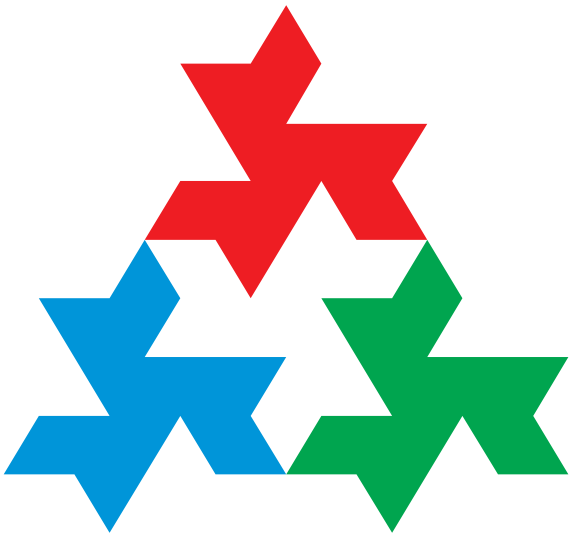
MINISTERSTWO EDUKACJI NARODOWEJ
UNIWERSYTET WROCŁAWSKI
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ



XV OLIMPIADA INFORMATYCZNA
2007/2008

Olimpiada Informatyczna jest organizowana przy współudziale

WARSZAWA, 2008

MINISTERSTWO EDUKACJI NARODOWEJ
UNIWERSYTET WROCŁAWSKI
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

XV OLIMPIADA INFORMATYCZNA
2007/2008

WARSZAWA, 2008

Autorzy tekstów:

Karol Cwalina
prof. dr hab. Zbigniew Czech
prof. dr hab. Krzysztof Diks
Marian M. Kędzierski
dr Marcin Kubica
Michał Pilipczuk
mgr Jakub Radoszewski
prof. dr hab. Wojciech Rytter
mgr Piotr Stańczyk
mgr Tomasz Waleń
Filip Wolski
Piotr Zieliński

Autorzy programów:

mgr Marek Cygan
mgr Krzysztof Dulęba
mgr Adam Iwanicki
mgr Anna Niewiarowska
mgr Jakub Pawlewicz
Michał Pilipczuk
mgr Jakub Radoszewski
mgr Piotr Stańczyk
Wojciech Tyczyński
Filip Wolski

Opracowanie i redakcja:

mgr Adam Iwanicki
dr Przemysława Kanarek
mgr Jakub Radoszewski

Opracowanie i redakcja treści zadań:

dr Marcin Kubica

Skład:

mgr Adam Iwanicki

Tłumaczenie treści zadań:

Tomasz Kulczyński
Jakub Łącki
mgr Jakub Radoszewski

Pozycja dotowana przez Ministerstwo Edukacji Narodowej.

Druk książki został sfinansowany przez 

ISBN 978–83–922946–4–1

Spis treści

Wstęp

Drogi Czytelniku!

Olimpiada Informatyczna ma już piętnaście lat. Została powołana 10 grudnia 1993 roku i od tego czasu w zawodach Olimpiady wzięło udział ponad 10 000 uczniów. Dzięki ich umiejętnościom i pracy, zaangażowaniu wielu wspaniałych nauczycieli, studentów, pracowników naukowych i przyjaciół Olimpiady, nasi olimpijczycy zdobyli w tym czasie w zawodach Międzynarodowej Olimpiady Informatycznej 23 złote medale, 23 srebrne medale i 17 brązowych medali. Dwukrotnie Polacy wygrywali te prestiżowe, międzynarodowe zawody — Filip Wolski w Meksyku w roku 2006 i Tomek Kulczyński w Chorwacji w roku 2007. Nie gorzej w roku 2008 spisali się laureaci XV Olimpiady Informatycznej. W XX Międzynarodowej Olimpiadzie Informatycznej (Kair, 16–23.08.2008 r.) Polacy zdobyli 3 złote i 1 srebrny medal, zajmując w klasyfikacji medalowej pierwsze miejsce wspólnie z Chinami. Indywidualnie Marcin Kościelnicki zajął trzecie miejsce (złoty medal), Marcin Andrychowicz miejsce 14. (złoty medal), Jarosław Błasiok miejsce 18. (złoty medal), Maciej Klimek miejsce 26. (srebrny medal). Bardzo dobrze Polacy wypadli także na Olimpiadzie Krajów Europy Środkowej (Marcin Andrychowicz — złoto, Marcin Kościelnicki — srebro oraz Jarosław Błasiok — brąz) i Olimpiadzie Krajów Bałtyckich (złote medale: Adam Polak, Tomasz Kłeczek, Jarosław Błasiok; srebrne medale: Mateusz Litwin, Anna Piekarska, Adam Iwaniuk).

Mam nadzieję, że uczestnicy XVI Olimpiady Informatycznej pójdą w ślady swoich poprzedników i osiągną w roku 2008/2009 nie gorsze wyniki. Na pewno pomocna w tym będzie ta książeczka, która zawiera szczegółową informację o przebiegu XV Olimpiady oraz starannie zredagowane opisy rozwiązań zadań konkursowych. Słowa podziękowania należą się autorom za interesujące i wyzywające intelektualnie zadania, jurorom za przygotowanie rozwiązań wzorcowych oraz redaktorom — Przemce Kanarek, Adamowi Iwanickiemu i Jakubowi Radoszewskiemu — za serce i olbrzymi wkład pracy w nadanie ostatecznego kształtu temu wydawnictwu.

W tym roku nastąpiły zmiany w Komitecie Głównym Olimpiady. Chciałbym gorąco podziękować doktorowi Andrzejowi Walatowi za piętnaście lat wspaniałej pracy na rzecz Olimpiady oraz doktor Mirosławie Skowrońskiej i profesorowi Jerzemu Nawrockiemu za serce i zaangażowanie w nasz ruch olimpijski. Od tego roku pan prof. Krzysztof Stencel przestaje pełnić obowiązki szefa jurorów Olimpiady Informatycznej. Będzie nam wszystkim brakowało jego gorących dysput z zawodnikami. Szczęśliwie nadal będzie pracował w Komitecie Głównym. Na koniec chciałbym powitać nowych członków Komitetu Głównego i życzyć im owocnej pracy: Szymona Acedańskiego, dr. Piotra Formanowicza, dr Barbarę Klunder i mgr. Jakuba Radoszewskiego.

Krzysztof Diks

Sprawozdanie z przebiegu XV Olimpiady Informatycznej w roku szkolnym 2007/2008

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 2007/2008 odbyły się zawody XV 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.

15 października 2007 r. rozesłano plakaty zawierające zasady organizacji zawodów I stopnia oraz zestaw 5 zadań konkursowych do 3721 szkół i zespołów szkół młodzieżowych ponadgimnazjalnych oraz do wszystkich kuratorów i koordynatorów edukacji informatycznej. Zawody I stopnia rozpoczęły się 22 października 2007 r. Ostatecznym terminem nadsyłania prac konkursowych był 19 listopada 2007 r.

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 i Wrocławiu oraz w Sopocie w dniach 06–08.02.2008 r., natomiast zawody III stopnia odbyły się w ośrodku firmy Combidata Poland SA w Sopocie, w dniach 01–05.04.2008r.

Uroczystość zakończenia XV Olimpiady Informatycznej odbyła się 05.04.2008 r. w Auli III Liceum Ogólnokształcącego im. Marynarki Wojennej RP w Gdyni.

SKŁAD OSOBOWY KOMITETÓW OLIMPIADY INFORMATYCZNEJ

Komitet Główny

przewodniczący:
prof. dr hab. Krzysztof Diks (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:
Szymon Acedański (student Uniwersytetu Warszawskiego)
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)
prof. dr hab. Krzysztof Loryś (Uniwersytet Wrocławski)
prof. dr hab. Jan Madey (Uniwersytet Warszawski)
dr hab. inż. Jerzy Nawrocki, prof. PP (Politechnika Poznańska)
Jakub Radoszewski (student Uniwersytetu Warszawskiego)
prof. dr hab. Wojciech Rytter (Uniwersytet Warszawski)
mgr Krzysztof J. Święcicki (emerytowany profesor Uniwersytetu Warszawskiego)
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 Łukasz Kowalik (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.

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:
mgr Grzegorz Gutowski (Uniwersytet Jagielloński)

dr Marcin Kozik (Uniwersytet Jagielloński)
mgr Jan Pawłowski (Kuratorium Oświaty w Krakowie)

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

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

Komitet Okręgowy w Poznaniu:
przewodniczący:
dr hab. inż. Małgorzata Sterna (Politechnika Poznańska)
zastępca przewodniczącego:
dr Jacek Marciniak (Uniwersytet Adama Mickiewicza w Poznaniu)
sekretarz:
mgr inż. Przemysław Wesołek (Politechnika Poznańska)
członkowie:
Hanna Ćwiek (Politechnika Poznańska)
mgr inż. Piotr Gawron (Politechnika Poznańska)
dr Maciej Machowiak (Politechnika Poznańska)
dr inż. Maciej Miłostan (Politechnika Poznańska)
Michał Połetek (Politechnika Poznańska)
Szymon Wąsik (Politechnika Poznańska)

Siedzibą Komitetu Okręgowego w Poznaniu jest Instytut Informatyki Politechniki Poznańskiej, ul. Piotrowo 2.
Strona internetowa Komitetu Okręgowego: http://www.cs.put.poznan.pl/oi/.

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:

mgr Michał Adamaszek
Kamil Anikiej
Michał Brzozowski
mgr Marek Cygan
Konrad Gołuchowski
mgr Adam Iwanicki
Marian Kędzierski
Tomasz Kulczyński
Karol Kurach
Jakub Łącki
Marek Marczykowski
Piotr Niedźwiedź
mgr Anna Niewiarowska
mgr Paweł Parys
mgr Jakub Pawlewicz
Krzysztof Pawłowski
mgr Marcin Pilipczuk
Michał Pilipczuk
Wojciech Śmietanka
Wojciech Tyczyński
Filip Wolski

ZAWODY I STOPNIA

W zawodach I stopnia XV Olimpiady Informatycznej wzięło udział 1000 zawodników. Decyzją Komitetu Głównego zdyskwalifikowano 26 zawodników. Ponadto dwóm zawodnikom nie uznano rozwiązań jednego z zadań. Powodem dyskwalifikacji była niesamodzielność rozwiązań zadań konkursowych.

Decyzją Komitetu Głównego do zawodów zostało dopuszczonych 44 uczniów gimnazjów. Byli to uczniowie z następujących szkół:

• Gimnazjum nr 24 przy III LO	Gdynia	7 uczniów
• Gimnazjum nr 16	Szczecin	7
• Gimnazjum nr 59	Warszawa	3
• Gimnazjum Dwujęzyczne nr 49	Wrocław	3
• Gimnazjum nr 50	Bydgoszcz	2
• Gimnazjum Akademickie	Toruń	2
• Gimnazjum nr 13	Warszawa	2
• Publiczne Gimnazjum nr 1	Białystok	1 uczeń
• Gimnazjum nr 4	Bielsko–Biała	1
• Gimnazjum nr 53	Bydgoszcz	1
• Gimnazjum nr 1 im. KEN	Ciechanów	1
• Gimnazjum nr 7	Gdańsk	1
• Gimnazjum nr 24	Katowice	1
• VI Publiczne Gimnazjum Akademickie	Kraków	1
• Gimnazjum nr 18	Kraków	1
• Gimnazjum nr 33	Kraków	1
• Gimnazjum nr 1	Mysłowice	1
• Gimnazjum Akademickie WSIiZ	Rzeszów	1
• Publiczne Gimnazjum nr 4	Siedlce	1
• Publiczne Gimnazjum	Skórcz	1
• Gimnazjum nr 4	Tomaszów Mazowiecki	1
• Gimnazjum	Trzebownisko	1
• Gimnazjum nr 14 im. L. Staffa	Warszawa	1
• Gimnazjum nr 5 SKiE	Warszawa	1
• Publiczne Gimnazjum nr 2	Ząbki	1

Kolejność województw pod względem liczby uczestników była następująca:

małopolskie	151 zawodników	zachodniopomorskie	53
mazowieckie	133	podlaskie	48
pomorskie	111	łódzkie	31
śląskie	103	lubelskie	30
dolnośląskie	86	świętokrzyskie	23
podkarpackie	69	warmińsko–mazurskie	15
wielkopolskie	65	opolskie	14
kujawsko–pomorskie	55	lubuskie	13

W zawodach I stopnia najliczniej reprezentowane były szkoły:

V LO im. Augusta Witkowskiego	Kraków	73 uczniów
III LO im. Marynarki Wojennej RP	Gdynia	47
XIV LO im. Stanisława Staszica	Warszawa	42
XIV LO im. Polonii Belgijskiej	Wrocław	34
I LO im. Adama Mickiewicza	Białystok	25
XIII Liceum Ogólnokształcące	Szczecin	19
VIII LO im. Adama Mickiewicza	Poznań	17
VIII LO im. Marii Skłodowskiej–Curie	Katowice	14
V Liceum Ogólnokształcące	Bielsko–Biała	12
VI LO im. J. i J. Śniadeckich	Bydgoszcz	12
VIII LO im. S. Wyspiańskiego	Kraków	12
V LO im. S. Żeromskiego	Gdańsk	11
IV LO im. Tadeusza Kościuszki	Toruń	11
I LO im. Marcina Kromera	Gorlice	10
I Liceum Ogólnokształcące	Legnica	9
Ogólnokształcące Liceum Jezuitów	Gdynia	8
VI LO im. Jana Kochanowskiego	Radom	8
XXVII LO im. T. Czackiego	Warszawa	8
I LO im. Mikołaja Kopernika	Krosno	7
I LO im. T. Kościuszki	Łomża	7
Gimnazjum nr 16	Szczecin	7
Gimnazjum i Liceum Akademickie	Toruń	7
Gimnazjum nr 24 przy III LO	Gdynia	6
I LO im. B. Nowodworskiego	Kraków	6
II LO im. Jana III Sobieskiego	Kraków	6
I LO im. Stanisława Staszica	Lublin	6
I LO im. Mikołaja Kopernika	Łódź	6
V Liceum Ogólnokształcące	Szczecin	6
LO im. Bogusława X	Białogard	5
I LO im. Władysława Jagiełły	Dębica	5
I LO im. Stefana Żeromskiego	Lębork	5
Zespół Szkół Komunikacji	Poznań	5
III LO im. św. Jana Kantego	Poznań	5
LO WSIiZ	Rzeszów	5

I LO im. B. Krzywoustego	Słupsk	5
VIII LO im. Władysława IV	Warszawa	5
VI LO im. Tadeusza Reytana	Warszawa	5
I LO im. C. K. Norwida	Bydgoszcz	4
I LO im. Juliusza Słowackiego	Chorzów	4
IV LO im. H. Sienkiewicza	Częstochowa	4
Gdańskie Liceum Autonomiczne	Gdańsk	4
I LO im. Mikołaja Kopernika	Gdańsk	4
I LO im. Mikołaja Kopernika	Jarosław	4
I LO im. K. S. Leszczyńskiego	Jasło	4
I LO im. Stefana Żeromskiego	Kielce	4
II LO im. K. K. Baczyńskiego	Konin	4
II LO im. Mikołaja Kopernika	Mielec	4
Zespół Szkół Elektronicznych	Radom	4
LO im. Komisji Edukacji Narodowej	Stalowa Wola	4
II LO im. Stanisława Staszica	Tarnowskie Góry	4
III LO im. Adama Mickiewicza	Tarnów	4
II Liceum Ogólnokształcące	Tarnów	4
LO im. M. Wadowity	Wadowice	4
X Liceum Ogólnokształcące	Wrocław	4
I LO Ogólnokształcące	Bełchatów	3
ZSZ nr 1 im. KEN	Biała Podlaska	3
III LO im. K. K. Baczyńskiego	Białystok	3
Zespół Szkół Elektryczno–Mechanicznych	Bielsko–Biała	3
Zespół Szkół Zawodowych	Brodnica	3
III LO im. Adama Mickiewicza	Bydgoszcz	3
II LO im. gen. W. Andersa	Chojnice	3
IX LO im. C. K. Norwida	Częstochowa	3
II Liceum Ogólnokształcące	Gdańsk	3
Zespół Szkół Techniczno–Informatycznych	Gliwice	3
V Liceum Ogólnokształcące	Gliwice	3
II LO im. Marii Skłodowskiej–Curie	Gorzów Wlkp.	3
I LO im. J. Kasprowicza	Inowrocław	3
I LO im. Stanisława Staszica	Jastrzębie Zdrój	3
Zespół Szkół Elektrycznych	Kielce	3
IV LO im. Hanki Sawickiej	Kielce	3
I LO im. Władysława Orkana	Limanowa	3
I Liceum Ogólnokształcące	Lubin	3
II LO im. Hetmana Zamoyskiego	Lublin	3
I LO im. Jana Długosza	Nowy Sącz	3
II LO im. Marii Konopnickiej	Nowy Sącz	3
IV Liceum Ogólnokształcące	Olsztyn	3
III LO im. M. Dąbrowskiej	Płock	3
I LO im. Karola Marcinkowskiego	Poznań	3
I LO im. J. Kasprowicza	Racibórz	3
Zespół Szkół Mechanicznych	Racibórz	3

II LO im. A. Frycza–Modrzewskiego	Rybnik	3
IV LO Mikołaja Kopernika	Rzeszów	3
Zespół Szkół Elektronicznych	Rzeszów	3
I LO im. Marii Konopnickiej	Suwałki	3
I LO im. Jana Kasprowicza	Świdnica	3
I LO im. K. Brodzińskiego	Tarnów	3
Zespół Szkół nr 8	Wałbrzych	3
I Liceum Ogólnokształcące	Wałbrzych	3
I SLO „Bednarska”	Warszawa	3
Gimnazjum nr 59	Warszawa	3
IX Liceum Ogólnokształcące	Wrocław	3
I Liceum Ogólnokształcące	Zielona Góra	3
Społeczne LO	Żary	3
Zespół Szkół Mechaniczno–Elektrycznych	Żywiec	3

Najliczniej reprezentowane były miasta:

Kraków	110 zawodników	Dębica	5 zawodników
Warszawa	96	Inowrocław	5
Gdynia	66	Jarosław	5
Wrocław	50	Konin	5
Szczecin	40	Mielec	5
Poznań	37	Skierniewice	5
Białystok	36	Wadowice	5
Gdańsk	26	Biała Podlaska	4
Bydgoszcz	23	Gorzów Wlkp.	4
Toruń	21	Jasło	4
Katowice	20	Koszalin	4
Bielsko–Biała	18	Limanowa	4
Lublin	14	Olsztyn	4
Radom	14	Rybnik	4
Rzeszów	14	Stalowa Wola	4
Częstochowa	13	Świdnica	4
Tarnów	13	Tarnowskie Góry	4
Gliwice	12	Wodzisław Śląski	4
Kielce	12	Zabrze	4
Łódź	12	Bełchatów	3
Legnica	11	Brodnica	3
Gorlice	10	Chełm	3
Opole	10	Chojnice	3
Łomża	8	Elbląg	3
Nowy Sącz	8	Jastrzębie Zdrój	3
Krosno	7	Lubin	3
Słupsk	7	Milanówek	3
Zielona Góra	7	Ostrowiec Św.	3
Lębork	6	Przemyśl	3

Płock	6	Sanok	3
Racibórz	6	Suwałki	3
Wałbrzych	6	Żary	3
Białogard	5	Żywiec	3
Chorzów	5		

Zawodnicy uczęszczali do następujących klas:

do klasy I gimnazjum	1
do klasy II gimnazjum	7
do klasy III gimnazjum	36
do klasy I szkoły średniej	210
do klasy II szkoły średniej	390
do klasy III szkoły średniej	249
do klasy IV szkoły średniej	7

W zawodach I stopnia zawodnicy mieli do rozwiązania pięć zadań: „Cło", „Klocki", „Plakatowanie", „Robinson" i „Szklana pułapka".

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.

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

- **CLO** — Cło

	CLO	
	liczba zawodników	czyli
100 pkt.	233	23,3%
75–99 pkt.	28	2,8%
50–74 pkt.	42	4,2%
1–49 pkt.	100	10,0%
0 pkt.	383	38,3%
brak rozwiązania	214	21,4%

- **KLO** — Klocki

	KLO	
	liczba zawodników	czyli
100 pkt.	183	18,3%
75–99 pkt.	55	5,5%
50–74 pkt.	27	2,7%
1–49 pkt.	406	40,6%
0 pkt.	166	16,6%
brak rozwiązania	163	16,3%

- **PLA** — Plakatowanie

	PLA	
	liczba zawodników	czyli
100 pkt.	450	45,0%
75–99 pkt.	10	1,0%
50–74 pkt.	2	0,2%
1–49 pkt.	314	31,4%
0 pkt.	188	18,8%
brak rozwiązania	36	3,6%

- **ROB** — Robinson

	ROB	
	liczba zawodników	czyli
100 pkt.	19	1,9%
75–99 pkt.	46	4,6%
50–74 pkt.	65	6,5%
1–49 pkt.	128	12,8%
0 pkt.	115	11,5%
brak rozwiązania	627	62,7%

- **SZK** — Szklana pułapka

	SZK	
	liczba zawodników	czyli
100 pkt.	129	12,9%
75–99 pkt.	18	1,8%
50–74 pkt.	9	0,9%
1–49 pkt.	132	13,2%
0 pkt.	61	6,1%
brak rozwiązania	651	65,1%

W sumie za wszystkie 5 zadań konkursowych:

SUMA	liczba zawodników	czyli
500 pkt.	13	1,3%
375–499 pkt.	101	10,1%
250–374 pkt.	129	12,9%
125–249 pkt.	200	20,0%
1–124 pkt.	420	42,0%
0 pkt.	137	13,7%

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 392 zawodników, którzy osiągnęli w zawodach I stopnia wynik nie mniejszy niż 145 pkt.

Czterech zawodników nie stawiło się na zawody. W zawodach II stopnia uczestniczyło 388 zawodników.

Zawody II stopnia odbyły się w dniach 6–8 lutego 2008 r. w siedmiu stałych okręgach oraz w Sopocie:

- w Gliwicach — 26 zawodników z następujących województw:
  - śląskie (26)
- w Krakowie — 65 zawodników z następujących województw:
  - małopolskie (65)
- w Poznaniu — 38 zawodników z następujących województw:
  - lubuskie (2)
  - łódzkie (6)
  - wielkopolskie (11)
  - zachodniopomorskie (19)

- w Rzeszowie — 45 zawodników z następujących województw:
  - lubelskie (8)
  - małopolskie (7)
  - podkarpackie (21)
  - śląskie (2)
  - świętokrzyskie (7)
- w Toruniu — 27 zawodników z następujących województw:
  - kujawsko–pomorskie (18)
  - mazowieckie (2)
  - podlaskie (5)
  - warmińsko–mazurskie (2)
- w Warszawie — 73 zawodników z następujących województw:
  - lubelskie (1)
  - łódzkie (1)
  - mazowieckie (51)
  - podkarpackie (1)
  - podlaskie (19)
- we Wrocławiu — 60 zawodników z następujących województw:
  - dolnośląskie (37)
  - lubuskie (2)
  - łódzkie (4)
  - opolskie (3)
  - śląskie (14)
- w Sopocie — 53 zawodników z następujących województw:
  - pomorskie (52)
  - warmińsko–mazurskie (1)

W zawodach II stopnia najliczniej reprezentowane były szkoły:

V LO im. Augusta Witkowskiego	Kraków	52 uczniów
III LO im. Marynarki Wojennej RP	Gdynia	33
XIV LO im. Stanisława Staszica	Warszawa	29
XIV LO im. Polonii Belgijskiej	Wrocław	22
I LO im. A. Mickiewicza	Białystok	18
XIII Liceum Ogólnokształcące	Szczecin	11
VIII Liceum Ogólnokształcące	Katowice	8
VI LO im. J. i J. Śniadeckich	Bydgoszcz	7
V Liceum Ogólnokształcące	Bielsko–Biała	6
I Liceum Ogólnokształcące	Legnica	5
VI LO im. Jana Kochanowskiego	Radom	5
XXVII LO im. T. Czackiego	Warszawa	5
I LO im. C. K. Norwida	Bydgoszcz	4
Ogólnokształcące Liceum Jezuitów	Gdynia	4
II LO im. Jana III Sobieskiego	Kraków	4
I LO im. Mikołaja Kopernika	Łódź	4
III LO im. Adama Mickiewicza	Tarnów	4
I Liceum Ogólnokształcące	Bełchatów	3
Gimnazjum nr 24	Gdynia	3
V Liceum Ogólnokształcące	Gliwice	3
I LO im. Marcina Kromera	Gorlice	3
I LO im. Mikołaja Kopernika	Jarosław	3
I LO im. Stefana Żeromskiego	Kielce	3

VIII LO im. S. Wyspiańskiego	Kraków	3
I LO im. B. Nowodworskiego	Kraków	3
PLO 2 w Opolu	Opole	3
LO WSIiZ	Rzeszów	3
VIII LO im. Władysława IV	Warszawa	3

Najliczniej reprezentowane były miasta:

Kraków	65 zawodników	Tarnów	6
Warszawa	44	Legnica	5
Gdynia	42	Gdańsk	4
Wrocław	28	Poznań	4
Białystok	23	Słupsk	4
Szczecin	16	Bełchatów	3
Bydgoszcz	12	Częstochowa	3
Bielsko–Biała	10	Gorlice	3
Katowice	9	Jarosław	3
Lublin	7	Nowy Sącz	3
Gliwice	6	Opole	3
Kielce	6	Rzeszów	3
Łódź	6	Toruń	3
Radom	6		

6 lutego 2008 r. odbyła się sesja próbna, na której zawodnicy rozwiązywali nie liczące się do ogólnej klasyfikacji zadanie „Blokada”. W dniach konkursowych zawodnicy rozwiązywali zadania: „BBB”, „Pociągi”, „Mafia” i „Ucieczka”, 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:

- **BLO — próbne** — Blokada

	BLO — próbne	
	liczba zawodników	czyli
100 pkt.	38	9,79%
75–99 pkt.	2	0,52%
50–74 pkt.	130	33,51%
1–49 pkt.	94	24,23%
0 pkt.	53	13,65%
brak rozwiązania	71	18,30%

- **BBB** — BBB

	BBB	
	liczba zawodników	czyli
100 pkt.	39	10,05%
75–99 pkt.	1	0,26%
50–74 pkt.	3	0,77%
1–49 pkt.	87	22,43%
0 pkt.	201	51,80%
brak rozwiązania	57	14,69%

- **POC** — Pociągi

	POC	
	liczba zawodników	czyli
100 pkt.	2	0,52%
75–99 pkt.	16	4,12%
50–74 pkt.	62	15,98%
1–49 pkt.	216	55,67%
0 pkt.	42	10,82%
brak rozwiązania	50	12,89%

- **MAF** — Mafia

	MAF	
	liczba zawodników	czyli
100 pkt.	52	13,40%
75–99 pkt.	2	0,52%
50–74 pkt.	3	0,77%
1–49 pkt.	119	30,67%
0 pkt.	147	37,89%
brak rozwiązania	65	16,75%

- **UCI** — Ucieczka

	UCI	
	liczba zawodników	czyli
100 pkt.	0	0%
75–99 pkt.	3	0,77%
50–74 pkt.	28	7,22%
1–49 pkt.	124	31,96%
0 pkt.	115	29,64%
brak rozwiązania	118	30,41%

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

SUMA	liczba zawodników	czyli
400 pkt.	0	0%
300–399 pkt.	5	1,29%
200–299 pkt.	26	6,70%
100–199 pkt.	72	18,56%
1–99 pkt.	243	62,63%
0 pkt.	42	10,82%

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 XV Olimpiady Informatycznej.

ZAWODY III STOPNIA

Zawody III stopnia odbyły się w ośrodku firmy Combidata Poland S.A. w Sopocie w dniach od 1 do 5 kwietnia 2008 r.

Do zawodów III stopnia zakwalifikowano 72 najlepszych uczestników zawodów II stopnia, którzy uzyskali wynik nie mniejszy niż 130 pkt.

Zawodnicy uczęszczali do szkół w następujących województwach:

pomorskie	16 zawodników	podlaskie	5
małopolskie	11	łódzkie	3

śląskie	9	wielkopolskie	2
mazowieckie	8	świętokrzyskie	2
dolnośląskie	6	lubuskie	2
podkarpackie	6	zachodniopomorskie	2

Niżej wymienione szkoły miały w finale więcej niż jednego zawodnika:

III LO im. Marynarki Wojennej RP	Gdynia	14 uczniów
V LO im. Augusta Witkowskiego	Kraków	9
XIV LO im. Stanisława Staszica	Warszawa	6
I LO im. A. Mickiewicza	Białystok	3
VIII Liceum Ogólnokształcące	Katowice	3
XIV LO im. Polonii Belgijskiej	Wrocław	3
I LO im. Stefana Żeromskiego	Kielce	2
LO im. KEN	Stalowa Wola	2
XIII Liceum Ogólnokształcące	Szczecin	2

1 kwietnia odbyła się sesja próbna, na której zawodnicy rozwiązywali nie liczące się do ogólnej klasyfikacji zadanie „Lampki". W dniach konkursowych zawodnicy rozwiązywali zadania: „Trójkąty", „Kupno gruntu", „Podział królestwa", „Stacja" i „Permutacja", 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:

- **LAM — próbne** — Lampki

	LAM — próbne	
	liczba zawodników	czyli
100 pkt.	5	6,94%
75–99 pkt.	1	1,39%
50–74 pkt.	6	8,33%
1–49 pkt.	41	56,94%
0 pkt.	9	12,50%
brak rozwiązania	10	13,90%

- **TRO** — Trójkąty

	TRO	
	liczba zawodników	czyli
100 pkt.	7	9,72%
75–99 pkt.	4	5,56%
50–74 pkt.	0	0%
1–49 pkt.	47	65,28%
0 pkt.	8	11,11%
brak rozwiązania	6	8,33%

- **KUP** — Kupno gruntu

	KUP	
	liczba zawodników	czyli
100 pkt.	3	4,17%
75–99 pkt.	3	4,17%
50–74 pkt.	6	8,33%
1–49 pkt.	39	54,17%
0 pkt.	14	19,44%
brak rozwiązania	7	9,72%

- **POD** — Podział królestwa

	POD	
	liczba zawodników	czyli
100 pkt.	13	18,06%
75–99 pkt.	3	4,17%
50–74 pkt.	6	8,33%
1–49 pkt.	24	33,33%
0 pkt.	17	23,61%
brak rozwiązania	9	12,50%

- **STA** — Stacja

	STA	
	liczba zawodników	czyli
100 pkt.	29	40,28%
75–99 pkt.	10	13,89%
50–74 pkt.	14	19,44%
1–49 pkt.	9	12,50%
0 pkt.	3	4,17%
brak rozwiązania	7	9,72%

- **PER** — Permutacja

	PER	
	liczba zawodników	czyli
100 pkt.	2	2,78%
75–99 pkt.	4	5,56%
50–74 pkt.	0	0%
1–49 pkt.	53	73,61%
0 pkt.	10	13,88%
brak rozwiązania	3	4,17%

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

SUMA	liczba zawodników	czyli
500 pkt.	0	0%
375–499 pkt.	5	6,94%
250–374 pkt.	15	20,83%
125–249 pkt.	31	43,06%
1–124 pkt.	20	27,78%
0 pkt.	1	1,39%

5 kwietnia 2008 roku w Auli III Liceum Ogólnokształcącego im. Marynarki Wojennej RP w Gdyni ogłoszono wyniki finału XV Olimpiady Informatycznej 2007/2008 i rozdano nagrody ufundowane przez: Asseco Poland SA, Wydawnictwa Naukowo–Techniczne, Ogólnopolską Fundację Edukacji Komputerowej i Olimpiadę Informatyczną.

Poniżej zestawiono listę laureatów i wyróżnionych finalistów:

(1) **Jarosław Błasiok**, 2 klasa, VIII Liceum Ogólnokształcące w Katowicach, 476 pkt., laureat I miejsca

(2) **Marcin Kościelnicki**, 3 klasa, I LO im. Juliusza Słowackiego w Chorzowie, 444 pkt., laureat I miejsca

(3) **Marcin Andrychowicz**, 3 klasa, XIV LO im. Stanisława Staszica w Warszawie, 441 pkt., laureat I miejsca

(4) **Maciej Klimek**, 3 klasa, II Liceum Ogólnokształcące w Gorzowie Wlkp., 396 pkt., laureat II miejsca

(5) **Maciej Andrejczuk**, 3 klasa, I LO im. A. Mickiewicza w Białymstoku, 383 pkt., laureat II miejsca

(6) **Tomasz Kłeczek**, 2 klasa, V LO im. Augusta Witkowskiego w Krakowie, 344 pkt., laureat II miejsca

(7) **Łukasz Marecik**, 3 klasa, III LO im. Adama Mickiewicza w Tarnowie, 328 pkt., laureat II miejsca

(8) **Marek Rogala**, 3 klasa, Ogólnokształcące Liceum Jezuitów w Gdyni, 307 pkt., laureat II miejsca

(9) **Jonasz Pamuła**, 3 klasa, V LO im. Augusta Witkowskiego w Krakowie, 299 pkt., laureat III miejsca

(10) **Anna Piekarska**, 1 klasa, XIV LO im. Polonii Belgijskiej we Wrocławiu, 299 pkt., laureatka III miejsca

(11) **Mateusz Litwin**, 2 klasa, III LO im. Marynarki Wojennej RP w Gdyni, 299 pkt., laureat III miejsca

(12) **Adam Polak**, 2 klasa, V LO im. Augusta Witkowskiego w Krakowie, 291 pkt., laureat III miejsca

(13) **Przemysław Mazur**, 3 klasa, II Liceum Ogólnokształcące w Krakowie, 287 pkt., laureat III miejsca

(14) **Mirosław Michalski**, 3 klasa, III LO im. Marynarki Wojennej RP w Gdyni, 285 pkt., laureat III miejsca

(15) **Adam Iwaniuk**, 2 klasa, VI Liceum Ogólnokształcące w Białymstoku, 282 pkt., laureat III miejsca

(16) **Tomasz Kociumaka**, 2 klasa, V Liceum Ogólnokształcące w Gliwicach, 268 pkt., laureat III miejsca

(17) **Bolesław Kulbabiński**, 3 klasa, I LO im. Stefana Żeromskiego w Kielcach, 266 pkt., laureat III miejsca

(18) **Wiktor Jakubiuk**, 2 klasa, Dulwich College w Londynie, 264 pkt., laureat III miejsca

(19) **Robert Obryk**, 3 klasa, V LO im. Augusta Witkowskiego w Krakowie, 256 pkt., laureat III miejsca

(20) **Michał Stachurski**, 3 klasa, I LO im. Mikołaja Kopernika w Jarosławiu, 255 pkt., laureat III miejsca

(21) **Jakub Tlałka**, 1 klasa, XIV LO im. Stanisława Staszica w Warszawie, 247 pkt., finalista z wyróżnieniem

(22) **Wojciech Baranowski**, 3 klasa, III LO im. Marynarki Wojennej RP w Gdyni, 242 pkt., finalista z wyróżnieniem

(23) **Jakub Pachocki**, 1 klasa, III LO im. Marynarki Wojennej RP w Gdyni, 241 pkt., finalista z wyróżnieniem

(24) **Jakub Oćwieja**, 2 klasa, V Liceum Ogólnokształcące w Bielsku–Białej, 231 pkt., finalista z wyróżnieniem

(25) **Adam Chrabąszcz**, 3 klasa, I LO im. B. Nowodworskiego w Krakowie, 227 pkt., finalista z wyróżnieniem

(26) **Piotr Leszczyński**, 3 klasa, I LO im. Marii Konopnickiej w Suwałkach, 222 pkt., finalista z wyróżnieniem

(27) **Dominik Dudzik**, 3 klasa, V LO im. Augusta Witkowskiego w Krakowie, 221 pkt., finalista z wyróżnieniem

(28) **Radosław Burny**, 3 klasa, LO im. S. Małachowskiego w Płocku, 219 pkt., finalista z wyróżnieniem

(29) **Adam Karczmarz**, 2 klasa, LO im. KEN w Stalowej Woli, 210 pkt., finalista z wyróżnieniem

(30) **Michał Iwaniuk**, 2 klasa, XIV LO im. Stanisława Staszica w Warszawie, 202 pkt., finalista z wyróżnieniem

(31) **Bartłomiej Wiśniewski**, 1 klasa, III LO im. Marynarki Wojennej RP w Gdyni, 201 pkt., finalista z wyróżnieniem

(32) **Michał Sapalski**, 2 klasa, V LO im. Augusta Witkowskiego w Krakowie, 199 pkt., finalista z wyróżnieniem

(33) **Joachim Jelisiejew**, 3 klasa, I LO im. A. Mickiewicza w Białymstoku, 190 pkt., finalista z wyróżnieniem

(34) **Jakub Adamek**, 2 klasa, V LO im. Augusta Witkowskiego w Krakowie, 189 pkt., finalista z wyróżnieniem

(35) **Łukasz Wołochowski**, 3 klasa, III LO im. Marynarki Wojennej RP w Gdyni, 189 pkt., finalista z wyróżnieniem

(36) **Mateusz Baranowski**, 3 klasa, III LO im. Marynarki Wojennej RP w Gdyni, 187 pkt., finalista z wyróżnieniem

(37) **Maciej Dębski**, 3 klasa, Gimnazjum nr 59 w Warszawie, 187 pkt., finalista z wyróżnieniem

Lista pozostałych finalistów w kolejności alfabetycznej:

- Przemysław Bągard, 3 klasa, Zespół Szkół w Strzyżowie
- Tomasz Boczkowski, 3 klasa, VIII Liceum Ogólnokształcące w Poznaniu
- Janek Burka, 3 klasa, III LO im. Marynarki Wojennej RP w Gdyni
- Szymon Cofalik, 2 klasa, II LO A. Frycza–Modrzewskiego w Rybniku
- Marcin Fatyga, 2 klasa, I LO im. Stefana Żeromskiego w Kielcach
- Grzegorz Graczyk, 3 klasa, I LO im. Mikołaja Kopernika w Łodzi
- Paweł Grynienko, 3 klasa, Zespół Szkół Ogólnokształcących w Strzegomiu
- Piotr Karasiński, 2 klasa, I Liceum Ogólnokształcące w Bełchatowie
- Karol Konaszyński, 2 klasa, XIV LO im. Polonii Belgijskiej we Wrocławiu
- Alan Kutniewski, 1 klasa, XIII Liceum Ogólnokształcące w Szczecinie
- Bartosz Lewiński, 3 klasa, III LO im. Marynarki Wojennej RP w Gdyni
- Sebastian Łach, 1 klasa, V LO im. Augusta Witkowskiego w Krakowie
- Daniel Malinowski, 2 klasa, VIII Liceum Ogólnokształcące w Katowicach
- Grzegorz Marcinkowski, 3 klasa, III LO im. Marynarki Wojennej RP w Gdyni
- Łukasz Mazurek, 3 klasa, XIV LO im. S. Staszica w Warszawie
- Adam Obuchowicz, 1 klasa, I LO w Zielonej Górze

- Krzysztof Pasek, 3 klasa, I LO im. S. Żeromskiego w Zawierciu
- Krzysztof Pieprzak, 2 klasa, XIV LO im. Polonii Belgijskiej we Wrocławiu
- Paweł Przytuła, 3 klasa, III LO im. Marynarki Wojennej RP w Gdyni
- Marek Rogalski, 3 klasa, XXXII LO im. H. Poświatowskiej w Łodzi
- Damian Rusak, 3 klasa, I LO im. T. Kościuszki w Legnicy
- Bartłomiej Ryniec, 1 klasa, I Liceum Ogólnokształcące w Łańcucie
- Arkadiusz Socała, 3 klasa, VIII Liceum Ogólnokształcące w Katowicach
- Filip Stachura, 3 klasa, III LO im. Marynarki Wojennej RP w Gdyni
- Piotr Suwara, 1 klasa, XIV LO im. Stanisława Staszica w Warszawie
- Jacek Tomasiewicz, 3 klasa, I LO im. A. Mickiewicza w Białymstoku
- Norbert Tusiński, 3 klasa, ZSM–E w Żywcu
- Paweł Walczak, 1 klasa, III LO im. Marynarki Wojennej RP w Gdyni
- Paweł Wanat, 2 klasa, V LO im. Augusta Witkowskiego w Krakowie
- Eryk Wieliczko, 2 klasa, III LO im. Marynarki Wojennej RP w Gdyni
- Michał Włodarczyk, 3 klasa, XIV LO im. Stanisława Staszica w Warszawie
- Konrad Zabłocki, 2 klasa, XIII Liceum Ogólnokształcące w Szczecinie
- Łukasz Zatorski, 3 klasa, X Liceum Ogólnokształcące we Wrocławiu
- Mateusz Zgierski, 3 klasa, LO im. KEN w Stalowej Woli
- Paweł Żuk, 1 klasa, I Liceum Ogólnokształcące w Słupsku

Komitet Główny Olimpiady Informatycznej przyznał następujące nagrody rzeczowe:

(1) puchar ufundowany przez Olimpiadę Informatyczną przyznano zwycięzcy XV Olimpiady — Jarosławowi Błasiokowi,

(2) roczny abonament na książki ufundowany przez Wydawnictwa Naukowo–Techniczne przyznano zwycięzcy XV Olimpiady — Jarosławowi Błasiokowi,

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

(4) laptopy (4 szt.) ufundowane przez Asseco Poland SA przyznano laureatom I miejsca i jednemu laureatowi II miejsca,

(5) komputery (8 szt.) ufundowane przez Asseco Poland SA przyznano laureatom II miejsca i czterem laureatom III miejsca,

(6) drukarki atramentowe (16 szt.) ufundowane przez Asseco Poland SA i Olimpiadę Informatyczną przyznano następnym laureatom III miejsca i ośmiu wyróżnionym finalistom,

(7) odtwarzacze mp3 (44 szt.) ufundowane przez Ogólnopolską Fundację Edukacji Komputerowej przyznano dziewięciu wyróżnionym finalistom i pozostałym finalistom,

(8) książki ufundowane przez Wydawnictwa Naukowo–Techniczne oraz książeczki „XIV Olimpiada Informatyczna” przyznano wszystkim uczestnikom zawodów finałowych.

Ogłoszono komunikat o powołaniu reprezentacji Polski na:

- **Międzynarodową Olimpiadę Informatyczną IOI’2008**, która odbędzie się w Egipcie, w Kairze w terminie 16–23 sierpnia 2008 r.:

(1) Jarosław Błasiok
(2) Marcin Kościelnicki
(3) Marcin Andrychowicz
(4) Maciej Klimek

rezerwowi:

(5) Maciej Andrejczuk
(6) Tomasz Kłeczek

- **Olimpiadę Informatyczną Krajów Europy Środkowej CEOI'2008**, która odbędzie się w Niemczech, w Dreźnie w terminie 6–12 lipca 2008 r.:

  (1) Jarosław Błasiok
  (2) Marcin Kościelnicki
  (3) Marcin Andrychowicz
  (4) Maciej Klimek

  rezerwowi:

  (5) Maciej Andrejczuk
  (6) Tomasz Kłeczek

- **Bałtycką Olimpiadę Informatyczną BOI'2008**, którą w tym roku organizowała Polska w Gdyni w terminie 17–23 kwietnia 2008 r. W związku z tym, że Polska była organizatorem BOI'2008, do zawodów mogły przystąpić dwie nasze drużyny, z tym, że udział drugiej drużyny miał charakter nieformalny.

  I drużyna:

  (1) Jarosław Błasiok
  (2) Tomasz Kłeczek
  (3) Mateusz Litwin
  (4) Anna Piekarska
  (5) Adam Polak
  (6) Adam Iwaniuk

  II drużyna:

  (1) Tomasz Kociumaka
  (2) Wiktor Jakubiuk
  (3) Jakub Tlałka
  (4) Jakub Pachocki
  (5) Jakub Oćwieja
  (6) Adam Karczmarz

  rezerwowi:

  (7) Michał Iwaniuk
  (8) Bartłomiej Wiśniewski

- **Obóz czesko–polsko–słowacki, Słowacja — Danisovce, 22–28 czerwca 2008 r.:**
  - członkowie reprezentacji oraz zawodnicy rezerwowi powołani na Międzynarodową Olimpiadę Informatyczną (IOI'2008) i Olimpiadę Informatyczną Krajów Europy Środkowej (CEOI'2008).

- **Obóz im. A. Kreczmara, Nowy Sącz, 17–29 lipca 2008 r.:**
  - reprezentanci na międzynarodowe zawody informatyczne, zawodnicy rezerwowi oraz wszyscy laureaci i finaliści, którzy uczęszczają do klas niższych niż maturalna.

Sekretariat wystawił łącznie 20 zaświadczeń o uzyskaniu tytułu laureata, 17 zaświadczeń o uzyskaniu tytułu wyróżnionego finalisty oraz 35 zaświadczeń o uzyskaniu tytułu finalisty XV Olimpiady Informatycznej.

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

- Robert Andrychowicz (Warszawa)
  - Marcin Andrychowicz — laureat I miejsca
- Ludmiła Bataj (X Liceum Ogólnokształcące we Wrocławiu)
  - Łukasz Zatorski — finalista
- Krzysztof Błasiok (Katowice)
  - Jarosław Błasiok — laureat I miejsca
- Florian Brom (V Liceum Ogólnokształcące w Gliwicach)
  - Tomasz Kociumaka — laureat III miejsca
- Przemysław Broniek (Kraków)
  - Jonasz Pamuła — laureat III miejsca
  - Dominik Dudzik — finalista z wyróżnieniem
- Ireneusz Bujnowski (I LO im. A. Mickiewicza w Białymstoku)
  - Maciej Andrejczuk — laureat II miejsca
  - Adam Iwaniuk — laureat III miejsca
  - Joachim Jelisiejew — finalista z wyróżnieniem
  - Jacek Tomasiewicz — finalista
- Czesław Drozdowski (XIII Liceum Ogólnokształcące w Szczecinie)
  - Konrad Zabłocki — finalista
- Jarosław Drzeżdżon (Ogólnokształcące Liceum Jezuitów w Gdyni)
  - Marek Rogala — laureat II miejsca
- Andrzej Dyrek (V LO im. A. Witkowskiego w Krakowie)
  - Tomasz Kłeczek — laureat II miejsca
  - Jonasz Pamuła — laureat III miejsca
  - Adam Polak — laureat III miejsca
  - Robert Obryk — laureat III miejsca
  - Adam Chrabąszcz — finalista z wyróżnieniem
  - Dominik Dudzik — finalista z wyróżnieniem
  - Michał Sapalski — finalista z wyróżnieniem
  - Paweł Wanat — finalista
- Jacek Fatyga (Kielce)
  - Marcin Fatyga (finalista)
- Alina Gościniak (VIII Liceum Ogólnokształcące w Poznaniu)
  - Tomasz Boczkowski — finalista
- Małgorzata Hil (Zespół Szkół Mechaniczno–Elektrycznych w Żywcu)
  - Norbert Tusiński — finalista
- Ryszard Jakubiuk (Zielona Góra)
  - Wiktor Jakubiuk — laureat III miejsca
- Witold Jarnicki (Uniwersytet Jagielloński w Krakowie)
  - Przemysław Mazur — laureat III miejsca
- Teresa Kaszuba (Liceum Ogólnokształcące im. KEN w Stalowej Woli)

  - Mateusz Zgierski — finalista
- Grażyna Kędzierska (XXXII LO im. H. Poświatowskiej w Łodzi)
  - Marek Rogalski — finalista
- Marek Klimek (Gorzów Wielkopolski)
  - Maciej Klimek — laureat II miejsca
- Rafał Kowalczyk (I Liceum Ogólnokształcące im. S. Żeromskiego w Zawierciu)
  - Krzysztof Pasek — finalista
- Joanna Krok–Rysz (Zespół Szkół w Strzyżowie)
  - Przemysław Bągard — finalista
- Rafał Kulbabiński (Kielce)
  - Bolesław Kulbabiński — laureat III miejsca
- Adam Lasko (III Liceum Ogólnokształcące im. A. Mickiewicza w Tarnowie)
  - Łukasz Marecik — laureat II miejsca
- Zbigniew Ledóchowski (I Liceum Ogólnokształcące w Słupsku)
  - Paweł Żuk — finalista
- Mirosław Malinowski (Mikołów)
  - Daniel Malinowski — finalista
- Piotr Mateja (I Liceum Ogólnokształcące im. M. Kopernika w Łodzi)
  - Grzegorz Graczyk — finalista
- Dawid Matla (XIV Liceum Ogólnokształcące im. Polonii Belgijskiej we Wrocławiu)
  - Anna Piekarska — laureatka II miejsca
  - Karol Konaszyński — finalista
  - Krzysztof Pieprzak — finalista
- Łucja Mularska (I Liceum Ogólnokształcące im. W. Broniewskiego w Bełchatowie)
  - Piotr Karasiński — finalista
- Sebastian Ostrowski (Liceum Ogólnokształcące im. S. Małachowskiego w Płocku)
  - Radosław Burny — finalista z wyróżnieniem
- Włodzimierz Raczek (V Liceum Ogólnokształcące w Bielsku–Białej)
  - Jakub Oćwieja — finalista z wyróżnieniem
- Kacper Ryniec (Kraków)
  - Bartłomiej Ryniec — finalista
- Rafał Spręga (XIII Liceum Ogólnokształcące w Szczecinie)
  - Konrad Zabłocki — finalista
- Bronisław Stachurski (Zapałów)
  - Michał Stachurski — laureat III miejsca
- Ryszard Szubartowski (III LO im. Marynarki Wojennej RP w Gdyni)
  - Mateusz Litwin — laureat III miejsca
  - Mirosław Michalski — laureat III miejsca
  - Wojciech Baranowski — finalista z wyróżnieniem
  - Jakub Pachocki — finalista z wyróżnieniem
  - Bartłomiej Wiśniewski — finalista z wyróżnieniem
  - Łukasz Wołochowski — finalista z wyróżnieniem
  - Mateusz Baranowski — finalista z wyróżnieniem
  - Janek Burka — finalista
  - Bartosz Lewiński — finalista

  - Grzegorz Marcinkowski — finalista
  - Paweł Przytuła — finalista
  - Eryk Wieliczko — finalista
- Joanna Śmigielska (XIV Liceum Ogólnokształcące im. S. Staszica w Warszawie)
  - Michał Iwaniuk — finalista z wyróżnieniem
  - Jakub Tlałka — finalista z wyróżnieniem
  - Piotr Suwara — finalista
- Katarzyna Tomczyk–Sulima (Zespół Szkół Ogólnokształcących w Strzegomiu)
  - Paweł Grynienko — finalista
- Paweł Walter (V Liceum Ogólnokształcące im. A. Witkowskiego w Krakowie)
  - Tomasz Kłeczek — laureat II miejsca
  - Robert Obryk — laureat III miejsca
  - Adam Polak — laureat III miejsca
  - Dominik Dudzik — finalista z wyróżnieniem
  - Michał Sapalski — finalista z wyróżnieniem
  - Jakub Adamek — finalista z wyróżnieniem
  - Paweł Wanat — finalista
- Dariusz Włodarczyk (Warszawa)
  - Michał Włodarczyk — finalista
- Teresa Wojciechowska (XIV Liceum Ogólnokształcące im. S. Staszica w Warszawie)
  - Łukasz Mazurek — finalista

Zgodnie z decyzją Komitetu Głównego z dnia 4 kwietnia 2008 r., opiekunowie naukowi laureatów i finalistów, będący nauczycielami szkół ponadgimnazjalnych, otrzymają nagrody pieniężne.
Komitet postanowił dodatkowo wyróżnić nagrodami pieniężnymi w wysokości 4 tysięcy złotych czterech nauczycieli, którzy wnieśli ogromny wkład w przygotowanie laureatów i finalistów do pięciu ostatnich Olimpiad. Są to następujący nauczyciele:

- Ryszard Szubartowski — nauczyciel III LO im. Marynarki Wojennej RP w Gdyni,
- Andrzej Dyrek — nauczyciel V LO im. Augusta Witkowskiego w Krakowie,
- Ireneusz Bujnowski — nauczyciel I LO im. Adama Mickiewicza w Białymstoku,
- Iwona Waszkiewicz — była dyrektor VI LO im. Jana i Jędrzeja Śniadeckich w Bydgoszczy, obecnie Kujawsko–Pomorska Kurator Oświaty.

