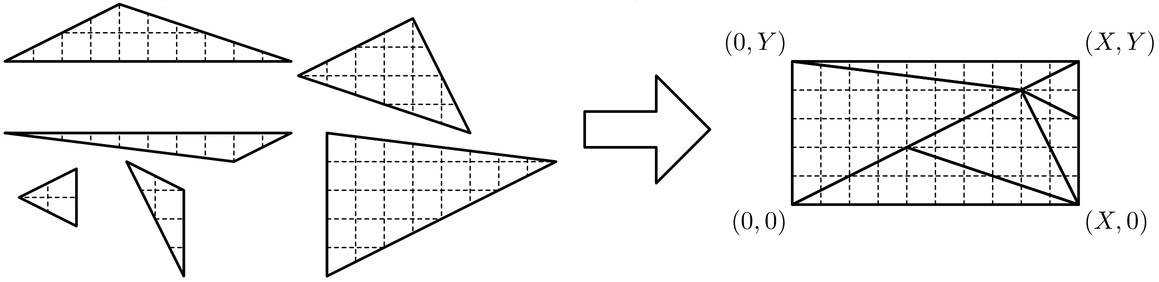
Zespół szybko zorientował się, że każdy kawałek jest trójkątem. Ponieważ na żaglach były wzorki w paski, więc można łatwo odtworzyć prawidłową orientację każdego z kawałków. Tak naprawdę żeglarze są zmuszeni zachować orientację wszystkich części, gdyż w przeciwnym razie żagiel może być mniej odporny na wiatr i podrzeć się ponownie. Żeglarze przez parę godzin próbowali odtworzyć pierwotny układ żagla, ale bez skutku. Wydaje im się, że komputer może im pomóc i proszą Ciebie o pomoc.

Zadanie

*Dany jest opis trójkątnych kawałków żagla (dla każdego wierzchołka trójkąta podane są jego współrzędne) oraz rozmiary oryginalnego żagla (jako dwie liczby całkowite X i Y). Twoim zadaniem jest przesunięcie wszystkich kawałków (***bez obracania***) tak, aby odtworzyć oryginalny żagiel (patrz rysunek). W szczególności trójkąty powinny zostać przemieszczone tak, aby:*

- *cały prostokąt był pokryty,*
- *żadne dwa trójkąty się nie przecinały, tj. nie istniał punkt należący do* **wnętrz** *dwóch różnych trójkątów, oraz*
- *żaden trójkąt nie wystawał poza prostokąt.*

Możesz założyć, że dla każdego danego wejścia istnieje rozwiązanie. W szczególności można założyć, że suma pól wszystkich kawałków równa się polu oryginalnego żagla. Może być wiele części o tym samym kształcie.



Wejście

Dla tego zadania nie należy wysyłać programu, który je rozwiązuje. Zamiast tego w katalogu `/mo/public/problems/sail` *znajdziesz pliki* `01.in`*,…,*`10.in` *z gotowymi wejściami, których struktura jest następująca. Pierwszy wiersz składa się z trzech liczb całkowitych N, X i Y pooddzielanych pojedynczymi odstępami. X i Y $(1 \leqslant X,Y \leqslant 1\,000\,000)$ określają rozmiary oryginalnego żagla, gdzie X jest długością, a Y wysokością. N, $2 \leqslant N \leqslant 1\,000$, oznacza liczbę trójkątnych części porwanego żagla.*

Następne N wierszy opisuje poszczególne kawałki: $(i+1)$-szy wiersz zawiera sześć liczb całkowitych $A_{i,x}$, $A_{i,y}$, $B_{i,x}$, $B_{i,y}$, $C_{i,x}$ i $C_{i,y}$ pooddzielanych pojedynczymi odstępami. Te sześć liczb opisuje trzy wierzchołki i-tego trójkąta: A_i, B_i i C_i. Wierzchołek A_i ma współrzędne $(A_{i,x},A_{i,y})$, wierzchołek B_i ma współrzędne $(B_{i,x},B_{i,y})$ oraz C_i ma współrzędne $(C_{i,x},C_{i,y})$. Możesz założyć, że $0 \leqslant A_{i,x},A_{i,y},B_{i,x},B_{i,y},C_{i,x},C_{i,y} \leqslant 1\,000\,000$.