Podobnie jak w ubiegłych latach w przygotowaniu jest publikacja zawierająca pełną informację o XV Olimpiadzie Informatycznej, zadania konkursowe oraz wzorcowe rozwiązania. W publikacji tej znajdą się także zadania z międzynarodowych zawodów informatycznych. Redaktorami wydawnictwa są dr Przemysława Kanarek, Adam Iwanicki i Jakub Radoszewski.

Programy wzorcowe w postaci elektronicznej i testy użyte do sprawdzania rozwiązań zawodników będą dostępne na stronie Olimpiady Informatycznej: www.oi.edu.pl.

Warszawa, 20 czerwca 2008 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 2007 r.

Zasady organizacji zawodów w roku szkolnym 2007/2008

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 2007/2008.

§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 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, zwany dalej Komitetem Głównym.

(2) Olimpiada Informatyczna jest trójstopniowa.

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

(4) Rozwiązaniem każdego z zadań zawodów I, II i III stopnia jest program (napisany w jednym z następujących języków programowania: *Pascal*, *C*, *C++* lub *Java*) 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 — 22.10–19.11.2007 r.
  ogłoszenie wyników:
  - w witrynie Olimpiady — 14.12.2007 r.,
  - pocztą — 28.12.2007 r.
- zawody II stopnia — 06–08.02.2008 r.
  ogłoszenie wyników:
  - w witrynie Olimpiady — 15.02.2008 r.
  - pocztą — 29.02.2008 r.
- zawody III stopnia — 01–05.04.2008 r.

§3 ROZWIĄZANIA ZADAŃ

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

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

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

(4) 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.

(5) 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`
Java	`java`

(6) 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. Programy w *Javie* będą kompilowane w systemie Linux za pomocą kompilatora z Sun JDK 6 Update 2. 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`
Dla `java`	`javac abc.java`

(7) Programy w Javie będą uruchamiane w systemie Linux za pomocą maszyny wirtualnej z Sun JDK 6 Update 2. Dla zadania abc klasa publiczna pliku źródłowego w Javie musi nosić nazwę `abc`. Uruchomieniu podlega treść publicznej funkcji statycznej `main(String[] args)` tej klasy.

(8) Limity czasowe i pamięciowe będą ustalane oddzielnie dla $C/C++$, *Pascala* i *Javy* z uwzględnieniem specyfiki kompilatorów używanych do oceny rozwiązań zadań napisanych w tych językach.

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

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

§4 ZAWODY 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 19 listopada 2007 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 19 listopada 2007 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) 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 6, 7 i 8.

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

(4) 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.

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

(6) Od dnia 3 grudnia 2007 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.

(7) Do dnia 7 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.

(8) Reklamacje złożone po 7 grudnia 2006 r. nie będą rozpatrywane.

§5 ZAWODY II I III STOPNIA

(1) Zawody II i III stopnia Olimpiady Informatycznej polegają na samodzielnym rozwiązywaniu zadań w ciągu dwóch pięciogodzinnych sesji odbywających się w różnych dniach.

(2) Rozwiązywanie zadań konkursowych poprzedzone jest trzygodzinną sesją próbną umożliwiającą uczestnikom zapoznanie się z warunkami organizacyjnymi i technicznymi Olimpiady. Wyniki sesji próbnej nie są liczone do klasyfikacji.

(3) W czasie rozwiązywania zadań konkursowych każdy uczestnik ma do swojej dyspozycji komputer z systemem Linux. Zawodnikom wolno korzystać wyłącznie ze sprzętu i oprogramowania dostarczonego przez organizatora. Stanowiska są przydzielane losowo.

(4) Komisja Regulaminowa powołana przez Komitet Główny czuwa nad prawidłowością przebiegu zawodów i pilnuje przestrzegania Regulaminu Olimpiady i Zasad Organizacji Zawodów.

(5) Zawody II i III stopnia są przeprowadzane za pomocą Serwisu Internetowego Olimpiady zwanego dalej SIO.

(6) Na sprawdzenie kompletności oprogramowania i poprawności konfiguracji sprzętu jest przeznaczone 45 minut przed rozpoczęciem sesji próbnej. W tym czasie wszystkie zauważone braki powinny zostać usunięte. Jeżeli nie wszystko uda się poprawić w tym czasie, rozpoczęcie sesji próbnej w tej sali może się opóźnić.

(7) W przypadku stwierdzenia awarii sprzętu w czasie zawodów, termin zakończenia pracy przez uczestnika zostaje odpowiednio przedłużony.

(8) W czasie trwania zawodów nie można korzystać z żadnych książek ani innych pomocy takich jak: dyski, kalkulatory, notatki itp. Nie wolno mieć na stanowisku komputerowym telefonu komórkowego ani innych własnych urządzeń elektronicznych.

(9) W ciągu pierwszej godziny każdej sesji nie wolno opuszczać przydzielonej sali zawodów. Zawodnicy spóźnieni więcej niż godzinę nie będą w tym dniu dopuszczeni do zawodów.

(10) W ciągu pierwszej godziny każdej sesji uczestnik może poprzez SIO zadawać pytania, na które otrzymuje jedną z odpowiedzi: *tak*, *nie*, *niepoprawne pytanie*, *odpowiedź wynika z treści zadania* lub *bez odpowiedzi*. Pytania mogą dotyczyć jedynie treści zadań.

(11) W czasie przeznaczonym na rozwiązywanie zadań jakikolwiek inny sposób komunikowania się z członkami Jury co do treści i sposobów rozwiązywania zadań jest niedopuszczalny.

(12) Komunikowanie się z innymi uczestnikami Olimpiady (np. ustnie, telefonicznie lub poprzez sieć) w czasie przeznaczonym na rozwiązywanie zadań jest zabronione pod rygorem dyskwalifikacji.

(13) Każdy zawodnik ma prawo wydrukować wyniki swojej pracy w sposób podany przez organizatorów.

(14) Każdy zawodnik powinien umieścić ostateczne rozwiązania zadań w SIO. Po zgłoszeniu rozwiązania każdego z zadań SIO dokona wstępnego sprawdzenia i udostępni jego wyniki zawodnikowi. Wstępne sprawdzenie polega na uruchomieniu programu zawodnika na testach przykładowych (wyniki sprawdzenia tych testów nie są liczone do końcowej klasyfikacji). Te same testy przykładowe są używane do wstępnego sprawdzenia w trybie weryfikacji rozwiązań na komputerze zawodnika.

(15) Każde zadanie można zgłosić w SIO co najwyżej 10 razy. Spośród tych zgłoszeń ocenianie jest jedynie najpóźniejsze.

(16) Jeżeli zawodnik nie zgłosił swoich rozwiązań w SIO, powinien je pozostawić w katalogu wskazanym przez organizatorów i niezwłocznie po zakończeniu sesji

a przed opuszczeniem sali zawodów wręczyć pisemne oświadczenie dyżurującemu w tej sali członkowi Komisji Regulaminowej. Oświadczenie to musi zawierać imię i nazwisko zawodnika oraz numer stanowiska. Złożenie takiego oświadczenia powoduje, że rozwiązanie złożone wcześniej w SIO nie będzie rozpatrywane.

(17) W sprawach spornych decyzje podejmuje Jury Odwoławcze, złożone z jurora nie zaangażowanego w rozważaną kwestię i wyznaczonego członka Komitetu Głównego. Decyzje w sprawach o wielkiej wadze (np. dyskwalifikacji) Jury Odwoławcze podejmuje w porozumieniu z przewodniczącym Komitetu Głównego.

(18) Każdego dnia zawodów około 2 godziny po zakończeniu sesji zawodnicy otrzymają raporty oceny swoich prac na niepełnym zestawie testów. Od tego momentu przez godzinę będzie czas na reklamację tej oceny, a w szczególności na reklamację wyboru rozwiązania, które ma podlegać ocenie.

§6 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 XX Międzynarodową Olimpiadę Informatyczną w 2008 roku na podstawie wyników zawodów III stopnia i regulaminu tej Olimpiady Międzynarodowej.

(7) Komitet Główny może przyznać nagrodę nauczycielom lub opiekunom naukowym, którzy przygotowywali laureatów lub finalistów 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 IX Obozie Naukowo–Treningowym im. Antoniego Kreczmara, który odbędzie się w czasie wakacji 2008 r.

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

§7 PRZEPISY KOŃCOWE

(1) 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; mają także zagwarantowane na czas tych zawodów bezpłatne zakwaterowanie, wyżywienie i zwrot kosztów przejazdu.

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

Zawody I stopnia

opracowania zadań

Michał Pilipczuk
Treść zadania, Opracowanie

Adam Iwanicki
Program

OI, Etap I, 22.10–19.11.2007

Cło

Król Bajtazar postanowił uporządkować kwestie związane z opłacaniem cła przez kupców Bajtocji. Bajtocja składa się z n miast połączonych m dwukierunkowymi drogami. Każda droga w Bajtocji łączy dwa różne miasta. Żadne dwa miasta nie są połączone więcej niż jedną drogą. Drogi mogą prowadzić przez tunele i estakady.

Dotychczas każde miasto w Bajtocji pobierało cło od każdego, kto do niego przyjeżdżał, i od każdego, kto z niego wyjeżdżał. Niezadowoleni z tej sytuacji kupcy wnieśli oficjalny protest, w którym sprzeciwili się wielokrotnemu pobieraniu cła. Król Bajtazar postanowił ograniczyć przywileje miast. Wedle nowego królewskiego edyktu, każde miasto może pobierać cło od kupców podróżujących dokładnie jedną z dróg prowadzących do niego (bez względu na kierunek ich podróży). Ponadto, dla każdej z dróg, podróżni podróżujący tą drogą nie mogą być zmuszeni do płacenia cła obu miastom, które ta droga łączy. Należy jeszcze podjąć decyzję, które miasto ma pobierać cło z której drogi. Rozwiązanie tego problemu król zlecił Tobie.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis układu dróg w Bajtocji,*
- *dla każdego miasta wyznaczy, na której drodze dane miasto będzie pobierać od kupców cło, lub stwierdzi, że wprowadzenie w życie edyktu nie jest możliwe,*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu wejścia znajdują się dwie liczby całkowite n i m ($1 \leqslant n \leqslant 100\,000$, $1 \leqslant m \leqslant 200\,000$), oznaczające odpowiednio liczbę miast oraz dróg w Bajtocji. Miasta są ponumerowane od 1 do n. W kolejnych m wierszach znajdują się opisy kolejnych dróg. W wierszu i znajdują się dwie liczby całkowite a_i i b_i ($1 \leqslant a_i < b_i \leqslant n$) oznaczające, że miasta a_i i b_i są połączone bezpośrednią drogą.

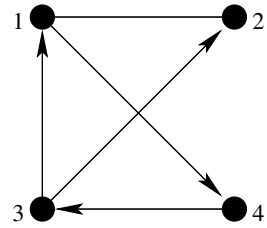
Wyjście

Jeśli pobieranie cła zgodnie z wymaganiami królewskiego edyktu nie jest możliwe, to w pierwszym i jedynym wierszu wyjścia Twój program powinien wypisać słowo `NIE`*. W przeciwnym przypadku, w pierwszym wierszu Twój program powinien wypisać słowo* `TAK`*, a w kolejnych n wierszach powinny się znaleźć informacje, które miasto, z jakiej drogi pobiera cło. W wierszu $i+1$ powinien znaleźć się numer miasta, do którego prowadzi droga, na której miasto numer i pobiera od kupców cło. W przypadku, gdy istnieje wiele rozwiązań, należy podać dowolne z nich.*

Przykład

Dla danych wejściowych:

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



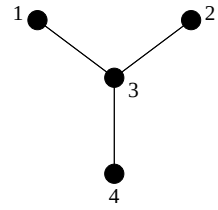
poprawnym wynikiem jest:

```
TAK
3
3
4
1
```

Strzałki na rysunku wskazują miasta pobierające cło od kupców podróżujących daną drogą. Zwróć uwagę, że kupcy podróżujący drogą łączącą miasta 1 i 2 nie płacą w ogóle cła.

Dla danych wejściowych:

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



poprawnym wynikiem jest:

```
NIE
```

Rozwiązanie

Wprowadzenie

Opiszmy problem z zadania w języku teorii grafów. Dany jest graf nieskierowany $G = (V, E)$, w którym wierzchołkami są miasta znajdujące się w Bajtocji, a krawędziami — drogi pomiędzy nimi. Każdemu wierzchołkowi należy przyporządkować jedną spośród incydentnych[1] z nim krawędzi, co odpowiada nadaniu miastu prawa do pobierania cła z drogi. Każda krawędź może być przyporządkowana co najwyżej jednemu wierzchołkowi — tak, aby podatek nie był pobierany na żadnej drodze dwukrotnie. Jeśli opisane przyporządkowanie nie istnieje, to również powinniśmy umieć to stwierdzić.

Poszukiwane rozwiązanie jest *funkcją różnowartościową* $f : V \to E$ taką, że dla każdego wierzchołka $v \in V$ wartość $f(v)$ jest krawędzią incydentną z v.

[1] Mówimy, że krawędź jest incydentna z wierzchołkiem, jeśli ten wierzchołek jest jednym z jej końców.

Rozwiązanie wzorcowe — proste przeszukiwanie

Zauważmy, że możemy zająć się poszukiwaniem funkcji przyporządkowującej krawędzie wierzchołkom oddzielnie dla każdej spójnej składowej grafu. Niech $G = H_1 \cup H_2 \cup \ldots \cup H_k$, gdzie H_i to spójne składowe G. Wtedy znajdując poszukiwane w zadaniu przyporządkowanie f_i dla każdej składowej H_i, możemy zdefiniować przyporządkowanie f dla całego grafu G jako f_i na każdym H_i. Oczywiście, jeśli dla pewnej składowej nie istnieje przyporządkowanie spełniające warunki zadania, to nie istnieje także przyporządkowanie dla całego grafu.

Przypadki bez rozwiązania

Zastanówmy się wpierw, dla jakich grafów odpowiedź na pytanie postawione w zadaniu jest negatywna. Pomocne będzie następujące proste spostrzeżenie.

Fakt 1. *Jeśli spójna składowa H grafu G jest drzewem, to nie istnieje dla niej poszukiwane przyporządkowanie.*

Dowód: Niech $H = (W, F)$ będzie drzewem. Oznaczmy przez $|W|$ i $|F|$ odpowiednio moc zbioru wierzchołków i krawędzi. Wiemy, że w każdym drzewie liczba krawędzi jest o jeden mniejsza niż liczba wierzchołków, więc $|F| = |W| - 1$. Zauważmy także, że jeśli funkcja f ma być różnowartościowa, to musi $|W|$ wierzchołkom przypisać dokładnie $|W|$ różnych krawędzi. Ponieważ w składowej H nie ma tylu krawędzi, więc jest to niemożliwe — stąd f nie istnieje.

■

Okazuje się, że fakt 1 zawiera w sobie już wszystkie przypadki, dla których odpowiedź jest negatywna. Dla grafów, w których żadna spójna składowa nie jest drzewem, pokażemy algorytm konstrukcji przyporządkowania krawędzi wierzchołkom dla poszczególnych składowych, a tym samym — przyporządkowania dla całego grafu.

Konstrukcja

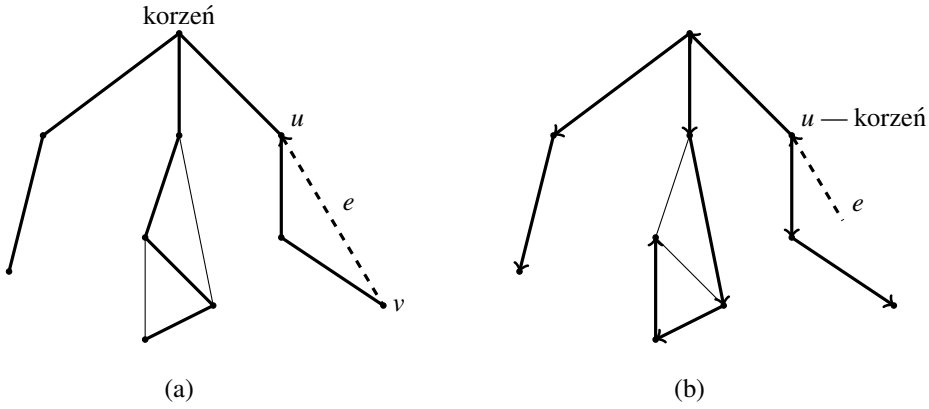
Rozważmy spójną składową $H = (W, F)$. Dla składowej tej istnieje *drzewo rozpinające*, które możemy *ukorzenić* w dowolnym wierzchołku. Wybór korzenia określa *orientację* w składowej — korzeń jest na samej górze, jego dzieci poniżej, ich dzieci jeszcze niżej itd. Każdemu z wierzchołków, oprócz korzenia, możemy przypisać krawędź łączącą go z jego ojcem w drzewie. W ten sposób dostajemy prawie dobre rozwiązanie — każdemu wierzchołkowi, oprócz korzenia, przypisaliśmy inną krawędź.

Aby uzyskać pełne rozwiązanie, musimy jeszcze znaleźć krawędź dla korzenia. Możemy w tym celu przed konstrukcją drzewa wybrać krawędź $e = (u, v) \in F$, której usunięcie nie rozspójni rozważanej składowej, a następnie:

- skonstruować drzewo rozpinające dla $H' = (W, F \setminus \{e\})$;
- za korzeń skonstruowanego drzewa wybrać jeden z wierzchołków końcowych krawędzi e, na przykład u;
- przypisać $f(u) = e$, a dla pozostałych wierzchołków $w \in W \setminus \{u\}$ za $f(w)$ przyjąć krawędź łączącą w z jego ojcem w drzewie.

Wszystkie powyższe operacje możemy wykonać, posługując się jednym z algorytmów przeszukiwania grafu: w głąb (DFS) lub wszerz (BFS), które są opisane w [20]. Najpierw musimy wybrać krawędź e, czyli dowolną krawędź należącą do jakiegokolwiek cyklu (co gwarantuje, że jej usunięcie nie spowoduje rozspójnienia składowej). Takie krawędzie łatwo rozpoznać w trakcie algorytmu przeszukiwania grafu — próba przejścia taką krawędzią prowadzi do wierzchołka już wcześniej odwiedzonego i zamyka cykl złożony z krawędzi drzewa. Poszukiwana krawędź nie istnieje zatem wtedy i tylko wtedy, gdy cała spójna składowa jest drzewem — jednak wówczas mamy do czynienia z przypadkiem negatywnym na mocy faktu 1. W przeciwnym razie znajdujemy odpowiednią krawędź $e = (u,v)$ i za korzeń drzewa rozpinającego wybieramy jeden z jej końców, na przykład u.

Teraz możemy usunąć krawędź e z grafu i ponownie uruchomić algorytm DFS lub BFS, startując z wierzchołka u — w ten sposób skonstruujemy drzewo rozpinające. Każdemu wierzchołkowi w możemy przypisać krawędź $f(w)$ — tę, którą weszliśmy do tego wierzchołka w trakcie przechodzenia grafu. Jest to krawędź łącząca wierzchołek w z jego ojcem w skonstruowanym drzewie rozpinającym.



Rys. 1: Przykład działania opisanej procedury dla spójnej składowej. W pierwszej iteracji (a) za pomocą przeszukiwania DFS znajdowane jest drzewo rozpinające (grubsze linie), a także wybierana jest pewna krawędź powrotna $e = (u,v)$, która staje się $f(u)$. W drugiej iteracji (b), w wyniku przeszukiwania rozpoczętego w u, każdemu wierzchołkowi poza u zostaje przypisana krawędź, którą do niego wchodzimy (przypisanie krawędzi oznaczono na rysunku strzałkami).

Algorytm wzorcowy

Opisany wyżej pomysł to nasz algorytm wzorcowy:

```
1: for s ∈ V
2:   if s nie przypisano jeszcze krawędzi then
3:   begin
4:     wykonaj przeszukiwanie z wierzchołka s:
5:       wyznacz krawędź e należącą do cyklu;
6:     if krawędź e nie istnieje then
7:     begin
8:       wypisz NIE;
```

```
9:          zakończ działanie programu;
10:       end else
11:       begin
12:         niech u będzie jednym z końców e;
13:         przypisz f(u) = e;
14:         usuń krawędź e z grafu;
15:         wykonaj przeszukanie z wierzchołka u:
16:           przypisz każdemu wierzchołkowi krawędź,
17:           którą do niego wchodzimy;
18:       end;
19:     end;
20: end;
21: wypisz TAK oraz zapamiętane przyporządkowanie f.
```

Algorytm automatycznie rozpoznaje spójne składowe grafu — przetwarzając wierzchołek s, odwiedzamy i przetwarzamy wszystkie wierzchołki należące do tej samej składowej, co on. Potem wracamy do głównej pętli algorytmu i poszukujemy nieprzetworzonego wierzchołka grafu, który wyznacza jeszcze nierozpatrzoną spójną składową.

Zauważmy, że w każdej spójnej składowej wykonujemy dwa przeszukiwania — każde o złożoności czasowej liniowej względem jej rozmiaru. Cały algorytm ma zatem złożoność liniową względem rozmiaru grafu, czyli $O(n+m)$. Złożoność pamięciowa jest taka sama, gdyż musimy pamiętać jedynie reprezentację grafu, zaznaczać odwiedzane wierzchołki i zapisywać konstruowane przyporządkowanie f. Wszystkie te dane mieszczą się w strukturach o rozmiarze $O(n+m)$.

Rozwiązania bazujące na powyższym schemacie mogą różnić się zastosowanym algorytmem przeszukiwania. Implementacje z przeszukiwaniem w głąb znajdują się w plikach: `clo6.c`, `clo7.cpp`, `clo8.pas` oraz `clo11.java`, natomiast z przeszukiwaniem wszerz — w plikach `clo.c`, `clo1.cpp`, `clo2.pas` oraz `clo9.java`.

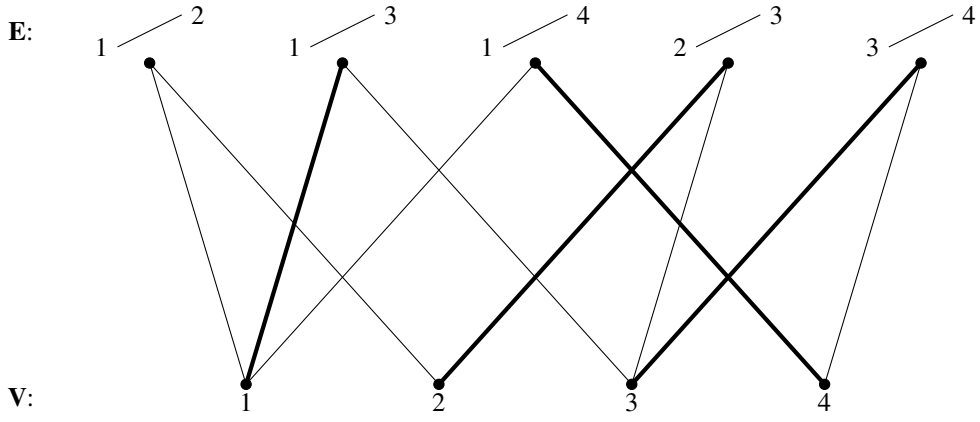
Rozwiązanie alternatywne — zastosowanie skojarzeń

Problem postawiony w zadaniu możemy rozwiązać także inną metodą — stosując bardziej skomplikowane pojęcia i otrzymując w wyniku algorytm nie lepszy niż wzorcowy. Uważamy jednak, że warto przedstawić także ten pomysł, gdyż pozwala on poznać kolejne ważne pojęcia i ciekawe algorytmy z teorii grafów.

Sprowadzimy zadanie do problemu znajdowania *maksymalnego skojarzenia w grafie dwudzielnym*. Oznaczmy oryginalny graf przez $G = (V,E)$. Zbudujmy dla niego graf dwudzielny $H = (V \cup E, F)$, gdzie $(v,e) \in F$ dla $v \in V$ oraz $e \in E$ wtedy i tylko wtedy, gdy v jest incydentne z e. Mówiąc prościej, graf H ma dwa zbiory wierzchołków: w jednym z nich są wierzchołki grafu G, a w drugim — krawędzie G. Wierzchołek v i krawędź e są połączone w grafie H krawędzią, jeśli v jest jednym z końców e w grafie G. A zatem H zawiera łącznie $n+m$ wierzchołków oraz $2m$ krawędzi.

Skojarzenie w grafie H to podzbiór krawędzi tego grafu, które nie mają wspólnych wierzchołków. Wierzchołki incydentne z krawędziami wybranymi do skojarzenia nazywamy *skojarzonymi* lub *pokrytymi* przez skojarzenie. Każde skojarzenie w grafie H możemy

zinterpretować jako różnowartościowe przyporządkowanie krawędzi wierzchołkom grafu G. Oczywiście zależy nam na znalezieniu jak największego skojarzenia — w końcu chcemy przypisać krawędź każdemu wierzchołkowi. Będziemy więc szukać skojarzenia maksymalnego pod względem liczności. Jeśli okaże się, że zawiera ono $|V|$ krawędzi, to będziemy mieli poszukiwane przyporządkowanie. W przeciwnym razie będziemy wiedzieli, że takie przyporządkowanie nie istnieje.



Rys. 2: Graf dwudzielny H skonstruowany dla grafu G z pierwszego przykładu z treści zadania. Dolne cztery wierzchołki H odpowiadają wierzchołkom G, natomiast górne pięć odpowiada krawędziom G. Pogrubione krawędzie reprezentują najliczniejsze skojarzenie w H, odpowiadające przyporządkowaniu wierzchołkom G takich samych krawędzi, jak w pierwszym przykładowym wyjściu.

Więcej o skojarzeniach i związanych z nimi algorytmach można przeczytać w [23].

Trochę kombinatoryki

Warunek określający istnienie skojarzenia w grafie H pokrywającego cały zbiór V jest oczywiście analogiczny do przedstawionego przy okazji algorytmu wzorcowego. Wystarczy, by żadna spójna składowa grafu G nie była drzewem — wtedy istnieje poszukiwane przyporządkowanie krawędzi wierzchołkom w grafie G, które z kolei odpowiada skojarzeniu w grafie H o mocy $|V|$. Warto jednak, korzystając z okazji, przytoczyć także inny warunek, oparty na twierdzeniu Halla, o którym można przeczytać np. w [31]. Pozwala on rozstrzygnąć o istnieniu skojarzenia pokrywającego jeden ze zbiorów wierzchołków także w szerszej klasie grafów, niż tutaj rozważane.

Twierdzenie 1 (Hall). *Niech $H = (A \cup B, E)$, gdzie $E \subseteq A \times B$, będzie grafem dwudzielnym. W H istnieje skojarzenie, w którym każdy wierzchołek ze zbioru A jest pokryty, wtedy i tylko wtedy, gdy zachodzi następujący warunek:*

dla każdego podzbioru $X \subseteq A$, zbiór tych wierzchołków $y \in B$, dla których istnieje $x \in X$, takie że $(x,y) \in E$, ma moc równą co najmniej $|X|$.

Korzystając z twierdzenia Halla, można napisać algorytm konstrukcji skojarzenia — niestety, jest on daleki od optymalnego. Wynaleziono jednak sporo innych, znacznie efektywniejszych rozwiązań tego problemu.

Algorytm Hopcrofta-Karpa

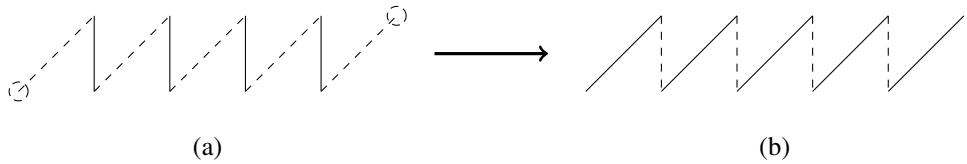
Najszybszą i nietrudną w implementacji metodą znajdowania maksymalnego skojarzenia w grafie dwudzielnym w ogólnym przypadku jest algorytm Hopcrofta-Karpa, opisany w [23]. Działa on w czasie $O(|E|\sqrt{|V|})$, gdzie V i E to odpowiednio zbiory wierzchołków i krawędzi. Przypomnijmy, że graf dwudzielny H zbudowany na podstawie grafu danego w zadaniu ma $n+m$ wierzchołków oraz $2m$ krawędzi. Zastosowanie do niego metody Hopcrofta-Karpa pozwala więc wyznaczyć rozwiązanie w czasie $O(m^{\frac{3}{2}})$, czyli istotnie gorszym od czasu działania algorytmu wzorcowego. Takie rozwiązanie jest zaimplementowane w pliku `clos1.cpp` — ze względu na kiepską złożoność nie uzyskuje ono kompletu punktów.

Turbo-matching

Turbo-matching to nieoficjalna nazwa nadana algorytmowi otrzymanemu przez drobną, acz brzemienną w skutki modyfikację podstawowego algorytmu znajdowania maksymalnego skojarzenia. Algorytm podstawowy jest dokładnie opisany w [20], więc tutaj tylko krótko omówimy schemat jego działania. Startujemy w nim od skojarzenia pustego, które stopniowo zwiększamy, wykorzystując tak zwane *ścieżki powiększające* (inaczej nazywane także *naprzemiennymi*). Ścieżkę w grafie nazwiemy powiększającą, jeśli:

- rozpoczyna się ona od wierzchołka nieskojarzonego jeszcze z żadnym innym;
- zawiera na przemian krawędzie nienależące do skojarzenia i należące do skojarzenia;
- kończy się także na wierzchołku nieskojarzonym.

Jeśli przez E_1 oznaczymy zbiór krawędzi ścieżki należących do skojarzenia, a przez E_2 — zbiór jej pozostałych krawędzi, to łatwo zauważyć, że $|E_1| = |E_2| - 1$. Ponadto, jeśli ze skojarzenia wyrzucimy krawędzie ze zbioru E_1, a dodamy krawędzie ze zbioru E_2, to dostaniemy nowe, większe skojarzenie! (patrz rys. 3) Można udowodnić, że dla dowolnego nie najliczniejszego skojarzenia istnieje ścieżka powiększająca, czyli że powiększając skojarzenie za pomocą ścieżek naprzemiennych, w końcu uzyskuje się najliczniejsze skojarzenie.



Rys. 3: Przykład ścieżki naprzemiennej (a) dla $|E_1| = 4$ i $|E_2| = 5$ (krawędzie ze skojarzenia oznaczone są liniami ciągłymi, a pozostałe — przerywanymi) oraz wyniku dołączenia krawędzi z E_2 do skojarzenia oraz „odkojarzenia" krawędzi z E_1 (b). Wierzchołki nieskojarzone zaznaczone są kółkami.

Niech $G = (V_1 \cup V_2, E)$ będzie danym grafem dwudzielnym. Schemat samego algorytmu wykorzystującego ścieżki powiększające jest następujący:

```
1:  podstaw za skojarzenie zbiór pusty;
2:  repeat
3:    znalezione := false;
4:    for v ∈ V1 do
5:      if v jest nieskojarzony then
6:      begin
7:        poszukaj ścieżki powiększającej z wierzchołka v;
8:        { w tym celu stosujemy przeszukiwanie w głąb (DFS) z wierzchołka v }
9:        if znaleziono ścieżkę then
10:       begin
11:         popraw skojarzenie, wykorzystując ścieżkę;
12:         znalezione := true;
13:       end;
14:       zaznacz odwiedzone wierzchołki jako nieodwiedzone;
15:     end;
16:   if not znalezione then przerwij;
17: end;
```

Powyższy algorytm działa w czasie $O(|V| \cdot |E|)$.
Algorytm turbo-matching niewiele różni się od poprzedniego:

```
1:  podstaw za skojarzenie zbiór pusty;
2:  repeat
3:    znalezione := false;
4:    for v ∈ V1 do
5:      if v jest nieskojarzony i nieodwiedzony then
6:      begin
7:        poszukaj ścieżki powiększającej z wierzchołka v procedurą DFS;
8:        if znaleziono ścieżkę then
9:        begin
10:         popraw skojarzenie, wykorzystując ścieżkę;
11:         znalezione := true;
12:       end;
13:     end;
14:   if not znalezione then przerwij;
15:   zaznacz odwiedzone wierzchołki jako nieodwiedzone;
16: end;
```

Istotną modyfikacją dokonaną w algorytmie podstawowym jest zmiana momentu „zerowania tablicy odwiedzin". W algorytmie turbo-matching zostało to przeniesione na zewnątrz pętli **for** (wiersz 15). To oznacza, że w jednej iteracji tej pętli w kolejnych wywołaniach przeszukiwania w głąb (wiersz 7) nie wchodzimy wielokrotnie do tych samych wierzchołków. Ta zmiana nie psuje poprawności algorytmu — jeśli istnieje ścieżka

powiększająca, to ją znajdziemy. Istotnie, jeśli w jednej iteracji pętli **for**, przechodząc przez jakiś wierzchołek, nie znaleźliśmy ścieżki powiększającej, to ponowne wchodzenie do niego w tej samej iteracji pętli nie ma sensu. Jeśli zatem istnieją ścieżki powiększające, to pierwsza z nich zostanie wyszukana bez potrzeby wchodzenia do odwiedzonych wcześniej wierzchołków.

Teoretycznie złożoność turbo-matchingu jest taka sama jak algorytmu podstawowego — w każdej iteracji pętli **for** przetwarzamy każdą krawędź i każdy wierzchołek co najwyżej raz, na co potrzebujemy czasu $O(|V|+|E|)$. Jednocześnie każde wykonanie tej pętli generuje nową ścieżkę powiększającą, zatem obrotów będzie co najwyżej $|V|$ — tyle, ile maksymalnie krawędzi ma skojarzenie. Cały algorytm ma złożoność $O(|V| \cdot (|V|+|E|))$, a ponieważ w sensownych grafach mamy $|V| = O(|E|)$, to daje ostatecznie złożoność turbo-matchingu równą $O(|V| \cdot |E|)$.

W praktyce turbo-matching okazuje się jednak dużo szybszy. Wprowadzona modyfikacja sprawia, że poszukując ścieżek powiększających, często pomijamy wierzchołki, o których wiadomo, że i tak nie ma sensu do nich wchodzić. Ze względu na małą stałą, algorytm ten dla rozsądnych danych zachowuje się nie gorzej niż algorytm Hopcrofta-Karpa. Jest za to prostszy i łatwiejszy do zaprogramowania.

Grafy dwudzielne, które występują w naszym zadaniu, są dość proste — wszystkie wierzchołki z jednej części (odpowiadające krawędziom z oryginalnego grafu) mają stopień równy 2. W takim przypadku turbo-matching działa dużo szybciej od algorytmu Hopcrofta-Karpa i nie udało się znaleźć przykładów, w których byłby istotnie wolniejszy od rozwiązań wzorcowych, liniowych. Dlatego jego poprawna implementacja, zawarta w pliku `clos3.cpp`, otrzymuje maksymalną liczbę punktów.

Testy

W powyższym opracowaniu zauważyliśmy, że rozwiązania szukamy praktycznie oddzielnie w każdej spójnej składowej. Dlatego grafy występujące w testach zostały skonstruowane z różnego rodzaju składowych:

- grafów losowych o zadanej gęstości;
- dużych cykli;
- drzew — używanych do testów z odpowiedzią negatywną;
- klik, czyli grafów pełnych;
- „drzew z cyklem” — w tej kategorii znalazły się dwa rodzaje grafów: drzewa z cyklem losowej długości i drzewa z cyklem długości 3.

Rozwiązania zawodników były sprawdzane na 10 grupach testów.

Nazwa	n	m	Opis
clo1a.in	40	60	mały test poprawnościowy z odpowiedzią pozytywną
clo1b.in	10	9	małe drzewo
clo2a.in	175	175	większy test poprawnościowy złożony z dwóch „drzew z cyklem”
clo2b.in	175	11 199	klika i drzewo
clo3.in	600	600	większy test poprawnościowy: cykl i dwa „drzewa z cyklami”
clo4.in	1 000	1 000	„drzewo z cyklem”
clo5.in	5 000	9 850	dwa „drzewa z cyklami”, cykl i klika
clo6.in	15 000	15 000	„drzewo z cyklem” i cykl
clo7.in	35 000	35 000	dwa „drzewa z cyklami” i cykl
clo8.in	50 000	50 000	dwa większe „drzewa z cyklami” i cykl
clo9.in	80 000	158 800	cykl, cztery małe kliki oraz dwa „drzewa z cyklami”
clo10a.in	100 000	180 000	„drzewo z cyklem” oraz losowy graf o dużej gęstości
clo10b.in	100 000	199 233	klika i duże drzewo

Tomasz Waleń
Treść zadania, Opracowanie

Krzysztof Dulęba, Jakub Radoszewski
Program

OI, Etap I, 22.10–19.11.2007

Klocki

Bajtazar jako małe dziecko uwielbiał bawić się klockami. Jego zabawa polegała na układaniu z klocków n kolumn o losowo wybranych wysokościach, a następnie ich porządkowaniu. Bajtazar wybierał liczbę k, a następnie starał się w minimalnej liczbie ruchów tak uporządkować klocki, by pewne k kolejnych kolumn klocków miało tę samą wysokość. Pojedynczy ruch polega na:

- *położeniu jednego klocka na szczycie wybranej kolumny klocków (Bajtazar posiadał ogromne pudło z zapasowymi klockami, więc ten ruch jest zawsze możliwy), lub*
- *zdjęciu jednego klocka ze szczytu wybranej kolumny.*

Bajtazar nigdy nie był pewien, czy wybrane przez niego rozwiązanie było optymalne, i poprosił Cię o napisanie programu, który pomoże mu rozwiązywać ten problem.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia liczbę k i opis początkowego układu klocków,*
- *wyznaczy rozwiązanie wymagające minimalnej liczby ruchów,*
- *wypisze otrzymane rozwiązanie na standardowe wyjście.*

Wejście

W pierwszym wierszu standardowego wejścia zapisane są dwie liczby całkowite n oraz k ($1 \leqslant k \leqslant n \leqslant 100\,000$), oddzielone pojedynczym odstępem. W kolejnych n wierszach zapisane są początkowe wysokości kolumn klocków; wiersz $(i+1)$-szy zawiera jedną liczbę całkowitą $0 \leqslant h_i \leqslant 1\,000\,000$ — wysokość i-tej kolumny klocków, czyli liczbę klocków, z których się ona składa.

Wyjście

Na standardowe wyjście należy wypisać optymalne rozwiązanie, to jest układ klocków, który:

- *zawiera k kolejnych kolumn o tej samej wysokości,*
- *można otrzymać z początkowego układu w minimalnej liczbie ruchów.*

Wyjście powinno składać się z $n+1$ wierszy, a każdy z nich powinien zawierać jedną liczbę całkowitą. W pierwszym wierszu należy wypisać minimalną liczbę ruchów, potrzebnych do uzyskania żądanego układu. W $(i+1)$-szym wierszu (dla $1 \leqslant i \leqslant n$) należy wypisać liczbę h'_i — końcową wysokość i-tej kolumny klocków. W przypadku, gdy istnieje wiele rozwiązań, należy podać dowolne z nich.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`5 3`	`2`
`3`	`3`
`9`	`9`
`2`	`2`
`3`	`2`
`1`	`2`

Rozwiązanie

Problem możemy podzielić na dwie części:

- *uporządkowanie* — dla wybranych, dowolnych k kolejnych kolumn możemy obliczyć minimalny koszt wyrównania ich wysokości,
- *wybór* — znając powyższe, możemy spośród wszystkich kolumn wybrać te, których uporządkowanie będzie miało najniższy koszt.

Dalsza część rozwiązania będzie przebiegać zgodnie z nakreślonym powyżej planem.

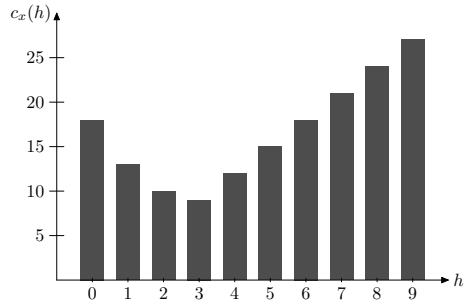
Uporządkowanie kolumn

Zajmijmy się ustalonymi k kolumnami klocków o wysokościach równych: $x_1, \ldots, x_k$. Zamierzamy je uporządkować, dodając i/lub zdejmując po jednym klocku tak, by ostatecznie były tej samej wysokości.

Niech $c_x(h)$ dla $x = (x_1, \ldots, x_k)$ oznacza koszt wyrównania rozważanych kolumn do wysokości h. Funkcję tę możemy wyrazić następującym wzorem:

$$c_x(h) = \sum_{i=1}^{k} |h - x_i|.$$

Przykład 1. Rozważmy konfigurację klocków podaną w przykładowych danych do zadania, czyli ciąg $x = (3,9,2,3,1)$. Na rys. 1 przedstawiony jest wykres funkcji $c_x(h)$ dla tego ciągu.



Rys. 1: Wykres wartości funkcji $c_x(h)$ dla ciągu $x = (3,9,2,3,1)$.

Na podstawie wykresu możemy zaobserwować, że funkcja $c_x(h)$ najpierw maleje, osiągając w pewnym punkcie minimum, a następnie rośnie. To, co nas interesuje, to wysokość h, dla której to minimum jest osiągane.

Poszukując „optymalnego" h, przyjrzyjmy się bliżej naszej funkcji. Najpierw zauważmy, że dla $h = 0$ wartością $c_x(0)$ jest:

$$c_x(0) = \sum_{i=1}^{k} x_i.$$

Następnie zobaczmy, jak zmienia się funkcja po zwiększeniu wysokości o 1. Otóż mając wartość $c_x(h-1)$ i chcąc obliczyć wartość $c_x(h)$:

- musimy dołożyć po jednym klocku do każdej kolumny, której wysokość jest mniejsza lub równa $h-1$,
- dla każdej kolumny, która początkowo miała wysokość większą lub równą h, oszczędzamy jeden ruch w porównaniu z uporządkowaniem jej na wysokości $h-1$ (musimy zdjąć jeden klocek mniej).

Bardziej formalnie:

$$\begin{aligned} c_x(h) &= c_x(h-1) + |\{x_i : x_i \leqslant h-1\}| - |\{x_i : x_i \geqslant h\}| \\ &= c_x(h-1) + |\{x_i : x_i \leqslant h-1\}| - (k - |\{x_i : x_i \leqslant h-1\}|) \\ &= c_x(h-1) + 2|\{x_i : x_i \leqslant h-1\}| - k. \end{aligned}$$

Niech $p_x(h)$ oznacza liczbę elementów ciągu x o wartości mniejszej lub równej h, czyli $p_x(h) = |\{x_i : x_i \leqslant h\}|$. Teraz wartości funkcji $c_x(h)$ możemy zapisać jako:

$$\begin{aligned} c_x(h) &= c_x(h-1) + 2p_x(h-1) - k \\ &= c_x(0) + \sum_{i=0}^{h-1} (2p_x(i) - k). \end{aligned}$$

Ponieważ funkcja $p_x(h)$ jest niemalejąca, to przy poszukiwaniu minimalnej wartości $c_x(h)$ opłaca się nam zwiększać h tak długo, jak długo składniki sumy w powyższym wzorze są ujemne. To oznacza, że $c_x(h)$ osiąga minimalną wartość dla wysokości $h = H$, takiej że

$$2p_x(H-1) - k < 0 \quad \text{oraz} \quad 2p_x(H) - k \geqslant 0. \tag{1}$$

Wartość H spełniająca nierówności (1) zawsze istnieje, ponieważ funkcja $p_x(i)$ zmienia wartości od 0 do k dla $i = -1, \ldots, \max\{x_j : 1 \leqslant j \leqslant k\}$, co oznacza, że funkcja $2p_x(i) - k$ dla tych samych argumentów przyjmuje wartości od $-k$ do k.

Spójrzmy raz jeszcze na nierówności (1). Dla poszukiwanej wysokości H chcemy mieć w ciągu x:

- mniej niż $\lceil k/2 \rceil$ kolumn o wysokościach mniejszych bądź równych $H-1$ oraz

- co najmniej $\lceil k/2 \rceil$ kolumn o wysokościach mniejszych bądź równych H.

Podstawiając za H wysokość $\lceil k/2 \rceil$-tej co do wysokości kolumny ciągu x, spełniamy oba powyższe warunki (a wyznaczony w ten sposób element nazywamy *medianą* ciągu x).

Wniosek 1. Funkcja $c_x(h)$ osiąga minimum dla H równego wartości $\lceil k/2 \rceil$-tego co do wielkości elementu ciągu x, gdzie k jest długością ciągu x.

Wybór kolumn

Korzystając z wniosku 1, możemy zapisać wstępny szkic algorytmu znajdowania najłatwiejszych do uporządkowania kolumn w całym ciągu $a_1, \ldots, a_n$:

```
1:  wynik := +∞;
2:  for i := 1,...,n+1−k do begin
3:    wyznacz medianę H ciągu x = (a_i,...,a_{i+k−1});
4:    oblicz optymalny koszt c_x(H) dla tego ciągu;
5:    wynik := min(wynik, c_x(H));
6:  end
```

Medianę ciągu k–elementowego potrafimy wyznaczyć w oczekiwanym czasie $O(k)$, korzystając z algorytmu Hoare'a (niestety pesymistyczny koszt tego algorytmu to $O(k^2)$). Możemy również zastosować algorytm Magicznych Piątek, którego pesymistyczny koszt wynosi $O(k)$ (niestety jest on trudniejszy w implementacji, a jego złożoność, choć teoretycznie dobra, jest opatrzona dużą stałą; przez to algorytm jest raczej mało praktyczny). Opisy obu algorytmów można odnaleźć w książkach [17, 20]. Wartości $c_x(H)$ można łatwo obliczyć (wprost z definicji) w czasie $O(k)$. To oznacza, że na razie mamy rozwiązanie zadania, którego złożoność czasowa wynosi $O(nk)$. Jeśli k jest małą stałą, to złożoność $O(nk)$ w zupełności nam wystarczy. Jednak według warunków zadania parametr k może osiągać wartości rzędu $\Theta(n)$, czyli złożoność czasową całego algorytmu musimy oszacować jako $O(n^2)$. W takim razie nasz algorytm wymaga jeszcze usprawnień.

Zauważmy, że kolejne ciągi x, dla których wyznaczamy medianę w kroku 3. algorytmu, są do siebie bardzo podobne. W i-tej iteracji analizujemy ciąg $(a_i, a_{i+1}, \ldots, a_{i+k-1})$, a w kolejnej — ciąg $(a_{i+1}, \ldots, a_{i+k-1}, a_{i+k})$. Ciągi te różnią się tylko dwoma elementami. Korzystając z tej obserwacji, możemy stworzyć wydajniejszy algorytm, jeśli tylko znajdziemy strukturę danych S, dla której będziemy potrafili efektywnie wykonać operacje:

- $przygotuj(x_1, x_2, \ldots, x_k)$ — umieszczenie w strukturze ciągu x o długości k,
- $dodaj(y)$ — dodanie do struktury elementu o wartości y,
- $usun(y)$ — usunięcie ze struktury elementu o wartości y,
- $mediana()$ — wyznaczenie mediany ciągu przechowywanego w strukturze,
- $minKoszt()$ — wyznaczenie wartości funkcji $c_x(mediana())$ dla ciągu przechowywanego w strukturze.

Używając struktury S, możemy zapisać następujący algorytm:

```
1:  S.przygotuj(a_1, ..., a_k);
2:  wynik := S.minKoszt();
3:  for i := 1, ..., n-k do begin
4:    S.usun(a_i);
5:    S.dodaj(a_{i+k});
6:    wynik := min(wynik, S.minKoszt());
7:  end
```

Pierwsza implementacja struktury — ulepszamy drzewa czerwono-czarne. Poszukajmy efektywnej implementacji struktury S. Jednym z możliwych rozwiązań jest zastosowanie struktury słownikowej (np. drzew czerwono-czarnych) wzbogaconej o dodatkowe możliwości. Sama struktura słownikowa pozwala nam efektywnie wyszukiwać, dodawać i usuwać elementy. W razie potrzeby możemy jednak do każdego węzła v dodać informacje typu: liczba czy suma elementów w poddrzewie ukorzenionym w v, minimalny lub maksymalny element w poddrzewie v. Informacje te stosunkowo łatwo można uaktualniać w czasie modyfikacji wykonywanych na strukturze (na przykład dodawania czy usuwania węzłów, rotacji itp.) bez zwiększania złożoności tych operacji. Dla naszych celów przydatne będzie zapisanie w każdym węźle v drzewa czerwono-czarnego:

- liczby elementów w poddrzewie ukorzenionym w v,
- sumy elementów w poddrzewie ukorzenionym w v.

Operacje dodawania i usuwania elementów wymagają oczywiście czasu $O(\log k)$. Korzystając z atrybutu *liczba elementów w poddrzewie*, możemy w czasie $O(\log k)$ odnaleźć medianę ciągu. Funkcję *minKoszt*() możemy również obliczyć w czasie $O(\log k)$. Wymaga to jedynie:

- odnalezienia węzła zawierającego medianę,
- wyznaczenia sumy oraz liczby kluczy leżących w drzewie na lewo od mediany (będą to elementy ciągu o wartości nie większej od mediany) — nazwijmy ten multizbiór[1] kluczy M,
- wyznaczenia sumy oraz liczby kluczy leżących w drzewie na prawo od mediany (będą to elementy ciągu o wartości nie mniejszej od mediany) — nazwijmy ten multizbiór kluczy W,
- obliczenia funkcji $c_x(mediana)$ za pomocą wzoru:

$$c_x(mediana) = (|M| \cdot mediana - \sum_{x \in M} x) + (\sum_{x \in W} x - |W| \cdot mediana).$$

Opisana implementacja struktury S pozwala zatem rozwiązać nasze zadanie w złożoności czasowej $O(n \log k)$. Niestety ma ona pewną dość istotną wadę — wymaga samodzielnej implementacji drzew czerwono-czarnych. Co prawda biblioteka STL (dla języka C++)

[1] Multizbiór to zbiór, w którym elementy o takiej samej wartości mogą występować wielokrotnie.

oferuje struktury słownikowe, jednak w chwili obecnej nie umożliwia wzbogacenia ich o wybrane atrybuty.

W przypadku naszego zadania można jednakże zaimplementować opisaną strukturę S znacznie prościej i w nieznacznie gorszej złożoności czasowej — $O(n \log n)$. W tym celu wystarczy na samym początku algorytmu posortować wszystkie wysokości kolumn klocków i zbudować statyczną, zrównoważoną strukturę drzewa — w postaci drzewa BST lub w postaci drzewa przedziałowego[2]. Wówczas wszystkie opisane operacje można zrealizować za pomocą analogicznego jak w powyższym opisie wzbogacenia tego drzewa. Struktura statyczna jest dużo przyjemniejsza w implementacji — można ją utrzymywać bezpośrednio w tablicy, a ponadto przy wstawianiu i usuwaniu elementów nie trzeba się martwić o operacje równoważące — i jedynie nieznacznie wolniejsza — wzrost kosztu czasowego wynika z konieczności wstępnego posortowania danych oraz stąd, że wysokość drzewa jest w tym przypadku rzędu $\Theta(\log n)$, a nie $\Theta(\log k)$ jak poprzednio. Implementację rozwiązania wykorzystującego statyczne drzewo zrównoważone można znaleźć w plikach `klo.cpp` oraz `klo1.pas`.

Druga implementacja struktury — po rozbiciu na części jest łatwiej. Strukturę S możemy także zaimplementować mniej pracochłonnie. Dla zbioru $(x_1, \ldots, x_k)$ stworzymy następujące „składowe" struktury:

- *mediana* — element stanowiący medianę ciągu,
- *mniejsze* — multizbiór elementów mniejszych bądź równych medianie,
- *wieksze* — multizbiór elementów większych bądź równych medianie,
- *sumaMniejsze*, *sumaWieksze* — odpowiednio suma elementów w multizbiorze *mniejsze*/*wieksze*.

Będziemy przy tym dbać, by w zbiorach *mniejsze* i *wieksze* było po tyle samo (z dokładnością do 1) elementów, czyli aby spełniony był niezmiennik:

$$|mniejsze| = \left\lceil \frac{|mniejsze| + |wieksze| + 1}{2} \right\rceil - 1.$$

Multizbiory *mniejsze*/*wieksze* możemy zaimplementować za pomocą struktury `multiset` z biblioteki STL.

Powracając do zadania, które musimy rozwiązać, pokażmy, jak za pomocą struktury S wykonać wszystkie potrzebne w algorytmie operacje:

- *przygotuj*(x) — możemy posortować zbiór x, wybrać jego medianę, a elementy od niej mniejsze (odpowiednio, większe) umieścić w multizbiorze *mniejsze* (odpowiednio, w multizbiorze *wieksze*), przy okazji zliczając sumy elementów;
- *mediana*$()$ — na to pytanie odpowiadamy w czasie stałym, podając wartość pola *mediana*;

[2] O statycznych drzewach przedziałowych można poczytać np. w opisie rozwiązania zadania Tetris 3D w [13].

- *minKoszt*() — wyznaczamy w czasie stałym wartość funkcji $c_x(mediana())$, korzystając z wartości *mediana*, *sumaMniejsze*, *sumaWieksze* oraz z liczby elementów w multizbiorach *mniejsze*/*wieksze*;

- *dodaj*(y) — najpierw dodajemy element do jednego z multizbiorów: jeśli $y \leqslant mediana$, to dodajemy element y do *mniejsze*, w przeciwnym przypadku dodajemy ten element do *wieksze*. Niestety, w ten sposób możemy zaburzyć niezmiennik:

$$|mniejsze| = \left\lceil \frac{|mniejsze| + |wieksze| + 1}{2} \right\rceil - 1.$$

  Aby go przywrócić, porównujemy moce zbiorów *mniejsze* i *wieksze*:

  – jeśli w zbiorze *mniejsze* brakuje jednego elementu, to dodajemy wartość *mediana* do zbioru *mniejsze*, ze zbioru *wieksze* usuwamy najmniejszą wartość i zapisujemy ją w polu *mediana*;
  – analogicznie, jeśli zbiór *mniejsze* zawiera o jeden element za dużo, to dodajemy wartość *mediana* do zbioru *wieksze*, ze zbioru *mniejsze* usuwamy największą wartość i zapisujemy ją w polu *mediana*.

  Struktura `multiset` z biblioteki STL pozwala wykonywać operacje dodawania, usuwania oraz znajdowania największego i najmniejszego elementu w czasie $O(\log k)$. Samodzielnie możemy ją także zaimplementować za pomocą drzew zrównoważonych lub odpowiednio wzbogaconych (np. podobnie jak w algorytmie Dijkstry) kopców[3].

- *usun*(y) — jeśli $y < mediana$, to usuwamy element y z *mniejsze*, jeśli $y > mediana$, to usuwamy element y z *wieksze*; w ostatnim możliwym przypadku mamy $y = mediana$ — usuwamy więc element y z pola *mediana*, a za nową wartość mediany przyjmujemy np. najmniejszy element ze zbioru *wieksze*. Na koniec równoważymy zbiory *mniejsze*/*wieksze*, podobnie jak to robiliśmy przy operacji *dodaj*.

W rozwiązaniu wzorcowym zastosowaliśmy wspomnianą strukturę `multiset` — jest ono zaimplementowane w pliku `klo2.cpp`. Ponieważ pesymistyczny koszt operacji *dodaj*/*usun* wynosi $O(\log k)$, to złożoność czasowa całego algorytmu to $O(n \log k)$.

Testy

Rozwiązania zawodników były sprawdzane na 28 testach, zgrupowanych w 11 zestawów.

Nazwa	n	k	Opis
klo1a.in	10	5	test poprawnościowy małego rozmiaru
klo1b.in	10	5	test poprawnościowy małego rozmiaru
klo1c.in	1	1	test poprawnościowy małego rozmiaru

[3] O wielu rodzajach drzew zrównoważonych i o kopcach można poczytać w podstawowych podręcznikach z algorytmiki: [15, 20].

Nazwa	n	k	Opis
klo1d.in	100	10	test poprawnościowy małego rozmiaru
klo2a.in	100	20	test poprawnościowy małego rozmiaru
klo2b.in	100	50	test poprawnościowy małego rozmiaru
klo2c.in	100	100	test poprawnościowy małego rozmiaru
klo3a.in	99	15	test poprawnościowy małego rozmiaru
klo3b.in	100	50	test poprawnościowy małego rozmiaru
klo4a.in	10 000	101	test poprawnościowy średniego rozmiaru
klo4b.in	10 000	5 000	test poprawnościowy średniego rozmiaru
klo5a.in	100 000	1	test poprawnościowy średniego rozmiaru
klo5b.in	100 000	1 000	test poprawnościowy średniego rozmiaru
klo5c.in	100 000	1 003	test poprawnościowy średniego rozmiaru
klo6a.in	100 000	2	test wydajnościowy
klo6b.in	100 000	10 001	test wydajnościowy
klo6c.in	99 889	1 000	test wydajnościowy
klo7a.in	100 000	10 000	test wydajnościowy
klo7b.in	100 000	10 000	test wydajnościowy
klo8a.in	100 000	10 000	test wydajnościowy
klo8b.in	100 000	10 000	test wydajnościowy
klo9a.in	90 000	90 000	test wydajnościowy
klo9b.in	100 000	10 000	test wydajnościowy
klo10a.in	99 999	101	test wydajnościowy
klo10b.in	100 000	10 000	test wydajnościowy
klo11a.in	95 000	1 001	test wydajnościowy
klo11b.in	100 000	10 000	test wydajnościowy
klo11c.in	100 000	10 000	test wydajnościowy

Jakub Radoszewski — Treść zadania
Michał Pilipczuk, Jakub Radoszewski — Opracowanie
Michał Pilipczuk — Program

OI, Etap I, 22.10–19.11.2007

Plakatowanie

Wszystkie budynki we wschodniej części Bajtogrodu zostały zbudowane zgodnie z zasadami starego bajtobudownictwa: stoją one jeden przy drugim (nie ma między nimi przerw). Razem tworzą bardzo długą ścianę budynków o zróżnicowanej wysokości, ciągnącą się ze wschodu na zachód.

Burmistrz Bajtogrodu, Bajtazar, postanowił, że ścianę budynków należy od północnej strony pokryć plakatami. Bajtazar zastanawia się, jaką minimalną liczbą plakatów można pokryć całą północną ścianę budynków. Plakaty powinny mieć kształt prostokątów o bokach pionowych i poziomych. Plakaty nie mogą zachodzić na siebie, natomiast mogą stykać się brzegami. Każdy plakat musi w całości przylegać do ścian pewnych budynków i cała powierzchnia północnych ścian budynków musi być pokryta plakatami.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opisy budynków,*
- *wyznaczy minimalną liczbę plakatów potrzebnych do całkowitego pokrycia ich północnych ścian,*
- *wypisze wynik na standardowe wyjście.*

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę całkowitą n ($1 \leqslant n \leqslant 250\,000$), oznaczającą liczbę budynków stojących w rzędzie. Kolejne n wierszy zawiera po dwie liczby całkowite d_i i w_i ($1 \leqslant d_i, w_i \leqslant 1\,000\,000\,000$), oddzielone pojedynczym odstępem i oznaczające długość i wysokość i-tego budynku w rzędzie.

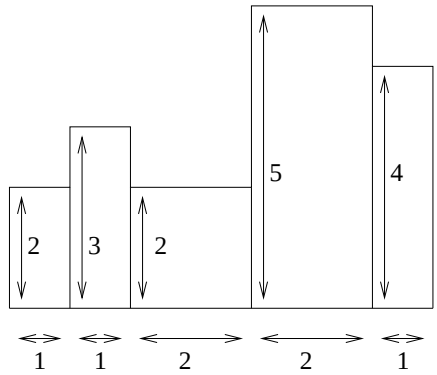
Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać jedną liczbę całkowitą — minimalną liczbę prostokątnych plakatów, którymi można całkowicie pokryć północne ściany budynków.

Przykład

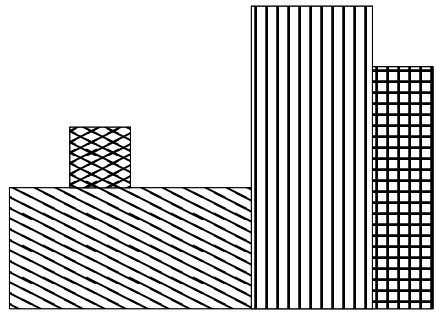
Dla danych wejściowych:

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



poprawnym wynikiem jest:

```
4
```



Na rysunkach została przedstawiona sama północna ściana rzędu budynków. Drugi z rysunków przedstawia przykładowe pokrycie ściany czterema plakatami.

Rozwiązanie

Dolne ograniczenie i uproszczenia

Na początek spróbujmy określić, ile co najmniej plakatów jest potrzebnych do pokrycia całej północnej ściany budynków, czyli jakie jest *dolne ograniczenie* na ich liczbę. Potem pokażemy algorytm, który wyznacza plakatowanie wykorzystujące dokładnie tyle plakatów.

Zauważmy, że północna ściana każdego budynku musi zostać w całości pokryta, więc każdemu budynkowi możemy jednoznacznie przyporządkować plakat, który pokrywa jego *lewy górny róg*. W takim przyporządkowaniu jeden plakat może odpowiadać dwóm różnym budynkom tylko wtedy, gdy są one tej samej wysokości i między nimi nie stoi żaden budynek niższy od nich. Wśród budynków, którym jest przyporządkowany ten sam plakat, możemy wyróżnić jeden — stojący najbardziej na lewo, czyli na zachód. Dokładniej określa to następująca definicja.

Definicja 1. Budynek nazywamy *redundantnym*, jeśli istnieje budynek położony na lewo od niego i takiej samej wysokości jak on, taki że pomiędzy tymi budynkami nie ma żadnego niższego od nich budynku. Jeżeli ten warunek dla danego budynku nie zachodzi, to nazywamy go *nieredundantnym*.

Przykład 1. W teście przykładowym z treści zadania trzeci budynek od lewej jest redundantny, a pozostałe są nieredundantne.

Łatwo zauważyć, że każdym dwóm budynkom nieredundantnym muszą być przyporządkowane różne plakaty, co pozwala nam sformułować następujący wniosek.

Obserwacja 1. *Liczba budynków nieredundantnych* jest *dolnym ograniczeniem na liczbę plakatów* użytych w dowolnym plakatowaniu.

Warto także poczynić kolejne obserwacje, które pozwalają lepiej określić, na czym polega trudność problemu.

Obserwacja 2. Jeśli wszystkie budynki w Bajtogrodzie są różnej wysokości, to wszystkie one są nieredundantne i do ich pokrycia potrzeba n plakatów! Takie pokrycie można łatwo skonstruować, naklejając na każdy budynek osobny plakat. Widać więc, że problem stanowią jedynie budynki tej samej wysokości.

Obserwacja 3. Szerokości budynków nie mają znaczenia dla rozwiązania, więc możemy założyć, że szerokość każdego budynku wynosi 1.

Algorytm

W rozwiązaniu wzorcowym będziemy przeglądać budynki od lewej do prawej. Za każdym razem, gdy napotkamy budynek nieredundantny, dodamy do rozwiązania nowy plakat rozpoczynający się w jego górnym, lewym narożniku. Rozmiar plakatu ustalimy tak, by stworzyć plakatowanie pokrywające wszystkie budynki. W szczególności, górny fragment każdego redundantnego budynku zostanie pokryty za pomocą plakatu, rozpoczynającego się na ostatnim budynku nieredundantnym o tej samej wysokości, położonym na lewo od niego. Założony cel osiągniemy, jeśli będziemy przestrzegać następujących kryteriów. Dla budynku nieredundantnego b i przyporządkowanego mu plakatu p_b:

(i) lewy, górny róg plakatu p_b będzie przyklejony dokładnie w lewym, górnym rogu budynku b;

(ii) dolny brzeg plakatu p_b będzie umieszczony najniżej, jak tylko się da — tak aby plakat nie nachodził na żaden z wcześniej umieszczonych plakatów (a zatem dolna krawędź świeżo przyklejonego plakatu będzie od dołu dotykać górnej krawędzi pewnego innego plakatu $p_{b'}$, dla budynku b' stojącego na lewo od b, albo ziemi);

(iii) prawy brzeg plakatu p_b będzie umieszczony najdalej, jak tylko się da — tak, aby plakat nie wystawał poza ścianę żadnego budynku; to oznacza, że p_b zostanie zakończony tuż przed rozpoczęciem pierwszego budynku b'' niższego od b i położonego na prawo od b.

Skonstruujemy teraz algorytm wyznaczający plakatowanie zgodne z kryteriami (i) – (iii). Plakaty, które będziemy przyklejać, będą „prawostronnie otwarte", czyli w momencie dodawania ich do rozwiązania nie będziemy zastanawiać się nad tym, gdzie dany plakat powinien mieć swój prawy brzeg. Określimy to w jednym z dalszych kroków algorytmu,

gdy napotkamy budynek, na którym plakat ten powinien zakończyć się zgodnie z kryterium (iii). W każdym momencie, wszystkie rozpoczęte i jeszcze nie zakończone plakaty będziemy trzymać na *stosie S*, uporządkowane od najwyżej do najniżej położonych. Zadbamy przy tym, by *każdy plakat znajdujący się na stosie stykał się dolnym brzegiem z górnym brzegiem pewnego innego plakatu ze stosu albo z ziemią*. Warunek ten będzie *niezmiennikiem* naszej procedury. Jego przestrzeganie zagwarantuje, że plakaty znajdujące się na stosie będą całkowicie pokrywały aktualnie rozważaną pozycję w rzędzie budynków, a na zakończenie algorytmu na ścianie nie pozostaną żadne nieoplakatowane „dziury".

Oznaczmy kolejne budynki opisane w danych zadania $b_1, b_2, \ldots, b_n$. Rozważmy teraz krok algorytmu, w którym przeanalizowaliśmy już $i-1$ budynków i aktualnie rozważamy budynek b_i o wysokości w_i. Zgodnie z niezmiennikiem algorytmu, stos S zawiera w tym momencie ciąg plakatów $p_1, \ldots, p_j$ takich, że p_1 styka się z ziemią, p_2 styka się dolnym brzegiem z górnym brzegiem p_1, itd. aż do p_j, który od dołu styka się z p_{j-1}. Aby zachować kryterium (iii), musimy w tym momencie zakończyć wszystkie otwarte plakaty sięgające wysokości większej niż w_i — usuwamy je więc ze stosu. Po tej operacji na stosie pozostają plakaty $p_1, \ldots, p_k$, dla pewnego $k \leqslant j$ (jeśli usuniemy ze stosu wszystkie plakaty, czyli $S = \emptyset$, to dla uproszczenia możemy przyjąć, że p_k oznacza poziom zerowy, czyli ziemię). Spełniają one oczywiście niezmiennik algorytmu, skoro spełniały go plakaty $p_1, \ldots, p_j$.

Niech $w \leqslant w_i$ oznacza wysokość, do jakiej sięga najwyższy pozostały na stosie plakat p_k. Zauważmy, że jeżeli budynek b_i jest redundantny, to musi zachodzić $w = w_i$. Faktycznie, niech b_l będzie najbliższym na lewo od b_i budynkiem nieredundantnym o wysokości $w_l = w_i$. Wówczas między b_l a b_i z definicji nie ma żadnego budynku o wysokości mniejszej niż w_i, czyli plakat rozpoczęty w lewym górnym rogu b_l nie został jeszcze prawostronnie zakończony. Ponieważ górny brzeg tego plakatu jest położony na wysokości w_i, to musi to być plakat p_k o wysokości $w = w_i$. To oznacza, że budynek b_i możemy pokryć, przedłużając w prawo o d_i wszystkie plakaty z S i nie dodając do rozwiązania żadnego nowego plakatu.

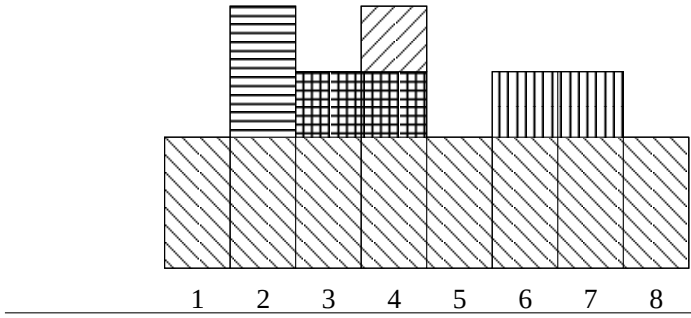
W przypadku, gdy budynek b_i jest nieredundantny, zachodzi natomiast nierówność $w < w_i$. Łatwo to wykazać, przeprowadzając rozumowanie analogiczne jak poprzednio. Z budynkiem b_i związemy więc plakat p, rozpoczynający się w jego lewym górnym rogu (zgodnie z kryterium (i)) i stykający się dolnym brzegiem z plakatem p_k (zgodnie z kryterium (ii)). Plakat p pozostawimy w tym momencie prawostronnie otwarty, tzn. na razie nie będzie nas interesować, kiedy i gdzie on się zakończy. Umieścimy go natomiast na szczycie stosu S, który od tej chwili zawiera kolejno plakaty: $p_1, \ldots, p_k, p$, a zatem wciąż spełnia niezmiennik algorytmu. Co więcej, przedłużenie wszystkich plakatów z S w prawo o szerokość budynku b_i spowoduje całkowite pokrycie ściany tego budynku.

W ten sposób opisaliśmy i-ty krok algorytmu, w którym uzupełniamy plakatowanie zgodnie z kryteriami (i) – (iii), a po jego wykonaniu budynek b_i jest całkowicie oplakatowany i ponadto zachowany jest niezmiennik algorytmu. Prosty dowód indukcyjny pozwala wykazać, że wykonanie kolejnych kroków algorytmu dla $i = 1, 2, \ldots, n$ doprowadza do konstrukcji poprawnego plakatowania o liczności równej liczbie nieredundantnych budynków w ciągu.

Poniżej przedstawiamy pseudokod opisanego algorytmu. Zmienna S oznacza w nim stos plakatów, które reprezentujemy za pomocą wysokości ich górnego brzegu. Na stosie możemy wykonywać podstawowe operacje: wstawienie elementu na szczyt (operacja *push*), odczytanie elementu ze szczytu (operacja *top*), usunięcie elementu ze szczytu (operacja *pop*) oraz sprawdzenie, czy stos jest pusty.

```
1:  S := ∅;
2:  p := 0; { liczba użytych plakatów }
3:  for i := 1 to n do
4:  begin
5:    while S ≠ ∅ and S.top() > w_i do
6:      S.pop();
7:    if S = ∅ or S.top() < w_i then
8:    begin
9:      { analizowany budynek jest nieredundantny }
10:     S.push(w_i);
11:     p := p + 1;
12:   end;
13: end;
14: return p;
```

Przykład 2. Na rys. 1 przedstawione jest znalezione przez opisany algorytm pokrycie rzędu budynków o wysokościach: 2, 4, 3, 4, 2, 3, 3, 2 za pomocą pięciu plakatów.



Rys. 1: Budynki nieredundantne na rysunku to budynki o numerach: 1, 2, 3, 4 i 6. W ich lewych górnych rogach przyklejone są lewe górne rogi pięciu plakatów. Pozostałe budynki (o numerach: 5, 7 i 8) są redundantne, a ich górne fragmenty są pokryte plakatami przyporządkowanymi budynkom nieredundantnym, odpowiednio 1, 6, 1.

Złożoność czasowa algorytmu

W jednym kroku pętli **for** (wiersze 3–13) łączna liczba wykonanych operacji może być nawet rzędu $O(n)$, ze względu na liczbę iteracji pętli **while** z wierszy 5–6. Daje to oszacowanie złożoności czasowej algorytmu równe $O(n^2)$. Przekonamy się jednak, że jest to szacowanie bardzo zawyżone, zliczając nieco dokładniej liczbę operacji wykonywanych w algorytmie (metodą zwaną *analizą kosztu zamortyzowanego*[1]).

Otóż okazuje się, że sumaryczna liczba obrotów pętli **while**, we wszystkich iteracjach pętli **for**, jest nie większa niż n. Faktycznie, w instrukcji **while** wykonujemy operację usunięcia plakatu ze stosu S. Operację tę możemy wykonać tylko n razy w całym algorytmie,

[1] Więcej informacji na temat analizy kosztu zamortyzowanego można znaleźć w [20]

bo najwyżej tyle plakatów zużyjemy, a każdy z nich tylko raz trafi na stos (operacja w wierszu 10) i tylko raz może zostać z niego zdjęty.

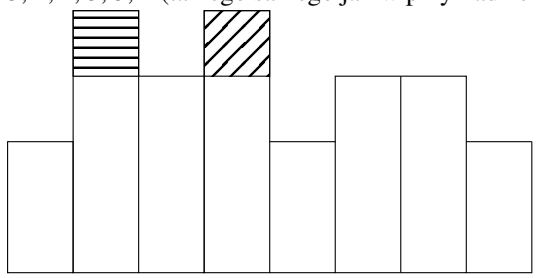
Wszystkie wymagane operacje na stosie potrafimy wykonywać w czasie stałym. Dodatkowo łączny koszt czasowy wszystkich operacji algorytmu wykonywanych poza pętlą **while** jest nie większy niż $O(n)$. Wynika stąd, że złożoność czasowa całego algorytmu wynosi $O(n)$. Opisane rozwiązanie zostało zaimplementowane w plikach `pla.c`, `pla1.pas` oraz `pla6.java`.

Inne rozwiązania

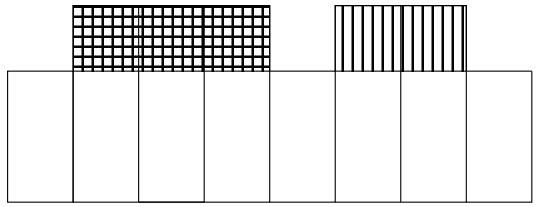
Metoda pokazana w algorytmie wzorcowym nie jest jedyna — istnieje wiele innych sposobów plakatowania, pozwalających uzyskać maksymalną liczbę punktów za zadanie. Naszkicujemy je poniżej, pomijając szczegółowe dowody poprawności. We wspomnianych rozwiązaniach kolejność przeglądania budynków jest inna niż w rozwiązaniu wzorcowym, a mianowicie: od najwyższych bądź od najniższych.

Plakatowanie „od góry". Na początku zajmujemy się budynkami najwyższymi — załóżmy, że każdy z nich ma wysokość h. Jeśli takie budynki stoją obok siebie, to „sklejamy" je w jeden. Na górną część każdego z najwyższych budynków nalepiamy nowy plakat. Plakat kończy się od dołu na wysokości wyższego z dwóch sąsiadów rozważanego budynku. Następnie „obcinamy" zalepione plakatami fragmenty najwyższych budynków i pozostaje nam do oplakatowania ciąg niższych budynków, z którym postępujemy analogicznie.

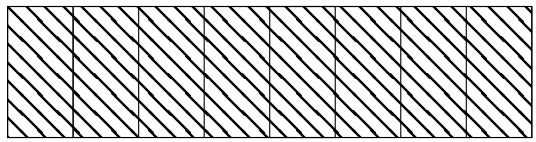
Przykład 3. Rozważmy kolejne kroki plakatowania „od góry" dla ciągu budynków o wysokościach 2, 4, 3, 4, 2, 3, 3, 2 (takiego samego jak w przykładzie 2).



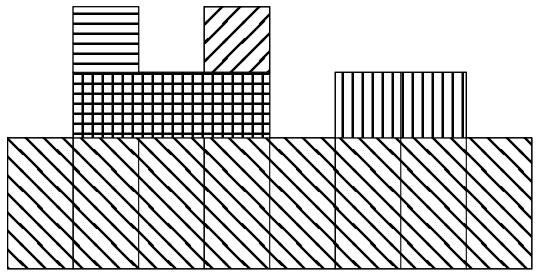
Rys. 2: Najwyższe budynki mają wysokość $h = 4$ i nie sąsiadują ze sobą, więc nie trzeba ich sklejać. Na ich górne części naklejamy dwa plakaty, a następnie obniżamy każdy z budynków do wysokości wyższego sąsiada, czyli $\max(2,3) = 3$.



Rys. 3: Po redukcji mamy rząd budynków, w którym najwyższe mają wysokość $h' = 3$. Tworzą one dwie grupy sąsiadujących budynków jednakowej wysokości — każdą z nich traktujemy jak jeden budynek. Na górną część każdej grupy naklejamy jeden plakat, redukując tym samym problem do wysokości 2.



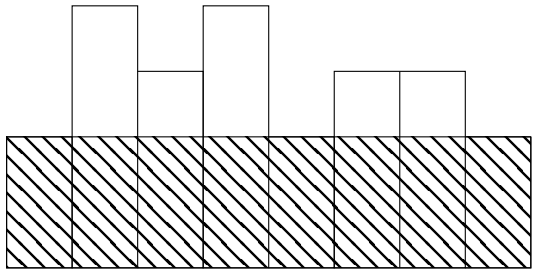
Rys. 4: Na koniec pozostaje nam rząd budynków o wysokości $h'' = 2$. Pokrywamy go jednym plakatem, co kończy działanie algorytmu.



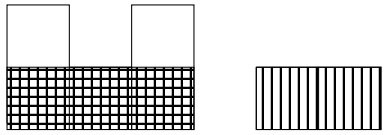
Rys. 5: Ostateczna postać rozwiązania skonstruowanego przez opisany algorytm, czyli plakatowania o liczności 5. Zauważmy, że uzyskane plakatowanie jest inne niż znalezione przez rozwiązanie wzorcowe, ale wykorzystuje tyle samo plakatów.

Plakatowanie „od ziemi". Optymalne rozwiązanie można także uzyskać, przeglądając budynki od najniższych do najwyższych. Rozpoczynamy od naklejenia tuż nad ziemią plakatu o wysokości równej wysokości najniższego budynku i szerokości równej sumarycznej szerokości wszystkich budynków. Następnie „ucinamy" oplakatowane części wszystkich budynków. W ten sposób dostajemy jedną bądź klika grup budynków o mniejszej wysokości niż wyjściowa. Każdy z uzyskanych podproblemów rozwiązujemy w analogiczny sposób, ponownie zaklejając jednym plakatem o szerokości równej szerokości całej grupy dolne części budynków.

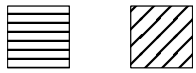
Przykład 4. Rozważmy kolejne kroki plakatowania „od ziemi" dla ciągu budynków o wysokościach 2, 4, 3, 4, 2, 3, 3, 2 (takiego samego jak w obydwu dotychczasowych przykładach).



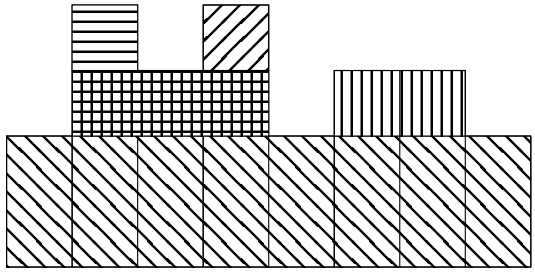
Rys. 6: Najniższy budynek ma wysokość 2. Przyklejamy zatem na dole na całej szerokości ściany plakat o wysokości 2. Potem usuwamy całą pokrytą powierzchnię, co powoduje powstanie dwóch grup budynków, które odtąd będziemy rozważać osobno.



Rys. 7: W każdej z grup najniższy budynek ma wysokość 1. Naklejamy zatem na całej szerokości każdej z grup plakat o tej wysokości. W ten sposób druga grupa jest całkowicie pokryta, natomiast z pierwszej powstają dwa samotnie stojące budynki o wysokości 1.



Rys. 8: Pokrycie dwóch samotnie stojących budynków wymaga oczywiście zużycia dwóch plakatów. Kończy to działanie algorytmu.



Rys. 9: Ostateczna postać rozwiązania skonstruowanego przez opisany algorytm to plakatowanie o liczności 5. Zauważmy, że jest ono takie samo, jak w przypadku poprzedniej metody.

W każdej z przedstawionych metod potrzebna jest odpowiednia struktura danych pozwalająca łatwo wyznaczać kolejne plakaty (czyli najwyższe bądź najwyższe budynki) w kolejnych podproblemach:

- w plakatowaniu „od ziemi” możemy wykorzystać statyczne drzewo przedziałowe; otrzymujemy w ten sposób algorytm o złożoności czasowej $O(n \log n)$, zastosowany w programach `pla2.cpp` i `pla7.java`;
- w plakatowaniu „od ziemi” możemy także wykorzystać statyczne drzewo licznikowe; otrzymujemy w ten sposób algorytm o złożoności czasowej $O(n \log n)$, zastosowany w programach `pla3.cpp` i `pla8.java`;
- w plakatowaniu „od góry” możemy wykorzystać listy wskaźnikowe; pozwala to znaleźć rozwiązanie w czasie $O(n)$ plus czas sortowania n budynków; algorytm jest zaimplementowany w plikach `pla4.cpp` i `pla9.java`;
- w plakatowaniu „od góry” możemy wykorzystać także strukturę danych Find-Union (patrz na przykład [15] lub [20]); pozwala to osiągnąć złożoność czasową $O(n \log^* n)$ plus czas sortowania budynków względem wysokości; implementacja tego rozwiązania znajduje się w plikach `pla5.cpp` i `pla10.java`.

Obie opisane metody mogą być także zaimplementowane prościej, bez odpowiednich struktur danych, co daje rozwiązania o złożoności czasowej $O(n^2)$. Ich implementacje można znaleźć w plikach `plas1.cpp-plas4.cpp`. Pozwalały one uzyskać na zawodach 40% punktów.

Testy

Zadanie było sprawdzane na dziesięciu grupach testów. Testy wydajnościowe były generowane tak, żeby sprawiały jak najwięcej problemów nawet najbardziej pomysłowym rozwiązaniom o złożoności $O(n^2)$.

Nazwa	n	Opis
pla1a.in	60	mały test poprawnościowy
pla1b.in	126	mały test wydajnościowy
pla1c.in	1	przypadek brzegowy — jeden budynek
pla2a.in	1 180	mały test poprawnościowy
pla2b.in	1 152	mały test wydajnościowy
pla2c.in	804	przypadek brzegowy — prawie wszystkie budynki równej wysokości
pla3a.in	1 990	mały test poprawnościowy
pla3b.in	2 445	mały test wydajnościowy
pla3c.in	1 708	przypadek brzegowy — prawie wszystkie budynki różnej wysokości
pla4a.in	3 941	średni test poprawnościowy
pla4b.in	3 951	średni test wydajnościowy

Nazwa	n	Opis
pla5a.in	145 941	średni test poprawnościowy
pla5b.in	163 330	średni test wydajnościowy
pla6a.in	236 836	duży test poprawnościowy
pla6b.in	201 424	duży test wydajnościowy
pla7a.in	246 755	duży test poprawnościowy
pla7b.in	224 023	duży test wydajnościowy
pla8a.in	240 497	duży test poprawnościowy
pla8b.in	250 000	duży test wydajnościowy
pla9a.in	243 763	duży test poprawnościowy
pla9b.in	230 031	duży test wydajnościowy
pla10a.in	247 951	duży test poprawnościowy
pla10b.in	249 089	duży test wydajnościowy

Karol Cwalina — Treść zadania, Opracowanie
Jakub Radoszewski — Program

OI, Etap I, 22.10–19.11.2007

Robinson

Rzucony przez sztorm na bezludną wyspę Robinson zbudował łódkę, na której postanowił wypłynąć w morze i szukać ludzkich siedzib. Jest on doświadczonym żeglarzem, więc jego łódź jest zbudowana zgodnie z zasadami sztuki: ma wzdłużną oś symetrii oraz odpowiedni kształt. Na dziobie jest wąska, stopniowo rozszerza się w stronę środka, by z kolei zacząć się zwężać w kierunku rufy. W szczególności oznacza to, że w pewnym miejscu w środku łódź jest szersza niż na dziobie i na rufie.

Pech chciał, że w miejscu, w którym Robinson spuścił swoją łódź na wodę, rosną bardzo gęste trzciny. Co gorsza, są one tak sztywne, że łódź nie jest w stanie ich złamać i przepłynąć przez zajmowane przez trzciny miejsca. Być może, odpowiednio manewrując łodzią, Robinson jest w stanie wypłynąć na otwarte morze.

Ze względu na słabą manewrowość, łódź może się poruszać do przodu, do tyłu, w prawo oraz w lewo, ale nie może skręcać. Oznacza to, że może się zdarzyć sytuacja, gdy łódź porusza się rufą lub którąś burtą do przodu.

Twoim zadaniem jest ocenić, czy Robinson może wydostać się na pełne morze.

Dla uproszczenia, wyspa wraz z otoczeniem jest w zadaniu reprezentowana przez kwadratową mapę podzieloną na kwadratowe pola jednostkowe, z których każde może być zajęte przez wodę, kawałek łodzi Robinsona bądź przez przeszkodę (np. ląd lub trzcinę). Łódź jest ustawiona równolegle do jednego z czterech kierunków świata i właśnie w tym kierunku przebiega przez środek łodzi pas pól jednostkowych, idealnie przepołowionych wzdłużną osią symetrii łodzi.

Przyjmujemy, że poza mapą znajduje się otwarte morze. Robinson może na nie wypłynąć, jeżeli łódź jest w stanie w całości opuścić teren pokazany na mapie. Pojedynczy ruch to przesunięcie się łodzi o jedno pole w wybranym kierunku (północ, południe, wschód lub zachód). Aby ruch był dozwolony, przed jego wykonaniem i po nim łódź musi w całości znajdować się na wodzie.

Zadanie

Napisz programu, który:

- *wczyta ze standardowego wejścia opis mapy,*
- *obliczy minimalną liczbę ruchów łodzi potrzebnych do wypłynięcia poza obszar pokazany na mapie,*
- *wypisze obliczoną liczbę na standardowe wyjście.*

Wejście

Pierwszy wiersz zawiera jedną liczbę całkowitą $3 \leqslant n \leqslant 2\,000$, oznaczającą długość boku mapy. W każdym z następnych n wierszy znajduje się po n znaków opisujących kolejne pola mapy:

i-ty znak w wierszu numer $j+1$ określa zawartość pola o współrzędnych (i,j). W wierszu mogą się pojawić następujące znaki:

- `.` *(kropka) — oznacza pole zajęte przez wodę,*
- `X` *— oznacza przeszkodę (trzcinę lub kawałek lądu),*
- `r` *— oznacza fragment łodzi Robinsona.*

Wyjście

Twój program powinien wypisać (w pierwszym i jedynym wierszu wyjścia) jedną dodatnią liczbę całkowitą, równą najmniejszej liczbie ruchów łodzi niezbędnych do opuszczenia przez nią w całości terenu przedstawionego na mapie. Jeśli wypłynięcie na otwarte morze nie jest możliwe, należy wypisać słowo „`NIE`*”.*

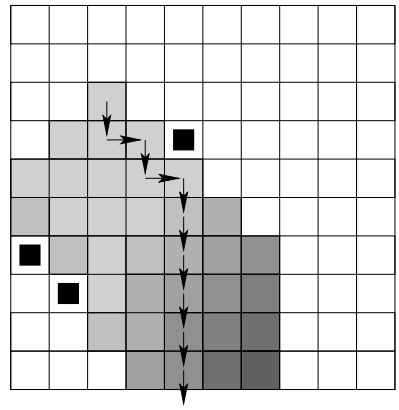
Przykład

Dla danych wejściowych:

```
10
..........
..........
..r.......
.rrrX.....
rrrrr.....
.rrr......
X.r.......
.Xr.......
..........
..........
```

poprawnym wynikiem jest:

```
10
```



Rozwiązanie

Zacznijmy od wybrania najistotniejszych informacji z treści zadania. Otóż:

- łódź może poruszać się w czterech głównych kierunkach, przy czym nie każdy ruch jest dozwolony;
- szukamy najkrótszej ścieżki, która doprowadzi do celu.

Choć bardzo uproszczony, opis ten jednoznacznie wskazuje, że mamy do czynienia z zagadnieniem znajdowania drogi w grafie. To bardzo naturalne dla doświadczonych zawodników spostrzeżenie i jednocześnie bardzo dobra wiadomość: jako jeden z klasycznych

problemów informatyki, zagadnienie to już dawno doczekało się wielu efektywnych algorytmów, na przykład BFS-a, czyli przeszukiwania wszerz[1].

Zanim przystąpimy do pracy nad rozwiązaniem zadania, zauważmy, że możliwe są dwa podstawowe przypadki — oś symetrii łodzi może mieć kierunek poziomy lub pionowy. W dalszej części opracowania założymy, że łódź ma poziomą oś symetrii. Przypadki łodzi o pionowej osi symetrii łatwo rozwiązać analogicznie lub przekształcić do przypadku o osi poziomej.

Dodatkowo, warto wyróżnić pewien punkt łodzi i utożsamiać jej położenie z położeniem tego punktu. Wyróżniony punkt możemy nazwać *punktem zakotwiczenia*, a o łodzi położonej na mapie w ten sposób, że jej punkt zakotwiczenia znajduje się na polu P mapy, powiemy, że *jest zakotwiczona* w polu P lub że *zajmuje pozycję* P. Wówczas wystarczy, byśmy potrafili określić dla każdego punktu na mapie, czy zakotwiczona w nim łódź nie koliduje z żadnymi przeszkodami — taki punkt nazwiemy *dozwolonym* — oraz rozpoznać, kiedy łódź wypłynęła poza mapę. Ponieważ graf nieskierowany, którego wierzchołki odpowiadają interesującym nas pozycjom łodzi, a krawędzie — możliwym ruchom łodzi, ma oczywiście rozmiar $O(n^2)$, to będziemy mogli go przeszukać w złożoności czasowej i pamięciowej $O(n^2)$, czyli najlepszej z możliwych.

W ten sposób sprowadziliśmy zadanie „zaledwie" do wyznaczenia podziału pól mapy na dozwolone i zabronione oraz… odszukania łodzi na mapie. W pierwszej kolejności skupimy się na wykrywaniu, czy pozycja na mapie jest dozwolona. Odnalezienie łodzi i odpowiednie jej przedstawienie w algorytmie zostawiamy na później.

Wyznaczenie pozycji zabronionych — wprawki

Zamiast zastanawiać się, czy określone położenie łodzi jest dozwolone, odwrócimy problem. Dla każdej przeszkody — pola zajętego przez ląd lub szuwary — wyznaczymy takie pola na mapie, że zakotwiczona w nich łódź koliduje z daną przeszkodą.

Rozważmy jedno pole z przeszkodą. Okazuje się, że pozycje łodzi wykluczone przez tę przeszkodę tworzą taki sam kształt jak łódź, tyle że odbity środkowosymetrycznie. Przekonają nas o tym przedstawione niżej obliczenia. Jeśli dla kogoś nie są one wystarczająco czytelne, to warto także przeanalizować rys. 1 (uwaga: przedstawia on łódź, która jest ogólniejszym przykładem, przez co nie spełnia warunków zadania).

Wprowadźmy na mapie naturalny układ współrzędnych, gdzie oś OX jest skierowana na wschód, a oś OY — na północ (czasem będziemy też używali określeń *poziomo* i *w prawo* lub *pionowo* i *do góry*). Za punkt zakotwiczenia łodzi przyjmijmy jej skrajnie lewy punkt — będziemy go nazywać także *dziobem*. Niech

$$L = \{(x_i, y_i) \ : \ i = 1, 2, \ldots\}$$

będzie zbiorem par liczb określających położenie punktów łodzi względem jej punktu zakotwiczenia. Wówczas, jeśli łódź jest na pozycji o współrzędnych (x, y), to zajmuje pola:

$$\{(x + x_i, y + y_i) \ : \ i = 1, 2, \ldots\},$$

[1] BFS i podobne zagadnienia są opisane w licznych podręcznikach algorytmiki, w tym w [20] i [17]. Występują także w wielu zadaniach z poprzednich edycji Olimpiady.

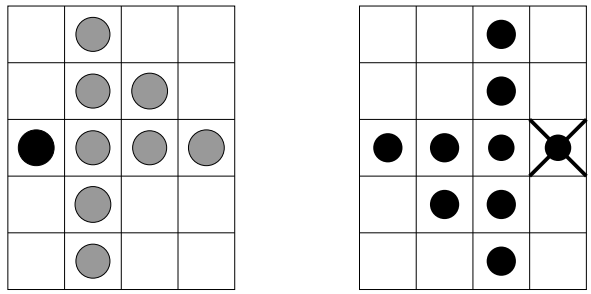
co zapiszemy krócej jako $(x,y)+L$. Jeśli (x_S,y_S) jest polem, na którym znajduje się przeszkoda, to koliduje ona z takimi pozycjami łodzi (x,y), że

$$(\exists i)\ (x,y)+(x_i,y_i)=(x_S,y_S) \Leftrightarrow (\exists i)\ (x,y)=(x_S,y_S)-(x_i,y_i),$$

co jest równoważne temu, że

$$(x,y) \in (x_S,y_S)+(-L).$$

Widzimy więc, że pozycje łodzi wykluczone przez jedną przeszkodę „mają kształt" określony przez zbiór $-L$, czyli środkowosymetryczne odbicie kształtu łódki.



(a) (b)

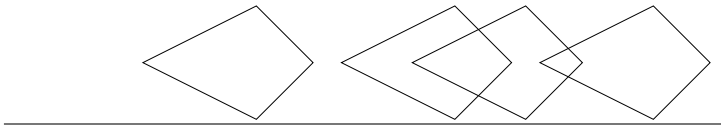
Rys. 1: Na rysunku (a) jest przedstawiona łódź (zbiór L) z wyróżnionym punktem zakotwiczenia — dziobem. Na rysunku (b) pokazano wszystkie pozycje łodzi, które kolidują z przeszkodą znajdującą się w punkcie zaznaczonym krzyżykiem. Widać, że pozycje te tworzą środkowosymetryczne odbicie kształtu łodzi. (Pokazana tu łódź nie jest osiowosymetryczna i przez to nie spełnia warunków zadania, ale pozwala za to lepiej zilustrować zależność pomiędzy zbiorem L a zabronionymi pozycjami).

Przy okazji warto zauważyć, że niektóre pozycje zabronione mogą znaleźć się poza mapą Robinsona. Nie stanowi to żadnego utrudnienia, lecz implementując rozwiązanie, trzeba będzie pamiętać o odpowiednim powiększeniu mapy.