Pozycja trójkąta na płaszczyźnie (podanego w pliku wejściowym) nie ma żadnego znaczenia, a jedynie istotny jest jego kształt. Na przykład, jeśli zastąpimy trzeci wiersz przykładowego wejścia przez `6 6 14 5 16 6`*, to dany zbiór trójkątów pozostanie niezmieniony.*

Wyjście

Dla każdego pliku wejściowego powinieneś utworzyć odpowiadający mu plik wyjściowy `01.out`*,…,*`10.out`*. Nie powinieneś wysyłać żadnego programu dla tego zadania.*

Plik wyjściowy `XX.out` *powinien składać się dokładnie z N wierszy opisujących ustawienie kawałków żagla tak, aby zostały spełnione wszystkie trzy podane warunki. Ściśle rzecz biorąc, i-ty wiersz powinien zawierać dwie liczby całkowite*

$A^*_{i,x}$ *i* $A^*_{i,y}$ *oddzielone pojedynczym odstępem — współrzędne punktu* A_i *po przesunięciu* i*-tego kawałka na właściwe miejsce. Zauważ, że lewy dolny róg żagla ma zawsze współrzędnie* $(0,0)$*, a prawy górny róg* (X,Y) *tak jak na rysunku.*

Przykład

Poniższe wejście i wyjście odpowiada rysunkowi (zauważ, że trójkąty po lewej przedstawiają tylko kształt kawałków żagla, a nie ich pozycję).

Wejście

```
6 10 5
4 0 12 4 4 5
4 5 12 4 14 5
0 3 10 3 4 5
4 5 10 3 8 7
7 7 7 10 5 11
0 1 2 0 2 2
```

Wyjście

```
0 0
0 5
0 0
4 2
10 0
8 4
```

Testowanie rozwiązań przed wysłaniem

Do pomocy w szukaniu rozwiązania udostępniono prosty program umożliwiający wizualizację rozwiązania. Program wywołuje się z dwoma parametrami za pomocą polecenia: `draw_sail XX.in XX.out`*, gdzie* `XX.in` *jest plikiem wejściowym, a* `XX.out` *odpowiadającym plikiem wyjściowym. Program wygeneruje i wyświetli plik PostScript pokazujący układ żagla oraz sprawdzi błędy. Pole żagla niepokryte żadnym trójkątem będzie czerwone, pole pokryte przez więcej niż jeden trójkąt będzie niebieskie, a części trójkątów leżące poza prostokątem będą zielone. Pamiętaj, że możesz powiększać i zmniejszać wybrane części w programie* `gv`*, aby zobaczyć szczegóły. Można również drukować bezpośrednio z* `gv`*.*

CEOI 2007, dzień drugi, 05.07.2007

Pokaz lotniczy

Zarząd linii lotniczych Podniebne Gołąby postanowił przygotować pokaz lotniczy jako element swojej kampanii reklamowej. Pokaz odbędzie się na lotnisku wyżej wzmiankowanej korporacji, które posiada N pasów startowych (ponumerowanych od 1 do N). W pokazie będą równolegle uczestniczyć dwie akrobatyczne grupy lotnicze. Oba występy mają z góry ustalony program, więc dla optymalnego efektu wizualnego samoloty muszą startować i lądować na odpowiednich pasach w określonej kolejności.

System ochrony lotniska śledzi, które pasy startowe są w użyciu i nie pozwala, aby jednocześnie korzystały z nich inne samoloty. Oba występy są przedstawiane ochronie w postaci ciągu rezerwacji i zwolnień określonych pasów. Niestety nie jest możliwe stwierdzić zawczasu, kiedy pojawią się powyższe żądania. Władze lotniska chciałyby wiedzieć, czy istnieje niebezpieczeństwo, że występy mogą potoczyć się w ten sposób, że w pewnym momencie nie będzie możliwa ich kontynuacja bez złamania zasad korzystania z pasów startowych.