Wyznaczenie pozycji zabronionych — zamiatamy

Wyznaczenie wszystkich pozycji zabronionych sprowadza się zatem do „naniesienia" na mapę pewnej liczby obrazów odbitych środkowosymetrycznie łódek. Prostym i narzucającym się rozwiązaniem jest bezpośrednie zaznaczenie na mapie każdego odbicia związanego z określoną przeszkodą, czyli zbioru $(x_S,y_S)-L$ dla każdego pola (x_S,y_S) zajętego przez przeszkodę. Ponieważ zarówno wielkość łódki, jak i liczba pól zajętych przez przeszkody, mogą być rzędu $O(n^2)$, to otrzymany w ten sposób algorytm ma złożoność $O(n^4)$. Jest więc zbyt wolny, choć oczywiście dla niewielkich testów poprawnościowych może być wystarczający.

Poszukując innej metody, rozważmy prostszy przypadek, gdy wszystkie pola z przeszkodami znajdują się na tej samej szerokości geograficznej, tzn. na tej samej prostej poziomej. Wówczas kształty obejmujące pozycje zabronione możemy przedstawić w uproszczeniu jak na rys. 2.



Rys. 2: Schematyczne przedstawienie zabronionych pozycji łodzi, generowanych przez przeszkody położone na jednej prostej poziomej.

Aby zaznaczyć wszystkie pola kolidujące z ułożonymi we wspomniany sposób przeszkodami, będziemy rozważać kolejne proste pionowe mapy i wykrywać pola zabronione położone na tych prostych. Tradycyjnie taki sposób przeglądania płaszczyzny nazywamy *zamiataniem*, a prostą (w tym przypadku pionową), którą zamiatamy całą płaszczyznę, przyjęło się określać mianem *miotły*.

Mapę będziemy zamiatać od lewej do prawej. W rozważanym przypadku, gdy wszystkie przeszkody i osie symetrii wszystkich odbić leżą na jednej prostej, w danym położeniu miotły jest tylko jedno odbicie, którego przekrój z miotłą trzeba rozważyć — to, dla którego ten przekrój jest największy. Takie odbicie nazwiemy *dominującym*. Zauważmy, że dla danego położenia miotły wszystkie odbicia łodzi możemy podzielić na trzy grupy:

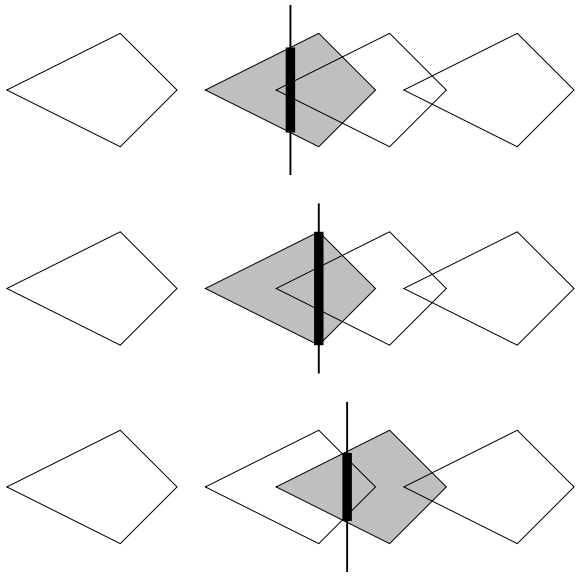
1. odbicia, które będą mogły stać się dominującymi dopiero, gdy miotła przesunie się na prawo;
2. odbicie aktualnie dominujące;
3. odbicia, które na pewno nie będą już dominowały.

Początkowo, gdy miotła znajduje się w skrajnie lewym położeniu, wszystkie odbicia należą do pierwszej grupy. W trakcie zamiatania każde odbicie przechodzi kolejno do drugiej i, dalej, do trzeciej grupy. Odbiciem, które jako kolejne opuszcza pierwszą grupę, jest zawsze to z tej grupy, które leży najbardziej na lewo. Przykłady sytuacji napotykanych w trakcie zamiatania są przedstawione na rys. 3.

Można już dostrzec, jak rozwiązać problem w rozważanym przypadku. Trzeba dysponować listą wszystkich pól z przeszkodami uporządkowaną rosnąco według pierwszej współrzędnej — początkowo wszystkie one (a właściwie związane z nimi odbicia łodzi) należą do pierwszej grupy. Następnie można zacząć rozważać kolejne pozycje miotły — począwszy od skrajnie lewej. Przy przejściu do kolejnej pozycji miotły należy sprawdzić, czy nie trzeba uaktualnić odbicia dominującego. Jest to jednak proste i przy odpowiedniej reprezentacji łodzi może być wykonane w czasie stałym — trzeba porównać aktualnie dominujące odbicie z pierwszym na liście w grupie pierwszej. Potem pozostaje tylko policzyć przekrój odbicia dominującego z miotłą.

Powróćmy do ogólnego przypadku, gdy przeszkody mogą zajmować różne szerokości geograficzne. Wówczas postępujemy bardzo podobnie, powtarzając to samo postępowanie niezależnie dla każdej prostej poziomej, na której leżą przeszkody. Trzeba jeszcze tylko zastanowić się, jak dla danej pozycji miotły szybko wyznaczyć wszystkie leżące na niej pozycje zabronione, tzn. jak połączyć wyniki otrzymane dla różnych prostych poziomych.

Gdybyśmy zaznaczali pola zabronione wynikające z przeszkód leżących na każdej prostej oddzielnie, to zaznaczenie wszystkich pól w jednym położeniu miotły mogłoby nam zająć czas $O(n^2)$ (może być bowiem około n przeszkód, każda wyznaczająca odbicie szerokości około n). Sumarycznie potrzebowalibyśmy wówczas czasu $O(n^3)$ — jest to na pewno wolniej niż byśmy sobie życzyli.



Rys. 3: Kolejne położenia miotły i wyróżnione na szaro odbicia dominujące w tych położeniach. Dla każdego odbicia dominującego, za pomocą czarnego prostokąta zaznaczone są pozycje zabronione, leżące na przecięciu tego odbicia z miotłą.

Czy zatem można lepiej? Na szczęście tak! W ustalonym położeniu miotły, zamiast nanosić na nią całe przekroje odbić dominujących, dla każdego odbicia zaznaczymy tylko jego skrajne punkty: górny i dolny. Możemy to oczywiście zrobić w czasie liniowym. Potem wystarczy raz przejrzeć wszystkie punkty miotły od dołu do góry, zliczając napotkane początki i końce odbić dominujących — jeśli początków jest więcej, to jesteśmy akurat w punkcie zabronionym. W przeciwnym przypadku, czyli gdy początków i końców jest tyle samo, jesteśmy w punkcie dozwolonym.

Postępując w opisany wyżej sposób, w każdym z $O(n)$ położeń miotły musimy:

1. dla każdej prostej poziomej uaktualnić odbicie dominujące — odbywa się to w czasie stałym;
2. dla każdego z tych odbić znaleźć początek i koniec jego przekroju z miotłą i oznaczyć je wszystkie na miotle; potrzebny na to czas zależy od przyjętej reprezentacji łodzi — musimy postarać się, by dało się to zrobić również w czasie stałym dla pojedynczego odbicia;
3. wyznaczyć wszystkie pozycje zabronione leżące na miotle — potrafimy to wykonać w czasie liniowym.

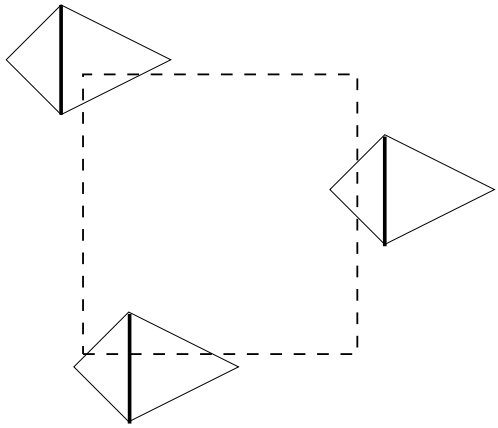
Tym samym wszystkie pola zabronione potrafimy odnaleźć w czasie rzędu $O(n^2)$. Potrzebna pamięć jest tej samej wielkości.

Reprezentacja łodzi

Ta część zadania wygląda na prostą, ale warto się nad nią zastanowić. Wiemy już, czego oczekujemy od naszej reprezentacji.

Łódź to dla nas osiowosymetryczny kształt z wyróżnionym punktem zakotwiczenia. Utożsamiamy go ze zbiorem par liczb L, który w razie potrzeby odbijamy środkowosymetrycznie, otrzymując zbiór $-L$. Umieszczając punkt zakotwiczenia odbicia we wszystkich punktach mapy, w których znajdują się przeszkody, będziemy chcieli oznaczyć wszystkie punkty zabronione na mapie. W tym celu musimy umieć wyznaczyć dowolny pionowy przekrój odbicia łodzi. Reprezentacja musi też pozwolić na szybkie rozpoznanie, czy w danym położeniu łódź wciąż znajduje się na mapie, czy już poza nią wypłynęła.

Jak widać nie mamy nadzwyczaj wygórowanych wymagań. Po wczytaniu mapy wystarczy jednokrotnie ją przejrzeć, by odnaleźć najmniejszy prostokąt zawierający łódź i stwierdzić, która z dwóch jego osi jest osią symetrii łodzi. Jeśli okaże się, że jest to oś pionowa, to odbijamy mapę, by łódź miała poziomą oś symetrii. Teraz pozostaje tylko zapamiętać ciąg liczb będących szerokościami kolejnych przekrojów poprzecznych łodzi. Przy takiej reprezentacji wyznaczanie przekroju łodzi z miotłą jest bardzo proste.



Rys. 4: Różne możliwe położenia poprzeczki łodzi względem obszaru objętego mapą: na lewo od pionowych granic mapy, pomiędzy nimi i na prawo od nich.

Rozstrzygnięcie, czy łódź wypłynęła poza planszę, wymaga rozważenia kilku przypadków. Dla ułatwienia, pionowy odcinek, prostopadły do osi wzdłużnej łodzi i łączący jej burty w najszerszym miejscu nazwijmy *poprzeczką*. Na rys. 4 widzimy kilka możliwości: poprzeczka łodzi może znajdować się na zachód od mapy, może być na długości geograficznej objętej przez mapę lub znajdować się na wschód od niej. Każda z tych możliwości wymaga nieco innego potraktowania, ale poza uważną implementacją nie jest przy tym potrzebna żadna szczególna pomysłowość:

1. gdy poprzeczka jest położona na zachód od mapy, to wystarczy sprawdzić, czy zachodni brzeg mapy jest rozłączny z łodzią, czyli czy nie przecina się z jej odpowiednim przekrojem;
2. gdy poprzeczka jest na długości geograficznej obejmowanej przez mapę, to łódź opuszcza mapę wtedy i tylko wtedy, gdy poprzeczka znajdzie się poza mapą;
3. gdy poprzeczka jest położona na wschód od mapy, to zachodzi przypadek analogiczny, jak pierwszy, tylko trzeba rozważyć wschodni brzeg mapy.

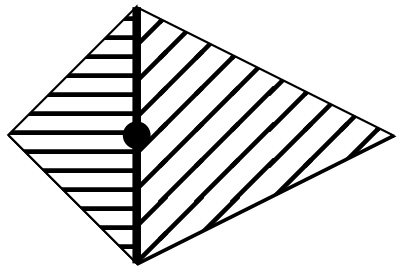
Opisane powyżej rozwiązanie zostało zaimplementowane w plikach `rob.cpp`, `rob1.c`, `rob2.pas` oraz `rob6.java`.

Jak już wspominaliśmy, przedstawione rozwiązanie to prosty przykład zastosowania techniki zamiatania[2].

Przegląd innych rozwiązań

Rozwiązanie $O(n^2)$

Okazuje się, że rozwiązanie o takiej samej złożoności jak wzorcowe można uzyskać, stosując inną metodę — *programowanie dynamiczne*. Pozwoli nam ona wyznaczyć bezpośrednio wszystkie dozwolone pozycje łodzi. Jak poprzednio, załóżmy, że oś symetrii łodzi jest położona poziomo, z dziobem skierowanym na zachód. Jak poprzednio, poprzeczką nazwijmy odcinek prostopadły do osi łodzi poprowadzony w jej najszerszym miejscu. Na potrzeby tego rozdziału część łodzi na lewo od poprzeczki (wraz poprzeczką) nazwijmy w całości *dziobem*, a część na prawo od poprzeczki (bez poprzeczki) — *rufą*. Dla wygody przyjmiemy także, że tym razem punktem zakotwiczenia łodzi będzie środkowy punkt poprzeczki (patrz rys. 5).



Rys. 5: Schemat łodzi z zaznaczonymi: poprzeczką (grubsza kreska), dziobem (zakreskowany poziomo, wraz z poprzeczką), rufą (zakreskowana ukośnie) i punktem zakotwiczenia (małe czarne kółko).

Istotą programowania dynamicznego jest wykorzystywanie wcześniejszych wyników do obliczenia kolejnych. W naszym przypadku chcielibyśmy coś powiedzieć o rozważanej pozycji łodzi w oparciu o wiedzę zgromadzoną w trakcie analizy poprzednich pozycji. Przypuśćmy, że będziemy rozważać pozycje na mapie kolejno wierszami, a w jednym wierszu od lewej do prawej. Gdy analizujemy określoną pozycję, znamy wynik dla pozycji leżącej w poprzedniej kolumnie i tym samym wierszu, co badana. Załóżmy, że była ona dozwolona. Wówczas jedyne pola, jakie trzeba sprawdzić w poszukiwaniu przeszkód dla nowej pozycji, to pola leżące bezpośrednio przy rufie łodzi. Kształt brzegu rufy może być jednak dość nieregularny.

Aby obejść tę trudność, rozważymy odrębnie dziób i rufę łodzi, wyznaczając wszystkie pozycje dozwolone dla każdej z tych części osobno. Zauważmy, że przesuwając w prawo sam dziób, mamy bardzo regularny kształt z prawej strony — odpowiada on przesuniętej

[2]Więcej na jej temat, wraz z przykładami, można znaleźć w [20] i [17]. Na Olimpiadzie również pojawiały się już zadania, w których znalazło się miejsce dla zamiatania: przede wszystkim zadanie *Lodowisko* na VI Olimpiadzie Informatycznej, ale także *Kopalnia złota*, *Wyspy* i *Gazociągi* podczas finałów VIII, XI i XIV Olimpiady.

o jedną kolumnę poprzeczce. Z kolei wyznaczając pozycje dozwolone dla rufy, możemy pola w każdym wierszu rozważać od prawej do lewej. Wtedy przesuwając rufę w lewo, także mamy regularny kształt — tym razem z lewej strony. Pozycjami dozwolonymi dla całej łodzi będą pozycje dozwolone i dla dziobu, i dla rufy.

W dalszej części skupimy się tylko na analizowaniu dziobu łodzi. Trzeba zastanowić się, jak efektywnie sprawdzić, czy dziób, który mieścił się bez przeszkód na określonej pozycji, da się bezkolizyjnie umieścić w nowej kolumnie. Niestety, przejście do nowej pozycji w sytuacji, gdy aktualna pozycja jest zabroniona, wydaje się być trudniejsze.

Aby poradzić sobie z problemem... uogólnimy go! Zamiast pytać, czy cały dziób łodzi mieści się bez przeszkód na mapie, zapytajmy, jaka jego część się mieści. *Prefiksem* dziobu długości ℓ nazwijmy fragment dziobu o długości ℓ (liczonej poziomo) począwszy od jego lewego, najwęższego krańca. Powiemy, że taki prefiks mieści się na pozycji (x,y) na mapie, jeśli można go umieścić bez przeszkód tak, że środek jego prawego boku wypada na pozycji (x,y). Dla pozycji (x,y), przez $pref_{x,y}$ oznaczmy długość najdłuższego prefiksu dziobu, który mieści się bez przeszkód na tej pozycji. Obliczając te wartości, dla każdej pozycji na mapie określimy, jak duży kawałek dziobu mieści się na niej. Dopiero na końcu wybierzemy jako dozwolone te pozycje, na których mieści się cały dziób.

Jak obliczyć wartość $pref_{x+1,y}$ na podstawie $pref_{x,y}$? Gdybyśmy wiedzieli, jaki jest najdalszy poprzeczny przekrój dziobu (licząc od lewej), który mieści się na pozycji (x,y) (oznaczmy tę liczbę jako $maxp_{x+1,y}$), to mielibyśmy gotowy wzór:

$$pref_{x+1,y} = \min(maxp_{x+1,y}, pref_{x,y}+1),$$

wskazujący na klasyczne zastosowanie techniki programowania dynamicznego!

Na szczęście obliczenie wartości $maxp_{x,y}$ również jest zagadnieniem, do którego można zastosować programowanie dynamiczne. Dla każdego pola (x,y) mapy możemy znaleźć najpierw *maksymalny promień* — maksymalną wartość r, taką że na polach $\{(x,y+i) : -r < i < r\}$ nie ma żadnych przeszkód. Potem wystarczy znaleźć najdalszy wśród najszerszych przekrój poprzeczny dziobu, który nie przekracza rozmiarów tego paska, czyli wartość $maxp_{x,y}$. Łatwo to wykonać w czasie stałym, jeśli wcześniej, w czasie $O(n)$, przygotujemy tablicę z odpowiednią informacją o łodzi.

Jak zatem znaleźć maksymalny promień dla pola mapy? Najpierw poszukajmy przeszkód ograniczających tę wartość, położonych nad danym polem. W tym celu przejrzyjmy kolumnę od góry w dół, cały czas pamiętając, gdzie po raz ostatni napotkaliśmy pole z przeszkodą — odległość do tego pola będzie ograniczeniem wartości promienia. Podobnie postępując, wyznaczymy odległość od najbliższej przeszkody na dole, ograniczającą wartość promienia. Za szukaną wartość promienia weźmiemy minimum z dwóch wyznaczonych liczb. Powyższe obliczenia dla jednej kolumny wykonujemy w czasie $O(n)$.

Pozostała część algorytmu, czyli odszukanie łodzi na mapie oraz przeszukiwanie wszerz w poszukiwaniu najkrótszej drogi, są takie same jak w opisanym wcześniej rozwiązaniu wzorcowym. Tak jak w tamtym rozwiązaniu, trzeba pamiętać, że należy rozważać położenia łodzi także poza obszarem objętym mapą.

Opisane tu rozwiązanie zostało zaimplementowane w plikach `rob3.cpp`, `rob4.pas` oraz `rob7.java`. Testy pokazują, że jest ono nieco wolniejsze (o ok. 50%) niż wzorcowe. Również jego wymagania pamięciowe są większe. Oczywiście, uważnie zaimplementowane, przechodzi pomyślnie wszystkie testy.

Rozwiązania zbyt wolne

Zadanie bez problemu można rozwiązać w czasie $O(n^4)$ — na przykład sprawdzając każde potencjalne położenie łodzi w czasie $O(n^2)$ lub implementując przeszukiwanie wszerz, w którym możliwość wykonania każdego kroku jest sprawdzana w czasie $O(n^2)$. Ze względu na dopuszczalną wielkość danych, są to rozwiązania zdecydowanie za wolne. Warto jednak wspomnieć o pewnej heurystyce, która, wykorzystana w takich prostych rozwiązaniach, może je istotnie usprawnić (nie zmieniając jednakże ich złożoności czasowej).

Dla łodzi Robinsona możemy zakreślić najmniejszy prostokąt, w którym się ona mieści. Gdyby dało się łatwo stwierdzić, że w tym prostokącie nie ma żadnych przeszkód, to moglibyśmy uniknąć pracochłonnego, „ręcznego" przeszukiwania tego fragmentu mapy. Badanie prostokątów okazuje się całkiem proste. Wystarczy dla każdego pola mapy (x,y) wyznaczyć wartość $ileszu_{x,y}$ oznaczającą liczbę pól z przeszkodami o obu współrzędnych nie większych niż, odpowiednio, x i y. Wartości $ileszu_{x,y}$ są powiązane zależnością

$$ileszu_{x,y} = ileszu_{x-1,y} + ileszu_{x,y-1} - ileszu_{x-1,y-1} + \delta_{x,y},$$

gdzie $\delta_{x,y}$ to 1 lub 0 w zależności od tego, czy na polu (x,y) jest przeszkoda, czy jej nie ma. Wszystkie wartości $ileszu_{x,y}$ można zatem wyznaczyć w czasie $O(n^2)$. Wówczas, dla określonej pozycji łodzi, liczbę przeszkód w ograniczającym ją prostokącie możemy wyrazić jako liniową kombinację czterech wartości typu $ileszu_{x,y}$[3]. Jeśli wartość ta jest niezerowa, to musimy sprawdzić poprawność pozycji ręcznie. Jeśli jednak wynosi zero, to jesteśmy pewni, że pozycja nie koliduje z żadnymi przeszkodami.

Rozwiązanie o złożoności $O(n^4)$, wykorzystujące opisaną heurystykę, zostało zaimplementowane w plikach `robs3.cpp`, `robs6.pas` oraz `robs8.c`.

Istnieje także nietrywialne rozwiązanie działające w czasie $O(n^3)$, którego pomysł warto przedstawić. Częścią decydującą o złożoności wszystkich opisanych rozwiązań jest określenie, które pozycje są dozwolone dla łodzi Robinsona. W dotychczas zaprezentowanych algorytmach było to wykonywane przed samym przeszukiwaniem wszerz. Tym razem będziemy tę informację wyliczać na bieżąco, w momencie przesuwania łodzi. Wykorzystamy przy tym własność, o której już pisaliśmy — aby przekonać się, czy można przesunąć łódź w określonym kierunku, sprawdzimy tylko, czy nie ma przeszkód wzdłuż odpowiedniego brzegu łodzi.

Niestety, ponieważ badany brzeg łodzi może mieć nieregularny kształt, sprawdzenie możliwości wykonania ruchu zajmuje czas $O(n)$. Cały algorytm, w którym analizujemy $O(n^2)$ pól, ma więc złożoność $O(n^3)$.

Powyższe rozwiązanie zostało zaimplementowane w pliku `robs5.cpp`, a połączone jeszcze z heurystyką „pustego prostokąta" — w plikach `robs4.cpp`, `robs7.pas` i `robs9.c`. Algorytm ten, dobrze zaimplementowany, otrzymywał $60-70\%$ punktów.

Rozwiązania niepoprawne

Przykrym błędem, jaki można popełnić, jest badanie brzegu łodzi (w rozwiązaniu $O(n^3)$) w niepoprawny sposób — poprzez sprawdzanie jedynie skrajnych punktów kolejnych

[3]Podobny przykład zastosowania metody programowania dynamicznego do wyznaczania pewnych sum dla prostokątów można znaleźć w opisie rozwiązania zadania *Mapa gęstości* z VIII Olimpiady Informatycznej.

przekrojów poprzecznych. Rozwiązanie to znajduje się w pliku `robb2.cpp`, będącym wariantem programu `robs4.cpp`.

Innym wartym wspomnienia błędem jest potraktowanie heurystyki „pustego prostokąta" jako jedynego kryterium tego, czy pozycja jest dozwolona. To oczywiście powoduje pominięcie niektórych dozwolonych pozycji. Podobny błąd można popełnić, rozstrzygając, czy łódź wypłynęła już na pełne morze, za pomocą przecięcia minimalnego prostokąta otaczającego łódź z mapą. Programy z opisanymi błędami zapisane są w plikach `robb1.cpp` i `robb3.cpp`.

Oprócz różnego rodzaju pomyłek implementacyjnych oraz opisanych wyżej przypadków, w rozwiązaniach zawodników nie wystąpiły żadne inne typowe błędy.

Testy

Przygotowane zostało 10 zestawów testowych, z których większość stanowią grupy kilku testów.

Testy od drugiego włącznie zostały wygenerowane automatycznie. Kilka z nich było generowanych za pomocą mechanizmu przekazywania procedurze pomocniczej położeń łodzi, które mają być na planszy dozwolone i nakazania jej wypełnienia prawie wszystkich (poza pewnymi losowymi) pozostałych pól przeszkodami. W procedurze tej do konstrukcji mapy zastosowano algorytm podobny do rozwiązania wzorcowego. Testy tego typu są nazwane *zadanymi polami*. W większości takich testów łódka ma kształt losowy, ale w miarę równomiernie rozszerzający się i zwężający.

W testach z odpowiedzią `NIE` wyjście poza mapę uniemożliwia zazwyczaj jedno lub kilka pól z sitowiem, tak aby trudno było tę sytuację wykryć.

Opis każdego testu zawiera kolejno: wartość n, wysokość i szerokość łodzi (w orientacji, w której oś symetrii przebiega poziomo), orientację osi symetrii łodzi (poziomą, pionową lub symetryczną, czyli posiadającą i pionową, i poziomą oś symetrii), wartość wyniku oraz krótki opis testu.

Nazwa	n	w	s	oś	wyn	Opis
rob1a.in	25	9	9	–	42	ręcznie generowany test poprawnościowy
rob1b.in	3	3	3	+	3	skrajnie mały test bez pól z sitowiem
rob2.in	100	5	5	–	135	ręcznie generowany test poprawnościowy zadany polami
rob3a.in	1 000	3	3	+	248 755	najmniejsza możliwa łódź, która musi przepłynąć przez całą powierzchnię mapy, aby się z niej wydostać
rob3b.in	1 000	3	3	+	*NIE*	taki jak poprzedni, ale z zablokowanym wyjściem

Nazwa	n	w	s	oś	wyn	Opis
rob4.in	1500	99	98	|	3949	bardzo zagmatwany test zadany polami z niewielką łódką
rob5.in	2000	793	601	|	2882	zadany polami test z dużą łódką, mającą dużo dłuższy dziób niż rufę
rob6a.in	999	273	300	–	1192	zadany polami test zawierający basen w kształcie przekrzywionego prostokąta
rob6b.in	999	305	300	|	*NIE*	jak poprzedni, ale bez możliwości wypłynięcia
rob7a.in	1000	401	401	–	1595	zadany polami test z łódką w kształcie strzałki
rob7b.in	1000	401	401	+	1599	zadany polami podobny test, ale z łódką w kształcie krzyża
rob7c.in	1000	401	401	+	*NIE*	jak 7b, ale bez możliwości wypłynięcia
rob8a.in	2000	1401	1472	|	2627	test z dużą łodzią, zadany polami
rob8b.in	2000	1503	1500	–	*NIE*	jak poprzedni, ale bez możliwości wypłynięcia
rob9a.in	1495	759	743	–	2100	duża łódź w prostokątnym basenie otoczonym murkiem z jedną dziurą
rob9b.in	1495	759	743	|	*NIE*	jak poprzedni, ale bez możliwości wypłynięcia
rob10a.in	2000	1981	1979	–	3001	ogromna łódź, zajmująca większą część planszy
rob10b.in	2000	1981	1979	|	*NIE*	jak poprzedni, ale bez możliwości wypłynięcia

Marian M. Kędzierski
Treść zadania, Opracowanie

Michał Pilipczuk
Program

OI, Etap I, 22.10–19.11.2007

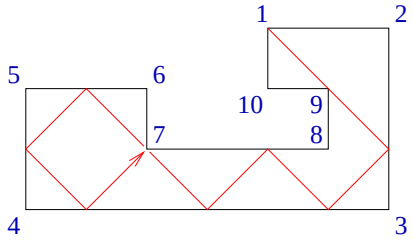
Szklana pułapka

W pałacu króla Bajtazara straszy duch. Złośliwi twierdzą, że jest to duch niedawno zmarłej (w podejrzanych okolicznościach) małżonki Bajtazara, gdyż tak jak ona, bardzo lubi przeglądać się w lustrach. Nie dziwi więc, że Bajtazar chętnie pozbyłby się ducha.

Bajtazar postanowił w jednej z komnat pałacu przygotować specjalną **szklaną pułapkę**. *Jest to zamknięta, dobrze oświetlona sala, której wszystkie ściany wewnętrzne pokryte są lustrami. Ponadto w każdym rogu sali może być umieszczony laser albo czujnik promieni laserowych. Gdy tylko duch przetnie wiązkę jednego z laserów, podniesie się alarm, a sprowadzeni przez Bajtazara pogromcy duchów rozprawią się z nim.*

Sala, w której powstaje szklana pułapka, ma kształt wielokąta o sąsiednich bokach prostopadłych do siebie. Ponadto długość każdej ściany (wyrażona w metrach) jest całkowita. Jeżeli w danym rogu sali ma być umieszczony laser, to jego promień musi być skierowany wzdłuż dwusiecznej kąta, jaki tworzą stykające się w tym rogu ściany (i zarazem równolegle do podłogi). Oczywiście, jeżeli promień lasera napotka na swojej drodze lustro, to odbija się od niego pod takim samym kątem, pod jakim pada, czyli 45 stopni. Promień wpadający wzdłuż dwusiecznej kąta w róg sali z czujnikiem zostaje całkowicie pochłonięty przez czujnik. Promień wpadający wzdłuż dwusiecznej kąta w róg sali bez czujnika (za to być może z innym laserem) jest odbijany o 180 stopni. W innych przypadkach promień kontynuuje wędrówkę po sali, nie zmieniając kierunku.

W sali należy rozmieścić maksymalnie dużo laserów i odpowiadających im czujników, w taki sposób, żeby promień każdego lasera wpadał do jakiegoś czujnika, promienie różnych laserów wpadały do różnych czujników oraz żeby do każdego czujnika wpadał promień pewnego lasera. Przy tym w każdym rogu można zainstalować tylko albo laser, albo czujnik, ale nie jedno i drugie.



Przykład szklanej pułapki z promieniem lasera biegnącym z rogu 1 do rogu 7.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis kształtu szklanej pułapki,*
- *wyznaczy sposób rozmieszczenia maksymalnej liczby laserów i czujników spełniający warunki zadania,*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu wejścia znajduje się jedna liczba całkowita n ($4 \leqslant n \leqslant 100\,000$). Jest to liczba ścian szklanej pułapki. W każdym z następnych n wierszy znajdują się po dwie liczby całkowite x_i i y_i oznaczające współrzędne rogu nr i ($1 \leqslant i \leqslant n$, $-1\,000\,000 \leqslant x_i, y_i \leqslant 1\,000\,000$), przedzielone pojedynczą spacją. Każde dwa kolejne rogi połączone są ścianą równoległą do jednej z osi współrzędnych. Żadne dwie ściany nie przecinają się i nie mają punktów wspólnych, z wyjątkiem dwóch kolejnych ścian, które stykają się we wspólnym rogu. Kolejne rogi pomieszczenia podane są w kolejności zgodnej z ruchem wskazówek zegara (tzn. przy obchodzeniu pomieszczenia wzdłuż ścian wnętrze znajduje się zawsze po prawej stronie). Łączna długość wszystkich ścian nie przekracza 300 000.

Wyjście

Twój program powinien w pierwszym wierszu wyjścia wypisać jedną liczbę całkowitą m, oznaczającą maksymalną liczbę par laser-czujnik, jakie można umieścić w szklanej pułapce. W każdym z następnych m wierszy powinna znaleźć się para liczb a_i i b_i oddzielonych pojedynczą spacją, gdzie a_i oznacza numer rogu, w którym powinien znaleźć się laser, zaś b_i — numer rogu, w którym powinien znaleźć się odpowiadający mu czujnik, przy czym $1 \leqslant a_i, b_i \leqslant n$. W przypadku, gdy istnieje wiele rozwiązań, należy podać dowolne z nich.

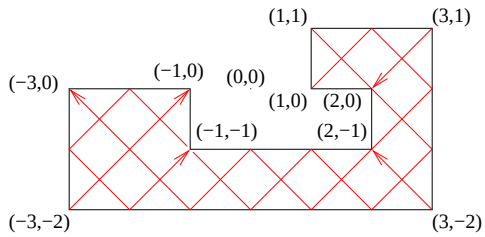
Przykład

Dla danych wejściowych:

```
10
1 1
3 1
3 -2
-3 -2
-3 0
-1 0
-1 -1
2 -1
2 0
1 0
```

poprawnym wynikiem jest:

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



Przykładowe rozmieszczenie systemu laserów i czujników w szklanej pułapce przedstawia rysunek.

Rozwiązanie

Wprowadzenie

Na pierwszym etapie LV Olimpiady Matematycznej pojawiło się następujące zadanie:

> Dany jest wielokąt o bokach długości wymiernej, w którym wszystkie kąty wewnętrzne są równe 90° lub 270°. Z ustalonego wierzchołka wypuszczamy promień świetlny do wnętrza wielokąta w kierunku dwusiecznej kąta wewnętrznego przy tym wierzchołku. Promień odbija się zgodnie z zasadą: kąt padania jest równy kątowi odbicia. Udowodnić, że promień trafi w jeden z wierzchołków wielokąta.

W dowodzie tego twierdzenia okazywało się ponadto, że wierzchołek, do którego trafiał promień świetlny, był różny od tego, z którego promień został wypuszczony. Nie jest to jednak jedyny ciekawy wniosek, jaki z tego dowodu można było wysnuć.

Jeśli przyjrzymy się treści powyższego zadania z Olimpiady Matematycznej, to zauważymy zapewne łudzące podobieństwo do naszego problemu ze szklaną pułapką. Nie bez powodu...

Okazuje się, że jeśli przekształcimy wielokąt z tego zadania przez jednokładność o skali równej wspólnemu mianownikowi wszystkich ułamków określających długości poszczególnych boków, to otrzymamy wielokąt podobny do oryginalnego, w którym długości wszystkich boków będą liczbami naturalnymi. Następnie, korzystając z faktu, że wszystkie kąty wewnętrzne wielokąta są wielokrotnościami 90°, możemy obrócić go tak, aby jego boki stały się równoległe do osi układu współrzędnych. Wystarczy jeszcze tylko zastosować przesunięcie (uczenie mówiąc: *translację*), takie aby pewien wierzchołek wielokąta stał się punktem kratowym (tzn. aby jego obie współrzędne stały się liczbami całkowitymi), i wtedy okaże się, że wszystkie wierzchołki wielokąta będą punktami kratowymi. Okazuje się, że tak przekształcony wielokąt jest dokładnie tej samej postaci, co szklana pułapka występująca w naszym zadaniu!

Ponieważ wykonane przekształcenia nie zmieniają kształtu figury (ściślej mówiąc, powstały wielokąt jest *podobny* do pierwotnego), to teza zadania z Olimpiady Matematycznej zachodzi również dla tak przekształconego wielokąta. W szczególności jest ona także prawdziwa dla szklanej pułapki.

Wiemy więc już, że jeżeli wystrzelimy promień lasera z wierzchołka a (wzdłuż dwusiecznej kąta), to trafi on w pewien wierzchołek b. Co więcej, będzie to wierzchołek różny od a. Łatwo też zauważyć, że promień biegnie w sposób symetryczny, tzn. wystrzelony z wierzchołka b trafi z powrotem do wierzchołka a.

To spostrzeżenie pozwala zauważyć, że wierzchołki dowolnego wielokąta można połączyć w jednoznaczny sposób w pary (a,b) takie, że promień lasera wystrzelony z wierzchołka a trafia w wierzchołek b, zaś wystrzelony z b trafia w a. To z kolei prowadzi do wniosku, że jeżeli dla każdej takiej pary umieścimy laser w jednym wierzchołku, a czujnik w drugim, to otrzymamy $\frac{n}{2}$ par laser-czujnik spełniających warunki zadania. Z drugiej strony w każdym rogu pomieszczenia może znaleźć się tylko jedno urządzenie, więc rozmieszczenie większej liczby par niż $\frac{n}{2}$ jest niemożliwe.

Tak więc jesteśmy już w stanie udzielić odpowiedzi na pierwszą część zadania i to jedynie na podstawie liczby rogów pomieszczenia — można zawsze rozmieścić dokładnie $\frac{n}{2}$ par urządzeń[1].

Wszystkie powyższe wnioski można było łatwo wyciągnąć na podstawie dowodu tezy przedstawionej w zadaniu z Olimpiady Matematycznej. Zadanie to nasuwa jednak na myśl kolejny problem. Skoro wiemy, że dla dowolnego wielokąta o wierzchołkach kratowych i bokach równoległych do osi układu współrzędnych istnieje jednoznaczny rozkład zbioru wierzchołków na pary „widzące się wzajemnie", to dlaczego nie pokusić się o wyznaczenie tego rozkładu dla zadanego wielokąta?

I tak oto wkraczamy w drugą, ciekawszą część zadania...

Rozważania matematyczne

Lepsze zrozumienie dowodu twierdzenia z Olimpiady Matematycznej powinno pomóc podzielić wierzchołki wielokąta na żądane pary. Przyjrzyjmy się więc dokładniej temu rozumowaniu.

Lemat 1. Wielokąt o bokach długości wymiernej, w którym wszystkie kąty wewnętrzne są równe 90° lub 270°, jest podobny do pewnego wielokąta o wierzchołkach kratowych i bokach równoległych do osi układu współrzędnych.

Dowód: Ciąg przekształceń geometrycznych opisanych w rozdziale „Wprowadzenie" pokazuje, jak dokonać przeprowadzenia pomiędzy takimi wielokątami, zachowując podobieństwo. ■

W dalszych rozważaniach będziemy już odnosić się do przekształconego wielokąta, który odpowiada założeniom o szklanej pułapce. Ponieważ wciąż mamy na myśli zadanie z Olimpiady Matematycznej, to w tym rozdziale będziemy zakładać, że promień, który wpada do wierzchołka, zawsze zostaje przez niego pochłonięty.

Definicja 1. Wszystkie sytuacje powstające w chwili zetknięcia promienia z obwodem wielokąta podzielimy na *zdarzenia* i *otarcia*. *Zdarzeniem* będziemy nazywali jedną z następujących sytuacji:

1. *wystrzelenie* promienia z wierzchołka (zgodnie z zasadami),
2. *odbicie* promienia od obwodu wielokąta,
3. *wpadnięcie* promienia do wierzchołka.

Otarciem będziemy nazywali sytuację, w której promień dotyka obwodu wielokąta, jednak nie zmienia to jego biegu.

Zauważmy, że terminy *zdarzenie* i *otarcie* nigdy nie opisują tej samej sytuacji i termin *zdarzenie* wyczerpuje wszystkie sytuacje, w których promień zmienia swój bieg.

Definicja 2. *Krokiem* biegu promienia będziemy nazywali odcinek jego toru zawarty pomiędzy dwoma zdarzeniami.

[1] Ciekawostka: przy okazji całe to rozumowanie uzasadnia, że liczba wierzchołków wyjściowego wielokąta musi być parzysta!

Lemat 2. Kierunek biegu promienia jest zawsze równoległy do jednej z prostych: $y = x$ lub $y = -x$.

Dowód: Dowiedziemy tego przez indukcję ze względu na kolejne kroki biegu promienia.

Baza indukcji. Dwusieczna kąta w każdym wierzchołku wielokąta jest zawsze równoległa do jednej z podanych prostych, więc promień wystrzelony z wierzchołka również.

Krok indukcyjny. Załóżmy, że promień spełnia tezę lematu i jest równoległy do prostej $y = x$. Po kolejnym zdarzeniu:

- jeżeli jest to odbicie, to kierunek po nim zmienia się na prostopadły do dotychczasowego, czyli będzie równoległy do prostej $y = -x$;
- jeżeli zaś jest to wpadnięcie, to po nim nie następuje żaden krok biegu promienia.

W przypadku, gdy przed zdarzeniem promień biegnie równolegle do prostej $y = -x$, dowód jest analogiczny.

Na mocy Twierdzenia o Indukcji Matematycznej, teza lematu jest więc prawdziwa w dowolnym kroku biegu promienia. ■

Oznaczmy przez B — brzeg wielokąta, przez W — jego wnętrze wraz z brzegiem, zaś przez $\mathbb{Z}$ — zbiór liczb całkowitych.

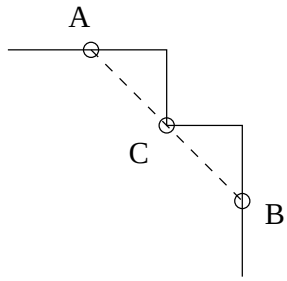
Z Lematu 2 wynika, że wszystkie możliwe trasy promieni są całkowicie zawarte w zbiorze:

$$X = W \cap \left(\bigcup_{a \in \mathbb{Z}} \{(x,y) : y = a - x\} \cup \bigcup_{a \in \mathbb{Z}} \{(x,y) : y = a + x\} \right)$$

Zauważmy, że $X \cap B$ składa się wyłącznie z punktów kratowych. Ponadto, zbiór X jest sumą odcinków, których końce są właśnie punktami kratowymi obwodu B.

Te spostrzeżenia przywodzą na myśl pewien pomysł, który jest istotą niniejszych rozważań i — jak się okaże później — także istotą rozwiązania całego problemu szklanej pułapki. Zbiór X można przedstawić jako graf, w którym wierzchołkami są punkty kratowe na obwodzie B rozpatrywanego wielokąta, zaś krawędzie łączą końce odcinków ze zbioru X.

Przy takiej konstrukcji grafu powstaje jednak pewien problem: nie jest jednoznacznie powiedziane, jak traktować otarcia. Na rys. 1 widać, że nie jest oczywiste, czy punkty A i B należy połączyć krawędzią, ignorując tym samym, że przechodzi ona przez punkt C, czy utworzyć dwie krawędzie: pomiędzy wierzchołkami A i C oraz B i C? Chociaż na tym etapie oba rozwiązania są równoważne, to lepiej jest przyjąć pierwsze z nich, gdyż uprości to dalsze rozważania.



Rys.1: Otarcie promienia o brzeg wielokąta.

Skonstruujmy więc graf, ignorując wszystkie otarcia. Nazwijmy go G. Łatwo zauważyć, że każdy wierzchołek odpowiadający narożnikowi wielokąta ma w grafie G stopień 1 (tzn. wychodzi z niego jedna krawędź). Natomiast wierzchołek odpowiadający dowolnemu innemu punktowi kratowemu na obwodzie ma w G stopień 2 (wychodzą z niego dwie krawędzie).

Okazuje się, że grafy o powyższej własności mają bardzo prostą postać.

Lemat 3. Graf, w którym każdy wierzchołek ma stopień 1 lub 2, jest zbiorem rozłącznych wierzchołkowo ścieżek i cykli.

Dowód: Udowodnimy ten fakt przez indukcję ze względu na liczbę wierzchołków grafu.

Baza indukcji. Najmniejszy graf spełniający założenia lematu to graf dwuwierzchołkowy mający jedną krawędź. Jest on oczywiście ścieżką.

Krok indukcyjny. Załóżmy, że teza zachodzi dla wszystkich grafów o $k < n$ wierzchołkach. Rozważmy graf n-wierzchołkowy. Jeśli graf ten nie ma wierzchołka stopnia 1, to wybierzmy dowolną krawędź e i usuńmy ją — w grafie mamy teraz dokładnie dwa wierzchołki stopnia 1.

Niech v będzie wierzchołkiem stopnia 1. Utwórzmy najdłuższy możliwy ciąg (v_i) różnych wierzchołków, takich że $v_1 = v$ oraz v_i jest połączony krawędzią z v_{i-1} dla $i > 1$. Powstały ciąg jest ścieżką i kończy się wierzchołkiem stopnia 1 innym niż v (dlaczego?) — oznaczmy go u. Po usunięciu z grafu ścieżki $v-u$ otrzymujemy graf o mniejszej liczbie wierzchołków, w którym również każdy wierzchołek ma stopień 1 lub 2, więc z założenia indukcyjnego jest także zbiorem rozłącznych ścieżek i cykli.

Graf wyjściowy zawiera dodatkowo ścieżkę $v-u$ i, ewentualnie, krawędź e. Zauważmy, że jeśli usunęliśmy z wyjściowego grafu krawędź e, to $e = (v,u)$ i zamyka ona ścieżkę $v-u$ w cykl. Stąd w obu przypadkach graf wyjściowy również jest zbiorem rozłącznych ścieżek oraz cykli. Z Twierdzenia o Indukcji Matematycznej teza lematu jest więc prawdziwa. ■

Teraz pozostaje już tylko zauważyć, że promień wypuszczony z dowolnego wierzchołka wielokąta biegnie dokładnie wzdłuż ścieżki grafu rozpoczynającej się w tym wierzchołku. Stąd już bezpośrednio wynika następujące Twierdzenie.

Twierdzenie 1 (Rozwiązanie zadania z LV OM). *Promień wypuszczony z dowolnego narożnika trafia w inny narożnik wielokąta.*

Dowód: Narożnikowi, z którego jest wypuszczany promień, odpowiada w grafie G wierzchołek v stopnia 1. Promień musi podążać po krawędziach grafu G ścieżką rozpoczynającą się w v, która kończy się w u — innym wierzchołku stopnia 1. Wierzchołkowi u także odpowiada narożnik wielokąta (oczywiście inny niż ten, z którego został wypuszczony promień). ■

Tak więc twierdzenie zostało ostatecznie udowodnione. Można je wyprowadzić krócej, jednak ten dowód jest lepszy z punktu widzenia zadania o szklanej pułapce. Jest w nim bowiem zakodowany algorytm pozwalający znaleźć żądane pary wierzchołków…

Wyznaczanie podziału wierzchołków na pary

Jak Czytelnik mógł się już domyślić, zamierzamy jawnie zbudować graf G, a następnie wyszukać w nim wszystkie ścieżki i połączyć w pary ich końce. Pozostaje jeszcze tylko pytanie, jak zbudować graf G?

Popatrzmy na konstrukcję zbioru X z poprzedniego rozdziału. Wynika z niej, że przecinając z wnętrzem wielokąta wszystkie proste równoległe do prostej $y = x$ przechodzące przez punkty kratowe, otrzymamy wszystkie krawędzie grafu G równoległe do tej prostej. Aby wyznaczyć resztę krawędzi, wystarczy tę samą procedurę powtórzyć dla wszystkich prostych równoległych do prostej $y = -x$.

Rozważmy jedną z interesujących nas prostych — równoległą do $y = x$ lub $y = -x$ — i oznaczmy ją przez ℓ. Powinniśmy teraz znaleźć wszystkie punkty, w których prosta ℓ przecina brzeg wielokąta, i pomiędzy nimi poprowadzić krawędzie grafu G. Zanim do tego przystąpimy, uściślijmy, które przecięcia nas interesują.

Definicja 3. Powiemy, że prosta ℓ *przebija* brzeg B wielokąta w punkcie x, jeżeli przechodzi ona przez punkt x i w każdym jego otoczeniu[2] ma punkty wspólne z wnętrzem oraz z zewnętrzem wielokąta.

Prawdziwa jest następująca, ciekawa własność.

Lemat 4. Dowolna prosta przebija brzeg wielokąta parzystą liczbę razy.

Dowód: Niech $V = \mathbb{R}^2 \setminus W$. Podział płaszczyzny $\mathbb{R}^2$ na zbiory W i V odpowiada jej podziałowi na wnętrze wielokąta i pozostały obszar. Brzeg wielokąta jest granicą pomiędzy zbiorami V i W (przypomnijmy, że brzeg wielokąta zaliczamy do zbioru W).

Obszar W jest ograniczony, więc rozważana prosta odpowiednio daleko zarówno z jednej, jak i z drugiej strony, leży w obszarze V. Prześledźmy jej wędrówkę, zaczynając odpowiednio daleko z jednej, a kończąc odpowiednio daleko z drugiej strony. Każdy punkt, w którym prosta przebija brzeg wielokąta, powoduje jej przejście do innego obszaru: z W do V lub z V do W. Ponieważ zaś „wędrówka" zaczyna i kończy się w obszarze V, to w jej trakcie prosta musiała przebić brzeg parzystą liczbę razy. ■

Załóżmy, że uda nam się wyznaczyć wszystkie punkty przebicia obwodu B przez prostą ℓ. Wówczas wystarczy je posortować w kolejności występowania na prostej ℓ, a następnie połączyć: pierwszy z drugim, trzeci z czwartym itd., tworząc kolejne krawędzie grafu G.

Takiego postępowania nie możemy oczywiście przeprowadzić dla wszystkich możliwych prostych, gdyż jest ich nieskończenie wiele. Ale możemy ograniczyć się tylko do tych prostych, które przecinają wielokąt. Jak je wyznaczyć?

Zauważmy, że proste równoległe do $y = x$ lub $y = -x$ mają odpowiednio równania $x - y = c$ oraz $x + y = d$ dla pewnych stałych c, d. W równaniach „skrajnych" prostych przecinających wielokąt, wartości stałych wynoszą $c_1 = \min_{(x,y)\in B}(x - y)$, $c_2 = \max_{(x,y)\in B}(x-y)$, $d_1 = \min_{(x,y)\in B}(x+y)$, $d_2 = \max_{(x,y)\in B}(x+y)$. Można zauważyć, że są to minimalne i maksymalne współrzędne wierzchołków wielokąta w nieco innym układzie współrzędnych, a mianowicie w układzie $(x-y, x+y)$ [3]. Po przedstawieniu wielokąta w tym układzie współrzędnych, łatwo znaleźć skrajne wartości współrzędnych wierzchołków. Gdy znamy wartości c_1, c_2, d_1 oraz d_2, wówczas trzeba wyznaczyć przecięcia z wielokątem tylko prostych o parametrach $c_1 \leqslant c \leqslant c_2$ oraz $d_1 \leqslant d \leqslant d_2$ $(c, d \in \mathbb{Z})$.

Pozostaje jeszcze kwestia sortowania punktów przecięcia na jednej prostej. Tutaj także warto zastosować układ współrzędnych $(x-y, x+y)$. Okazuje się, że jeśli rozważana prosta

[2] Czytelnik niezaznajomiony z pojęciem *otoczenia* może sobie zastąpić ten termin w niniejszej definicji przez *koło bez brzegu*.

[3] Ćwiczenie dla Czytelnika: czym jest przekształcenie płaszczyzny, które punktowi o współrzędnych (x, y) przyporządkowuje właśnie punkt $(x-y, x+y)$?

jest postaci $x - y = c$, to punkty należy posortować względem wartości $x + y$. W nowym układzie oznacza to, że punkty na prostych pionowych należy posortować względem rzędnych. Dla prostych postaci $x + y = d$ punkty przecięcia należy posortować względem wartości $x - y$. W nowym układzie przenosi się to na sortowanie punktów na prostych poziomych względem odciętych.

Otrzymaliśmy więc pierwszy algorytm, który pozwala na rozwiązanie zadania.

Algorytm I

Przyjmijmy, że współrzędne wierzchołków są przechowywane odpowiednio w tablicach $x[1..n]$ oraz $y[1..n]$. Graf będziemy budowali w słowniku *sąsiedzi*, który współrzędnym punktu kratowego obwodu będzie przypisywał jego listę sąsiadów w grafie G (tzn. listę współrzędnych punktów kratowych obwodu, z którymi dany punkt jest połączony krawędzią). Aby wykonać sortowanie punktów na prostej, będziemy potrzebowali jeszcze pomocniczej tablicy $srt[1..MAXL]$, gdzie $MAXL$ oznacza maksymalną liczbę punktów kratowych na krawędzi (czyli 300 000). W tablicy tej będziemy zapisywali współrzędne punktów przecięcia z obwodem B prostych skośnych w każdym z dwóch kierunków.

Przyjmijmy, że funkcje pomocnicze sortuj_względem_sumy(L : *integer*) oraz sortuj_względem_różnicy(L : *integer*) sortują pierwsze L rekordów w tablicy $srt[1..MAXL]$, posługując się przy porównaniach wartościami odpowiednio $x + y$ lub $x - y$.

Otrzymujemy zatem następujący algorytm budowania grafu:

```
{ Znajdujemy ekstremalne parametry prostych c1, c2, d1 i d2. }
c1 := d1 := +∞;
c2 := d2 := −∞;
for i := 1 to n do
begin
  if x[i] − y[i] < c1 then c1 := x[i] − y[i];
  if x[i] − y[i] > c2 then c2 := x[i] − y[i];
  if x[i] + y[i] < d1 then d1 := x[i] + y[i];
  if x[i] + y[i] > d2 then d2 := x[i] + y[i];
end
{ Dla każdej prostej przecinającej wielokąt }
{ wyznaczamy zawarte w niej krawędzie grafu G. }
for c := c1 to c2 do
begin
  { L oznacza liczbę znalezionych w danym kroku przecięć. }
  L := 0;
  { Rozważamy prostą x − y = c. }
  { Znajdujemy wszystkie jej punkty przecięcia z obwodem wielokąta. }
  for i := 1 to n do
  begin
    { Badamy bok między wierzchołkami i oraz (i mod n) + 1. }
    next_i := (i mod n) + 1;
    if bok (i, next_i) przecina się z prostą x − y = c then
    begin
      L := L + 1;
```

```
26:          srt[L] := punkt_przecięcia(bok (i, next_i), prosta x − y = c);[4]
27:        end
28:      end
29:      { Sortujemy przecięcia srt[1..L] względem wartości x + y. }
30:      sortuj_względem_sumy(L);
31:      { Usuwamy wszystkie pozostałe punkty otarcia, }
32:      { czyli te punkty, które występują dwa razy obok siebie. }
33:      L2 := 1;
34:      i := 1;
35:      while i ⩽ L do
36:        if i < L and srt[i] = srt[i + 1] then i := i + 2
37:        else
38:        begin
39:          srt[L2] := srt[i];
40:          i := i + 1;
41:          L2 := L2 + 1;
42:        end
43:      L := L2;
44:      { Wyznaczamy krawędzie. }
45:      i := 1;
46:      while i ⩽ L do
47:      begin
48:        dodaj_do_listy(sąsiedzi[srt[i]], srt[i + 1]);
49:        dodaj_do_listy(sąsiedzi[srt[i + 1]], srt[i]);
50:        i := i + 2;
51:      end
52:  end
53:  for d := d1 to d2 do
54:  begin
55:      { Powtarzamy wszystko analogicznie dla prostej x + y = d, }
56:      { sortując względem różnicy x − y zamiast względem sumy. }
57:      ...
58:  end
```

Powyższa procedura konstruuje graf G, który wystarczy tylko przeszukać, aby znaleźć pary wierzchołków odpowiadające końcom ścieżek.

Przeanalizujmy złożoność tego algorytmu. Niech l oznacza liczbę punktów kratowych na obwodzie wielokąta. Dla każdej prostej przecinającej wielokąt (może być ich potencjalnie $O(l)$) sprawdzamy wszystkie boki (jest ich $O(n)$) w celu wykrycia możliwych przecięć. Tak więc samo wyznaczanie przecięć ma w tej metodzie koszt $O(n \cdot l)$. Koszt wykonywanych dwóch sortowań to $O(l \cdot \log l)$, jeżeli użyjemy jednego z szybkich algorytmów sortowania[5]. Trzeba jeszcze uwzględnić operacje na słowniku *sąsiedzi*, które są wykonywane $O(l)$ razy.

[4] Przy wyznaczaniu przecięć zakładamy, że odcinki zawierają ten z dwóch końców, który ma mniejszą wartość $x - y$, zaś drugiego nie zawierają. Dzięki temu łatwo jest odsiać punkty otarcia, gdyż są one albo wykrywane jako przecięcia dwukrotnie, albo w ogóle, podczas gdy każde inne przecięcie jest wykrywane dokładnie raz.

[5] Np. MergeSort bądź HeapSort — oba algorytmy są opisane w [17] i w [20].

Struktury słownikowe można tak zaimplementować, aby każda operacja na nich miała koszt $O(\log l)$[6]. Jednak nawet tak finezyjna ich realizacja niewiele nam pomoże, gdyż sumaryczna złożoność algorytmu już i tak wynosi $O(n \cdot l)$, co w naszym wypadku jest kosztem zbyt dużym.

Jeżeli jednak powrócimy do wspomnianego wcześniej przekształcenia układu współrzędnych (x,y) w układ $(x-y, x+y)$, to możemy znaleźć sposób na przyspieszenie algorytmu...

Rozwiązanie wzorcowe

Zamiast przykładać do każdej prostej każdy bok wielokąta, żeby sprawdzić, czy aby przypadkiem się nie przetną, możemy przetwarzać zarówno proste, jak i możliwe punkty przecięcia w pewnych analogicznych do siebie porządkach. Przecież każdy z możliwych punktów przecięcia, czyli punktów kratowych na obwodzie wielokąta, jest punktem przecięcia dla pewnej prostej. Tak więc zamiast szukać przecięć prostych z poszczególnymi bokami, możemy wziąć wszystkie punkty kratowe pojawiające się na obwodzie i przetwarzać je w takiej kolejności, w jakiej pojawiałyby się w poprzednim algorytmie. Jaka to kolejność?

Skupmy się na pierwszym kierunku prostych ($x - y = c$). Punkty kratowe obwodu są tam przetwarzane w blokach odpowiadających kolejnym wartościom c (dla kolejnych prostych). W każdym zaś z tych bloków są one sortowane względem wartości $x+y$. Tak więc szukany porządek to punkty posortowane w pierwszej kolejności względem „współrzędnej" $x-y$, zaś w drugiej kolejności (tzn. w przypadku takich samych wartości $x-y$) względem „współrzędnej" $x+y$.

Analogicznie postępujemy dla prostych biegnących w kierunku prostopadłym do poprzedniego (czyli prostych postaci $x+y=d$) — punkty kratowe obwodu są przetwarzane w porządku posortowanym w pierwszej kolejności względem $x+y$, zaś w drugiej kolejności względem $x-y$.

Możemy więc na samym początku analizy każdego z dwóch kierunków prostych posortować wszystkie punkty kratowe na obwodzie w odpowiedni sposób, a następnie przetwarzać je w tak otrzymanym porządku dokładnie taką samą metodą jak poprzednio. Trochę więcej uwagi trzeba jednak w tej sytuacji poświęcić odróżnianiu otarć od przebić. Zauważmy jednak, że jeżeli analizujemy punkty posortowane w pierwszej kolejności względem jednej z naszych przekształconych „współrzędnych" ($x-y$ lub $x+y$), a następnie względem drugiej, to punkt kratowy obwodu jest punktem otarcia wtedy i tylko wtedy, gdy jego sąsiedzi na obwodzie wielokąta (tj. poprzedni i następny punkt kratowy) mają takie same wartości pierwszej „współrzędnej". Analiza kilku możliwych przypadków potwierdza prawdziwość tego spostrzeżenia, a jego prostota bardzo ułatwi nam implementację.

Algorytm II — wzorcowy

W tablicach *brzeg_x*$[1..MAXL]$, *brzeg_y*$[1..MAXL]$ zapisujemy wszystkie punkty kratowe obwodu, które następnie będziemy sortować. Funkcje sortujące w tym algorytmie działają podobnie jak poprzednio na tablicy *srt*$[1..MAXL]$, która jednak tym razem zawiera tylko indeksy punktów na obwodzie w odpowiednio posortowanej kolejności.

Przy konstrukcji grafu od razu określamy dla każdego punktu, czy jest on punktem otarcia w każdym z kierunków. W tablicach wartości logicznych *otarcie_dla_sumy*$[1..MAXL]$ oraz

[6] O tym, jak to zrobić, można przeczytać w [17] lub w [20].

otarcie_dla_różnicy[1..*MAXL*] zapisujemy dla każdego punktu, czy jest on otarciem dla prostych w kierunku $x+y=d$ oraz w kierunku $x-y=c$.

W tablicy wartości logicznych *wierzchołek*[1..*MAXL*] pamiętamy, czy dany punkt kratowy obwodu jest wierzchołkiem wielokąta czy nie. Natomiast w tablicy wartości logicznych *odwiedzony*[1..*MAXL*], wypełnionej początkowo wartościami **false**, będziemy zaznaczać, które z punktów kratowych obwodu wielokąta zostały odwiedzone przy przeszukiwaniu grafu — ostatniej fazie algorytmu, mającej na celu wypisanie wszystkich żądanych par wierzchołków. Tak naprawdę wystarczy jako odwiedzone zaznaczać tylko wierzchołki wielokąta, czyli końce ścieżek, i tak też będziemy robić w poniższym algorytmie.

Zmienia się także reprezentacja grafu. Nie potrzebujemy już list sąsiadów w postaci słowników — dla każdego wierzchołka wystarczy nam zwykła tablica indeksowana numerami punktów na obwodzie, ponieważ zbiór tych punktów jest określany wspólnie dla obu kierunków prostych.

```
{ Znajdujemy wszystkie punkty kratowe na obwodzie. }
L := 0;
for i := 1 to n do
begin
  { Punkty kratowe na boku (i, (i mod n)+1). }
  next_i := (i mod n)+1;
  k := max(abs(x[i]-x[next_i]), abs(y[i]-y[next_i]));
  for j := 1 to k do
  begin
    L := L+1;
    brzeg_x[L] := (x[i]·(k-j)+x[next_i]·j)/k;
    brzeg_y[L] := (y[i]·(k-j)+y[next_i]·j)/k;
    wierzchołek[L] := (j=k);
  end
end
{ Zaznaczamy wszystkie punkty otarć. }
for i := 1 to L do
begin
  next_i := i+1;
  if next_i>n then next_i := 1;
  prev_i := i-1;
  if prev_i=0 then prev_i := n;
  otarcie_dla_różnicy[i] :=
    (brzeg_x[prev_i]-brzeg_y[prev_i]=brzeg_x[next_i]-brzeg_y[next_i]);
  otarcie_dla_sumy[i] :=
    (brzeg_x[prev_i]+brzeg_y[prev_i]=brzeg_x[next_i]+brzeg_y[next_i]);
end
{ Analizujemy proste postaci x-y=c. }
{ Sortujemy w pierwszej kolejności po x-y, a w drugiej po x+y. }
{ Wykonujemy to przez dwukrotne sortowanie — drugie musi być stabilne! }
```

[7] Sortowanie stabilne nie zmienia względnej kolejności elementów równych. Jeśli drugie z wykonanych sortowań jest stabilne, to posortowanie najpierw względem $x+y$, a później względem $x-y$ daje ciąg posortowany w pierwszej kolejności względem $x-y$, a w drugiej względem $x+y$. Naturalne, stabilne algorytmy sortowania to: sortowanie przez zliczanie, kubełkowe czy MergeSort. Jednak także

```
31: for i := 1 to n do srt[i] := i;
32: sortuj_względem_sumy(L);
33: sortuj_względem_różnicy(L);
34: { Łączymy punkty niebędące otarciami w pary, tworząc krawędzie grafu G. }
35: i := 0;
36: while i < L do
37: begin
38:   do i := i+1 while i < L and otarcie_dla_różnicy[srt[i]];
39:   początek := i;
40:   do i := i+1 while i < L and otarcie_dla_różnicy[srt[i]];
41:   koniec := i;
42:   if koniec ⩽ L then
43:   begin
44:     dodaj_do_listy(sąsiedzi[srt[początek]], srt[koniec]);
45:     dodaj_do_listy(sąsiedzi[srt[koniec]], srt[początek]);
46:   end
47: end
48: { Powtarzamy wszystko analogicznie dla prostych postaci x+y=d. }
49: ...
50: { Możemy umieścić n/2 par urządzeń. }
51: wypisz(n/2);
52: { Przeszukujemy wszystkie ścieżki w grafie i wypisujemy żądane pary. }
53: for i := 1 to L do
54:   if wierzchołek[i] and not odwiedzony[i] then
55:   begin
56:     last_j := i;
57:     j := początek_listy(sąsiedzi[i]);
58:     while rozmiar_listy(sąsiedzi[j]) = 2 do
59:     begin
60:       if początek_listy(sąsiedzi[j]) = last_j then
61:       begin
62:         last_j := j; j := koniec_listy(sąsiedzi[j]);
63:       end else
64:       begin
65:         last_j := j; j := początek_listy(sąsiedzi[j]);
66:       end
67:     end
68:     odwiedzone[i] := true;
69:     odwiedzone[j] := true;
70:     wypisz(i, j);
71:   end
```

Przeanalizujmy złożoność tego algorytmu. Wyznaczenie wszystkich punktów kratowych obwodu i oznaczenie, które z nich są otarciami, ma koszt $O(l)$. Sortowania wymagają

każdy z innych algorytmów sortowania, w tym QuickSort i HeapSort, można łatwo przekształcić w stabilny.

$O(l \cdot \log l)$ operacji. Następnie łączenie punktów obwodu w pary ma koszt liniowy ze względu na ich liczbę, czyli $O(l)$. Tak więc sumaryczna złożoność tego algorytmu to $O(l \cdot \log l)$, co przy ograniczeniach z zadania w zupełności wystarcza.

Rozwiązanie wzorcowe zostało zaimplementowane w plikach `szk.cpp` (bez użycia STL), `szk1.pas`, `szk2.cpp` (z użyciem STL) oraz `szk4.java`.

Inne rozwiązania

Oprócz wyżej podanych rozwiązań istnieje wiele innych wolniejszych. Tutaj wspomnimy o dwóch z nich.

W bardzo bezpośrednim podejściu do rozwiązania zadania można wykonać symulację biegu promienia z każdego wierzchołka „krok po kroku". Można ją wykonywać na dwa sposoby.

Symulacja I

Pierwszy z nich polega na symulowaniu ruchu promienia pomiędzy kolejnymi punktami kratowymi obwodu. W tym podejściu musimy jednak przy każdym kroku przeglądać wszystkie boki wielokąta, aby sprawdzić, z którym z nich następuje kolejne przecięcie promienia. Ponieważ takie sprawdzenie potencjalnie może być wykonywane dla każdego punktu kratowego obwodu, to złożonością takiego rozwiązania jest $O(n \cdot l)$.

Tego typu rozwiązania otrzymywały około 18 punktów. Przykład realizacji tego podejścia znajduje się w pliku `szks3.cpp`.

Symulacja II

Drugie podejście polega na symulacji ruchu promienia pomiędzy wszystkimi punktami kratowymi wewnątrz wielokąta, a nie tylko na obwodzie. Po starcie z danego wierzchołka przesuwamy pozycję promienia świetlnego w każdym kroku o jeden na każdej ze współrzędnych (w odpowiednim kierunku).

W każdym kroku musimy wtedy badać, czy nie trafiliśmy w wierzchołek lub obwód wielokąta. Możemy robić to bezpośrednio, przeglądając wszystkie wierzchołki. Ponieważ każdy punkt kratowy wewnątrz wielokąta zostanie w tym podejściu odwiedzony co najwyżej 4 razy (w czterech kierunkach), to metoda ta ma złożoność $O(S \cdot l)$, gdzie S — pole wielokąta. Rozwiązanie to znajduje się w pliku `szks2.cpp`.

Aby usprawnić sprawdzanie przecięć promienia z obwodem, możemy na początku wykonywania algorytmu umieścić wszystkie punkty kratowe obwodu oraz wszystkie wierzchołki w dwóch zrównoważonych drzewach BST (na przykład drzewach czerwono-czarnych zaimplementowanych w C++ w bibliotece STL pod nazwą *set*). Dzięki temu sprawdzenie przynależności punktu do obwodu oraz sprawdzenie, czy jest on wierzchołkiem, możemy wykonywać w czasie $O(\log l)$. więc otrzymamy algorytm o złożoności $O(S \cdot \log l)$. Został on zaimplementowany w pliku `szks1.cpp`.

Rozwiązanie to jest bardzo wydajne dla małych testów, jednak dla testów z dodanym do wielokąta fragmentem o dużym (rzędu kilkuset milionów) polu jest bardzo nieefektywne. Programy zaimplementowane w ten sposób otrzymywały około 30 punktów.

Testy

Programy zawodników były testowane na zestawie 15 testów.

Nazwa	n	l	Opis
szk1.in	380	796	mały test poprawnościowy
szk2.in	840	2 016	mały test poprawnościowy
szk3.in	4 310	9 540	mały test poprawnościowy
szk4.in	7 302	17 026	mały test poprawnościowo-wydajnościowy
szk5.in	1 512	10 188	mały test poprawnościowy
szk6.in	14 404	92 714	średni test poprawnościowy
szk7.in	16 002	107 762	średni test poprawnościowy
szk8.in	24 748	120 354	średni test poprawnościowy
szk9.in	32 610	225 910	duży test poprawnościowy
szk10.in	12 008	250 558	duży test poprawnościowy
szk11.in	43 666	295 342	duży test poprawnościowo-wydajnościowy
szk12.in	93 004	289 242	duży test wydajnościowy
szk13.in	97 328	290 010	duży test wydajnościowy
szk14.in	84 722	261 730	duży test poprawnościowo-wydajnościowy
szk15.in	28 654	285 884	duży test poprawnościowo-wydajnościowy

Testy 1–4 są obrazami narysowanymi za pomocą programu gimp przekształconymi do formatu z treści zadania. Pozostałe testy były tworzone przez kombinację różnego rodzaju sposobów generowania fragmentów obwodu wielokąta spełniającego warunki zadania i dodawanie do niego artefaktów. Wśród artefaktów były:

- „schodki” — struktura polegająca na naprzemiennych krokach losowej długości w określoną stronę, za pomocą której można budować długie fragmenty obwodu;
- „krzyże” — rekurencyjnie zagłębiająca się struktura fraktalna;
- „spirale” — rekurencyjna struktura w pełni oddająca kształtem swoją nazwę;
- „falbanki” — losowo „poszarpany” fragment obwodu;
- „losowa tuleja” — struktura polegająca na wytworzeniu wąskiego korytarza o losowym brzegu;
- „tuleja” — struktura polegająca na powtórzeniu pewnego fragmentu figury wiele razy i wytworzeniu w ten sposób długiego korytarza, w którym kilka promieni odbija się wielokrotnie w nietrywialny sposób.

W testach 6–15 dodatkowo zadbano, aby figura miała duże pole, co umożliwiło odsianie rozwiązań nieefektywnych o złożoności zależnej od tej wielkości.

Zawody II stopnia

opracowania zadań

Piotr Zieliński
Treść zadania, Opracowanie

Marek Cygan
Program

OI, Etap II, dzień próbny, 6.02.2008

Blokada

W Bajtocji znajduje się dokładnie n miast. Między niektórymi parami miast biegną dwukierunkowe drogi. Poza miastami nie ma skrzyżowań, lecz mogą istnieć mosty, tunele i estakady. Między każdymi dwoma miastami może istnieć co najwyżej jedna droga. Z każdego miasta da się dojechać — bezpośrednio bądź też pośrednio — do każdego innego.

W każdym mieście mieszka dokładnie jeden mieszkaniec. Mieszkańcom doskwiera samotność. Okazuje się, że każdy z mieszkańców chce odwiedzić każdego innego w jego mieście, i to dokładnie raz. A zatem powinno odbyć się $n \cdot (n-1)$ spotkań. Tak, tak, **powinno**. *Niestety w Bajtocji trwają protesty programistów, domagających się wprowadzenia interwencyjnego skupu oprogramowania. Programiści planują, w ramach protestu, zablokowanie możliwości wjeżdżania do jednego z miast Bajtocji, wyjeżdżania z niego i przejeżdżania przez nie. Zastanawiają się teraz, które miasto wybrać, tak aby jak najbardziej uprzykrzyć życie mieszkańcom Bajtocji.*

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis sieci dróg Bajtocji,*
- *dla każdego miasta obliczy, ile spotkań nie mogłoby się odbyć, gdyby owe miasto zostało zablokowane przez programistów,*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu wejścia znajdują się dwie liczby naturalne oddzielone pojedynczym odstępem: n oraz m ($2 \leqslant n \leqslant 100\,000$, $1 \leqslant m \leqslant 500\,000$) oznaczające odpowiednio liczbę miast oraz liczbę dróg. Miasta są ponumerowane od 1 do n. W kolejnych m wierszach znajdują się opisy dróg. Każdy opis składa się z dwóch liczb a oraz b ($1 \leqslant a < b \leqslant n$) oddzielonych pojedynczym odstępem i oznacza, że istnieje droga między miastami o numerach a i b.

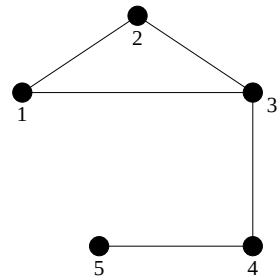
Wyjście

Twój program powinien wypisać dokładnie n liczb naturalnych, po jednej w wierszu. W i-tym wierszu powinna znaleźć się liczba spotkań, które nie odbyłyby się, gdyby programiści zablokowali miasto nr i.

Przykład

Dla danych wejściowych:

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



poprawnym wynikiem jest:

```
8
8
16
14
8
```

Rozwiązanie

Analiza problemu

Nawet początkujący uczestnik Olimpiady domyśla się z pewnością, że skoro w zadaniu jest mowa o miastach połączonych drogami, to rozwiązania należy szukać w teorii grafów. Tak jest w istocie — na Bajtocję możemy patrzeć jak na graf nieskierowany G, w którym miasta są wierzchołkami, a drogi krawędziami. Powiemy, że wierzchołek v jest osiągalny z wierzchołka u, jeśli istnieje w grafie G ścieżka zaczynająca się w u i kończąca się w v.

G jest grafem spójnym i nieskierowanym, co oznacza, że wszystkie wierzchołki są wzajemnie osiągalne. Dla każdego wierzchołka x chcemy znaleźć $wynik[x]$ — liczbę takich par (u,v), że po usunięciu z G wierzchołka x (wraz z incydentnymi krawędziami) v nie będzie osiągalne z u.

Bezpośrednia symulacja

Spróbujmy rozwiązać zadanie bezpośrednio. Aby obliczyć wynik dla wierzchołka x, usuniemy go z grafu, a następnie dla wszystkich par (u,v) sprawdzimy, czy są one wzajemnie osiągalne. Takie sprawdzenie możemy wykonać, przeszukując graf dowolną metodą, np. DFS. Złożoność testu dla jednej pary (u,v) będzie wówczas równa $O(n+m)$. Obliczenie wyniku dla jednego wierzchołka wymaga przeprowadzenia $O(n^2)$ sprawdzeń, czyli obliczenia dla całego grafu G zajmą czas $O(n^3 \cdot (n+m))$. Patrząc na rozmiar danych wejściowych, dochodzimy do wniosku, że to stanowczo za wolno.

Rozwiązanie wolne

Para wierzchołków u, v jest wzajemnie osiągalna w grafie G wtedy i tylko wtedy, gdy oba znajdują się w tej samej spójnej składowej. Gdybyśmy znali rozmiary spójnych składowych, na które rozpada się G po usunięciu z niego wierzchołka x, moglibyśmy w prosty sposób obliczyć dla niego wynik:

$$wynik[x] := 2 \cdot (n-1) + \sum_{i=1}^{k} s_i \cdot (n-1-s_i) \tag{1}$$

gdzie s_i to rozmiary spójnych składowych grafu z usuniętym wierzchołkiem x, a k to liczba owych składowych. Pierwszy składnik w powyższym wzorze to oczywiście liczba „uniemożliwionych" przez blokady spotkań „mieszkańca" wierzchołka x z wszystkimi pozostałymi. Następnie, i-ty składnik sumy to liczba „uniemożliwionych" spotkań „mieszkańców" i-tej składowej z „mieszkańcami" z pozostałych składowych.

Rozmiary spójnych składowych możemy łatwo wyznaczyć, korzystając np. z przeszukiwania grafu w głąb (czyli ponownie DFS). W ten sposób obliczamy odpowiedź dla każdego wierzchołka w czasie $O(n+m)$, co daje łączną złożoność czasową algorytmu $O(n \cdot (n+m))$. Algorytm ten uzyskiwał około połowy wszystkich punktów. Jego implementacja znajduje się w pliku `blos4.cpp`.

Rozwiązanie wzorcowe

Nietrudno zauważyć, że interesują nas tak naprawdę te wierzchołki, po usunięciu których G rozpada się na więcej niż jedną spójną składową — dla pozostałych wierzchołków wynik jest równy $2 \cdot (n-1)$. Wierzchołki takie w teorii grafów nazywa się *punktami artykulacji*. Okazuje się, że wszystkie punkty artykulacji grafu możemy znaleźć w czasie $O(n+m)$, korzystając z procedury przeszukiwania w głąb (znowu DFS).

W tym miejscu warto przypomnieć pewne pojęcia występujące w tej procedurze. Przeszukiwanie rozpoczynamy w wybranym wierzchołku, który nazwiemy *korzeniem*. W trakcie przeszukiwania, gdy po raz pierwszy odwiedzamy jakiś wierzchołek, to wchodzimy do niego z wierzchołka, który nazwiemy jego *ojcem*. Tak zdefiniowany korzeń i pojęcie ojca wyznaczają *drzewo przeszukiwania*, inaczej *drzewo DFS*. W trakcie przeszukiwania, będąc w wierzchołku v, staramy się odwiedzić wszystkich sąsiadów v z wyjątkiem jego ojca. Gdy okazuje się, że sąsiad u był już odwiedzony, to krawędź (v,u) nazywamy *krawędzią powrotną*. Ostatecznie w wyniku przeszukania grafu metodą DFS każda krawędź zostaje sklasyfikowana albo jako drzewowa, albo powrotna, tzn. każda krawędź łączy jakiś wierzchołek z pewnym jego potomkiem w drzewie DFS. To w szczególności oznacza, że nie ma tzw. krawędzi *skośnych*, czyli łączących wierzchołki z różnych poddrzew DFS.

Dodatkowo, w trakcie przeszukiwania możemy ponumerować wierzchołki grafu zgodnie z kolejnością, w jakiej je po raz pierwszy odwiedzamy. Kolejność ta jest zgodna z porządkiem *preorder* w drzewie przeszukiwań, a wartość przypisaną wierzchołkowi v oznaczymy $numer[v]$.

W trakcie procedury DFS możemy wyznaczyć sporo innych ciekawych własności drzewa przeszukiwań i grafu, w szczególności dla każdego wierzchołka v możemy obliczyć wartość funkcji *low* zdefiniowanej jako:

$$low[v] = \min\{numer(u) \mid \text{z } v \text{ da się dojść do } u, \text{ idąc krawędziami drzewa przeszukiwania w dół, a następnie co najwyżej raz krawędzią powrotną}\}.$$

Aby trochę lepiej zrozumieć sens powyższej definicji, przyjrzyjmy się pseudokodowi zmodyfikowanej procedury DFS, za pomocą której można wyznaczyć $low[v]$ (i oczywiście także $numer[v]$) dla każdego wierzchołka v grafu G.

```
1:  nr := 0; { zmienna pomocnicza do numeracji preorder }
2:  odwiedzony :=(false dla każdego wierzchołka);
3:  procedure DFS(v, ojciec)
4:  { Wywołanie: DFS(korzeń, nieistniejący_wierzchołek). }
5:  begin
6:    odwiedzony[v] :=true;
7:    nr := nr + 1; numer[v] := nr;
8:    low[v] := numer[v];
9:    for u ∈ Sąsiedzi(v) do
10:     if u ≠ ojciec then begin
11:       if not odwiedzony[u] then begin
12:         DFS(u, v);
13:         low[v] := min(low[v], low[u]);
14:       end else
15:         low[v] := min(low[v], numer[u]);
16:     end
17: end
```

Dla każdego wierzchołka v, $low[v]$ wyznaczamy w wywołaniu rekurencyjnym odpowiadającym v. Zaczynamy od $low[v] = numer[v]$ (wiersz 8.), a następnie próbujemy tę wartość jak najbardziej zmniejszyć. W tym celu analizujemy wszystkie wierzchołki u sąsiadujące z v różne od jego ojca i dla każdego z nich sprawdzamy, czy można poprowadzić przez niego ścieżkę z v typu *najpierw w dół drzewa, a potem co najwyżej jedna krawędź powrotna* do wierzchołka o mniejszym niż dotychczas znalezione numerze preorder. Jeżeli analizowany wierzchołek u jest już odwiedzony (wiersz 15.), to mamy krawędź powrotną, a zatem jedyną ścieżką spełniającą opisane warunki jest $v \to u$, prowadząca do wierzchołka o numerze $numer[u]$. Jeżeli zaś u nie jest odwiedzony (wiersze 12.–13.), to u znajdzie się w poddrzewie drzewa DFS ukorzenionym w v, czyli krawędź (v,u) prowadzi w dół drzewa. To oznacza, że szukaną ścieżkę z v możemy otrzymać, przedłużając o krawędź (v,u) dowolną ścieżkę spełniającą nasze kryterium wychodzącą z u — najlepsza z tych ścieżek prowadzi do wierzchołka o numerze $low[u]$. W momencie odwiedzenia v wierzchołek u nie jest jeszcze odwiedzony, więc musimy najpierw wykonać z niego rekurencyjnie przeszukiwanie (wiersz 12.), aby wartość $low[u]$ została policzona.

Jaki jest związek między funkcją *low* a punktami artykulacji? Okazuje się, że v jest punktem artykulacji G wtedy i tylko wtedy, gdy:

- jeżeli v nie jest korzeniem, to istnieje syn u wierzchołka v w drzewie DFS, dla którego zachodzi nierówność $low[u] \geqslant numer[v]$;
- jeżeli v jest korzeniem, to v ma co najmniej dwóch synów w drzewie DFS.

Zastanówmy się najpierw nad pierwszym z powyższych podpunktów. Jeśli zachodzi nierówność $low[u] \geqslant numer[v]$, to w wyniku usunięcia z grafu G wierzchołka v, u znajdzie się w odrębnej składowej wraz z ukorzenionym w nim poddrzewem DFS. Dlaczego? Ponieważ z żadnego wierzchołka w poddrzewie o korzeniu u nie wychodzi krawędź powrotna do żadnego przodka wierzchołka v (oraz dlatego, że w drzewie DFS nie ma krawędzi skośnych). Faktycznie, gdyby jakaś krawędź powrotna istniała, to wówczas zachodziłaby nierówność $low[u] < numer[v]$. Rozumując podobnie dla wszystkich wierzchołków u spełniających nierówność $low[u] \geqslant numer[v]$, otrzymamy, że usunięcie v spowoduje, że znajdą się one wraz z ukorzenionymi w nich poddrzewami DFS w osobnych spójnych składowych, a reszta grafu znajdzie się w jednej, osobnej składowej. To w szczególności oznacza, że jeżeli nie istnieje żaden u spełniający opisaną nierówność, to usunięcie v nie rozspójni G.

Przedstawiona argumentacja nie działa oczywiście, jeżeli v jest korzeniem — stąd druga część powyższego kryterium. Jej uzasadnienie jest całkiem proste. Jeżeli korzeń ma tylko jednego syna w drzewie DFS, to oczywiście jego usunięcie nie rozspójni grafu. Z kolei jeżeli synów korzenia jest więcej, to poddrzewa DFS w nich ukorzenione nie są połączone żadnymi krawędziami, gdyż byłyby to krawędzie skośne, a takich w przeszukiwaniu DFS nie ma. To oznacza, że usunięcie korzenia powoduje w tym przypadku rozkład grafu na składowe odpowiadające właśnie tym poddrzewom.

Dzięki zastosowaniu funkcji *low* potrafimy zatem rozpoznać w grafie punkty artykulacji i obliczać wynik zgodnie ze wzorem (1) jedynie dla nich. Nie polepsza to niestety złożoności pesymistycznej rozwiązania, która wynosi nadal $O(n \cdot (n+m))$, ale pozwala w niektórych przypadkach przyspieszyć rozwiązanie i wskutek tego zdobyć nieco więcej punktów. Implementację tego pomysłu można znaleźć w pliku `blos3.cpp`.

Maksymalną liczbę punktów można zdobyć, jeśli poczyni się jeszcze jedno spostrzeżenie. Już wcześniej zauważyliśmy, że spójne składowe grafu G powstałe po usunięciu z niego pewnego wierzchołka to (poza ewentualnie jedną) pewne poddrzewa DFS. Procedurę DFS można łatwo zmodyfikować tak, by przy okazji przechodzenia grafu obliczać jednocześnie dla każdego wierzchołka rozmiar poddrzewa DFS w nim ukorzenionego (przeprowadzenie tej modyfikacji pozostawiamy Czytelnikowi jako ćwiczenie). Jeśli mamy już tę informację (oraz policzoną funkcję *low*), to bez trudu możemy dla każdego punktu artykulacji v obliczyć wynik za pomocą wzoru (1), założywszy na przykład, że $s_1, \ldots, s_{k-1}$ ze wzoru odpowiadają pewnym poddrzewom DFS ukorzenionym w synach wierzchołka v, natomiast

$$s_k = n - 1 - s_1 - \ldots - s_{k-1}$$

(jeżeli v jest korzeniem, to $s_k = 0$). Rozwiązanie takie przechodzi wszystkie testy i można je znaleźć: zaimplementowane w C++ w pliku `blo.cpp`, w Pascalu — w pliku `blo2.pas` oraz w Javie — w pliku `blo3.java`. Działa oczywiście w czasie $O(n+m)$, gdyż wymaga zaledwie jednego przeszukania DFS grafu.

Wszystkie przedstawione w tym opracowaniu rozwiązania mają złożoność pamięciową $O(n+m)$, jako że przechowują graf w pamięci w postaci list sąsiedztwa. Podczas implementacji rozwiązania należało zwrócić uwagę na to, że obliczenia należy przeprowadzać na liczbach 64-bitowych, gdyż wynik dla pojedynczego wierzchołka może być nawet rzędu $O(n^2)$.

Testy

Testy zostały wygenerowane losowo. W poniższym opisie n oznacza liczbę wierzchołków w grafie, m — liczbę krawędzi, natomiast a — liczbę punktów artykulacji w grafie. Rozwiązania wolniejsze o złożoności $O(n \cdot (n+m))$ przechodziły testy 1–5. Dzięki zastosowaniu usprawnienia polegającego na przeszukiwaniu grafu jedynie dla punktów artykulacji możliwe było również zdobycie punktów za test 6.

Nazwa	n	m	a
blo1.in	8	12	2
blo2.in	30	50	6
blo3.in	100	150	26
blo4.in	250	800	24
blo5.in	500	10 000	22
blo6.in	10 000	90 000	774
blo7.in	40 000	100 000	2 937
blo8.in	100 000	200 000	6 762
blo9.in	100 000	350 000	7 260
blo10.in	100 000	500 000	6 946

Jakub Radoszewski, Wojciech Rytter — Treść zadania, Opracowanie
Michał Pilipczuk — Program

OI, Etap II, dzień pierwszy, 7.02.2008

BBB

Bajtazar ma konto w Bajtockim Banku Bitowym (w skrócie BBB). Na początku na koncie było p, a na końcu q bajtalarów. Wszystkie transakcje polegały na wpłacie bądź wypłacie jednego bajtalara. Bilans konta nigdy nie zszedł poniżej zera. Kasjer przygotował wyciąg z konta: pasek papieru z ciągiem plusów i minusów (plus oznacza wpłatę, a minus wypłatę jednego bajtalara). Okazało się, że operacje wykonywane na koncie nie zawsze były poprawnie księgowane. Kasjer nie może wydrukować nowego wyciągu, lecz musi poprawić ten już wydrukowany. Wyciąg nie musi być zgodny z rzeczywistością; wystarczy, że ciąg operacji na wyciągu będzie spełniał następujące dwa warunki:

- *saldo końcowe będzie się zgadzać z saldem początkowym i ciągiem operacji na wyciągu,*
- *zgodnie z ciągiem operacji na wyciągu, saldo konta nigdy nie spadło poniżej zera.*

Twoim zadaniem jest policzenie minimalnego czasu, jaki kasjer potrzebuje na korektę wyciągu. Kasjer może:

- *w ciągu x sekund zmienić wybraną operację na wyciągu na przeciwną, lub*
- *w ciągu y sekund przenieść ostatnią operację na początek wyciągu.*

Jeśli $p = 2$, $q = 3$, to na przykład $--++-+-++-+-+$ jest poprawnym wyciągiem. Natomiast wyciąg $---++++++$ nie jest poprawny, gdyż po trzeciej operacji saldo konta spadłoby poniżej zera, a ponadto saldo końcowe powinno wynosić 3, a nie 5. Możemy go jednak poprawić, zmieniając przedostatni symbol na przeciwny i następnie przenosząc ostatnią operację na początek wyciągu.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia aktualny wygląd wyciągu z konta Bajtazara (ciąg plusów i minusów) oraz liczby p, q, x i y,*
- *wypisze na standardowe wyjście minimalną ilość czasu, potrzebną na taką korektę wyciągu, żeby początkowy i końcowy bilans zgadzały się oraz żeby saldo konta w żadnym momencie nie spadło poniżej zera.*

Wejście

Pierwszy wiersz zawiera 5 liczb całkowitych n, p, q, x oraz y ($1 \leqslant n \leqslant 1\,000\,000$, $0 \leqslant p,q \leqslant 1\,000\,000$, $1 \leqslant x,y \leqslant 1000$), pooddzielanych pojedynczymi odstępami i oznaczających odpowiednio: liczbę transakcji wykonanych przez Bajtazara, początkowe i końcowe saldo konta oraz liczby sekund potrzebne na wykonanie pojedynczej zamiany i przesunięcia operacji na wyciągu. Drugi wiersz zawiera ciąg n plusów i/lub minusów, niezawierający odstępów.

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać jedną liczbę całkowitą, równą minimalnemu czasowi potrzebnemu na poprawienie wyciągu. Jeśli wyciąg nie wymaga poprawek, tą liczbą powinno być zero. Możesz założyć, że odpowiedni ciąg modyfikacji wyciągu będzie istniał dla każdych danych testowych.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`9 2 3 2 1` `---++++++`	`3`

Rozwiązanie

O co chodzi w zadaniu?

Przedstawmy zadanie BBB w nieco krótszy i przystępniejszy sposób, co ułatwi nam poszukiwanie jego rozwiązania. Mamy dany ciąg a złożony z plusów i minusów (oznaczających odpowiednio plus jedynki i minus jedynki). Taki ciąg wygodnie jest przedstawić jako wykres stanu konta w zależności od czasu, na którym + jest zilustrowany jako odcinek jednostkowy skierowany w górę pod kątem 45°, natomiast - jako podobny odcinek, tyle że skierowany w dół (przykłady takich wykresów znajdują się na rysunkach w dalszej części rozwiązania). Mamy do dyspozycji operacje zamiany znaku na przeciwny (koszt x) oraz przesunięcia cyklicznego ciągu o 1, czyli przesunięcia ostatniego znaku na początek (koszt y). Chcielibyśmy doprowadzić zadany wykres do postaci, w której zaczyna się na wysokości p, kończy na wysokości q oraz nie schodzi nigdy poniżej zera. Dodatkowo, mamy w zadaniu zagwarantowane to, że poszukiwane przekształcenie ciągu jest możliwe.

Na skróty

Dla niecierpliwych Czytelników już na samym początku umieszczamy krótki opis algorytmu rozwiązującego niniejsze zadanie. Jeśli to Wam wystarczy, to gratulujemy. W przeciwnym razie zapraszamy do lektury dalszej części opracowania, gdzie znajdziecie dokładne omówienie tego rozwiązania.

Przypomnijmy, że dla ciągu $a_1, a_2, \ldots, a_n$ ciąg sum częściowych $b_1, b_2, \ldots, b_n$ jest zdefiniowany (dla każdego i) następująco:

$$b_i = a_1 + a_2 + \ldots + a_i.$$

Algorytm wzorcowy

1. Wyznaczamy tablicę `MinPrefSuma`$[0..n-1]$. `MinPrefSuma`$[i]$ to minimalna spośród sum częściowych ciągu $a_{n-i+1}, \ldots, a_n, a_1, \ldots, a_{n-i}$.

2. Na podstawie powyższej tablicy obliczamy tablicę `ZamianyBezObrotów`$[0..n-1]$. `ZamianyBezObrotów`$[i]$ to minimalna liczba zamian znaków potrzebna, aby ciąg $a_{n-i+1},\ldots,a_n,a_1,\ldots,a_{n-i}$ przekształcić w poprawny wyciąg z konta.

3. Wyznaczamy wynik równy:

$$\min\{x\cdot \texttt{ZamianyBezObrotów}[i]+y\cdot i:\ 0\leqslant i<n\}.$$

Sposób realizacji punktu pierwszego w złożoności czasowej $O(n)$ został opisany w rozdziale *Ostatni problem*. Związek między pierwszym i drugim punktem jest wyjaśniony w rozdziale *Wersja ogólna*. Sposób wyznaczenia wartości `ZamianyBezObrotów`$[i]$ na podstawie `MinPrefSuma`$[i]$ w czasie stałym (czyli realizacja punktu drugiego) znajduje się w rozdziale *Łatwiejszy problem*.

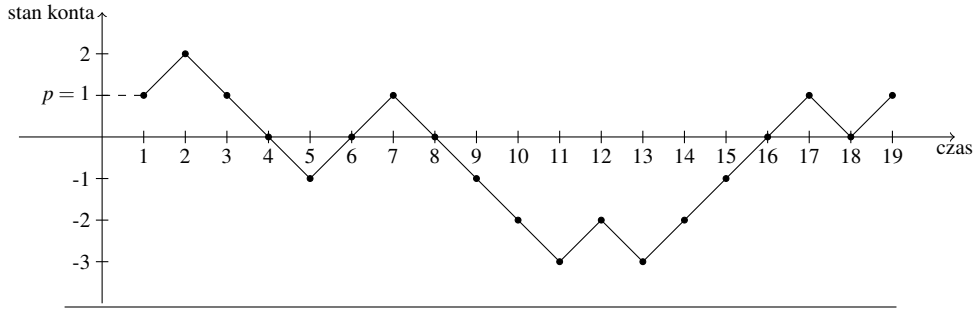
Łatwiejszy problem

Zastanawiając się nad rozwiązaniem niniejszego zadania, warto rozpocząć od uproszczenia problemu. Spróbujmy więc najpierw ograniczyć zakres dopuszczalnych operacji na ciągu i skupmy się jedynie na takich przekształceniach, które polegają tylko na zamianach znaków. W związku z tym możemy na jakiś czas zapomnieć o kosztach operacji x, y i dążyć jedynie do minimalizacji liczby wykonanych zamian. Niech

$$W=\texttt{ZamianyBezObrotów}[0].$$

Pokażemy, że wartość W zależy tylko od p i q oraz dwóch dodatkowych parametrów wyjściowego ciągu:

- różnicy salda końcowego i początkowego, czyli sumy wszystkich plus jedynek i minus jedynek z ciągu (s);
- minimalnej wartości salda w ciągu, czyli najniższego punktu na wykresie stanu konta (m [1]).



Rys. 1: Przykładowy wykres stanu konta w zależności od czasu ($n=18$, $p=1$) dla wyciągu `+---++----+-++++-+`. Dla tego wykresu mamy $s=0$, $m=-3$. Wyciąg z wykresu jest błędny, gdyż występują w nim ujemne salda.

[1] Warto zauważyć, że $m=\texttt{MinPrefSuma}[0]+p$. Pozwala to śledzić związek aktualnych rozważań z przedstawionym na początku algorytmem.

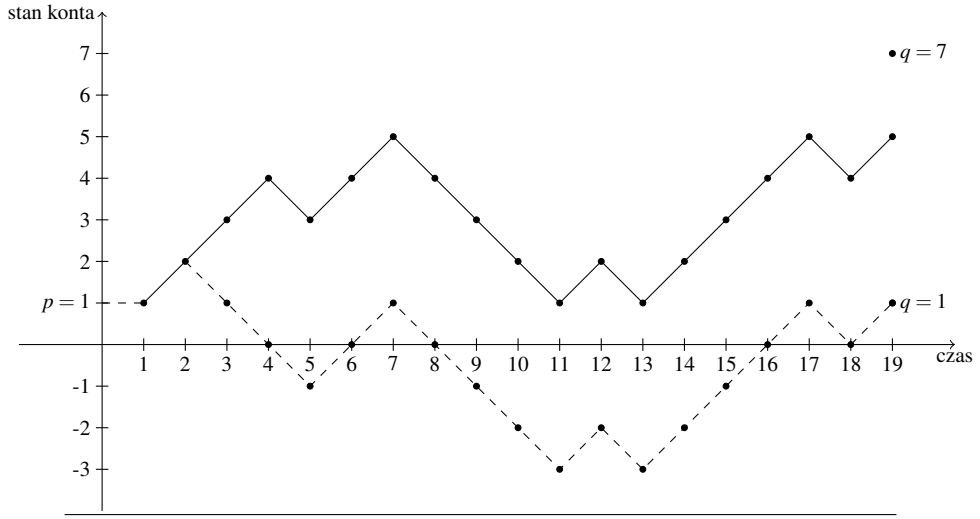
Aby skorygować ciąg operacji, musimy zapewnić właściwe saldo na końcu i nieujemny stan konta w całym okresie, którego dotyczy wyciąg. Policzymy najpierw wartość w — minimalną liczbę zmian znaków w ciągu konieczną do tego, aby stan konta był zawsze nieujemny. Jeżeli $m \geqslant 0$, to oczywiście żadne zamiany nie są potrzebne, czyli $w = 0$. W przeciwnym przypadku wystarczy zamienić $w = \lceil \frac{-m}{2} \rceil$ początkowych minusów na plusy[2]. Każda taka zamiana podnosi o 2 jednostki dalszą część wykresu stanu konta, więc po wykonaniu w takich zamian:

- początkowy odcinek ciągu, w którym dokonujemy korekty, składa się z samych plusów, więc wykres nie może na tym odcinku spaść poniżej zera;
- cały wykres po ostatniej operacji zamiany podniesie się o

$$2 \cdot w = 2 \cdot \left\lceil \frac{-m}{2} \right\rceil \geqslant -m,$$

  a zatem na tym odcinku wykres także nie spada poniżej zera.

Warto także zauważyć, że mniejsza liczba operacji zamiany nie wystarcza do skorygowania wyciągu.



Rys. 2: Wyciąg z rys. 1 poprawiony tak, by stan konta nie spadał poniżej zera. Dwa pierwsze minusy zostały zamienione na plusy, więc reszta wykresu podniosła się o 4 jednostki. W tym wyciągu saldo końcowe nie musi się jednak zgadzać z wymaganym (może być zbyt duże, np. dla $q = 1$, lub zbyt małe, np. dla $q = 7$).

Teraz pozostaje nam zmienić jak najmniej znaków, aby doprowadzić saldo końcowe wyciągu do wymaganej wartości. Musimy przy tym uważać, żeby nie popsuć już osiągniętej właściwości nieujemności stanu konta.