Zadanie

Twoim zadaniem jest napisać program, który sprawdzi, czy występy mogą przebiegać w taki sposób, że w pewnym momencie nie będzie możliwa ich kontynuacja. Dla każdego z dwóch występów masz daną listę żądań rezerwacji i zwolnień pasów. Dla każdego z występów kolejność żądań spełnia następujące warunki:

- *zarezerwowany pas nie jest rezerwowany do czasu jego zwolnienia,*
- *tylko uprzednio zarezerwowany i niezwolniony pas może być zwolniony oraz*
- *wszystkie zarezerwowane pasy muszą być zwolnione przed końcem występu.*

Jeśli jakiś pas jest potrzebny dla jednego występu i jest on właśnie zarezerwowany przez drugi występ, to pierwszy występ może być wstrzymany do czasu, gdy drugi występ zwolni dany pas. Do tego momentu drugi występ może rezerwować i zwalniać szereg innych pasów.

Jednakże występy mogą „zablokować” lotnisko, jeżeli jeden z nich rezerwuje pas A, jednocześnie trzymając rezerwację pasa B, a drugi rezerwuje pas B, jednocześnie trzymając rezerwację pasa A. W takiej sytuacji żaden z występów nie może być już kontynuowany bez łamania kolejności startów i lądowań.

Zauważ, że Twoim zadaniem **nie** *jest sprawdzenie, czy istnieje taki przebieg występów, który nie zablokuje lotniska.*

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę całkowitą N ($1 \leqslant N \leqslant 1\,000$), oznaczającą liczbę pasów startowych na lotnisku. Po nim następują opisy dwóch występów. Każdy opis zaczyna się wierszem zawierającym parzystą liczbę całkowitą L ($2 \leqslant L \leqslant 5\,000$), oznaczającą liczbę rezerwacji i zwolnień pasów dokonywanych podczas występu. Każdy z kolejnych L wierszy zawiera trzyliterowy napis `RES` *lub* `REL` *i liczbę całkowitą A ($1 \leqslant A \leqslant N$), oddzielone pojedynczym odstępem. Wiersze rozpoczynające się od* `RES` *oznaczają zapotrzebowania na rezerwację pasa, natomiast te, które rozpoczynają się od* `REL`*, oznaczają zwolnienie pasa. Liczba A jest numerem rezerwowanego lub zwalnianego pasa.*

Wyjście

Jeśli występy zawsze kończą się bez blokowania lotniska (niezależnie od tego, kiedy pojawiają się poszczególne żądania), wyjście powinno składać z pojedynczego wiersza zawierającego zdanie „`The performances will always finish.`*”.*

W przypadku, gdy istnieje taki przebieg występów, który prowadzi do blokady lotniska, należy go wypisać (może istnieć wiele takich przebiegów, wtedy należy wypisać dowolny z nich). Opis przebiegu występów reprezentowany jest przez ciąg $a_1 \ldots a_k$ liczb 1 i 2 (bez odstępów między liczbami). Liczby te reprezentują kolejność, w jakiej występują żądania występów (liczby 1 i 2 reprezentują odpowiednio pierwszy i drugi występ): wpierw jest przyjmowane żądanie występu a_1, następnie przyjmowane jest pierwsze niewykonane żądanie występu a_2 (jeżeli $a_1 \neq a_2$, to jest to pierwsze żądanie występu a_2, a jeśli $a_1 = a_2$, to jest to drugie żądanie występu a_2) itd. Ciąg ten musi reprezentować poprawny ciąg operacji w tym sensie, że każdy pas startowy może być zarezerwowany przez co najwyżej jeden występ. Ponieważ ciąg ma opisywać taki przebieg występów, który prowadzi do blokady lotniska, więc następne żądanie pierwszego występu nie znajdujące się w ciągu powinno być rezerwacją pasa zarezerwowanego przez drugi występ i następne żądanie drugiego występu powinno być rezerwacją pasa zarezerwowanego przez pierwszy.

Przykład 1

Wejście

```
2
4
RES 1
RES 2
REL 1
REL 2
4
RES 2
RES 1
REL 2
REL 1
```

Możliwe są dwa wyjścia dla powyższych danych wejściowych.