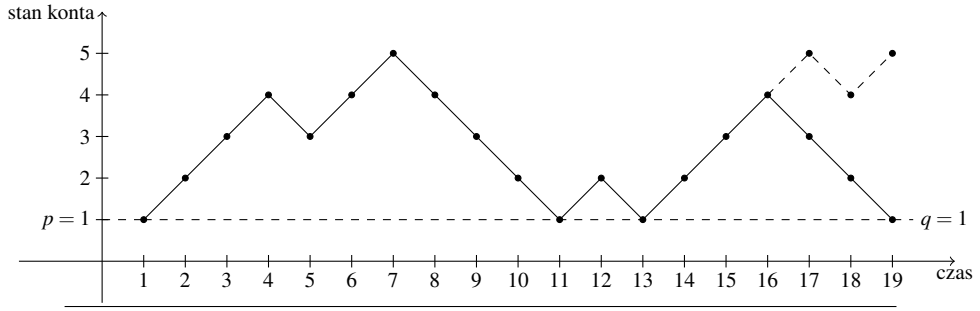
[2] Dla formalności dodajmy, że w ciągu znajduje się co najmniej $-m$ minusów, gdyż startujemy od nieujemnego stanu konta $p \geqslant 0$.

Po wykonaniu zamian z poprzedniej fazy różnica między saldem końcowym a początkowym wynosi $s+2w$, a powinna wynosić $q-p$. Z porównania tych liczb wychodzą dwa proste przypadki do rozważenia.

Jeżeli $q-p < s+2w$, to musimy obniżyć aktualne saldo końcowe, czyli zamienić

$$\frac{(s+2w)-(q-p)}{2} \tag{1}$$

plusów na minusy. Aby nie doprowadzić przy tym do ujemnego stanu konta, do zamiany wybierzemy ostatnie tyle plusów z ciągu. W ten sposób oczywiście osiągniemy pożądane saldo. Dodatkowo, ponieważ końcowy fragment wyciągu będzie składał się z samych minusów, więc na pewno wprowadzone zmiany nie spowodują zejścia wykresu poniżej zera.

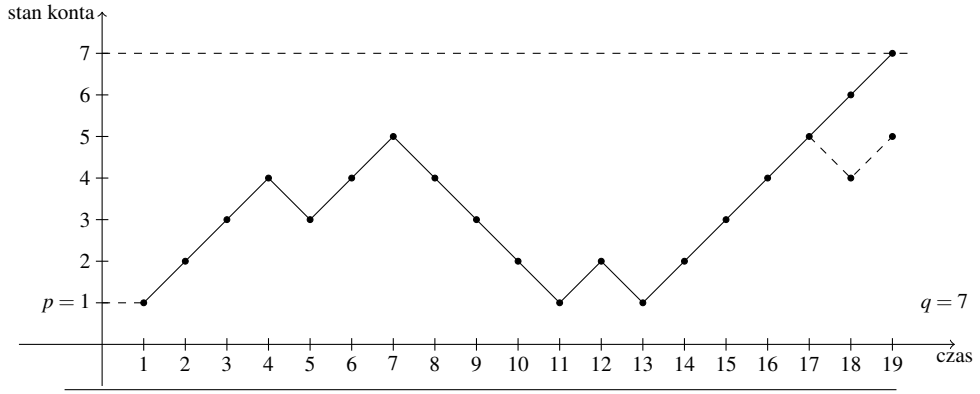


Rys. 3: Jeżeli $q=1$, to zachodzi pierwszy przypadek: $q-p=0<4=s+2w$. Aby naprawić sytuację, zmieniamy $4/2=2$ ostatnie plusy na minusy.

Jeżeli zaś $q-p \geqslant s+2w$, to musimy podwyższyć aktualne saldo końcowe, czyli zamienić

$$\frac{(q-p)-(s+2w)}{2} \tag{2}$$

minusów na plusy. Ponieważ żadna taka zamiana nie może spowodować obniżenia wykresu, w szczególności jego zejścia poniżej zera, to możemy do tego celu wybrać dowolne minusy z ciągu operacji, na przykład te końcowe.



Rys. 4: Jeżeli $q=7$, to zachodzi drugi przypadek: $q-p=6>4=s+2w$. Aby naprawić sytuację, zmieniamy $2/2=1$ ostatni minus na plus.

Podobnie jak poprzednio, łatwo sprawdzić, że wszystkie wykonane na tym etapie zamiany znaków są konieczne. Trzeba jeszcze chwilkę zastanowić się, czy są możliwe. Aby dało się je wykonać, wyrażenia (1) oraz (2) muszą mieć całkowite wartości i wartości te muszą być nie większe niż odpowiednio liczby plusów i minusów w wyjściowym ciągu. Okazuje się, że własności te wynikają stąd, że w zadaniu mamy zagwarantowane istnienie jakiegoś rozwiązania. Faktycznie:

- Żadna z operacji dopuszczonych w zadaniu nie zmienia parzystości sumy wszystkich plusów i minusów. Jeżeli więc jakieś rozwiązanie istnieje, to liczby s (wartość początkowa tej sumy) oraz $q-p$ są tej samej parzystości, a zatem wartości (1) i (2) są całkowite.
- Na mocy poczynionego założenia musi zachodzić $q-p \in [-n,n]$. Do tego samego przedziału musi oczywiście należeć także $s+2w$. Wystarczy teraz zauważyć, że z ciągu plus i minus jedynek o dowolnej sumie z przedziału $[-n,n]$ da się dojść do ciągu o dowolnej innej sumie z tego samego przedziału i o tej samej parzystości za pomocą zamian znaków.

Podsumowaniem powyższych rozważań jest następująca obserwacja:

Obserwacja 1. Przy oznaczeniu

$$w = \begin{cases} \lceil \frac{-m}{2} \rceil & \text{jeżeli } m < 0 \\ 0 & \text{w przeciwnym przypadku,} \end{cases} \tag{3}$$

minimalna liczba operacji korekty wyciągu, gdy nie dopuszczamy przesunięć cyklicznych, a zezwalamy jedynie na zamiany znaków, wynosi

$$W = \begin{cases} w + \frac{s+2w-q+p}{2} & \text{jeżeli } q-p < s+2w, \\ w + \frac{q-p-s-2w}{2} & \text{w przeciwnym przypadku.} \end{cases} \tag{4}$$

Zauważmy, że jeżeli znamy wartości s oraz m, to W możemy wyznaczyć w czasie stałym.

Wersja ogólna

Najwyższa pora powrócić do oryginalnej wersji zadania i przypomnieć sobie, że mamy do dyspozycji także cykliczne przesunięcia ciągu. Łatwo zauważyć, że zamiany i przesunięcia są całkowicie niezależne od siebie nawzajem — możemy założyć, że najpierw wykonywane są wszystkie przesunięcia cykliczne, a dopiero potem zamiany znaków. Możemy więc rozważyć wszystkich możliwe przesunięcia cykliczne zadanego ciągu (kolejno o 0 znaków, 1 znak, 2 znaki...), policzyć wartość W dla każdego z nich i wybrać optymalne rozwiązanie, uwzględniając koszty x, y operacji przesunięcia i zamiany. Odpowiada to obliczeniu zawartości tablicy `ZamianyBezObrotów`$[0..n-1]$ i wyznaczeniu końcowego minimum za pomocą wzoru z ostatniego punktu algorytmu wzorcowego.

Jak już zauważyliśmy, do efektywnego policzenia wartości W dla danego ciągu wystarczy znajomość wartości s oraz m dla tego ciągu. Oczywiście s nie zmienia się przy przesunięciu cyklicznym ciągu, więc możemy ją sobie policzyć raz na początku.

Większym problemem jest wyznaczenie wartości m, gdyż one mogą ulegać zmianie przy przesuwaniu cyklicznym ciągu. Musimy jednak jakoś sobie z nimi poradzić. Przede

wszystkim pozbądźmy się parametru p ze wzoru na m. Przy liczeniu m możemy po prostu założyć, że $p = 0$, wyznaczyć przy tym założeniu wartość m', a następnie przyjąć $m = m' + p$. Dla $p = 0$ wartość m możemy zdefiniować jako minimum w ciągu sum częściowych wyjściowego ciągu plus i minus jedynek. W ten sposób dochodzimy do problemu wyznaczenia wartości minimalnych ciągów sum częściowych dla wszystkich przesunięć cyklicznych wyjściowego ciągu, czyli tablicy `MinPrefSuma`$[0..n-1]$ z algorytmu wzorcowego. Można to oczywiście obliczyć bezpośrednio, lecz wtedy otrzymamy rozwiązanie o złożoności czasowej $O(n^2)$ (zaimplementowane w plikach `bbbs1.cpp` oraz `bbbs2.pas`). Rzut oka na ograniczenia z zadania pozwala stwierdzić, że taka metoda nie wystarcza (dodatkowo warto zauważyć, że od tego fragmentu zależy w tej chwili efektywność całego rozwiązania, gdyż wszystkie pozostałe obliczenia potrafimy już wykonać w czasie liniowym).

Ostatni problem

Sformułujmy problem, który pozostał nam do rozwiązania, abstrahując od plusów i minusów, sald bankowych oraz wszystkich parametrów z zadania. Mamy dany ciąg $a_1, \ldots, a_n$. Chcemy dla każdego przesunięcia cyklicznego tego ciągu o i znaleźć minimalną wartość `MinPrefSuma`$[i]$ występującą w jego ciągu sum częściowych.

Niech $b_1, \ldots, b_n$ będzie ciągiem sum częściowych wyjściowego ciągu $a_1, \ldots, a_n$. Jak wówczas wygląda ciąg sum częściowych $d^i_1, \ldots, d^i_n$ dla ciągu a, w którym i końcowych wyrazów przerzucimy na początek? Jest to, jak łatwo sprawdzić:

$$\begin{aligned}
d^i_1 &= b_{n-i+1} - b_{n-i} &&= a_{n-i+1} \\
d^i_2 &= b_{n-i+2} - b_{n-i} &&= a_{n-i+1} + a_{n-i+2} \\
&\vdots \\
d^i_i &= b_n - b_{n-i} &&= a_{n-i+1} + a_{n-i+2} + \ldots + a_n \\
d^i_{i+1} &= b_1 + (b_n - b_{n-i}) &&= a_{n-i+1} + a_{n-i+2} + \ldots + a_n + a_1 \\
d^i_{i+2} &= b_2 + (b_n - b_{n-i}) &&= a_{n-i+1} + a_{n-i+2} + \ldots + a_n + a_1 + a_2 \\
&\vdots \\
d^i_n &= b_{n-i} + (b_n - b_{n-i}) &&= a_{n-i+1} + a_{n-i+2} + \ldots + a_n + a_1 + a_2 + \ldots + a_{n-i}
\end{aligned} \tag{5}$$

Innymi słowy:

a) W przesunięciu cyklicznym i końcowych wyrazów ciągu a przerzucamy na początek. To powoduje, że i początkowych wyrazów ciągu sum częściowych jest obliczanych na podstawie tych wyrazów. Dodatkowo, tych i początkowych wyrazów różni się od i końcowych wyrazów ciągu sum częściowych oryginalnego ciągu a tylko o sumę

$$a_1 + a_2 + \ldots + a_{n-i} = b_{n-i}.$$

b) Kolejnych $n - i$ wyrazów ciągu sum częściowych różni się od oryginalnych, początkowych $n - i$ wyrazów ciągu sum częściowych dla ciągu a o wartość

$$a_{n-i+1} + a_{n-i+2} + \ldots + a_n = b_n - b_{n-i}.$$

Teraz przydadzą nam się dwa dodatkowe ciągi $pref$ oraz suf, zdefiniowane dla każdego j następująco:

$$\begin{aligned} pref_j &= \min\{b_1, b_2, \ldots, b_j\} \\ suf_j &= \min\{b_j, b_{j+1}, \ldots, b_n\}. \end{aligned}$$

Zauważmy, że minimum ciągu (5) możemy za ich pomocą wyrazić następująco:

$$\min(suf_{n-i+1} - b_{n-i},\ pref_{n-i} + (b_n - b_{n-i})). \tag{6}$$

Pierwszy człon w powyższym zapisie pochodzi z wyrazów przerzuconych w trakcie przesunięcia cyklicznego (opisanych w podpunkcie a)), natomiast drugi — z pozostałych (opisanych w podpunkcie b)).

Widzimy, że jeżeli znamy ciągi b, $pref$ oraz suf, to minimum dla ciągu przesuniętego cyklicznie o i liter możemy obliczyć za pomocą wzoru (6) w złożoności czasowej $O(1)$. Pozostaje zauważyć, że każdy z tych ciągów możemy wyznaczyć w złożoności $O(n)$ — najpierw liczymy b od lewej do prawej, a potem, na podstawie b, $pref$ również od lewej do prawej oraz suf od prawej do lewej.

W ten sposób uzyskujemy rozwiązanie wzorcowe, o złożoności czasowej i pamięciowej $O(n)$. Zostało ono zaimplementowane w plikach `bbb.cpp`, `bbb1.pas` oraz `bbb2.java`.

Inne rozwiązania

Zdecydowana większość alternatywnych rozwiązań tego zadania różni się od wzorcowego jedynie metodą podejścia do ostatniej fazy, czyli do wyznaczania minimów w ciągach sum częściowych dla przesunięć cyklicznych zadanego ciągu. Zauważmy, że wszystkie przesunięcia cykliczne ciągu a występują jako spójne n-elementowe fragmenty ciągu aa (czyli ciągu powstałego przez sklejenie dwóch kopii ciągu a). Niech $c_1, \ldots, c_{2n}$ oznacza ciąg sum częściowych ciągu aa. Wówczas łatwo zauważyć, że jeżeli od każdego wyrazu ciągu

$$c_i, c_{i+1}, \ldots, c_{i+n-1}$$

(gdzie $i \leqslant n$) odejmiemy sumę $a_1 + \ldots + a_{i-1} = c_{i-1}$, to otrzymamy ciąg sum częściowych dla przesunięcia cyklicznego ciągu a o $n-i$ wyrazów. To oznacza, że gdybyśmy potrafili dla każdego „okna” długości n w ciągu c wyznaczyć minimum, to na tej podstawie bylibyśmy w stanie łatwo obliczać szukane minima ciągów sum częściowych dla wszystkich przesunięć cyklicznych wyjściowego ciągu.

Jeszcze inny problem

Sprowadziliśmy zatem podproblem wyjściowego problemu do… jeszcze innego problemu. Tym razem mamy dany ciąg $c_1, \ldots, c_{2n}$ i chcielibyśmy poznać minimalną wartość w każdym oknie (tzn. spójnym fragmencie) długości n tego ciągu.

Ten klasyczny problem można rozwiązać na wiele sposobów. Prostą metodę (działającą także wówczas, gdy interesują nas okna różnych rozmiarów) o złożoności $O(n \log n)$ otrzymujemy przy użyciu statycznego drzewa przedziałowego czy też licznikowego,

w którego liściach zapisujemy elementy ciągu c, natomiast w węzłach wewnętrznych — minima z poddrzew im odpowiadających[3].

Bardziej bezpośrednie podejście do problemu polega na utrzymywaniu zbioru n kolejnych elementów ciągu c w strukturze danych, która pozwala na dodawanie i usuwanie elementów (co odpowiada przesunięciu okna o jedną pozycję w prawo) oraz na odpowiadanie na pytania o minimalną wartość zawartą w zbiorze. Taką strukturą danych, po dodaniu odpowiedniej tablicy, w której zapamiętamy bieżące pozycje elementów w strukturze, może być kopiec binarny (stóg)[4]. To rozwiązanie, o złożoności czasowej $O(n \log n)$, zostało zaimplementowane w plikach `bbb6.cpp` oraz `bbb7.pas` — uzyskiwało ono na zawodach maksymalną liczbę punktów. Zamiast kopca możemy zastosować także zrównoważone drzewo BST (otrzymamy wówczas dokładnie tę samą złożoność czasową). Programujący w C++ mogą wykorzystać gotową implementację, tj. strukturę `multiset` z biblioteki STL. Takie rozwiązanie znajduje się w pliku `bbbs7.cpp` — nie uzyskiwało ono na zawodach maksymalnej liczby punktów ze względu na nadmierne zapotrzebowanie na pamięć.

Jest także kolejne rozwiązanie zadania o złożoności czasowej $O(n)$, do implementacji którego wystarczy nam jedna lista, którą oznaczymy przez L. W tym rozwiązaniu wykorzystujemy obserwację, że jeżeli w oknie znajdują się dwa elementy c_i oraz c_j, takie że $i < j$ oraz $c_i \geqslant c_j$, to c_i nie będzie miało wpływu na wynik ani dla bieżącej pozycji okna, ani też dla żadnej kolejnej (dlaczego tak jest?). Na liście L będziemy więc przechowywać tylko te elementy z aktualnego okna, które są mniejsze od wszystkich znajdujących się na prawo od nich w oknie. Łatwo widać, że L będzie zawsze posortowana rosnąco, więc jej pierwszy element będzie zarazem elementem najmniejszym dla rozważanego okna.

W jaki sposób należy zmodyfikować L przy przesunięciu okna o jedną pozycję w prawo? Jeżeli pierwszy (i zarazem najmniejszy) element L wypada przy przesunięciu poza okno, to należy go usunąć z L. Z kolei element, który dochodzi do okna z prawej strony, wymusi usunięcie z L wszystkich elementów większych od niego (czyli pewnej liczby elementów z końca listy), po czym sam zostanie dodany na końcu listy L.

Poza operacją usuwania elementów z końca listy L, wszystkie pozostałe ewidentnie sumują się do złożoności czasowej $O(n)$. Do przeanalizowania usuwania z listy wykorzystamy analizę kosztu zamortyzowanego, która w tym przypadku jest bardzo prosta: każdy element zostaje w algorytmie wstawiony na listę dokładnie raz, a łączna liczba usunięć z listy jest nie większa niż liczba wstawień na listę. To pokazuje, że również sumaryczny koszt usunięć można oszacować przez $O(n)$, co kończy analizę złożoności tego rozwiązania. Jego implementację można znaleźć w pliku `bbb3.cpp`.

Okazuje się, że jeżeli przypomnimy sobie o jednej szczególnej własności, którą posiada ciąg c z naszego zadania, to rozważany problem można rozwiązać jeszcze łatwiej, nadal w czasie liniowym. Zauważmy mianowicie, że c to ciąg sum częściowych ciągu złożonego tylko z wyrazów $+1$ i -1, czyli że każde dwa kolejne wyrazy ciągu c różnią się co najwyżej o 1. W takim przypadku zbiór liczb zawartych w oknie możemy zapamiętać w prostej tablicy zliczającej. Dopracowanie szczegółów tego pomysłu pozostawiamy jako ćwiczenie. W jego wykonaniu mogą być pomocne rozwiązania z plików `bbb4.cpp` oraz `bbb5.pas`.

Na koniec dodajmy, że dość dokładna analiza większości z opisanych tu rozwiązań, a także jeszcze jedno ciekawe rozwiązanie liniowe, znajdują się w opracowaniu zadania

[3] Więcej o statycznych drzewach przedziałowych można przeczytać np. w opisie rozwiązania zadania *Tetris 3D* z książeczki XIII Olimpiady Informatycznej [13].

[4] Więcej o kopcach można przeczytać np. w książce [20].

Sound z Bałtyckiej Olimpiady Informatycznej 2007. Materiały te można znaleźć (tylko w wersji anglojęzycznej) np. na stronie `http://www.boi2007.de/tasks/book.pdf`.

Testy

Rozwiązania zawodników były sprawdzane na 13 zestawach testów, których opisy są zawarte w poniższej tabeli.

Nazwa	n	q − p	Opis
bbb1a.in	22	2	mały test poprawnościowy
bbb1b.in	1	−1	przypadek brzegowy
bbb2a.in	61	1	mały test poprawnościowy
bbb2b.in	47	25	mały test poprawnościowy
bbb3a.in	288	12	mały test poprawnościowy
bbb3b.in	375	211	mały test poprawnościowy
bbb4a.in	2761	11	mały test poprawnościowy
bbb4b.in	2741	719	mały test poprawnościowy
bbb5a.in	8921	−41	mały test poprawnościowy
bbb5b.in	7654	5298	mały test poprawnościowy
bbb6.in	98867	19465	średni test
bbb7.in	201346	−50	średni test
bbb8.in	361555	201	średni test
bbb9a.in	499822	−10230	średni test
bbb9b.in	478963	108939	średni test
bbb10.in	808669	687	duży test
bbb11.in	912880	−2238	duży test
bbb12.in	970293	−853	duży test
bbb13a.in	999879	−4789	duży test
bbb13b.in	1000000	340354	duży test

Piotr Stańczyk
Treść zadania, Opracowanie

Piotr Stańczyk
Program

OI, Etap II, dzień pierwszy, 7.02.2008

Pociągi

W Bajtocji odbędzie się Parada Kolorowych Pociągów. Na torach technicznych bajtockiego dworca trwają intensywne przygotowania. Na dworcu jest n równoległych torów, ponumerowanych od 1 do n. Na i-tym torze ustawiono pociąg o numerze i. Każdy pociąg składa się z l wagonów, z których każdy jest pomalowany na jeden z 26 kolorów (oznaczonych małymi literami alfabetu angielskiego). Mówimy, że dwa pociągi **wyglądają identycznie**, *jeśli ich kolejne wagony są tego samego koloru.*

Parada będzie polegać na tym, że co minutę stacyjny dźwig zamieni miejscami pewną parę wagonów. Prawdziwa parada odbędzie się jednak dopiero jutro. Dziś dyżurny ruchu Bajtazar bacznie przyglądał się próbie generalnej. Dokładnie zapisał sobie ciąg kolejno zamienianych par wagonów.

Bajtazar nie lubi, gdy zbyt wiele pociągów wygląda identycznie. Chciałby, żebyś dla każdego pociągu p policzył maksymalną liczbę pociągów, które w pewnym momencie wyglądają identycznie jak pociąg p w owym momencie.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opisy pociągów stojących na torach oraz ciąg wykonywanych zamian wagonów,*
- *dla każdego pociągu wyznaczy maksymalną liczbę pociągów, które w pewnym momencie wyglądają tak samo jak on,*
- *wypisze wynik na standardowe wyjście.*

Wejście

Pierwszy wiersz wejścia zawiera trzy liczby naturalne n, l oraz m ($2 \leqslant n \leqslant 1\,000$, $1 \leqslant l \leqslant 100$, $0 \leqslant m \leqslant 100\,000$), oznaczające odpowiednio liczbę pociągów, ich długość oraz liczbę wykonywanych zamian wagonów. W kolejnych n wierszach znajdują się opisy kolejnych pociągów stojących na torach. k-ty z tych wierszy składa się z l małych liter alfabetu angielskiego reprezentujących kolory kolejnych wagonów k-tego pociągu. Za opisami pociągów znajduje się m wierszy zawierających opisy kolejnych zamian, w kolejności ich wykonywania. W r-tym z tych wierszy znajdują się cztery liczby całkowite p_1, w_1, p_2, w_2 ($1 \leqslant p_1, p_2 \leqslant n$, $1 \leqslant w_1, w_2 \leqslant l$, $p_1 \neq p_2$ lub $w_1 \neq w_2$) i opisują one r-tą operację wykonywaną przez dźwig — zamianę wagonu numer w_1 z pociągu p_1 z wagonem w_2 pociągu p_2.

Wyjście

Twój program powinien wypisać dokładnie n wierszy. k-ty wiersz powinien zawierać jedną liczbę całkowitą — liczbę pociągów wyglądających tak samo jak pociąg numer k w pewnym momencie czasu.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
5 6 7	3
ababbd	3
abbbbd	3
aaabad	2
caabbd	3
cabaad	
2 3 5 4	
5 3 5 5	
3 5 2 2	
1 2 4 3	
2 2 5 1	
1 1 3 3	
4 1 5 6	

Oto wygląd pociągów w kolejnych fazach próby generalnej:

```
tor 1:  ababbd    ababbd    ababbd    ababbd    aaabbd    aaabbd    aaabbd    aaabbd
tor 2:  abbbbd    ababbd    ababbd    aaabbd    aaabbd    acabbd    acabbd    acabbd
tor 3:  aaabad -> aaabad -> aaabad -> aaabbd -> aaabbd -> aaabbd -> aaabbd -> aaabbd
tor 4:  caabbd    caabbd    caabbd    caabbd    cabbbd    cabbbd    cabbbd    dabbbd
tor 5:  cabaad    cabbad    caabbd    caabbd    caabbd    aaabbd    aaabbd    aaabbc

           (0)       (1)       (2)       (3)       (4)       (5)       (6)       (7)
```

Dla pociągów 1, 2 i 3 najwięcej podobnych było np. w momencie (4) *(były one nawzajem do siebie podobne). Dla pociągu 5 najwięcej mu podobnych było w momentach* (5) *i* (6)*. Dla pociągu 4 najwięcej podobnych było np. w momencie* (2)*.*

Rozwiązanie

Pierwszy pomysł przychodzący na myśl po przeczytaniu treści zadania to rozwiązanie go za pomocą prostej symulacji. Wystarczy napisać program, który będzie wykonywał przestawienia wagonów zgodnie z opisem zawartym w danych i po każdej zamianie będzie sprawdzał, dla każdego pociągu, ile pociągów wygląda tak samo jak on. Będzie także na bieżąco wyznaczał dla każdego pociągu *wynik* — maksymalną liczbę pociągów do niego podobnych w trakcie dotychczas wykonanych kroków symulacji. Dla różnych sposobów realizacji tego pomysłu otrzymamy algorytmy o różnej złożoności czasowej. Na efektywność obliczeń mają wpływ reprezentacja pociągów (czyli struktura danych) oraz

sposoby wykonywania przestawiania wagonów i wyznaczania liczby podobnych pociągów. Rozpocznijmy od analizy prostej symulacji, którą następnie, w kolejnych podrozdziałach opracowania, będziemy stopniowo ulepszać.

W opisie rozwiązania będziemy stosować takie same oznaczenia, jak w treści zadania, a więc n będzie liczbą pociągów, l — długością pociągu (wszystkie są jednakowej długości), natomiast m — liczbą wykonywanych przestawień wagonów. Wprowadzamy także jedno dodatkowe oznaczenie

$$f = nl + m\log l, \tag{1}$$

z którego będziemy korzystać w trakcie analizy złożoności kilku algorytmów (wtedy wyjaśni się, skąd się ta wartość bierze).

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

Najbardziej naturalną reprezentacją danych jest zapisanie pociągu jako ciągu l małych liter alfabetu angielskiego, czyli słowa długości l. Taka reprezentacja jest prosta w implementacji i na dodatek pozwala łatwo i efektywnie przestawiać wagony w pociągach — wystarczy zamienić miejscami dwie litery w słowach. Bardziej czasochłonna okazuje się druga operacja — wyznaczenie dla pociągu liczby pociągów do niego podobnych. Dla konkretnego pociągu możemy porównać opisujące go słowo z pozostałymi w czasie $O(n \cdot l)$ (w pesymistycznym przypadku porównanie dwóch słów wymaga l porównań liter), zatem czas potrzebny dla wszystkich pociągów wynosi $O(n^2 \cdot l)$. Ponieważ porównania musimy przeprowadzać po każdym przestawieniu wagonów, a przestawień wykonujemy łącznie m, to ostateczna złożoność tego algorytmu wynosi $O(n^2 \cdot l \cdot m)$ (jego implementacja znajduje się w pliku `pocs1.cpp`). Porównawszy tę złożoność z zakresem danych wejściowych, można spodziewać się, że rozwiązanie takie niestety nie uzyska zbyt dużej liczby punktów. Zastanówmy się zatem, jak je przyspieszyć.

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

W poprzednim rozwiązaniu dużo czasu zajęło nam porównywanie pociągów. Zauważmy jednak, że nie wszystkie wykonywane porównania są konieczne. Po przestawieniu dwóch wagonów może zmienić się liczba podobnych pociągów tylko dla tych, które uległy zmianie, i dla tych, które są do nich podobne. Z tego wynika, że wystarczy wyznaczyć podobne pociągi tylko dla (co najwyżej) dwóch zmienianych pociągów (przed przestawieniem wagonów i po nim), a następnie uaktualnić wynik dla nich i dla wszystkich pociągów do nich podobnych (tzn. podobnych przed przestawieniem bądź po nim). Symulacja przestawienia dwóch wagonów wymaga więc już tylko czasu $O(n \cdot l)$, czyli symulację wszystkich m kroków wykonujemy w czasie $O(n^2 \cdot l + n \cdot l \cdot m)$, gdzie składnik $O(n^2 \cdot l)$ to czas potrzebny na wyznaczenie liczb podobnych pociągów przed rozpoczęciem symulacji — tę operację wykonujemy tak samo jak poprzednio, przez porównanie wszystkich par pociągów.

To rozwiązanie zostało zaimplementowane w pliku `pocs2.cpp`.

Rozwiązanie o złożoności czasowej $O(n \cdot l + n \cdot l \cdot m) = O(n \cdot l \cdot m)$

Poprzednio udało nam się przyspieszyć fazę aktualizacji wyniku, spróbujmy teraz poprawić efektywność obliczania wartości początkowych. Aby wyznaczyć dla każdego pociągu

liczbę pociągów do niego podobnych, możemy najpierw posortować pociągi (a dokładniej słowa, które je opisują). Sortowanie można zrealizować za pomocą dowolnego algorytmu wykonującego $O(n \log n)$ porównań sortowanych obiektów, czyli działającego w naszym przypadku w czasie $O(l \cdot n \log n)$. Jednak dla słów nad niewielkim alfabetem znacznie bardziej praktyczne jest sortowanie pozycyjne, które działa w czasie $O(n \cdot l + l \cdot |\Sigma|)$, gdzie $|\Sigma|$ to moc alfabetu[1]. Gdy pociągi są posortowane, można łatwo w czasie $O(n \cdot l)$ wyznaczyć dla każdego z nich liczbę pociągów podobnych. W ten sposób uzyskujemy algorytm o całkowitej złożoności $O(n \cdot l + n \cdot l \cdot m) = O(n \cdot l \cdot m)$.

Rozwiązanie o złożoności czasowej $O(n \cdot l + n \cdot m)$?

Skonstruowaliśmy już algorytm n razy szybszy niż wyjściowa symulacja. Patrząc jednak na możliwe zakresy liczb n, m oraz l, należy spodziewać się, że autorzy zadania oczekują jeszcze lepszego rozwiązania. W szczególności warto zastanowić się, czy nie da się szybciej sprawdzać, czy dwa pociągi wyglądają tak samo. Porównań pociągów wykonujemy bardzo dużo — aż $O(n \cdot m)$ — zatem przyspieszenie tej operacji w istotny sposób wpłynie na złożoność całego algorytmu.

Pierwszym z pomysłów jest zastosowanie *haszowania*. Operacja ta polega na przypisaniu obiektom nowych reprezentacji (możemy je nazwać etykietami), o jak najmniejszej wartości. Zazwyczaj jest to funkcja $h : D \mapsto \{0, \ldots, N\}$, gdzie D jest zbiorem obiektów, a N — stosunkowo małą liczbą naturalną. Oczywiście haszowanie (jako funkcja) przypisuje jednakowe wartości jednakowym argumentom. Nie zawsze natomiast, choć jest to pożądane, funkcja haszująca jest różnowartościowa na zbiorze D [2]. W naszym zadaniu wykorzystamy modularną funkcję haszującą, zadaną wzorem

$$h(x) = x \bmod P,$$

gdzie P jest pewną liczbą pierwszą. Aby zastosować h dla naszych obiektów, musimy słowa opisujące pociągi zinterpretować jako liczby. W tym celu przyporządkujemy literom wartości liczbowe zadane funkcją g ($g(a) = 0$, $g(b) = 1$, …, $g(z) = 25$), a następnie rozszerzymy funkcję g na słowa, przypisując słowu $s = c_1 c_2 \ldots c_l$ wartość:

$$g(c_1 \ldots c_l) = \sum_{k=1}^{l} g(c_k) \cdot 26^{k-1}.$$

Z powyższego wzoru widać, że $g(s)$ może być bardzo dużą liczbą, zatem jej wyznaczenie wymaga zaimplementowania własnej arytmetyki na długich liczbach. Ponieważ jednak nie potrzebujemy wartości $g(s)$, a jedynie $h(g(s))$, sprawa znacznie się upraszcza:

$$\begin{aligned} h(g(c_1 \ldots c_k)) &= h\Big(\sum_{k=1}^{l} g(c_k) \cdot 26^{k-1}\Big) \\ &= \Big(\sum_{k=1}^{l} g(c_k) \cdot 26^{k-1}\Big) \bmod P \\ &= \sum_{k=1}^{l} \big((g(c_k) \bmod P) \cdot (26^{k-1} \bmod P)\big) \bmod P. \end{aligned}$$

[1] O wszystkich wspomnianych metodach sortowania można przeczytać w [15], [17] czy [20].

[2] Więcej na temat haszowania, w szczególności o dobrych funkcjach haszujących, można przeczytać np. w [20].

Co więcej, raz obliczoną wartość $h(g(s))$ można łatwo aktualizować po zamianie wagonu w pociągu. Załóżmy, że w pociągu $s = c_1 \ldots c_l$ wagon na pozycji k został zmieniony na wagon c'_k; niech s' będzie słowem opisującym pociąg po zmianie. Wówczas

$$h(g(s')) = (h(g(s)) + (c'_k - c_k) \cdot 26^{k-1}) \bmod P.$$

Zauważmy, że jeżeli na samym początku spamiętamy reszty z dzielenia potęg $26^0, 26^1, \ldots, 26^{l-1}$ przez P, to $h(g(s'))$ będziemy mogli wyznaczyć na podstawie $h(g(s))$ w czasie stałym.

Aby porównać dwa pociągi, można wstępnie sprawdzić, czy wartości ich funkcji haszujących są równe. Jeśli nie są, to pociągi na pewno są różne. Jeśli natomiast wartości są równe, to być może pociągi są takie same, ale nie mamy co do tego pewności.

W tym momencie trzeba podjąć decyzję. Albo chcemy mieć program, którego wynikom zawsze ufamy, albo też jesteśmy gotowi podjąć ryzyko błędu. W pierwszym przypadku, po pozytywnym wyniku porównania etykiet słów (czyli wartości funkcji haszującej) należy wykonać jeszcze dokładne porównanie samych słów w czasie $O(l)$. W drugim przypadku możemy pominąć sprawdzenie, uznając, że słowa o równych etykietach są równe. Wówczas nasz program może zwracać błędne wyniki, gdy zawiedzie przyjęte założenie (ma to miejsce zazwyczaj, gdy dane są dobrane „złośliwie" do funkcji lub funkcja haszująca jest po prostu źle wybrana).

W pierwszym przypadku otrzymujemy algorytm, który w najgorszym razie działa tak samo długo jak poprzednie rozwiązanie (to rozwiązanie zostało zaimplementowane w pliku `pocs3.cpp`). Sytuacja taka ma miejsce, gdy występuje wiele par podobnych pociągów. Jeśli natomiast zaryzykujemy pominięcie dokładnego sprawdzenia, to wykonujemy każde porównanie pociągów w czasie $O(1)$, czyli obliczamy rozwiązanie w sumarycznym czasie $O(n \cdot l + m \cdot n)$. Czas $O(n \cdot l)$ jest nam tym razem potrzebny na wyznaczenie początkowych wartości funkcji haszujących oraz, jak poprzednio, na policzenie początkowych wyników. Niestety otrzymujemy program *de facto* niepoprawny i testy do zadania, jeśli są odpowiednio dobrane, mogą to wykazać. Implementacja tego rozwiązania znajduje się w pliku `pocb1.cpp`.

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

Przedstawione przed chwilą rozwiązanie „ryzykowne", pomimo że szybkie i łatwe w implementacji, nie może oczywiście być rozwiązaniem wzorcowym. Możemy jednak spróbować je zmodyfikować, przypisując słowom-pociągom etykiety w sposób różnowartościowy. Jeśli zadbamy, by etykiety były krótkie i łatwe do modyfikacji przy zamianie wagonów, to będziemy mogli szybko porównywać pociągi w czasie symulacji.

Rozpatrzmy pociąg reprezentowany przez ciąg liter $s_0 = c_1 c_2 \ldots c_l$. By uzyskać etykietę słowa, będziemy wielokrotnie skracać tę reprezentację, za każdym razem zastępując pary liter pojedynczymi znakami. Do tego celu potrzebna nam będzie funkcja różnowartościowa

$$F : Id \times Id \to Id,$$

gdzie Id jest zbiorem identyfikatorów zawierającym alfabet małych liter angielskich (a raczej przypisanych im kodów $0, 1, \ldots, 25$). Teraz możemy pogrupować litery słowa w pary:

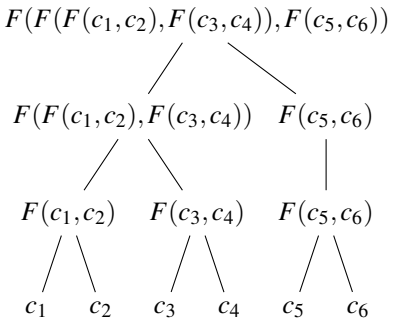
$(c_1,c_2),(c_3,c_4),\ldots$ i zastąpić słowo l-literowe przez słowo złożone z $\lceil l/2 \rceil$ liter (jeśli słowo wyjściowe było nieparzystej długości, to ostatnią literę pozostawiamy niezmienioną):

$$s_1 \;=\; F(c_1,c_2)F(c_3,c_4)\ldots.$$

Powyższą metodę możemy następnie zastosować do słowa s_1, otrzymując s_2, potem do s_2 itd., aż uzyskamy reprezentację s_k o długości 1 (oczywiście $k = O(\log l)$ i wszystkie słowa s_i są nad alfabetem Id). Z różnowartościowości F wynika od razu, że pociągom opisywanym różnymi słowami przypiszemy w ten sposób różne etykiety. Wstępne wyznaczenie etykiety dla jednego pociągu o długości l wymaga wyliczenia

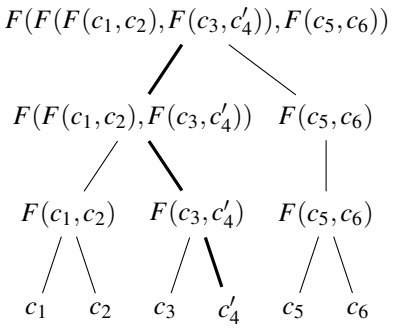
$$\left\lceil \frac{l}{2} \right\rceil + \left\lceil \frac{l}{4} \right\rceil + \ldots = O(l)$$

razy wartości funkcji F.



Rys. 1: Sposób wyznaczania etykiety dla słowa $s_0 = c_1c_2\ldots c_6$ za pomocą funkcji F.

Następnie, po zamianie pojedynczego wagonu w pociągu, wystarczy zaktualizować po jednej literze w każdym ze słów $s_0, s_1, \ldots, s_k$ — to wymaga zaledwie $O(\log l)$ wywołań F.



Rys. 2: Sposób aktualizacji słów s_i po zmianie czwartej litery wyjściowego słowa.

W całym algorytmie będziemy musieli zatem wyliczyć wartość F łącznie

$$O(nl + m\log l) = O(f)$$

razy (patrz wzór (1)).

Pozostaje zatem ostatnie pytanie — w jaki sposób zdefiniować funkcję F. Jedynym wymaganiem, jakie sobie postawiliśmy, jest jej różnowartościowość. W tej sytuacji najprostszym rozwiązaniem jest definiowanie wartości F dla kolejnych par argumentów, które się pojawiają, jako kolejnych liczb naturalnych. Możemy to zrobić, przechowując w strukturze słownikowej wszystkie napotkane dotychczas pary wraz z przypisanymi im wartościami funkcji F. Niech M oznacza maksymalną wartość funkcji przechowywaną w danym momencie w słowniku. Gdy pojawi się kolejna para (c,d), wówczas szukamy jej w słowniku. Jeśli tam się znajduje, to znamy wartość $F(c,d)$. W przeciwnym przypadku nadajemy wartość funkcji $F(c,d) = M$, dodajemy parę (c,d) wraz z wyznaczoną wartością do słownika i zwiększamy M o jeden. Słownik możemy zaimplementować w postaci zrównoważonego drzewa poszukiwań binarnych[3]. Czas wyszukiwania i wstawania elementów wynosi wówczas $O(\log r)$, gdzie r to liczba elementów w słowniku[4]. Maksymalna wielkość słownika w trakcie algorytmu jest rzędu takiego samego, jak liczba wywołań funkcji F, czyli $O(f)$. Stąd złożoność czasowa algorytmu wynosi

$$O(f \log f + n \cdot l + n \cdot m) = O((nl + m \log l) \log(nl + m \log l) + n \cdot m),$$

gdzie pierwszy składnik to sumaryczny czas odwołań do słownika w celu wyznaczenia funkcji F, drugi — $O(n \cdot l)$ — to czas policzenia wyników dla pociągów w początkowym ustawieniu, wreszcie trzeci — $O(n \cdot m)$ — to sumaryczny czas wszystkich wykonywanych porównań etykiet zmienianych pociągów i aktualizacji wyników. Zauważmy, że $n \cdot l = O(f)$, więc drugi składnik nie pojawia się w ostatecznym wzorze na złożoność.

Rozwiązanie wzorcowe o złożoności czasowej $O(f \log f)$

Czy można uzyskać jeszcze szybszy algorytm? Nie, jeśli będziemy nadal trzymać się pomysłu z bezpośrednią symulacją. Wymaga ona $n \cdot m$ operacji, ponieważ w każdym kroku liczba pociągów podobnych do zmienianych może być równa nawet n, a dla wszystkich tych pociągów musimy przynajmniej uaktualnić wynik. Ale jeśli przestaniemy traktować każdy pociąg „indywidualnie" i przyjrzymy się grupom jednakowych pociągów, to pojawia się szansa na przyspieszenie obliczeń.

Załóżmy więc, że wszystkie pociągi występujące w trakcie symulacji podzielimy na *grupy* jednakowych pociągów, które będziemy identyfikować etykietami pociągów-reprezentantów, obliczonymi tak, jak w poprzednim rozdziale. Po przestawieniu dwóch wagonów musimy zmienić przynależność do grupy (co najwyżej) dwóch zmienionych pociągów. Jeśli prześledzimy, do jakich grup należał określony pociąg w trakcie całej symulacji, to wynikiem dla niego będzie oczywiście moc najliczniejszej spośród tych grup. Trzeba więc znaleźć sposób, by śledzić, jak zmieniają się moce grup w czasie, i dodatkowo, a może przede wszystkim, wykonywać na nich operacje dodawania oraz usuwania elementów-pociągów. Zamiast dotychczasowego prostego podejścia do symulacji, odrębnie prześledzimy zmiany zachodzące w grupach i odrębnie przeanalizujemy historię poszczególnych pociągów.

[3] Najbardziej znanymi przykładami drzew zrównoważonych są drzewa AVL, które są opisane w [17], i drzewa czerwono-czarne, o których można przeczytać w [20]. W języku C++ można także skorzystać z gotowej implementacji, a mianowicie z kontenera `set` z biblioteki STL.

[4] Można także zastosować … funkcję haszującą. Jednak tym razem ewentualne „błędy" przełożyłyby się jedynie na dłuższy czas wyszukiwania i/lub wstawiania elementów do słownika. Oczekiwany czas tych operacji to jednak $O(1)$. O zastosowaniu funkcji haszujących do implementacji słownika także można przeczytać więcej w [20].

Pierwszy przebieg symulacji — śledzimy całe grupy. Moc grupy będziemy śledzić, zapisując na specjalnej liście wszystkie wykonane dla niej akcje dodania/usunięcia pociągu. Dla grupy g taką listę oznaczymy przez L_g. Dodanie nowego pociągu w kroku t zapiszemy w postaci pary $(t,1)$, natomiast usunięcie pociągu w kroku t — w postaci pary $(t,-1)$. Na podstawie listy L_g możemy łatwo wyznaczyć moc grupy g w dowolnym momencie t (oznaczmy ją $f(g,t)$) — jest ona równa sumie drugich składowych par $(t_i,x_i) \in L_g$, dla których $t_i \leqslant t$. Aby efektywnie wyznaczać wartości $f(g,t)$, trzeba zadbać, by lista

$$L_g = ((t_1,x_1),(t_2,x_2),(t_3,x_3),\ldots)$$

była posortowana rosnąco według pierwszej składowej, a następnie zastąpić drugie składowe elementów ich sumami częściowymi[5]. W ten sposób otrzymujemy ciąg par

$$M_g = ((t_1,f(g,t_1)),(t_2,f(g,t_2)),\ldots).$$

Ponieważ w trakcie śledzenia historii określonego pociągu podczas symulacji interesuje nas maksymalna moc grupy g w okresie, gdy pociąg ten do niej należał, musimy jeszcze umieć sprawnie odpowiadać na pytanie o maksymalną moc grupy g w zadanym przedziale czasowym. Do tego celu możemy wykorzystać *drzewo przedziałowe*. Jest to zrównoważone, statyczne drzewo binarne, zawierające w liściach pary $(t_i,f(g,t_i))$, a w węzłach wewnętrznych minimum i maksimum z pierwszego elementu par oraz maksimum z drugiego elementu par z poddrzewa tego węzła. Taka implementacja pozwala wyznaczać maksymalną moc grupy w zadanym przedziale czasowym w czasie logarytmicznym ze względu na liczbę operacji dotyczących danej grupy.

Drugi przebieg symulacji — śledzimy pociągi. Jedyne, co pozostaje zrobić, to przeanalizować „historię" każdego pociągu w czasie symulacji. Przyjmijmy, że zarówno pociągi, jak i grupy będziemy oznaczać etykietami obliczonymi w ten sam sposób, co w poprzednim rozdziale. Wówczas wstępne wyznaczenie etykiety pociągu wymaga $O(l)$ wywołań funkcji F, jej aktualizacja po zmianie jednego wagonu — $O(\log l)$ wywołań F, a porównanie dwóch pociągów lub grup — czasu $O(1)$. Na początku odnotowujemy, że każdy pociąg od chwili 0 należy do swojej grupy. Rozważmy teraz przestawienie wagonów pomiędzy pociągami p_1 i p_2 w chwili t_x. Jeśli wiemy, że p_1 przed przestawieniem należał do grupy g_1 i trafił do niej w momencie t_1 (i było to ostatnie „zdarzenie" w historii tego pociągu), to należy wyznaczyć maksymalną wartość $f(g_1,t)$ w okresie $[t_1,t_x]$ i wykorzystać ją do uaktualnienia wyniku dla p_1. Analogicznie postępujemy dla p_2. Następnie odnotowujemy dla p_1 i p_2, że od tej chwili należą do nowych grup.

Postarajmy się przeanalizować złożoność czasową przedstawionego rozwiązania. Wyznaczenie etykiet wszystkich pociągów wymaga czasu $O(f \log f)$. W trakcie pierwszej symulacji stworzymy listy L_g akcji wykonywanych na poszczególnych grupach. Początkowe listy grup (odpowiadające chwili 0) tworzymy w czasie $O(n+f)$. Następnie symulujemy kolejne przestawienia wagonów i przypisujemy kolejne akcje do poszczególnych grup — złożoność

[5] Dla ciągu (a_n) ciąg jego sum częściowych to (s_n), gdzie $s_n = \sum_{i=1}^{n} a_i$.

czasowa tego kroku to $O(m)$. Gotowe listy L_g mają sumaryczną długość $O(n+m)$ — przekształcenie ich w listy z sumami częściowymi, a następnie w drzewa przedziałowe zajmie więc dokładnie tyle samo czasu.

Mając gotowe drzewa przedziałowe dla grup, możemy przystąpić do drugiego przebiegu symulacji. W trakcie tego przebiegu ponownie śledzimy wszystkie zdarzenia, ale tym razem w każdym kroku aktualizujemy wynik i grupę dla zmienionych pociągów. Ze względu na konieczność wykonywania zapytań w drzewach przedziałowych wymaga to czasu $O((n+m)\log(n+m))$ — składnik $n\log(n+m)$ w tym zapisie wziął się stąd, że na końcu symulacji dla każdego pociągu musimy uwzględnić ostatnią grupę, w której się znalazł.

Sumaryczna złożoność całego rozwiązania wynosi więc

$$T(n) \quad = \quad O(f\log f + n + f + m + (n+m)\log(n+m)),$$

czyli

$$\begin{aligned} T(n) &= O((nl + m\log l)\log(nl + m\log l) + n + f + m + (n+m)\log(n+m)) \\ &= O((nl + m\log l)\log(nl + m\log l)) \\ &= O(f\log f). \end{aligned}$$

Widzimy więc, że nowe podejście do symulacji pozwoliło nam zmniejszyć najbardziej kłopotliwy składnik $n \cdot m$ do $(n+m)\log(n+m)$, wskutek czego „ukrył się" w składniku wynikającym z wyliczania etykiet. Rozwiązanie wzorcowe zostało zaimplementowane w plikach `poc0.cpp` i `poc3.java`.

Rozwiązanie alternatywne

Zauważmy, że usprawnienie, które doprowadziło nas do rozwiązania wzorcowego, mogłoby zostać równie dobrze zastosowane do rozwiązania używającego haszowanie — zamiast etykiet pojawiłyby się w nim po prostu funkcje haszujące pociągów. W ten sposób otrzymuje się efektywne, lecz niestety niepewne rozwiązanie — jedynie w przypadku wykorzystania stosunkowo dużej liczby pierwszej P (np. rzędu 10^{16}) istniała duża szansa na zdobycie przez nie maksymalnej punktacji. Sposób poradzenia sobie z tym, że dla tak dużej wartości P liczba możliwych etykiet jest stosunkowo duża, oraz oszacowanie złożoności tego rozwiązania pozostawiamy Czytelnikowi jako ćwiczenie. Pomóc w tym mogą jego implementacje zawarte w plikach `poc.pas`, `poc1.cpp` i `poc2.cpp`.

Testy

Testy do zadania zostały wygenerowane losowo, przy użyciu czterech niezależnych metod. Każda z nich została skonstruowana w celu wychwycenia nieprawidłowych bądź zbyt wolnych rozwiązań różnego rodzaju.

- **s** — generator testów, których celem jest wykrywanie programów o nieefektywnie zaimplementowanej fazie aktualizacji wyników dla pociągów. W testach tego typu występuje wiele identycznych pociągów, przez co rozwiązania nieefektywnie aktualizujące wyniki po każdej zamianie wagonów muszą wykonywać wiele operacji.

- **h** — generator testów, których zadaniem jest wykrycie programów wykorzystujących metodę haszowania. Testy tego typu zawierają wiele podobnie wyglądających pociągów, różniących się kolorem tylko kilku wagonów. Istnieje szansa, że programy wykorzystujące metodę haszowania sklasyfikują podobnie wyglądające pociągi jako takie same (oczywiście bardzo dużo zależy od jakości funkcji haszującej, jakiej zawodnik użył w swoim programie).
- **l** — generator testów losowych. Wybiera on kolejno po dwa pociągi i stara się doprowadzić jeden z nich do postaci drugiego, wykorzystując w tym celu wagony ze specjalnie przygotowanego zbioru pociągów zapasowych.
- **p** — generator testów, których zadaniem jest wykrycie drobnych błędów w programach. W testach tych pociągi różnią się na co najwyżej kilku wagonach i wykonywanych jest bardzo dużo zamian właśnie tych wagonów. Odpowiedzi dla różnych pociągów są stosunkowo zróżnicowane, przez co większość błędnych programów daje na nich złe odpowiedzi.

Poniższa lista zawiera podstawowe informacje dotyczące testów. Przez n została oznaczona liczba pociągów, przez l — ich długość, przez m — liczba wykonywanych zamian, natomiast przez r — rodzaj testu (**s**, **h**, **l** lub **p**).

Nazwa	n	l	m	*r*
poc1 .in	10	10	100	*l*
poc2 .in	10	10	1 000	*l*
poc3a.in	110	10	304	*s*
poc3b.in	100	10	300	*l*
poc3c.in	100	10	300	*p*
poc4a.in	100	10	1 204	*s*
poc4b.in	100	10	3 000	*l*
poc4c.in	100	10	3 000	*p*
poc5a.in	990	100	1 009	*s*
poc5b.in	1 000	100	1 000	*l*
poc5c.in	1 000	100	1 000	*l*
poc5d.in	1 000	100	1 000	*p*
poc6a.in	110	100	30 006	*s*
poc6b.in	100	100	9 996	*h*
poc6c.in	1 000	100	5 000	*l*
poc6d.in	100	100	30 000	*p*

Nazwa	n	l	m	*r*
poc7a.in	995	100	10 004	*s*
poc7b.in	1 000	100	9 996	*h*
poc7c.in	1 000	100	10 000	*l*
poc7d.in	100	100	10 000	*p*
poc8a.in	960	10	83 409	*s*
poc8b.in	920	10	72 996	*h*
poc8c.in	990	10	82 000	*l*
poc8d.in	980	10	90 000	*p*
poc9a.in	990	10	80 009	*s*
poc9b.in	100	90	75 000	*h*
poc9c.in	100	90	83 000	*l*
poc9d.in	100	100	80 000	*p*
poc10a.in	1 000	100	99 988	*s*
poc10b.in	1 000	100	99 996	*h*
poc10c.in	1 000	100	100 000	*l*
poc10d.in	1 000	100	100 000	*p*

Wojciech Rytter
Treść zadania, Opracowanie

Jakub Radoszewski
Program

OI, Etap II, dzień drugi, 8.02.2008

Mafia

Ostatnio Bajtocją Równikową wstrząsają mafijne porachunki. Do Bajtogrodu, stolicy tego kraju, zjechali się przywódcy mafijni, aby zakończyć spory. W pewnym momencie rozmowy stały się bardzo nerwowe i uczestnicy spotkania wyciągnęli broń. Każdy z nich trzyma w ręku pistolet i celuje w jednego innego uczestnika. Jeśli dojdzie do strzelaniny, to zgodnie z kodeksem honorowym:

- *uczestnicy będą strzelać w pewnej kolejności, przy czym jednocześnie może strzelać tylko jeden z nich,*
- *żaden z nich nie pudłuje, a ofiara natychmiast umiera i nie może strzelać,*
- *każdy z nich strzeli dokładnie jeden raz, o ile wcześniej sam nie zostanie zastrzelony,*
- *żaden uczestnik nie może zmienić wybranego na początku celu, nawet jeżeli osoba, do której celuje, zostanie zastrzelona (w takim wypadku strzał nie zmienia liczby ofiar).*

Przypadkowo sytuację obserwuje przedsiębiorca pogrzebowy, zaprzyjaźniony ze wszystkimi obecnymi. Chce on oszacować możliwą liczbę ofiar: minimalną i maksymalną. Przedsiębiorca pogrzebowy wie, kto w kogo celuje, ale nie zna kolejności strzelania. Twoim zadaniem jest napisanie programu, który te liczby wyliczy.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis celów, jakie obrali sobie poszczególni mafiozi,*
- *wyznaczy minimalną i maksymalną liczbę ofiar strzelaniny,*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu wejścia zapisana jest liczba uczestników n ($1 \leqslant n \leqslant 1\,000\,000$). Są oni ponumerowani od 1 do n. W drugim wierszu znajduje się n liczb całkowitych $s_1, s_2, \ldots, s_n$ ($1 \leqslant s_i \leqslant n$), pooddzielanych pojedynczymi odstępami. Liczba s_i jest numerem uczestnika, w którego celuje uczestnik nr i. Może się zdarzyć, że $s_i = i$.

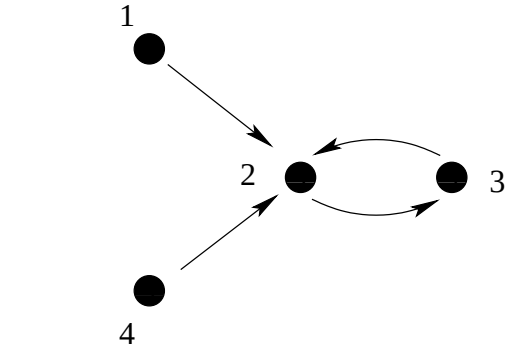
Wyjście

Twój program powinien wypisać w pierwszym i jedynym wierszu wyjścia dwie liczby całkowite, oddzielone pojedynczym odstępem: minimalną i maksymalną liczbę ofiar, będących końcowym wynikiem strzelaniny.

Przykład

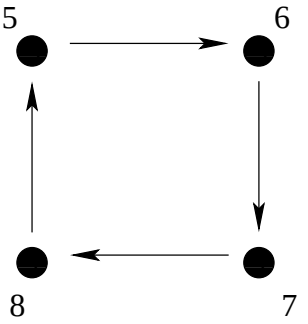
Dla danych wejściowych:

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



poprawnym wynikiem jest:

```
3 5
```



Rozwiązanie

Strzelanina w języku grafów

Zacznijmy od zapisania zadania w języku teorii grafów. Po pierwsze pozwoli nam to jaśniej przedstawić problem i skorzystać z rozwiązań znanych w tej dziedzinie. Po drugie uwolnimy się w ten sposób trochę od dość ponurej treści zadania. Niech $G = (V, E)$ będzie grafem skierowanym, w którym wierzchołki V odpowiadają uczestnikom spotkania, a krawędź $(i, j) \in E$ oznacza, że uczestnik i celuje w uczestnika j. Dla krawędzi (i, j) będziemy mówić, że:

- i wskazuje na j albo
- i jest *synem* j, a j jest *ojcem* i, co zapiszemy $parent(i) = j$.

Poszukiwane rozwiązanie zadania to dwa zbiory wierzchołków: o minimalnej oraz maksymalnej liczności, takie że... No właśnie, jeszcze dokładnie nie wiadomo, co to za zbiory — powrócimy do tej sprawy później.

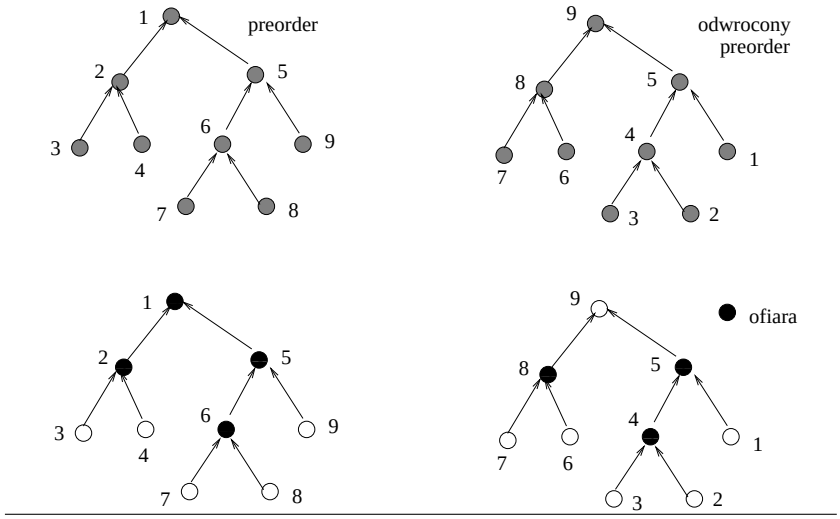
W grafie G każdy wierzchołek ma stopień wyjściowy równy 1, co istotnie wpływa na jego postać. Rozważmy jedną słabo spójną składową G (czyli spójną składową w wersji grafu G, w której ignorujemy skierowanie krawędzi). Jest ona drzewem z jedną dodatkową krawędzią, która łączy korzeń (*root*) z pewnym wierzchołkiem wewnątrz drzewa. Każdą składową możemy zająć się oddzielnie, więc od teraz przyjmiemy, że $G = T = (V, E)$ jest drzewem z dodatkową krawędzią $(root, parent(root))$, którą nazwiemy *złą krawędzią*. Załóżmy też, że graf T ma co najmniej dwa wierzchołki (w przeciwnym przypadku rozwiązanie obu części zadania jest dla niego trywialne).

Niech $|V| = n$. Usuńmy na chwilę złą krawędź z T i ponumerujmy węzły w porządkach *preorder* i *postorder*. Każda z tych numeracji przypisuje węzłom różne wartości ze zbioru

$\{1,2,\ldots n\}$ tak, by dla każdego węzła $i \neq root$ spełnione były warunki:

$$preorder(parent(i)) < preorder(i), \tag{1}$$
$$postorder(parent(i)) > postorder(i). \tag{2}$$

Jest wiele funkcji, które spełniają powyższe wymagania — w szczególności numeracja odwrotna do *preorder* jest pewnym porządkiem *postorder* i odwrotnie (na rys. 1 są pokazane przykładowe numeracje).



Rys. 1: Rozważmy graf powstały ze słabo spójnej składowej przez usunięcie złej krawędzi. W górnym rzędzie, po lewej stronie jest przedstawiona numeracja *preorder*, a po prawej — numeracja *postorder*. W dolnym rzędzie zaznaczono efekty strzelaniny: zaczernione węzły to ofiary, gdyby uczestnicy strzelali odpowiednio w porządku *preorder* i *postorder*.

Minimalna liczba ofiar. *Zbiór niezależny* węzłów grafu definiujemy jako zbiór $N \subseteq V$, taki że

$$i \in N \quad \Rightarrow \quad parent(i) \notin N. \tag{3}$$

Co ciekawe, alternatywna definicja zbioru niezależnego:

$$parent(i) \in N \quad \Rightarrow \quad i \notin N$$

jest równoważna definicji (3).

Można łatwo spostrzec, że zbiór węzłów, które przetrwają strzelaninę, musi być zbiorem niezależnym — żaden z „ocalałych" węzłów nie może celować do innego „ocalałego". Niestety, nie dla każdego zbioru niezależnego da się zaplanować taki przebieg strzelaniny, by ocalały dokładnie wierzchołki z tego zbioru — na przykład pojedynczy wierzchołek jest zbiorem niezależnym, a rzadko kiedy uda się trafić wszystkich oprócz niego. Jednak jeśli mamy maksymalny w sensie zawierania zbiór niezależny zawierający wszystkie liście T (czyli wierzchołki o stopniu wejściowym 0), to taki przebieg już istnieje.

Fakt 1. *Jeśli N jest maksymalnym (w sensie zawierania) zbiorem niezależnym zawierającym wszystkie liście w grafie T, to istnieje taki przebieg strzelaniny, w wyniku którego ocaleją dokładnie węzły ze zbioru N.*

Dowód: Wystarczy zaplanować przebieg strzelaniny tak, by najpierw trafić wszystkie wierzchołki celujące w jakiekolwiek wierzchołki ze zbioru N (oznaczmy zbiór tych wierzchołków „zagrażających" N przez C). Zauważmy, że każdy wierzchołek ze zbioru C może zostać trafiony, gdyż:

- nie mogą to być liście;
- stąd w każdy węzeł $v \in C$ celuje jakiś węzeł $w \in V \setminus C$; oczywiście w nie celuje wówczas w żaden wierzchołek należący do N.

Pozwólmy więc w pierwszej fazie strzelić wszystkim, którzy zlikwidują wierzchołki zagrażające elementom zbioru N. Teraz już zbiór N jest bezpieczny i dalszy (właściwie dowolny) przebieg strzelaniny zawsze pozostawi zbiór niezależny zawierający w sobie zbiór N. Z założenia o maksymalności N wynika, że musi to być dokładnie zbiór N. ■

Na mocy powyższego faktu możemy stwierdzić, że istnieje w T kolejność strzelania, w wyniku której przeżyją wszystkie wierzchołki z pewnego *najliczniejszego* zbioru niezależnego zawierającego wszystkie liście. Łatwo zauważyć, że jest to maksymalna liczba ocalałych, jaką da się uzyskać w T.

W przykładzie przedstawionym na rys. 1 bliski optymalnemu efekt uzyskujemy, „strzelając" w porządku *postorder* (z dokładnością do usuniętej złej krawędzi). Nie jest to przypadek — wykażemy potem, że właśnie porządek postorder pozwala łatwo wyznaczyć najliczniejszy zbiór niezależny.

Maksymalna liczba ofiar. Poszukując maksymalnej liczby ofiar, nie będziemy musieli uciekać się do żadnych zaawansowanych pojęć. Wystarczy zauważyć, że „strzelając" w pewnym porządku preorder[1], można zmaksymalizować liczbę ofiar — da się trafić wszystkich oprócz liści (do których nikt nie celuje, więc nikt ich nie trafi) i być może jednego wierzchołka (nie byłoby tego warunku, gdyby nie zła krawędź).

Algorytm MAX-MIN

Oznaczmy przez *liczba-ofiar*(π) liczbę ofiar, gdy uczestnicy strzelają w kolejności π. Rozważmy porządek postorder wyznaczony w drzewie, tj. w grafie T z pominiętą złą krawędzią. Oznaczmy przez $postorder(V)$ ciąg węzłów posortowany rosnąco względem wartości postorder.

```
1: ALGORYTM MAX-MIN(T)
2:     { Graf T jest pojedynczym drzewem o n węzłach z jedną złą krawędzią. }
3:     { Wynikiem są dwie szukane liczby: }
4:     {   MAX — maksymalna liczba ofiar, }
```

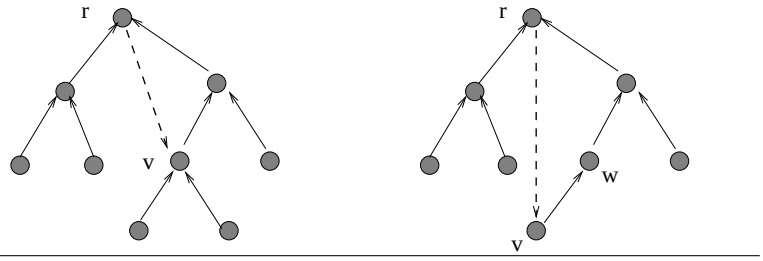
[1] tj. w dowolnym porządku preorder dla pewnego ukorzenienia T — dokładny opis doboru odpowiedniego korzenia znajduje się w dalszej części tekstu.

```
5:  { MIN — minimalna liczba ofiar. }
6:  k := liczba liści w grafie T;
7:  MAX := min(n − k, n − 1);
8:  π := postorder(V);
9:  π′ := cykliczna rotacja π;
10: { π′ tworzymy z π, przesuwając ostatni element na początek. }
11: MIN := min(liczba-ofiar(π), liczba-ofiar(π′));
12: return (MAX, MIN);
```

Wartość MAX jest poprawna

Dowód poprawności tej części algorytmu jest stosunkowo łatwy. Jeśli T jest pojedynczym cyklem zawierającym $n > 1$ węzłów (przypomnijmy, że przypadek $n = 1$ odrzuciliśmy w początkowej analizie), to maksymalna liczba ofiar wynosi $n - 1$. Załóżmy zatem, że T jest nietrywialnym drzewem z cyklem, tzn. ma $k > 0$ liści. Niewątpliwie każdą strzelaninę „przeżyją" wszystkie liście, ponieważ nikt do nich nie celuje.

Zauważmy, że każdy z pozostałych węzłów może zostać trafiony. Faktycznie, gdyby w T nie było złej krawędzi, to w wyniku strzelaniny przeprowadzonej w porządku preorder przy życiu pozostałyby jedynie liście (a dokładniej, liście w podgrafie T niezawierającym złej krawędzi). Zła krawędź $(root, parent(root))$ może zmodyfikować końcowy efekt strzelania — *root* strzela jako pierwszy, eliminuje wierzchołek $parent(root)$, przez co $parent(parent(root))$ może ocaleć. Zauważmy jednak, że wszystko będzie dobrze, jeśli węzeł $parent(parent(root))$ ma jakiegoś syna oprócz $parent(root)$ — wówczas w trakcie strzelaniny w porządku preorder $parent(parent(root))$ zostanie trafiony mimo tego, że $parent(root)$ ginie już w pierwszym strzale (patrz także rys. 2). Jeżeli poprzedni warunek nie zachodzi, to na cyklu zaczynającym się w korzeniu drzewa możemy znaleźć jakiś inny węzeł w taki, że $parent(w)$ ma jeszcze jednego syna $w' \neq w$ (gdyż T jest nietrywialnym drzewem z cyklem). Ponieważ każdy wierzchołek z cyklu może zostać uznany za korzeń drzewa, możemy wybrać także poprzednika w na tym cyklu. Zauważmy, że wówczas po strzelaninie zgodnie z porządkiem preorder pozostaną nietrafione tylko liście (patrz rys. 2).



Rys. 2: Na rysunku po lewej porządek preorder wystarczy, by trafić wszystkie węzły oprócz liści. Na rysunku po prawej najpierw musimy inaczej wybrać korzeń drzewa — powinien nim być wierzchołek, którego ojciec ma „brata" (tzn. ojciec ojca korzenia powinien mieć drugiego syna). Jako korzeń wybieramy zatem poprzednika w na cyklu, czyli w tym przypadku v — wówczas $w = parent(v)$ nie jest „jedynakiem" i „strzelamy" w porządku preorder wyznaczonym w drzewie bez złej krawędzi (v, w).

Wartość MIN jest poprawna

Ta część algorytmu jest mniej trywialna. Możemy jednak wykazać jej poprawność, bazując na prostszym przypadku drzewa bez złej krawędzi. Oznaczmy, jak poprzednio, przez $T = (V, E)$ drzewo ze złą krawędzią (r, p). Ustaliliśmy już, że zbiór „pozostałych przy życiu wierzchołków" musi być najliczniejszym zbiorem niezależnym zawierającym wszystkie liście w grafie.

Najliczniejszy zbiór niezależny w drzewie. Wyrzućmy na moment z grafu T złą krawędź (r, p); niech $T' = T \setminus \{(r, p)\}$. Rozważmy następujący algorytm zachłanny.

Algorytm A**:** Konstruuj zbiór niezależny w T', wybierając do niego za każdym razem najgłębiej położony niewybrany jeszcze wierzchołek, którego żaden z synów nie znajduje się jeszcze w zbiorze.

Okazuje się, że:

Fakt 2. *Algorytm A jest optymalny dla T', to znaczy konstruuje najliczniejszy zbiór niezależny w T'.*

Dowód: Niech N oznacza zbiór niezależny skonstruowany przez A, natomiast N' — dowolny najliczniejszy zbiór niezależny w T'. Pokażemy, że jeżeli $N \neq N'$, to można bez zmniejszenia liczności N' sprowadzić ten zbiór do N, a zatem że N jest również najliczniejszy.

Niech v będzie najgłębiej położonym wierzchołkiem drzewa, który odróżnia N i N', czyli najgłębiej położonym wierzchołkiem ze zbioru $N \oplus N' = (N \setminus N') \cup (N' \setminus N)$. Na mocy kryterium wyboru v, żaden z synów v nie należy ani do N, ani do N'. Ze względu na metodę zachłanną wykorzystywaną w algorytmie A, mamy zatem $v \in N$, czyli $v \notin N'$.

Gdyby ojciec w wierzchołka v nie należał do N' (albo, tym bardziej, gdyby w nie istniał), to $N' \cup \{v\}$ byłby zbiorem niezależnym, co przeczyłoby temu, że N' jest najliczniejszym zbiorem niezależnym. Zamieńmy zatem N' na $N'' = (N' \setminus \{w\}) \cup \{v\}$. Otrzymamy wówczas inny najliczniejszy zbiór niezależny, który ma o jeden więcej wierzchołek wspólny z N (gdyż oczywiście $w \notin N$). Kontynuując ten proces, po maksymalnie $|N|$ krokach bez usuwania wierzchołków otrzymamy zbiór N. To kończy dowód optymalności A. ■

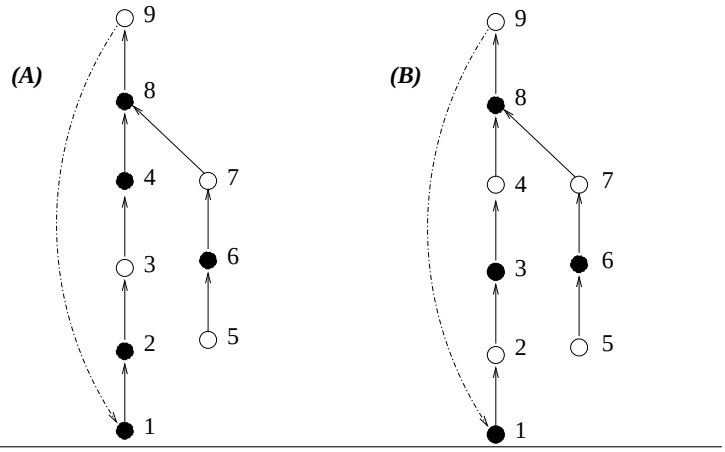
Jak praktycznie wykorzystać algorytm A? Otóż można zauważyć, że *porządek postorder* wyznacza kolejność strzelania, w wyniku której przy życiu pozostaną wierzchołki z dokładnie tego zbioru niezależnego, który jest konstruowany przez *algorytm A* (dowód tego faktu pozostawiamy Czytelnikowi jako ćwiczenie). Związek z algorytmem A w szczególności implikuje jedną istotną własność strzelania w porządku postorder, która przyda nam się w dalszej analizie.

Obserwacja 1. Jeżeli w T' istnieje najliczniejszy zbiór niezależny niezawierający korzenia, to zostanie on skonstruowany w wyniku strzelaniny w porządku postorder.

Najliczniejszy zbiór niezależny w drzewie ze złą krawędzią. Powróćmy teraz do drzewa T ze złą krawędzią (r, p). Zauważmy, że wyznaczony dla niego zbiór niezależny musi być także niezależny ze względu na krawędzie drzewa. Możemy więc rozpocząć od wyznaczenia najliczniejszego zbioru niezależnego N' w drzewie $T \setminus \{(r, p)\}$. Daje to nam jeden z dwóch przypadków:

- do zbioru N' nie należą jednocześnie węzły r i p, więc dołączenie do drzewa złej krawędzi nie psuje niezależności zbioru — to oznacza, że zbiór N' jest niezależny i optymalny także w grafie ze złą krawędzią;
- do zbioru N' należą jednocześnie r i p, więc w drzewie ze złą krawędzią zbiór N' nie jest już zbiorem niezależnym — moglibyśmy wyrzucić jeden z końców krawędzi (r,p), ale nie możemy być do końca pewni, czy dostajemy w ten sposób rozwiązanie optymalne.

Przyjrzyjmy się dokładniej drugiemu spośród powyższych przypadków, w którym obecność p oraz r w N' wymusza na nas wyrzucenie jednego z tych wierzchołków. Zauważmy, że drzewo może mieć kilka różnych najliczniejszych zbiorów niezależnych, więc warto sprawdzić, czy nie można się obejść bez wierzchołka p lub r. Na mocy Obserwacji 1, jeżeli zbiór niezależny N' skonstruowany za pomocą porządku postorder zawiera r, to każdy inny najliczniejszy zbiór niezależny w drzewie także zawiera r. Może jednak istnieć możliwość obejścia się bez p — oznaczmy najliczniejszy zbiór niezależny dla grafu T z usuniętym wierzchołkiem p przez N''. Porównując zbiory N'' i N' z ewentualnie usuniętym jednym z węzłów r, p, dostajemy zatem zbiór N — najliczniejszy zbiór niezależny dla drzewa za złą krawędzią.



Rys. 3: Węzły zaznaczone na czarno oznaczają węzły trafione (A): w porządku postorder $\pi = 1,2,3,4,5,6,7,8,9$ (5 trafień) oraz (B): w porządku $\pi' = 9,1,2,3,4,5,6,7,8$ (4 trafienia).

Skoro już wiemy, jak wyznaczyć najliczniejszy zbiór niezależny, to sprawdźmy, czy robi to algorytm MAX-MIN. Zauważmy, że:

- jeśli węzły strzelają w „zwykłym" porządku postorder π, to jako wynik dostajemy liczność zbioru niezależnego N', ewentualnie z usuniętym węzłem p, jeśli krawędź (r,p) psuła niezależność zbioru N';
- jeśli węzły strzelają w przesuniętym cyklicznie porządku postorder, czyli π', to wyznaczamy optymalny zbiór niezależny w grafie z wstępnie usuniętym węzłem p (r trafia go na samym początku) — jest to zbiór N''.

Widzimy więc, że algorytm bada wszystkie konieczne przypadki i zwraca optymalną z możliwych wartości, czyli jest poprawny!

Na rysunku 3 jest pokazany przypadek, gdy optymalny jest właśnie porządek π'.

Alternatywne nieformalne uzasadnienie dla wersji MIN. Jeżeli Czytelnikowi nie przypadł do gustu dowód poprawności wersji MIN wykorzystujący zbiory niezależne, to na koniec prezentujemy w skrócie alternatywne uzasadnienie tej części algorytmu MAX-MIN.

Zauważmy, że graf T o korzeniu r składa się z cyklu

$$C = r \to v_1 \to \ldots \to v_k \to r$$

i pewnego zbioru D drzew podwieszonych do C. Oznaczmy przez

$$A_{C1} : \; v_1 \to \ldots \to v_k \to r, \;\; A_{C2} : \; r \to v_1 \to \ldots \to v_k$$

dwie interesujące kolejności strzelania na cyklu C oraz niech A_D będzie kolejnością strzelania w drzewach ze zbioru D zgodną z porządkiem postorder.

Obserwacja 2. Algorytm MAX-MIN daje ten sam zbiór ocalałych, co algorytm minimalizujący liczbę ofiar z dwóch kolejności: $(A_D; A_{C1})$, $(A_D; A_{C2})$.

Krótkie uzasadnienie. Zauważmy, że pierwsza z tych kolejności odpowiada π, natomiast druga *prawie* odpowiada π' (gdzie π, π' — jak w algorytmie MAX-MIN). Rozpatrzmy dwa przypadki. Jeśli po wykonaniu A_D wierzchołek r jest martwy, to kolejność $(A_D; A_{C1})$ minimalizuje liczbę ofiar. W tym przypadku jest to po prostu kolejność postorder, optymalna dla drzewa $T \setminus \{(r,p)\}$, a r nie wykonuje strzału, więc zła krawędź niczego nie psuje. W przeciwnym przypadku π' odpowiada pod względem zbioru ofiar kolejności $(A_D; A_{C2})$, tzn. strzał wykonywany w π' przez r możemy przesunąć po A_D.

Obserwacja 3. Istnieje optymalna kolejność strzelania B, w której strzały z A_D padają na samym początku, tj. $B = (A_D; B_C)$.

Wynika to stąd, że każda strategia, w której wybrany liść strzela jako pierwszy, może być zawsze rozszerzona do algorytmu optymalnego. Można to udowodnić indukcyjnie ze względu na liczbę liści.

Wystarczy teraz stwierdzić następujący fakt.

Obserwacja 4. Jeden z ciągów A_{C1}, A_{C2} jest optymalny dla T będącego cyklem, w którym pewne wierzchołki mogą już być martwe, tzn. jeden z ciągów A_{C1}, A_{C2} jest równoważny ciągowi B_C w pewnym algorytmie optymalnym B.

Wynika to stąd, że w takim cyklu dowolny ciąg kolejnych (wzdłuż cyklu) strzelań zaczynający się od martwego wierzchołka (albo jakikolwiek taki ciąg, jeżeli nie istnieją martwe wierzchołki) jest optymalny. W ciągach A_{C1}, A_{C2} strzelanie zaczynamy z sąsiednich wierzchołków v_1 i r, dzięki czemu co najmniej raz otrzymamy ten sam zbiór ofiar, co przy strzelaniu optymalnym.

Złożoność algorytmu

Algorytm wymaga jedynie przejścia grafu w porządku postorder. Jest to prosta operacja wykonywana w czasie $O(n)$, czyli bez problemu radzimy sobie z danymi o zadanym rozmiarze.

Testy

Rozwiązania zawodników były sprawdzane na 13 zestawach testów. W poniższej tabelce znajdują się ich krótkie opisy.

Nazwa	n	Opis
maf1.in	9	mały losowy test
maf2.in	1 000	zróżnicowany średni test losowy (7 cykli, 21 pętli, 74 drzewa z cyklami)
maf3.in	2 000	zróżnicowany średni test losowy (10 cykli, 30 pętli, 169 drzew z cyklami)
maf4.in	3 000	zróżnicowany średni test losowy (17 cykli, 61 pętli, 336 drzew z cyklami)
maf5.in	3 500	zróżnicowany średni test losowy (19 cykli, 120 pętli, 462 drzewa z cyklami)
maf6.in	4 000	zróżnicowany średni test losowy (22 cykle, 97 pętli, 421 drzew z cyklami)
maf7.in	60 006	test złożony z 10 000 6-wierzchołkowych składowych z małym bonusem
maf8.in	10 000	test losowy z dodanymi cyklami (w tym jednowierzchołkowymi)
maf9.in	100 000	test losowy z dodanymi ponad 7 000 wierzchołków na cyklach oraz z 1 200 8-wierzchołkowymi składowymi
maf10.in	400 000	test losowy z dodanymi cyklami, zawierającymi łącznie 150 000 wierzchołków
maf11a.in	1 000 000	test losowy z dodanymi ponad 10 000 9-wierzchołkowych składowych oraz 120 000 wierzchołków leżących na cyklach
maf11b.in	1 000 000	test, w którym kilka wierzchołków ma bardzo duże stopnie wejściowe
maf12a.in	1 000 000	test losowy
maf12b.in	1 000 000	ścieżka długości milion dochodząca do pętli
maf13a.in	1 000 000	test losowy z dodanymi 10 000 10-wierzchołkowych składowych oraz 100 000 wierzchołków na cyklach
maf13b.in	1 000 000	cykl długości milion

W trakcie generowania testów wyszedł na jaw empiryczny fakt, że losowy graf (tj. graf, któremu odpowiada wejście w postaci losowego n-elementowego ciągu nad $\{1,2,\ldots,n\}$) składa się z małej liczby słabo spójnych składowych, które zazwyczaj są nietrywialnymi drzewami z cyklami (i oczywiście pewne z nich muszą być stosunkowo duże). Z tego względu najtrudniejsze testy w zestawie poszerzały testy losowe o słabo spójne składowe będące cyklami (jedno- i wielowierzchołkowymi) oraz o wiele bardzo małych losowych spójnych składowych — celem tych drugich było wykrycie rozmaitych rozwiązań błędnych.

Marian M. Kędzierski	Marian M. Kędzierski, Filip Wolski	Filip Wolski
Treść zadania	Opracowanie	Program

OI, Etap II, dzień drugi, 8.02.2008

Ucieczka

Al Bajtone, znany na całym świecie złodziej, planuje napad na bank. Wie, że jak tylko obrabuje bank, rozpocznie się policyjny pościg. Al Bajtone jest niestety kiepskim kierowcą i skręcanie w lewo sprawia mu kłopoty. Chce więc tak zaplanować ucieczkę, aby przez każde skrzyżowanie przejeżdżać na wprost lub skręcać w prawo. Wie też, że jeżeli przez jakieś skrzyżowanie raz przejedzie, to policja będzie tam już na niego czyhać. Nie może więc dwa razy przejeżdżać przez to samo skrzyżowanie. Dodatkowo, na niektórych skrzyżowaniach zawsze stoją patrole policji, więc takie skrzyżowania musi omijać (na skrzyżowaniach, przy których znajdują się bank i kryjówka, nie ma patroli policji).

Al Bajtone planuje trasę ucieczki z banku do swojej kryjówki. Nieoczekiwanie złożył Ci wizytę i przedstawił „propozycję nie do odrzucenia": musisz policzyć, ile jest różnych tras ucieczki prowadzących z banku do kryjówki, spełniających podane warunki.

Ulice Bajtogrodu tworzą regularną prostokątną siatkę. Wszystkie ulice prowadzą w kierunku północ-południe lub wschód-zachód, a na przecięciu każdych dwóch ulic znajduje się skrzyżowanie. Bank znajduje się na południe od skrzyżowania wysuniętego najbardziej na południowy zachód. Al Bajtone rozpocznie ucieczkę w kierunku północnym.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia położenie kryjówki, opis skrzyżowań, przy których stoją patrole policji, oraz pewną dodatnią liczbę całkowitą k,*
- *wyznaczy liczbę wszystkich tras ucieczki z banku do kryjówki, które spełniają podane warunki,*
- *wypisze na standardowe wyjście resztę z dzielenia wyniku przez k.*

Wejście

W pierwszym wierszu standardowego wejścia znajdują się trzy liczby całkowite n, m i k ($1 \leqslant n,m \leqslant 100$, $1 \leqslant k \leqslant 10^9$). Liczby n i m to odpowiednio liczby ulic, które prowadzą w kierunku wschód-zachód oraz północ-południe. W drugim wierszu znajdują się dwie liczby całkowite x i y ($1 \leqslant x \leqslant m$, $1 \leqslant y \leqslant n$). Reprezentują one położenie kryjówki — znajduje się ona przy skrzyżowaniu x-tej ulicy prowadzącej w kierunku północ-południe i y-tej ulicy prowadzącej w kierunku wschód-zachód. Ulice są numerowane z zachodu na wschód i z północy na południe, odpowiednio od 1 do m i od 1 do n.

W każdym z następnych n wierszy znajduje się m znaków „$*$" i/lub „$+$". Jest to mapa miasta. Znak w i-tym wierszu i j-tej kolumnie tej mapy określa skrzyżowanie i-tej ulicy prowadzącej z zachodu na wschód z j-tą ulicą prowadzącą z północy na południe, przy czym „$*$" oznacza, że na skrzyżowaniu stoi patrol policji, a „$+$" oznacza, że przez skrzyżowanie może prowadzić trasa ucieczki.

Al Bajtone rozpoczyna ucieczkę, wjeżdżając na skrzyżowanie o współrzędnych $(1,n)$ z kierunku południowego, tj. z nieistniejącego skrzyżowania $(1,n+1)$.

Wyjście

Twój program powinien wypisać w pierwszym i jedynym wierszu resztę z dzielenia liczby wszystkich możliwych tras ucieczki przez k.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`3 5 10` `4 2` `+++++` `++*++` `++++*`	`2`

Rozwiązanie

Wprowadzenie

Na II etapie II Olimpiady Informatycznej pojawiło się zadanie „Klub Prawoskrętnych Kierowców". W zadaniu tym należało stwierdzić dla wczytanej prostokątnej mapy pól (spośród których niektóre są zajęte, zaś inne wolne), jaka jest najkrótsza trasa z pewnego pola A do pewnego innego pola B, pod warunkiem, że nie dopuszczamy skrętów w lewo oraz jazdy przez zajęte pola.

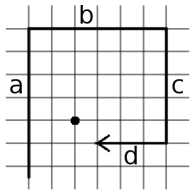
W naturalny sposób można zapytać o liczbę możliwych „prawoskrętnych" tras na takiej mapie. To pytanie stało się inspiracją do stworzenia tego zadania.

Rozwiązanie prawie wzorcowe — jeszcze nieoptymalne

Kluczowym warunkiem w zadaniu jest założenie, że zliczamy tylko trasy bez samoprzecięć. Sprawia to, że dozwolone są jedynie trasy szczególnej postaci.

Podzielmy drogę Ala Bajtone na części, z których każda jest prostym odcinkiem pomiędzy dwoma skrętami w prawo. Oznaczmy cztery pierwsze odcinki literami: *a*, *b*, *c* i *d* (gdy będzie nam wygodnie, to takimi samymi literami będziemy oznaczać także ulice, na których leżą odcinki). Przykład drogi z podziałem na odcinki jest przedstawiony na rys. 1.

Zauważmy, że skoro *a* zaczyna się w południowo-zachodnim końcu miasta, a *d* jest na pewno na południe od *b*, to przedłużenie odcinka *d* przecina odcinek *a*. W takim razie Al Bajtone musi zjechać z ulicy *d* (w prawo, oczywiście) przed dojechaniem do ulicy *a*. Oznaczmy kolejny odcinek trasy Ala Bajtone literą *e*. Łatwo zauważyć, że biegnie on w tym samym kierunku, co odcinek *a*, i całkowicie zawiera się w prostokącie ograniczonym ulicami *a*, *b*, *c* i *d*. Równie łatwo można dowieść metodą indukcji, że jeśli literami *u*, *v*, *w*, *x*, *y* i *z* oznaczymy sześć kolejnych (niekoniecznie początkowych) odcinków trasy Ala Bajtone, to:



Rys. 1: Początek trasy Ala Bajtone.

- ostatni odcinek z szóstki (z) jest w całości zawarty w prostokącie ograniczonym ulicami v, w, x i y;
- pierwszy odcinek szóstki (u) leży w całości poza prostokątem ograniczonym ulicami v, w, x i y.

Trasa ucieczki Ala Bajtone ma więc kształt spirali skręcającej w prawo. Ponadto ucieczka kończy się w kryjówce, stąd do powyższych trzeba dodać jeszcze jeden warunek:

- kryjówka musi należeć do każdego z prostokątów, utworzonych przez cztery kolejne odcinki ucieczki.

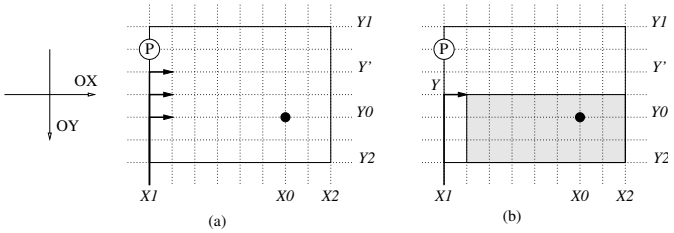
Z powyższych własności widać, że trasę ucieczki Ala Bajtone można przedstawić jako prosty odcinek zakończony skrętem i następnie trasę całkowicie zawartą w mniejszym prostokącie. To oznacza, że problem można spróbować rozwiązać techniką programowania dynamicznego — wyznaczając liczbę możliwych tras dla wszystkich prostokątów mapy zawierających kryjówkę Ala Bajtone. Prostokąty trzeba przy tym rozważać w takiej kolejności, aby wynik dla każdego prostokąta można było łatwo wyznaczyć na podstawie wyników dla prostokątów wcześniej przeanalizowanych.

Weźmy więc dowolny prostokąt $P = [x_1, x_2] \times [y_1, y_2]$, ograniczony ulicami x_1 i x_2 (pionowymi) oraz y_1 i y_2 (poziomymi). Załóżmy, że kryjówka Ala Bajtone leży na tym obszarze w punkcie (x_0, y_0), czyli $x_1 \leqslant x_0 \leqslant x_2$ i $y_1 \leqslant y_0 \leqslant y_2$. Przypomnijmy, że osie układu współrzędnych są zgodnie z treścią zadania skierowane odpowiednio na wschód i na południe. Chcemy obliczyć liczbę tras spełniających warunki zadania, wchodzących do P od południa w lewym dolnym rogu, czyli w punkcie (x_1, y_2).

Możliwe są dwa przypadki.

1. Kryjówka leży na ulicy x_1, czyli $x_0 = x_1$. Wtedy Al Bajtone może do niej dojechać tylko jedną drogą i to tylko wówczas, gdy na odcinku od skrzyżowania (x_1, y_2) do skrzyżowania (x_1, y_0) nie ma żadnych patroli policyjnych.

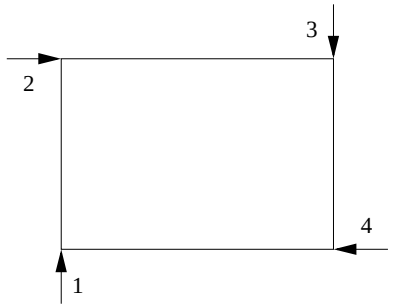
2. Kryjówka znajduje się na wschód od ulicy x_1, czyli $x_0 > x_1$. Wtedy Al Bajtone musi wykonać przynajmniej jeden skręt w prawo. Ponieważ zakładamy, że będzie poruszał się tylko w prostokącie P, to może skręcić na wschód w ulicę y dla $y_1 \leqslant y \leqslant y_2$. Dodatkowo $y' \leqslant y \leqslant y_0$, gdzie $y' - 1$ jest najdalej wysuniętym na południe skrzyżowaniem na ulicy x_1 pomiędzy ulicami y_1 i y_2, na którym stoi policja; jeżeli na odcinku od (x_1, y_1) do (x_1, y_2) nie ma policji, to przyjmujemy zamiast tego $y' = y_1$ (patrz też rys. 2a).



Rys. 2: Możliwości ruchu Ala Bajtone w prostokącie P (a) i podprostokąt P_y, w którym odbywa się dalsza ucieczka, po wyborze określonego y (b).

W pierwszym przypadku mamy rozwiązanie, równe zero lub jeden, gdy tylko sprawdzimy obecność patroli policyjnych na odcinku od (x_1, y_0) do (x_1, y_2). Poszukując rozwiązania w nietrywialnym, drugim przypadku, możemy przeglądać wartości $y = y_2, y_2 - 1, \ldots, y_1$, dopóki nie napotkamy pierwszego patrolu — na tej podstawie wyznaczamy y'. Dla wartości y z przedziału $[y', y_0]$ sumujemy wyniki dla prostokątów $P_y = [x_1 + 1, x_2] \times [y, y_2]$, zakładając, że wjeżdżamy do takiego prostokąta od strony zachodniej w lewym górnym rogu $(x_1 + 1, y)$ (patrz także rys. 2b).

W tym momencie widzimy, że informacja o liczbie tras wchodzących do prostokąta w lewym dolnym rogu od południa nie jest wystarczająca do obliczeń dynamicznych. Potrzebne są także wyniki w przypadku, gdy Al Bajtone wjeżdża przez lewy górny narożnik od zachodu. Powtarzając poprzednie rozumowanie dla takich przypadków, zauważymy szybko, że trzeba rozważyć także przypadki, gdy Al Bajtone wjeżdża w prostokątny obszar w prawym górnym rogu od północy i w prawym dolnym rogu od wschodu. Przypadki te oznaczymy: (1), (2), (3) i (4) — są one przedstawione na rys. 3.



Rys. 3: Konieczne do rozważenia sposoby rozpoczęcia trasy wewnątrz prostokąta.

Prosta implementacja przedstawionego pomysłu przebiega następująco. Rozpoczynamy od prostokąta $[x_0, x_0] \times [y_0, y_0]$ składającego się jedynie ze skrzyżowania, przy którym jest kryjówka — liczba tras we wszystkich czterech przypadkach jest równa 1. Następnie, dla kolejnych prostokątów o coraz większym polu (można je przeglądać, na przykład, zwiększając w każdym etapie jeden z wymiarów prostokąta o jeden) obliczamy każdy z czterech wyników. W pesymistycznym przypadku obliczenia dla jednego prostokąta wymagają przejrzenia wszystkich skrzyżowań znajdujących się na jego obwodzie.

Ponieważ na mapie mamy w sumie $O(n^2 \cdot m^2)$ prostokątów (dlaczego?) i dla każdego z nich należy rozważyć $O(n + m)$ potencjalnych miejsc skrętu, to złożoność czasowa

zaproponowanego rozwiązania wynosi $O(n^2m^2(n+m))$, jest więc zbyt duża jak na ograniczenia z zadania. Również konieczność pamiętania $O(n^2m^2)$ wyników dla wszystkich prostokątów przekracza dostępne zasoby pamięci.

Rozwiązanie wzorcowe — po poprawkach

Przedstawione rozwiązanie możemy jednak poprawić. Uważny Czytelnik dostrzeże bowiem, że nie ma konieczności sumowania $O(n+m)$ wyników dla mniejszych prostokątów, aby otrzymać wynik dla większego. Powróćmy do prostokąta $P=[x_1,x_2]\times[y_1,y_2]$ i obliczania dla niego wyniku w przypadku (1) — w poprzednim rozwiązaniu wymagało to dodawania wielu wyników typu (2) dla prostokątów postaci $[x_1+1,x_2]\times[y,y_2]$ dla $y' \leqslant y \leqslant y_0$. Zauważmy jednak, że wszystkie rozważane tu trasy możemy podzielić na dwie grupy:

- skręcające w prawo przed dojechaniem do ulicy y_1;
- skręcające w prawo na skrzyżowaniu (x_1,y_1).

Liczba tras z pierwszej grupy to w rzeczywistości liczba tras typu (1) dla prostokąta $[x_1,x_2]\times[y_1+1,y_2]$. Natomiast liczba tras z drugiej grupy to liczba tras typu (2) dla prostokąta $[x_1+1,x_2]\times[y_1,y_2]$, o ile tylko cały odcinek od (x_1,y_1) do (x_1,y_2) jest wolny od policji. Analogicznie można uprościć obliczenia we wszystkich czterech przypadkach — za każdym razem wynik dla prostokąta i jego wybranego rogu będzie wyznaczany w czasie stałym. Pozostaje więc tylko wymyślić metodę rozpoznawania w czasie stałym, czy dowolny odcinek jest wolny od policji, i otrzymamy rozwiązanie zadania działające w czasie $O(n^2m^2)$.

Wszystkie „bezpieczne" dla Ala Bajtone odcinki ulic możemy wyznaczyć na początku działania algorytmu, ponownie stosując programowanie dynamiczne. Wystarczy, że dla każdego punktu na mapie obliczymy, jak daleko na południe i na wschód ciągnie się bezpieczna strefa. Wyznaczając wartość (na przykład w kierunku wschodnim) dla skrzyżowania (x,y), wystarczy skorzystać z wartości dla skrzyżowania $(x+1,y)$. Łatwo zauważyć, że wszystkie wartości można wyliczyć w czasie $O(nm)$ i zapisać w pamięci rozmiaru $O(nm)$. Natomiast odpowiadając na pytanie, czy odcinek od (x_1,y_1) do (x_2,y_1), dla $x_1 \leqslant x_2$, jest bezpieczny, wystarczy porównać rozmiar bezpiecznej strefy w kierunku wschodnim dla skrzyżowania (x_1,y_1) z wartością x_2-x_1+1.