Wyjście 1

```
12
```

Wyjście 2

```
21
```

Przykład 2

Wejście

```
4
8
RES 1
RES 2
RES 3
REL 3
REL 2
RES 2
REL 1
REL 2
4
RES 3
REL 3
RES 4
REL 4
```

Wyjście

```
The performances will always finish.
```

Tomasz Idziaszek, Jakub Pawlewicz
Tłumaczenie

CEOI 2007, dzień drugi, 05.07.2007

Naszyjniki

I wtedy Alicja powiedziała: — Mam najpiękniejszy naszyjnik na świecie. Od lewej do prawej składa się z dwóch czerwonych pereł, z dwóch zielonych i kolejnej czerwonej.

Beatka szybko dodała: — Mój jest jeszcze lepszy. Wygląda prawie jak Twój, ale musiałabyś usunąć dwie skrajnie prawe perły i zastąpić je niebieskimi.

Za chwilę do dyskusji włączyła się Cecylka: — Żaden z nich nie może się równać z moim naszyjnikiem. Mam o jedną żółtą perłę więcej z lewej strony.

Zapewne nie będziesz zdziwiony, gdy powiem, że Dominika nie przemilczała piękna swojej ozdoby: — To wszystko nic. Aby dostać mój naszyjnik, musiałybyście wziąć naszyjnik Beatki, usunąć po perle z lewej i prawej strony i dodać dwie czarne na lewy koniec.

I tak dyskutowały sobie zawzięcie, aż w pewnym momencie Zuzia zapytała: — Troszkę się pogubiłam. Powiedzcie jakiego koloru była skrajnie lewa perła w naszyjniku Edytki?

Odpowiedziała jej cisza.

Zadanie

Twoim zadaniem jest napisanie **biblioteki** *(modułu w Pascalu), który umożliwi tego typu dysputy. Poniżej jest opisany interfejs tej biblioteki; szablony znajdziesz w katalogach* `/mo/public/necklace/c`, `/mo/public/necklace/cpp` *oraz* `/mo/public/necklace/pas`*. Podkatalogi dla C i C++ zawierają również plik nagłówkowy* `necklace.h`*. Nie ma wejścia ani wyjścia.*

Twój program będzie pracował ze zbiorem **naszyjników**. *Naszyjnik jest ciągiem nieujemnych liczb całkowitych, uporządkowanych od lewej do prawej. Każdy naszyjnik jest identyfikowany przez nieujemną liczbę całkowitą. Początkowo jest dostępny naszyjnik o numerze 0 składający się z pustego ciągu.*

Na samym początku, program oceniający wywoła dokładnie raz procedurę `init`*. Następnie będzie wywoływał w pewnym porządku funkcje* `create` *i* `pearl`*. W sumie, program oceniający odwoła się do biblioteki co najwyżej 1 000 000 razy w każdym teście.*

Wywołanie procedury `create` *oznacza utworzenie nowego naszyjnika. Numer nowego naszyjnika jest o jeden większy od największego numeru spośród już istniejących naszyjników, tzn. pierwsze wywołanie* `create` *tworzy naszyjnik o numerze 1, drugie wywołanie tworzy naszyjnik o numerze 2 itd. Nowy naszyjnik jest utworzony z naszyjnika* `from` *poprzez operację opisaną parametrami* `operation`, `on_left` *i* `param`*:*

- *Jeżeli parametr* `operation` *jest literą* `R` *(ang.* **remove***), jedna liczba jest usuwana z odpowiedniego końca naszyjnika. Parametr* `param` *jest w tym przypadku ignorowany.*
- *Jeżeli parametr* `operation` *jest literą* `A` *(ang.* **add***), liczba całkowita* `param` *jest dodawana na odpowiednim końcu naszyjnika.*

Jeśli parametr `on_left` *jest prawdą (jest niezerowy w C/C++), wtedy powyższa operacja jest wykonywana na lewym końcu naszyjnika, w przeciwnym wypadku jest wykonywana na prawym końcu. Parametr* `from` *w wywołaniu procedury* `create` *będzie zawsze mniejszy niż liczba naszyjników utworzonych przez tę procedurę, tzn. będzie odnosić się do istniejącego naszyjnika. Możesz założyć, że nie będzie prób usuwania perły z pustego naszyjnika.*

Funkcja `pearl` *powinna zwrócić liczbę znajdującą się na lewym końcu naszyjnika* `neck_id` *jeżeli parametr* `on_left` *jest prawdą (jest niezerowy w C/C++), a w przeciwnym wypadku liczbę na prawym końcu naszyjnika* `neck_id`*. Funkcja* `pearl` *będzie wywoływana tylko dla niepustych naszyjników, które zostały wcześniej utworzone przez wywołanie procedury* `create`*. Ta funkcja nie zmienia naszyjników.*

Opis interfejsu

W języku C/C++

```
extern void init (void);
extern void create (int from, char operation, int on_left,
        int param);
extern int pearl (int neck_id, int on_left);
```

W języku Pascal

```
unit necklace;
interface
```

```
procedure init;
procedure create (from : longint; operation : char; on_left : boolean;
       param : longint);
function pearl (neck_id : longint; on_left : boolean) : longint;
```

Pamiętaj, że w dostarczanej przez siebie bibliotece możesz utworzyć dodatkowe procedury, funkcje i zmienne globalne

Przykład

Poniższy przykład pokazuje możliwy ciąg wywołań funkcji i zwracanych przez nie wartości:

W języku C/C++

```
init ();
create (0, ’A’, 1, 5);
create (1, ’A’, 1, 3);
pearl (2, 0);  /* zwraca 5 */
create (2, ’R’, 0, 0);
pearl (3, 0);  /* zwraca 3 */
pearl (2, 0);  /* zwraca 5 */
```

W języku Pascal

```
init;
create (0, ’A’, true, 5);
create (1, ’A’, true, 3);
pearl (2, false);  { zwraca 5 }
create (2, ’R’, false, 0);
pearl (3, false);  { zwraca 3 }
pearl (2, false);  { zwraca 5 }
```

Testowanie Twojej biblioteki

Oprócz szablonów dla Twojej biblioteki, podkatalogi katalogu `/mo/public/necklace` *zawierają pliki* `neck_main.c`, `neck_main.cpp` *i* `neck_main.pas`, *będące plikami źródłowymi programów, które mogą zostać skompilowane z Twoją biblioteką i będą wywoływać jej funkcje w sposób pokazany na przykładzie (programy powiadomią Cię, czy wartości zwracane przez funkcję* `pearl` *są poprawne).*

Skrypt `compile` *skompiluje Twoją bibliotekę, i utworzony tym sposobem program* `necklace` *pozwoli Ci przetestować Twoją bibliotekę w sposób opisany poniżej. Jeżeli program jest uruchomiony bez parametrów, użyje Twojej biblioteki w sposób opisany w przykładzie i sprawdzi, czy wartości zwracane przez funkcję* `pearl` *są poprawne (jeśli nie, na standardowe wyjście błędów zostanie wypisany komunikat). Program można również uruchomić w* **trybie interakcyjnym** *wywołując go z parametrem* `-i`. *W tym trybie zapytania wczytywane są ze standardowego wejścia, przekazywane do biblioteki, a odpowiedzi wypisywane są na standardowe wyjście.*

Program na początku wywołuje procedurę `init` *i, po jej zakończeniu, wypisuje* `Initiated`. *Następnie oczekuje zapytań (każdego w nowym wierszu); możliwe są następujące rodzaje:*

- `create` from operation on_left param
  To zapytanie powoduje, że program wywołuje procedurę `create` *z parametrami* **from**, **operation**, on_ left *oraz* **param**. *Po zakończeniu procedury, program wypisuje* `Done`.
- `pearl` neck_id on_left
  To zapytanie powoduje, że program wywołuje funkcję `pearl` *z parametrami* neck_id *oraz* on_left. *Po zakończeniu funkcji, program wypisuje* `Returns` **X**, *gdzie* **X** *jest liczbą zwracaną przez funkcję.*
- `quit`
  To zapytanie powoduje zakończenie działania programu.

Zauważ, że program w trybie interakcyjnym tylko wypisuje odpowiedzi zwracane przez Twoją bibliotekę, natomiast nie sprawdza, czy są one poprawne.

Poniżej przedstawione zostały zapytania i odpowiedzi programu w trybie interakcyjnym, odpowiadające ciągowi wywołań z przykładu.

W języku C/C++

```
Initiated
create 0 A 1 5
Done
create 1 A 1 3
Done
pearl 2 0
Returns 5
create 2 R 0 0
Done
pearl 3 0
Returns 3
pearl 2 0
Returns 5
quit
```

W języku Pascal

```
Initiated
create 0 A true 5
Done
create 1 A true 3
Done
pearl 2 false
Returns 5
create 2 R false 0
Done
pearl 3 false
Returns 3
pearl 2 false
Returns 5
quit
```

Tomasz Idziaszek, Jakub Pawlewicz
Tłumaczenie

CEOI 2007, dzień drugi, 05.07.2007

Królewski skarbiec

Pewnego razu, w odległym królestwie, skarbiec zaczął świecić pustkami. Król postanowił położyć kres tej sytuacji i obmyślił nowy sposób współpracy z urzędnikami Ministerstwa Skarbu. Urzędnicy mają podzielić się w pary (przekupić dwóch urzędników jest trudniej i drożej), a każda para ma składać się z urzędnika i jego bezpośredniego podwładnego. Twoim zadaniem jest wyznaczyć, mając daną strukturę Ministerstwa Skarbu, największą liczbę par, które mogą zostać utworzone w ten sposób, jak również na ile sposobów można to zrobić.

Zadanie

Ministerstwo Skarbu jest zarządzane przez Ryszarda Zimnoskórego. Każdy urzędnik ma pod sobą kilku podwładnych, a zarazem jest podwładnym dokładnie jednego urzędnika (za wyjątkiem Zimnoskórego, który odpowiada jedynie przed samym królem). Liczba urzędników nie przekracza 1 000. Twoim zadaniem jest obliczyć największą liczbę par, w które można pogrupować urzędników w taki sposób, że każda para składa się z urzędnika i jego bezpośredniego podwładnego. Dodatkowo, powinieneś obliczyć, na ile sposobów można dokonać takiego pogrupowania. Zauważ, że nie wszyscy urzędnicy Ministerstwa muszą zostać sparowani.

Wejście

W pierwszym wierszu wejścia znajduje się liczba N *oznaczająca liczbę urzędników* ($1 \leqslant N \leqslant 1\,000$)*. Urzędnicy mają przypisane unikalne numery z zakresu od 1 do* N*. Ryszard Zimnoskóry ma przypisany numer* 1*. Każdy z kolejnych* N *wierszy opisuje pojedynczego urzędnika: zawiera jego numer, liczbę* K *jego podwładnych* ($0 \leqslant K \leqslant 999$) *i numery jego podwładnych, pooddzielane pojedynczymi odstępami. Możesz założyć, że wiersz opisujący danego urzędnika nie pojawi się przed wierszem opisującym jego przełożonego.*

Wyjście

Wyjście powinno składać się z dokładnie dwóch wierszy. W pierwszym wierszu należy wypisać jedną liczbę M *oznaczającą największą liczbę par, które mogą utworzyć urzędnicy. Drugi wiersz powinien zawierać liczbę sposobów utworzenia przez urzędników* M *par (zgodnie z powyższymi wymaganiami króla).*

Punktacja

Za to zadanie możesz zdobyć punkty częściowe. Jeżeli Twój program poprawnie wyznaczy największą liczbę par, które mogą utworzyć urzędnicy lub liczbę sposobów utworzenia tych par, ale niepoprawnie wyznaczy drugą z tych liczb, otrzymasz 40% punktów przyznawanych za dany test.

Przykład

Wejście

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

Wyjście

```
3
4
```

Bibliografia

[1] *I Olimpiada Informatyczna 1993/1994*. Warszawa–Wrocław, 1994.

[2] *II Olimpiada Informatyczna 1994/1995*. Warszawa–Wrocław, 1995.

[3] *III Olimpiada Informatyczna 1995/1996*. Warszawa–Wrocław, 1996.

[4] *IV Olimpiada Informatyczna 1996/1997*. Warszawa, 1997.

[5] *V Olimpiada Informatyczna 1997/1998*. Warszawa, 1998.

[6] *VI Olimpiada Informatyczna 1998/1999*. Warszawa, 1999.

[7] *VII Olimpiada Informatyczna 1999/2000*. Warszawa, 2000.

[8] *VIII Olimpiada Informatyczna 2000/2001*. Warszawa, 2001.

[9] *IX Olimpiada Informatyczna 2001/2002*. Warszawa, 2002.

[10] *X Olimpiada Informatyczna 2002/2003*. Warszawa, 2003.

[11] *XI Olimpiada Informatyczna 2003/2004*. Warszawa, 2004.

[12] *XII Olimpiada Informatyczna 2004/2005*. Warszawa, 2005.

[13] *XIII Olimpiada Informatyczna 2005/2006*. Warszawa, 2006.

[14] A. V. Aho, J. E. Hopcroft, J. D. Ullman. *Projektowanie i analiza algorytmów komputerowych*. PWN, Warszawa, 1983.

[15] L. Banachowski, A. Kreczmar. *Elementy analizy algorytmów*. WNT, Warszawa, 1982.

[16] L. Banachowski, K. Diks, W. Rytter. *Algorytmy i struktury danych*. WNT, Warszawa, 1996.

[17] J. Bentley. *Perełki oprogramowania*. WNT, Warszawa, 1992.

[18] I. N. Bronsztejn, K. A. Siemiendiajew. *Matematyka. Poradnik encyklopedyczny*. PWN, Warszawa, wydanie XIV, 1997.

[19] T. H. Cormen, C. E. Leiserson, R. L. Rivest. *Wprowadzenie do algorytmów*. WNT, Warszawa, 1997.

[20] D. Harel. *Algorytmika. Rzecz o istocie informatyki*. WNT, Warszawa, 1992.

[21] J. E. Hopcroft, J. D. Ullman. *Wprowadzenie do teorii automatów, języków i obliczeń*. PWN, Warszawa, 1994.

[22] W. Lipski. *Kombinatoryka dla programistów*. WNT, Warszawa, 2004.

[23] E. M. Reingold, J. Nievergelt, N. Deo. *Algorytmy kombinatoryczne*. WNT, Warszawa, 1985.

[24] R. Sedgewick. *Algorytmy w C++. Grafy*. RM, 2003.

[25] M. M. Sysło. *Algorytmy*. WSiP, Warszawa, 1998.

[26] M. M. Sysło. *Piramidy, szyszki i inne konstrukcje algorytmiczne*. WSiP, Warszawa, 1998.

[27] M. M. Sysło, N. Deo, J. S. Kowalik. *Algorytmy optymalizacji dyskretnej z programami w języku Pascal*. PWN, Warszawa, 1993.

[28] R. J. Wilson. *Wprowadzenie do teorii grafów*. PWN, Warszawa, 1998.

[29] N. Wirth. *Algorytmy + struktury danych = programy*. WNT, Warszawa, 1999.

[30] Ronald L. Graham, Donald E. Knuth, Oren Patashnik. *Matematyka konkretna*. PWN, Warszawa, 1996.

[31] K. A. Ross, C. R. B. Wright. *Matematyka dyskretna*. PWN, Warszawa, 1996.

[32] Donald E. Knuth. *Sztuka programowania*. WNT, Warszawa, 2002.

[33] Steven S. Skiena, Miguel A. Revilla. *Wyzwania programistyczne*. WSiP, Warszawa, 2004.

[34] Witold Lipski, Witold Marek. *Analiza kombinatoryczna*. PWN, 1986.

[35] John H. Conway, Richard K. Guy. *Księga liczb*. WNT, 2004.

[36] Vijay V. Vazirani. *Algorytmy aproksymacyjne*. WNT, 2004.

Niniejsza publikacja stanowi wyczerpujące źródło informacji o zawodach XIV Olimpiady Informatycznej, przeprowadzonych w roku szkolnym 2006/2007. Książka zawiera informacje dotyczące organizacji, regulaminu oraz wyników zawodów. Zawarto w niej także opis rozwiązań wszystkich zadań konkursowych. Programy wzorcowe w postaci elektronicznej i testy użyte do sprawdzania rozwiązań zawodników będą dostępne na stronie Olimpiady Informatycznej: www.oi.edu.pl.

Książka zawiera też zadania z XVIII i XIX Międzynarodowej Olimpiady Informatycznej, XIII Bałtyckiej Olimpiady Informatycznej oraz XIV Olimpiady Informatycznej Krajów Europy Środkowej.

XIV 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 978–83–922946–2–7