Uzyskaliśmy już satysfakcjonującą złożoność czasową. Niestety, złożoność pamięciowa całego algorytmu nadal jest rzędu $O(n^2m^2)$, co przekracza limit pamięciowy w zadaniu, który wynosi 64 MB. Problem stanowi tu duża liczba prostokątów, dla których zapisujemy wyniki częściowe. Oczywiście ostatecznie interesuje nas tylko wynik typu (1) dla prostokąta $[1,m]\times[1,n]$, ale wcześniej wyliczamy i tablicujemy wszystkie wartości typu (1)–(4) dla prostokątów $[x_1,x_2]\times[y_1,y_2]$, gdzie $1 \leqslant x_1 \leqslant x_2 \leqslant m$ i $1 \leqslant y_1 \leqslant y_2 \leqslant n$. Aby zredukować liczbę pamiętanych wartości, można obliczać wyniki dla prostokątów w takiej kolejności, by w każdym momencie korzystać tylko ze stosunkowo niewielu wyników wcześniejszych. Wówczas wszystkie wyniki starsze będzie można zapomnieć i do obliczeń wystarczy nam mniejsza pamięć. Znalezienie odpowiedniej kolejności przetwarzania prostokątów wymaga pewnej pomysłowości, gdyż nie jest ona najbardziej typowa.

Zauważmy, że obliczając liczbę tras (na przykład typu (1)) dla prostokąta $P=[x_1,x_2]\times[y_1,y_2]$, korzystamy z wyników dla prostokątów $[x_1,x_2]\times[y_1+1,y_2]$

i $[x_1+1, x_2] \times [y_1, y_2]$. Są to co prawda prostokąty o polach znacznie mniejszych niż badany, ale każdy z nich ma obwód mniejszy dokładnie o 2 od prostokąta P! Wystarczy więc pogrupować prostokąty ze względu na obwód i wyznaczać wyniki kolejno dla grup $G_0, G_2, G_4, \ldots, G_{2(n+m)}$, gdzie G_k to wszystkie prostokąty o obwodzie k. Widać, że obliczenia dla grupy G_k wymagają odwołania się jedynie do wartości z grupy G_{k-2} i każda grupa ma najwyżej $O((n+m)nm)$ elementów, co mieści się już w limicie pamięciowym dla zadania. Jako ćwiczenie pozostawiamy Czytelnikowi opracowanie efektywnej metody wyznaczenia i poindeksowania wszystkich prostokątów należących do ustalonej grupy G_k.

Opisane rozwiązanie wzorcowe zostało zaimplementowane w plikach `uci.cpp`, `uci1.pas` oraz `uci2.java`.

Rozwiązania nieoptymalne

Oprócz rozwiązania wzorcowego istnieją również algorytmy o gorszej złożoności.

Jednym z nich jest opisany jako pierwszy algorytm o złożoności czasowej $O(n^2m^2(n+m))$, który został zaimplementowany w plikach `ucib2.cpp`, `ucib3.pas` oraz `ucib4.java`. Przechodzi on testy 1–10.

Inne rozwiązanie to algorytm siłowy, czyli bezpośrednio wyszukujący wszystkie możliwe ścieżki ucieczki Ala Bajtone. Został on zaimplementowany w plikach `ucis3.cpp`, `ucis4.pas`, `ucis5.java`. Przechodzi on testy 1–7.

Testy

Rozwiązania zawodników były sprawdzane na następujących zestawach testowych:

Nazwa	n	m	Opis
uci1.in	5	5	mały test poprawnościowy z jednym posterunkiem policji
uci2a.in	5	5	mały test poprawnościowy z jedną trasą ucieczki
uci2b.in	5	5	mały test poprawnościowy z jedną trasą ucieczki
uci2c.in	5	5	mały test poprawnościowy bez trasy ucieczki
uci3a.in	10	10	średni test poprawnościowy — miasto bez policji
uci3b.in	10	10	średni test poprawnościowy — miasto z jednym patrolem policji
uci4.in	10	10	średni test poprawnościowy — kilka patroli, kryjówka blisko południowo-zachodniego krańca
uci5.in	10	10	średni test poprawnościowy — kryjówka blisko centrum
uci6.in	20	20	średni test poprawnościowo-wydajnościowy — dużo policji w mieście, kryjówka tuż obok banku
uci7a.in	20	20	średni test poprawnościowo-wydajnościowy
uci7b.in	20	10	średni test poprawnościowy — miasto puste

Nazwa	n	m	Opis
uci8.in	40	40	średni test poprawnościowo-wydajnościowy
uci9.in	40	40	średni test poprawnościowo-wydajnościowy
uci10.in	40	40	średni test poprawnościowo-wydajnościowy
uci11.in	80	80	duży test wydajnościowy
uci12.in	80	80	duży test wydajnościowy
uci13.in	100	100	duży test wydajnościowy
uci14.in	100	100	duży test wydajnościowy
uci15.in	100	100	duży test wydajnościowy
uci16.in	100	100	duży test wydajnościowy
uci17.in	100	100	maksymalny możliwy test, z maksymalną możliwą odpowiedzią

Zawody III stopnia

opracowania zadań

Michał Pilipczuk
Treść zadania, Opracowanie

Jakub Radoszewski
Program

OI, Etap III, dzień próbny, 01.04.2008

Lampki

Mały Jaś dostał na święta nietypowy prezent. Po odpakowaniu kolorowego kartonu jego oczom ukazał się napis „Nieskończony łańcuch lampek choinkowych". Zainteresowany chłopiec od razu rozłożył nową zabawkę na podłodze.

Łańcuch Jasia ma formę kabla posiadającego początek, ale nieposiadającego końca. Do kabla są podłączone lampki ponumerowane (zgodnie z kolejnością na kablu) kolejnymi liczbami naturalnymi, poczynając od 0. Kabel podłączony jest do panelu sterowania. Na panelu znajduje się pewna liczba przycisków, każdy w innym kolorze i przy każdym znajduje się inna dodatnia liczba naturalna. Liczby umieszczone przy różnych przyciskach są parami względnie pierwsze.

W momencie rozpakowania prezentu żadna lampka się nie świeciła. Jaś, nie myśląc wiele, przycisnął po kolei wszystkie przyciski na panelu sterującym, od pierwszego aż do ostatniego. Ze zdumieniem zaobserwował, że naciśnięcie i-tego przycisku powoduje zapalenie się wszystkich lampek o numerach podzielnych przez liczbę p_i znajdującą się przy tym przycisku. Co więcej, wszystkie te lampki zaczynają świecić kolorem k_i, takim jak kolor przycisku. W szczególności, wszystkie lampki o numerach podzielnych przez p_i, które były już uprzednio zapalone, zmieniają swój kolor na kolor k_i.

Teraz Jaś patrzy urzeczony na nieskończony wielobarwny łańcuch i zastanawia się, jaka część lampek pali się poszczególnymi kolorami. Oznaczmy przez $L_{i,r}$ liczbę lampek palących się na kolor k_i, spośród lampek o numerach $0, 1, \ldots, r$. Formalnie, ułamek C_i lampek, które palą się kolorem k_i, to:

$$C_i = \lim_{r \to \infty} \frac{L_{i,r}}{r}$$

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opisy przycisków na panelu sterowania,*
- *dla każdego koloru k_i obliczy ułamek C_i, mówiący jaka część lampek pali się kolorem k_i,*
- *wypisze wyniki na standardowe wyjście.*

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę całkowitą n ($1 \leqslant n \leqslant 1\,000$), oznaczającą liczbę przycisków znajdujących się na panelu sterowania. Kolejne n wierszy zawiera po jednej liczbie całkowitej p_i ($1 \leqslant p_i \leqslant 1\,000\,000\,000$), oznaczającej, że przyciśnięcie i-tego przycisku powoduje zapalenie się na kolor k_i lampek o numerach podzielnych przez p_i. Liczby p_i są podane w kolejności naciskania przycisków przez Jasia. Liczby p_i są parami względnie pierwsze i różne.

Wyjście

Twój program powinien wypisać na wyjście dokładnie n wierszy. W i-tym z nich powinien się znaleźć ułamek C_i, mówiący jaka część lampek pali się kolorem k_i, zapisany w formie nieskracalnego ułamka a/b, gdzie a jest całkowite, b całkowite dodatnie oraz a i b są względnie pierwsze. Jeśli $C_i = 0$, to ułamek ten powinien być wypisany jako $0/1$.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`3` `2` `3` `5`	`4/15` `4/15` `1/5`

Rozwiązanie

Wprowadzenie

Rozwiązanie zadania składa się z dwóch części. Pierwszą z nich jest analiza problemu od strony teoretycznej, w celu wyprowadzenia wzoru na średnie liczby lampek poszczególnych kolorów, które będą świecić się po naciśnięciu wszystkich przycisków. Potem należy zastanowić się, jak efektywnie obliczać wartości wyznaczonej formuły tak, by zmieścić się w limitach czasowych nawet dla największych danych.

Analiza teoretyczna

Dla uproszczenia opisu przyjmiemy, że lampka zgaszona też ma pewien kolor, a dokładniej, wszystkie zgaszone lampki są tego samego koloru. Ponadto będziemy stosować oznaczenie $a \perp b$, gdy liczby a i b są względnie pierwsze.

Pozbądźmy się nieskończoności

Oznaczmy

$$\Pi = p_1 p_2 \ldots p_n.$$

Zauważmy, że kolory lampek w ciągu powtarzają się z okresem Π. Istotnie, dla każdego $q \in \{0, 1, 2, \ldots, \Pi - 1\}$, lampki q, $q + \Pi$, $q + 2\Pi$ itd. są tego samego koloru, gdyż ich numery dzielą się przez te same spośród czynników $p_1, p_2, \ldots, p_n$, jako że Π dzieli się przez nie wszystkie. Stąd ułamek C_i, oznaczający jak w treści zadania część lampek świecących na końcu i-tym kolorem, jest równy stosunkowi liczby takich lampek o numerach od 0 do $\Pi - 1$ do liczby Π. Intuicyjnie jest to oczywiste — jeśli nieskończenie wiele razy powtarzamy ten sam cykl, to lampki w danym kolorze występują w nieskończonym ciągu średnio z taką samą częstotliwością, z jaką występują w cyklu.

Aby formalnie potwierdzić nasze podejrzenia, oznaczmy przez c_i liczbę lampek w kolorze k_i spośród lampek o numerach od 0 do $\Pi-1$. Spróbujmy oszacować $L_{i,r-1}$. Jeśli $r=a\Pi+b$, gdzie $0\leqslant b<\Pi$, to $ac_i\leqslant L_{i,r-1}\leqslant(a+1)c_i$, gdyż lampek w rozważanym kolorze jest co najmniej tyle, ile w a powtórzeniach cyklu, a nie więcej niż w $a+1$ powtórzeniach. Stąd mamy, że $\frac{a}{r}c_i\leqslant\frac{L_{i,r-1}}{r}\leqslant\frac{a+1}{r}c_i$, czyli

$$\frac{1}{\Pi+\frac{b}{a}}c_i=\frac{a}{a\Pi+b}c_i\leqslant\frac{L_{i,r-1}}{r}\leqslant\frac{a+1}{a\Pi+b}c_i=\frac{1+\frac{1}{a}}{\Pi+\frac{b}{a}}c_i.$$

Wraz z $r\to\infty$ również $a\to\infty$, zaś b pozostaje ograniczone, stąd zarówno górne, jak i dolne ograniczenie ułamka $\frac{L_{i,r-1}}{r}$ zbiega do $\frac{c_i}{\Pi}$. Zatem:

$$\lim_{r\to\infty}\frac{L_{i,r}}{r}=\lim_{r\to\infty}\frac{r+1}{r}\frac{L_{i,r}}{r+1}=\lim_{r\to\infty}\frac{L_{i,r}}{r+1}=\lim_{r\to\infty}\frac{L_{i,r-1}}{r}=\frac{c_i}{\Pi}.$$

Sprowadziliśmy zatem problem do rozważania jedynie Π początkowych lampek. Zobaczmy teraz, które spośród nich zapalają się po naciśnięciu odpowiednich przycisków na panelu. Ze względu na to, że liczy się ostateczny kolor lampki, będziemy rozważać przyciski od ostatniego do pierwszego. Po wciśnięciu n-tego przycisku kolorem k_n zaświeci co p_n-ta lampka, stąd $C_n=\frac{1}{p_n}$. Po wciśnięciu $(n-1)$-szego przycisku zaświeci się co p_{n-1}-sza lampka, ale potem niektóre z nich zmienią kolor na k_n. W ogólności, przy wyznaczaniu ułamka lampek, które ostatecznie będą świeciły kolorem k_i, musimy uwzględnić, że niektóre z nich zmienią potem swój kolor. Nie jest to jednak skomplikowane — naciśnięcie i-tego przycisku zapala bowiem lampki o numerach podzielnych przez p_i, ale tylko te z nich, których numery nie dzielą się przez żadne p_j dla $j>i$, pozostaną w tym kolorze do końca. Dokładne wyliczenie ułamka lampek C_i wymaga zastosowania pewnego prostego aparatu matematycznego.

Odrobina teorii liczb

Wpierw zauważmy, że jeżeli $a\perp b$ oraz $a|c$ i $b|c$, to również $ab|c$. Jest to jasne — skoro a i b nie mają w rozkładzie na czynniki pierwsze żadnych wspólnych czynników, zaś w rozkładzie c występują zarówno czynniki liczby a, jak i b, to wszystkie czynniki iloczynu ab także występują w rozkładzie liczby c. Analogicznie, jeśli $a_1,a_2,\ldots,a_n$ są parami względnie pierwsze oraz $a_i|c$ dla każdego i, to c jest podzielne przez $a_1a_2\ldots a_n$.

Weźmy teraz wszystkie liczby ze zbioru $\{0,1,\ldots,\Pi-1\}$ i dla każdej z nich wyliczmy ciąg reszt z dzielenia jej kolejno przez $p_1,p_2,\ldots,p_n$. Załóżmy, że dla dwóch różnych liczb $a,b\in\{0,1,\ldots,\Pi-1\}$ otrzymamy takie same ciągi reszt. Co to oznacza? Przyjmijmy bez straty ogólności, że $a\leqslant b$, i popatrzmy na liczbę $b-a$. Skoro a i b dają taką samą resztę z dzielenia przez p_i, to p_i dzieli $b-a$. Skoro zachodzi to dla każdego p_i, to Π dzieli $b-a$. Jednak $b-a$ jest liczbą ze zbioru $\{0,1,\ldots,\Pi-1\}$, więc $a=b$.

W ten sposób dochodzimy do wniosku, że każda z liczb od 0 do $\Pi-1$ ma inny ciąg reszt. Dodatkowo zauważmy, że jest dokładnie $p_1p_2\ldots p_n=\Pi$ różnych ciągów reszt, gdyż pierwszą resztę można wybrać na p_1 sposobów, drugą na p_2 sposobów itd. To oznacza, że ciągów reszt jest tyle samo, co rozważanych liczb, więc każdy ciąg jest ciągiem reszt dokładnie jednej liczby. Czytelnik, który spotkał się wcześniej z podstawami teorii liczb, z pewnością zauważył, że właśnie udowodniliśmy Chińskie twierdzenie o resztach, o którym

można też przeczytać w opracowaniu zadania *Permutacja* w niniejszej książeczce czy w książce [20].

Powróćmy teraz do problemu znalezienia liczby elementów zbioru $\{0,1,\ldots,\Pi-1\}$, które dzielą się przez p_k i nie dzielą się przez żadne p_j dla $j>k$. Dzięki udowodnionemu twierdzeniu, zamiast zliczać liczby z rozważanego zbioru, możemy zliczać ciągi reszt. Te, które nas interesują, muszą mieć następujące własności:

- reszta z dzielenia przez p_j dla $j>k$ musi być niezerowa — możemy więc wybrać ją na p_j-1 sposobów,
- reszta z dzielenia przez p_k musi być równa 0, czyli mamy tylko 1 możliwość,
- reszta z dzielenia przez p_j dla $j<k$ może być dowolna, co daje p_j możliwości.

Jest więc $c_k = p_1\cdot p_2\cdot\ldots\cdot p_{k-1}\cdot 1\cdot(p_{k+1}-1)\cdot\ldots\cdot(p_n-1)$ poszukiwanych ciągów reszt, zatem:

$$C_k=\frac{c_k}{\Pi}=\frac{p_1p_2\ldots p_{k-1}(p_{k+1}-1)\ldots(p_n-1)}{p_1p_2\ldots p_n}=\frac{(p_{k+1}-1)(p_{k+2}-1)\ldots(p_n-1)}{p_kp_{k+1}\cdots p_n}.$$

Algorytm wzorcowy

Wzór jest gotowy, ale czy to koniec zadania? Już na pierwszy rzut oka widać, że występujące w nim liczby (nawet po skróceniu liczników i mianowników) będą iloczynami około 1 000 liczb rzędu 1 000 000 000, a więc będą duże. Stąd na pewno trzeba będzie zaimplementować własną arytmetykę. Przy obliczaniu wyniku będziemy musieli uporać się z dwoma problemami:

- jak zaplanować obliczenia tak, by algorytm był odpowiednio efektywny (w sensie złożoności czasowej),
- jak zapewnić względną pierwszość liczników i mianowników.

Zastanówmy się, czy wystarczające byłoby podejście bezpośrednie — wyznaczenie wszystkich liczników i mianowników przez proste wymnożenie czynników i następnie skrócenie wyniku przez *NWD* licznika i mianownika znalezione za pomocą algorytmu Euklidesa. Powiedzmy, że występująca w obliczeniach liczba jest dla nas *duża*, jeśli potencjalnie może mieć $9n$ cyfr (zagadka dla Czytelnika: skąd wzięła się magiczna stała 9?). Liczbę uznamy za *małą*, jeśli mieści się w typie 32-bitowym ze znakiem.

Zauważmy, że dla każdego ułamka musimy wykonać:

- $O(n)$ mnożeń dużej liczby przez małą (domnażając wynik kolejno przez małe czynniki), każde domnożenie ma złożoność $O(n)$,
- wyznaczenie *NWD* dwóch dużych liczb w standardowej złożoności $O(n^2)$,
- podzielenie dwóch dużych liczb przez ich *NWD*, każde dzielenie w standardowej złożoności $O(n^2)$.

Daje to nam całkowitą złożoność $O(n^3)$, gdyż powyższe operacje wykonujemy dla wszystkich n ułamków. Nie jest to zatem rozwiązanie wystarczająco szybkie.

Pierwszy pomysł na przyspieszenie obliczeń to skorzystanie z wartości C_{k+1} w trakcie obliczania C_k. Pozwala na to następująca reguła rekurencyjna, wynikająca prosto ze wzoru na C_k:

$$C_k = \frac{p_{k+1}-1}{p_k} C_{k+1}$$

dla $k < n$ oraz $C_n = \frac{1}{p_n}$. Możemy więc obliczać szukane ułamki, poczynając od ostatnich, domnażając licznik i mianownik uzyskanego już C_{k+1} przez pewne małe wartości. W ten sposób wykonujemy w sumie $O(n)$ mnożeń dużej liczby przez małą, każdorazowo w złożoności $O(n)$. Niestety, aby skrócić wynikowe ułamki, nadal musimy stosować kosztowne obliczanie *NWD* oraz dzielenie dużych liczb.

W tym miejscu pojawia się kolejny pomysł, pozwalający jeszcze lepiej wykorzystać C_{k+1} w trakcie obliczania C_k. Jeśli mamy już C_{k+1} przedstawione jako $\frac{a_{k+1}}{b_{k+1}}$, gdzie $a_{k+1} \perp b_{k+1}$, to analogiczny ułamek reprezentujący $C_k = \frac{a_k}{b_k}$ spełnia następującą zależność:

$$\frac{a_k}{b_k} = \frac{p_{k+1}-1}{p_k} \frac{a_{k+1}}{b_{k+1}}.$$

Obliczenie go przez zwykłe wymnożenie może sprawić, że licznik i mianownik przestaną być względnie pierwsze z kilku powodów:

1. $p_{k+1}-1$ i p_k nie są względnie pierwsze,
2. $p_{k+1}-1$ i b_{k+1} nie są względnie pierwsze,
3. p_k i a_{k+1} nie są względnie pierwsze.

Warto więc przed wymnożeniem poskracać przez największe wspólne dzielniki kolejno:

1. ułamek $\frac{p_{k+1}-1}{p_k}$,
2. licznik (skrócony) powyższego ułamka i b_{k+1},
3. mianownik (skrócony) powyższego ułamka i a_{k+1}.

Wykonane operacje spowodują, że po prostym wymnożeniu skróconych liczb dostaniemy wartość C_k w formie ułamka nieskracalnego.

Powyższy algorytm wymaga skrócenia dwóch małych liczb (czyli wykonania algorytmu Euklidesa i dzielenia), co robimy w czasie stałym, oraz dwukrotnie skrócenia liczby małej i dużej, co robimy w czasie $O(n)$. W ten sposób dla każdego ułamka C_k wykonujemy operacje o złożoności $O(n)$, co prowadzi do wzorcowego algorytmu o złożoności $O(n^2)$.

Kilka słów o dużych liczbach

Opisując rozwiązanie wzorcowe, stwierdziliśmy, że potrafimy wykonać następujące operacje na dużych liczbach:

- pomnożenie dużej liczby przez małą w złożoności liniowej względem długości zapisu (liczby cyfr) dużej liczby,

- podzielenie dużej liczby przez małą w takiej samej złożoności,
- obliczenie *NWD* dużej liczby i małej w takiej samej złożoności,
- podzielenie dużej liczby przez dużą w złożoności kwadratowej od długości ich zapisu,
- obliczenie *NWD* dwóch dużych liczb w takiej samej złożoności.

Warto się chwilę zastanowić, czy rzeczywiście umiemy to zrobić. Mnożenie dużej liczby przez małą w czasie liniowym można zrealizować, naśladując szkolne mnożenie pisemne — łatwo zauważyć, że jest to algorytm liniowy, jeśli liczba, przez którą mnożymy, ma ustaloną, małą liczbę cyfr. Podobnie efektywne jest dzielenie dużej liczby przez małą metodą „pod kreską".

Oczywiście powyższe dzielenie możemy wykorzystać do obliczania reszt z dzielenia dużej liczby przez małą[1]. Z kolei umiejętność obliczania reszty przydaje się do wyznaczenia *NWD*. Jeśli a jest dużą liczbą, zaś b małą, to $NWD(a,b) = NWD(a \bmod b, b)$. Od tego momentu mamy już do czynienia tylko z małymi liczbami, gdyż $a \bmod b < b$, więc ich *NWD* liczymy w czasie stałym. Stąd *NWD* dużej i małej liczby wyznaczamy w czasie liniowym.

Czwartą operację, czyli dzielenie dużej liczby przez inną dużą liczbę, wykonujemy także podobnie do dzielenia pisemnego, które tym razem ma jednak złożoność kwadratową względem długości dzielnej.

Euklides nie zadbał o naprawdę duże liczby

Ostatnia, piąta operacja jest nieco trudniejsza. Standardowa implementacja algorytmu Euklidesa, polegająca na redukcji $NWD(a,b) = NWD(a \bmod b, b)$, może wymagać wykonania tylu kroków, ile cyfr ma krótsza liczba. W każdym kroku liczymy resztę modulo dla dwóch dużych liczb. Wykonanie tego poprzez dzielenie liczb zajmuje czas kwadratowy względem długości tych liczb. Zatem cały algorytm może działać nawet w czasie proporcjonalnym do sześcianu długości liczb.

Okazuje się jednak, że można osiągnąć złożoność kwadratową liczenia *NWD*. Trzeba się jednak przy tym wykazać sporym sprytem! Niech a i b będą dużymi liczbami, których *NWD* chcemy obliczyć. Przedstawmy a i b w systemie dwójkowym — potrafimy wykonać to w czasie kwadratowym: po prostu dzielimy liczbę a czy b wielokrotnie przez dwa, za każdym razem sprawdzając parzystość (czyli ostatni bit) wyniku i ustalając w ten sposób kolejne bity przekształcanej liczby. Następnie skorzystajmy z kilku prostych obserwacji:

1. Jeśli $b = 0$, to $NWD(a,b) = a$ (i symetrycznie w przypadku $a = 0$).
2. Jeśli a i b są parzyste, to $NWD(a,b) = 2 \cdot NWD(\frac{a}{2}, \frac{b}{2})$.
3. Jeśli a jest parzyste, a b nie, to $NWD(a,b) = NWD(\frac{a}{2}, b)$.
4. Jeśli b jest parzyste, a a nie, to $NWD(a,b) = NWD(a, \frac{b}{2})$.
5. Jeśli a i b są nieparzyste i $a > b$, to $NWD(a,b) = NWD(a-b, b)$.

[1] Nieco bardziej elegancko można to zrealizować, traktując dużą liczbę jak wielomian i wyznaczając jej wartość modulo mała za pomocą schematu Hornera.

Podobnie jak to ma miejsce w klasycznym algorytmie Euklidesa, wśród powyższych mamy własności pozwalające rekurencyjnie zmniejszać a i b (2–5) oraz przypadek brzegowy (1). Z własności 2–4 korzystamy, gdy na końcu zapisu przynajmniej jednej z liczb występuje zero. Z własności 5 — gdy mamy dwie liczby nieparzyste. Redukcje 2–4 polegają na podzieleniu przez 2 liczb a i/lub b (robimy to w czasie stałym, obcinając bit zero na końcu liczby) i, w przypadku drugim, wymnożeniu wyniku wywołania rekurencyjnego przez 2 — robimy to w czasie liniowym. W przypadku piątym potrzebujemy liniowego czasu, by obliczyć różnicę a i b. Oczywiście przy stosowaniu reguł 2-4 łączna liczba cyfr binarnych rozważanych liczb zmniejsza się co najmniej o 1. Gdy jesteśmy zmuszeni zastosować regułę 5 (która nie powoduje wzrostu liczby cyfr binarnych rozważanych liczb), to zaraz po niej wystąpi przypadek trzeci lub czwarty. Zatem najwyżej dwa ruchy są potrzebne, by zmniejszyć sumę długości liczb binarnych a i b co najmniej o 1. Stąd wynika, że algorytm składa się co najwyżej z liniowej liczby kroków. Ponieważ każdy krok potrafimy wykonać w czasie liniowym ze względu na sumę długości liczb, to uzyskany algorytm jest kwadratowy. Praktyka pokazuje, że jest on również stosunkowo prosty w implementacji, dzięki uniknięciu kłopotliwej operacji wyznaczania reszty z dzielenia dla dużych liczb.

Dobre rady na przyszłość

Skoro już jesteśmy przy dużych liczbach, to warto wspomnieć, że zapis binarny pojawia się w ich implementacji raczej rzadko. Wiele algorytmów na dużych liczbach ma złożoności kwadratowe i większe, dlatego w praktyce istotne jest zapisanie liczby w jak najkrótszej postaci (np. skrócenie zapisu c razy przekłada się na skrócenie czasu działania algorytmu kwadratowego o czynnik $\frac{1}{c^2}$). Zazwyczaj stosuje się system pozycyjny o podstawie 10^9, aby „cyfry" liczby można było zapisać za pomocą 32 bitów. Nie wolno wówczas zapomnieć, by mnożenie takich „cyfr" wykonywać na 64 bitach. Czasem, przy bardziej skomplikowanych algorytmach, używa się podstawy reprezentacji równej 10^6, aby pomieścić na 64 bitach wynik mnożenia trzech takich cyfr. Zaleca się przy tym używanie jako podstawy potęgi 10, aby uniknąć dodatkowych przekształceń systemu dziesiętnego w binarny i odwrotnie przy wczytywaniu i wypisywaniu liczb.

Zauważmy także, że w algorytmie wzorcowym zyskaliśmy sporo na czasie, zastępując operacje na dwóch długich argumentach operacjami, w których tylko jedna liczba była długa. O ile bowiem podzielenie dużej liczby przez małą jest stosunkowo proste, o tyle dzielenie dwóch dużych liczb zajmuje sporo czasu (cennego zwłaszcza na zawodach) i często jest mało efektywne. To samo podejście okazuje się skuteczne w wielu innych przypadkach.

Testy

Wszystkie testy zostały wygenerowane losowo przy pomocy dwóch parametrów — liczby lampek **n** i maksymalnej wielkości liczby przy przycisku $\mathbf{max_p}$. Poniższa tabelka zawiera listę wartości tych parametrów w dziesięciu testach użytych do sprawdzania rozwiązań.

Nazwa	n	$\mathbf{max_p}$	Opis
lam1.in	5	20	test poprawnościowy
lam2.in	5	100	test poprawnościowy

Nazwa	n	$\max_p$	Opis
lam3.in	10	200	test poprawnościowy
lam4.in	100	20 000	test poprawnościowy
lam5.in	200	20 000	test poprawnościowy
lam6.in	200	1 000 000	test wydajnościowy
lam7.in	500	1 000 000	test wydajnościowy
lam8.in	1 000	1 000 000	test wydajnościowy
lam9.in	1 000	100 000 000	test wydajnościowy
lam10.in	1 000	1 000 000 000	test wydajnościowy

Ciekawostki

Dzień próbny finału XV OI odbył się pierwszego kwietnia. Z tej okazji uczestnicy przez pierwsze pół godziny zawodów dysponowali jedynie przykładowymi wejściami i wyjściami, bez historyjki stojącej za nimi. Mimo to, jednemu zawodnikowi udało się w tym czasie odgadnąć wzór oraz wysłać rozwiązanie go obliczające, chociaż nie używające dużych liczb. Szacun.

Zbigniew Czech
Treść zadania, Opracowanie

Michał Pilipczuk
Program

OI, Etap III, dzień pierwszy, 02.04.2008

Podział Królestwa

Król Bajtocji, Bajtazar, postanowił przejść na emeryturę. Ma on dwóch synów. Nie może się jednak zdecydować, który z nich powinien być jego następcą. Postanowił więc podzielić królestwo na dwie połowy i uczynić swoich synów ich władcami.

Po podziale królestwa, na drogach łączących obie połowy należy zbudować strażnice. Ponieważ wiąże się to z kosztami, dróg łączących obie połowy powinno być możliwie najmniej.

Bajtocja składa się z parzystej liczby miast połączonych drogami. W wyniku podziału w każdej połowie powinna znaleźć się połowa miast. Każda droga łączy dwa miasta. Drogi nie łączą ani nie krzyżują się poza miastami, mogą natomiast występować wiadukty tudzież tunele. Każde dwa miasta mogą być bezpośrednio połączone co najwyżej jedną drogą.

Przy podziale królestwa istotne jest, które miasta znajdą się w której połowie. Możesz założyć, że teren poza miastami można tak podzielić, że drogi łączące miasta leżące w tej samej połowie nie będą przecinać granicy. Natomiast na każdej drodze łączącej miasta z różnych połówek trzeba wybudować jedną strażnicę.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis miast i łączących je dróg,*
- *wyznaczy taki podział królestwa na dwie połowy, że w każdej połowie będzie tyle samo miast, a liczba dróg łączących miasta leżące w różnych połowach będzie minimalna,*
- *wypisze wyznaczony wynik na standardowe wyjście.*

Jeżeli istnieje wiele poprawnych podziałów królestwa, Twój program powinien wyznaczyć którykolwiek z nich.

Wejście

W pierwszym wierszu wejścia zapisane są dwie liczby całkowite n i m oddzielone pojedynczym odstępem, równe odpowiednio liczbie miast i liczbie łączących je dróg, $2 \leqslant n \leqslant 26$, $2|n$, $0 \leqslant m \leqslant \frac{n \cdot (n-1)}{2}$. Miasta są ponumerowane od 1 do n. W kolejnych m wierszach zapisane są po dwie liczby całkowite oddzielone pojedynczym odstępem. W wierszu $(i+1)$-szym (dla $i = 1, 2, \ldots, m$) zapisane są liczby u_i i v_i, $1 \leqslant u_i < v_i \leqslant n$. Reprezentują one drogę łączącą miasta u_i i v_i.

Wyjście

Twój program powinien wypisać jeden wiersz zawierający $\frac{n}{2}$ liczb całkowitych pooddzielanych pojedynczymi odstępami. Powinny to być numery miast należących do tej połówki królestwa, do której należy miasto nr 1, podane w rosnącej kolejności.

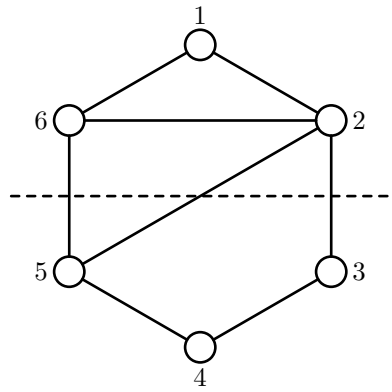
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
1 2 6
```



Na rysunku linią przerywaną zaznaczono optymalny podział, który wymaga zbudowania 3 strażnic.

Rozwiązanie

Rozwiązanie wzorcowe

Zadanie, jakie ma do rozwiązania król Bajtocji, dotyczy problemu minimalnego połowienia (bisekcji) grafu, który formułuje się następująco. Dany jest graf $G = (V, E)$, gdzie V jest zbiorem wierzchołków, a E zbiorem krawędzi. Zakładamy, że liczba miast $|V| = n$ jest parzysta. Należy podzielić zbiór wierzchołków grafu na dwa równoliczne podzbiory Y_1 i Y_2 tak, aby liczba krawędzi łączących wierzchołki podzbiorów Y_1 i Y_2 była minimalna. Problem minimalnego połowienia grafu należy do klasy NP-zupełnych. Przedstawienie programu rozwiązującego większe przykłady problemu w sposób optymalny w rozsądnym czasie jest więc niemożliwe, o ile P≠NP[1]. Jednak dla tak niewielkich wartości n, jakie występują w treści zadania, można zaryzykować rozwiązanie w czasie wykładniczym polegające na przejrzeniu wszystkich możliwych bisekcji. Oznaczmy liczbę wierzchołków grafu przez

[1]Do klasy P należą zadania, które potrafimy rozwiązać w czasie wielomianowym: $O(n^2)$, $O(n^3)$ czy nawet $O(n^{100})$. Do klasy NP należą zadania, których na razie nie potrafimy tak szybko rozwiązywać — potrafimy poradzić sobie w czasie około 2^n (przekonaj się sam, od jakiego n zaczyna się przewaga 2^n nad n^{100} i jak szybko potem się ona zwiększa). W klasie NP występują zadania, które nieformalnie można nazwać „kluczowymi", a formalnie określa się je jako NP-zupełne. Gdybyśmy dysponowali programem rozwiązującym dowolne zadanie NP-zupełne w czasie wielomianowym, to wykorzystując go jako procedurę, potrafilibyśmy w czasie wielomianowym rozwiązać każde zadanie z klasy NP i okazałoby się, że P=NP. Naukowcy skłaniają się ku podejrzeniom, że klasy P i NP są różne. Gdy więc napotkasz na problem NP-zupełny, rozsądek nakazuje wierzyć, że uda Ci się napisać program, który znajduje rozwiązanie tego problemu: dla małych przypadków w czasie wykładniczym, dla charakterystycznych przypadków w czasie wielomianowym, ewentualnie z pewnym przybliżeniem (za pomocą algorytmu aproksymacyjnego) lub gdy przypisze Ci szczęście (za pomocą algorytmu losowego). Więcej o klasach P i NP oraz problemach NP-zupełnych można przeczytać w [15] i [20] (przyp. red.).

$|V| = n = 2N$. Wówczas liczba możliwych połowień grafu, jakie należy sprawdzić, wynosi:

$$\binom{n}{n/2} = \binom{2N}{N} = \frac{(2N)!}{N!N!}.$$

Powyższą wartość można oszacować, stosując wzór Stirlinga:

$$\frac{(2N)!}{N!N!} = \frac{\sqrt{2\pi 2N}\frac{(2N)^{2N}}{e^{2N}}(1+o(1))}{2\pi N\frac{N^{2N}}{e^{2N}}(1+o(1))^2} = \frac{2^{2N}(1+o(1))}{\sqrt{\pi N}} = \Theta\left(\frac{2^{2N}}{\sqrt{N}}\right) \tag{1}$$

Poszukując jak najlepszego rozwiązania, zastosujemy przeszukiwanie wyczerpujące. Nie możemy liczyć na istotne zmniejszenie liczby przypadków do rozważenia, ale możemy starać się uzyskać jak najniższy współczynnik proporcjonalności poprzedzający czynnik wykładniczy w funkcji złożoności. Poniższy lemat pokazuje, jak w efektywny sposób generować wszystkie interesujące nas połowienia grafu.

Lemat 1. Dla dowolnych liczb naturalnych k,l istnieje ciąg (a_i), którego elementami są wszystkie słowa zerojedynkowe długości $k+l$ mające dokładnie k jedynek, gdzie każde takie słowo występuje dokładnie raz, a_1 jest dowolnym takim słowem oraz każde dwa kolejne elementy ciągu różnią się na dokładnie dwóch pozycjach.

Dowód: Przeprowadzimy dowód indukcyjny ze względu na sumę $k+l$. Najpierw zauważmy, że jeśli $k = 0$ lub $l = 0$, to teza jest oczywista — mamy po prostu jedno słowo składające się z samych zer lub samych jedynek. Załóżmy teraz, że dla wszystkich par liczb k',l' o sumie mniejszej od $k+l$ teza jest prawdziwa oraz że $k > 0$ i $l > 0$. Weźmy jako a_1 dowolne słowo ze zbioru, który mamy wygenerować. Bez straty ogólności możemy przyjąć, że ostatnim znakiem a_1 jest 1. Następnie:

- Zastosujmy założenie indukcyjne dla pary $k-1,l$, generując w pożądany sposób wszystkie słowa zerojedynkowe ciągu (a_i) zawierające na końcu jedynkę.
- Niech a będzie ostatnim wygenerowanym dotychczas słowem. Wyszukajmy pierwszy znak zero w słowie a (na pewno taki istnieje, bo $l > 0$). Utwórzmy kolejne słowo ciągu a', zamieniając miejscami w a znalezione zero i jedynkę z ostatniej pozycji.
- Następnie zastosujmy założenie indukcyjne dla pary $k,l-1$ i słowa początkowego a', uzyskując w ten sposób wszystkie szukane słowa z ostatnim znakiem równym zero.

Widzimy, że generując ciąg w podany wyżej sposób, dbamy o zachowanie wszystkich koniecznych warunków:

- każde wymagane słowo pojawia się w ciągu,
- żadne słowo nie powtarza się,
- każde dwa kolejne słowa ciągu różnią się dokładnie dwoma znakami.

■

Wniosek 1. Przedstawiony w powyższym dowodzie sposób konstrukcji ciągu (a_i) jest w istocie algorytmem rekurencyjnym, za pomocą którego można wygenerować taki ciąg w czasie $O\left((k+l)\frac{(k+l)!}{k!l!}\right)$. Drugi człon w funkcji złożoności jest liczbą wygenerowanych elementów ciągu (a_i), zaś pierwszy wynika z liniowego czasu szukania zera zamienianego z ostatnim znakiem.

Algorytm wzorcowy

1. *Wczytaj dane wejściowe i zbuduj graf o* $2N$ *wierzchołkach.*
2. *Ustal pierwszy podział, wybierając do pierwszej połówki wierzchołki* $1,2,\ldots,N$*, a resztę — do drugiej. Dla każdego wierzchołka oblicz, ilu ma sąsiadów w pierwszej połówce, a ilu w drugiej. Wyznacz także liczbę krawędzi przebiegających pomiędzy połowami.*
3. *Generuj kolejne podziały królestwa (właściwie* N*-elementowe podzbiory zbioru wierzchołków stanowiące pierwszą połowę), wykorzystując procedurę z Lematu 1. Zauważ, że dwa kolejne podziały różnią się wymianą jednej pary wierzchołków* (u,v) *pomiędzy połowami.*
   a. *Popraw liczby sąsiadów z odpowiednich połówek dla wierzchołków sąsiadujących z* u *i* v *oraz dla samych* u *i* v*.*
   b. *Uaktualnij także liczbę krawędzi pomiędzy połowami. Jeśli jest mniejsza niż dotychczasowe minimum, to zapamiętaj aktualny podział.*
4. *Wypisz jako wynik najlepszy znaleziony w kroku* 3. *podział.*

W algorytmie wykorzystaliśmy procedurę z Lematu 1, więc widać, że rozważyliśmy wszystkie możliwe podziały i wybraliśmy optymalny. Zastanówmy się jeszcze, ile czasu nam to zajęło. Dla każdego podziału aktualizacja statystyk dla wierzchołków i wyniku (liczby krawędzi łączących połówki) mogą być wykonane w czasie $O(N)$. Tyle samo czasu potrzebujemy, by przejść do kolejnego podziału. Wszystkich podziałów jest oczywiście $\frac{(2N)!}{N!N!}$. To oznacza, że mamy algorytm o złożoności czasowej $O\left(N\frac{(2N)!}{N!N!}\right)$ — ze wzoru (1) wynika, że jest to funkcja rzędu $O(\sqrt{N}2^{2N}) = O(\sqrt{|V|}2^{|V|})$.

Złożoność pamięciowa jest $O(N^2)$, gdyż największą strukturą, którą należy zapamiętać, jest reprezentacja grafu. Podziały zbioru wierzchołków mogą być zapamiętane w postaci wektora-słowa długości n, więc nie mają istotnego wpływu na złożoność pamięciową. Bez problemu mieścimy się więc nie tylko w limicie pamięciowym 64 MB, ale także w znacznie mniejszych.

W rozwiązaniu wzorcowym zostało dodane jeszcze jedno usprawnienie. Ze względu na symetrię połówek grafu, można na początku ustalić przynależność jednego wierzchołka i rozpatrywać podzbiory N-elementowe zbioru $(2N-1)$-elementowego. Nie poprawia to asymptotycznej złożoności algorytmu, ale dwukrotnie przyspiesza jego działanie.

Implementacja algorytmu wzorcowego znajduje się w plikach:

- `pod.cpp` — implementacja w języku C++,
- `pod1.pas` — implementacja w języku Pascal,
- `pod2.cpp` — implementacja w języku C++ z użyciem biblioteki STL.

Inne rozwiązania

Rozwiązania alternatywne

Algorytm 2.1. Algorytm ten jest implementacją pomysłu wzorcowego z zastosowaniem operacji na bitach do zapisu i modyfikacji podziału zbioru wierzchołków. Jeśli zapiszemy podział jako słowo maszynowe o $O(|V|)$ bitach (nazwijmy je *maską podziału*), wymianę wierzchołków i znajdowanie pierwszego zera w reprezentacji zbioru możemy wykonać za pomocą kilku operacji procesora.

W tym celu dla każdego wierzchołka zapisujemy jego stopień i maskę bitową reprezentującą jego sąsiadów (nazwijmy to słowo *maską sąsiedztwa wierzchołka*). Po przeniesieniu wierzchołka z jednej połowy do drugiej, chcielibyśmy w czasie stałym odpowiedzieć na pytanie, ilu sąsiadów z danej połówki ma przeniesiony wierzchołek. Można to zrobić, mnożąc bitowo (tj. stosując operację AND) maskę podziału przez maskę sąsiedztwa tego wierzchołka i następnie znajdując liczbę jedynek w tak otrzymanym słowie. Aby nie zliczać jedynek w słowie za każdym razem od nowa, można wcześniej, na początku algorytmu, utworzyć tablicę $L[0..2^{16}-1]$, zawierającą na pozycji i liczbę jedynek w reprezentacji binarnej liczby i (z przedziału $[0..2^{16}-1]$.) Wówczas aby obliczyć liczbę jedynek w podwójnym słowie a, wystarczy podzielić je na dwa słowa: a_1 złożone z początkowych 16 bitów i a_2 złożone z kolejnych 16 bitów, a następnie odczytać i dodać wartości $L[a_1]$ i $L[a_2]$. Przygotowanie tablicy L wymaga 2^{16} prostych operacji, a sprawdzenie liczby jedynek w każdej masce sąsiedztwa wierzchołka wykonujemy w czasie $O(1)$.

Operację przejścia do kolejnego podziału zbioru wierzchołków także realizujemy, wykorzystując operacje maszynowe i wykonując odpowiednie obliczenia wstępne. Obliczamy tablicę $P[0..2^{16}-1]$, gdzie $P[i]$ to pierwsza pozycja, na której w binarnej postaci liczby i występuje zero. Chcąc odszukać pozycję pierwszego zera w liczbie 32-bitowej, sprawdzamy wpierw, czy dolne 16 bitów liczby nie składa się z samych jedynek, i odpowiednio odczytujemy wartość z tablicy P dla dolnych lub górnych bitów liczby[2].

Przedstawiony sposób pozwala nam uniknąć liniowego kosztu wyszukiwania związanego z aktualizacją wyniku i przejściem do kolejnego podziału w zamian za kilka operacji bitowych i wstępne, jednorazowe obliczenia. Otrzymujemy w ten sposób algorytm o złożoności $O\left(2^{|V|}/\sqrt{|V|}\right)$. Jest on obarczony jednak sporą stałą, więc w rezultacie okazuje się tylko około 2 razy szybszy od programów wzorcowych. Implementacja tego rozwiązania znajduje się w plikach `pod3.cpp` i `pod4.pas`.

Algorytm 2.2. Kolejnym pomysłem jest przejrzenie wszystkich możliwych podziałów zbioru wierzchołków, a nie tylko tych, w których liczby wierzchołków w obu częściach są jednakowe. Taka ewidentna „rozrzutność" (rozważamy w końcu wiele niepotrzebnych podziałów) pozwala jednak łatwo śledzić liczby wierzchołków w obu częściach i inne własności masek podziału, gdyż zmieniają się one w sposób bardzo regularny. Obliczając koszty tylko tych podziałów, które są połowieniami (stosując w tym celu ten sam pomysł,

[2]Ciekawostka: w C/C++ w kompilatorach z serii GCC istnieją bardzo wygodne wbudowane instrukcje realizujące obie powyższe operacje. Funkcja `__builtin_popcount(x)` zlicza liczbę zapalonych bitów w liczbie `x` — jest ona realizowana podobnie, jak to zostało opisane powyżej. Z kolei funkcja `__builtin_ctz(x)` (ang. *count trailing zeros*) zwraca liczbę zer znaczących na końcu `x`, np. dla `x`=8 zwraca 3 (uwaga: funkcja jest niezdefiniowana dla `x`=0). W przeciwieństwie do poprzedniej, ta funkcja w procesorach z architekturą x86 jest realizowana za pomocą pojedynczej instrukcji procesora. Z kolei liczbą zawierającą jedynie najmniejszy zapalony bit liczby `x` jest `x & ~(x - 1)` (przyp. red.).

co w poprzednim algorytmie) uzyskujemy ostatecznie rozwiązanie o złożoności takiej jak rozwiązanie wzorcowe, choć w praktyce nieco wolniejsze. Podobnie jak poprzednie metody, ta także pozwalała uzyskać 100% punktów. Jej implementacja znajduje się w pliku `pod5.cpp`.

Rozwiązania wolniejsze

Rozwiązania wolniejsze są w zasadzie różnymi wariantami siłowego przeszukiwania zbioru podziałów grafu, w związku z czym ich złożoność różni się złożoności algorytmu wzorcowego jedynie czynnikiem poprzedzającym wyraz 2^n.

Algorytm 3.1. Algorytm polega na podobnym jak w rozwiązaniu wzorcowym przeszukiwaniu tylko tych podziałów, które są połowieniami. Dla każdego połowienia w czasie $O(|E|)$ obliczana jest liczba krawędzi pomiędzy połówkami grafu. W rezultacie złożoność algorytmu wynosi $O\left(|E|2^{|V|}/\sqrt{|V|}\right) = O(|V|^{\frac{3}{2}}2^{|V|})$. Przekłada się to na kilkunastokrotnie gorszy czas działania w stosunku do algorytmu wzorcowego.

Algorytm pozwalał uzyskać 30%–40% punktów, a jego implementacja znajduje się w plikach `pods1.cpp` i `pods4.pas`.

Algorytm 3.2. Algorytm jest prostszy, choć na pierwszy rzut oka bardziej rozrzutny niż poprzedni. Przeglądamy wszystkie podziały zbioru wierzchołków grafu i dla każdego podziału będącego połowieniem od nowa zliczamy liczbę krawędzi pomiędzy połówkami. W ten sposób dla $(2N)!/(N!N!)$ podziałów będących połowieniami wykonujemy operacje w czasie $O(|V|+|E|)$, a dla pozostałych — w czasie $O(|V|)$ (generacja kolejnego podziału). W rezultacie dostajemy algorytm o złożoności $O\left(|V|2^{|V|} + \frac{|E|}{\sqrt{|V|}}2^{|V|}\right)$, czyli $O(|V|^{\frac{3}{2}}2^{|V|})$. Złożoność tego algorytmu jest taka sama jak poprzedniego, lecz stała proporcjonalności jest nieco gorsza. Pozwalał on jednak także uzyskać 30%–40% punktów. Jego implementacja znajduje się w plikach `pods2.cpp` i `pods5.pas`.

Algorytm 3.3. Algorytm polega na przeglądaniu wszystkich podzbiorów zbioru wierzchołków, ale w uporządkowany sposób. Wszystkie słowa zerojedynkowe określonej długości można ułożyć w ciąg w ten sposób, by kolejne dwa różniły się dokładnie jednym bitem (podobny efekt uzyskaliśmy w Lemacie 1, ale dotyczył on słów o ustalonej liczbie jedynek i wymagał zmiany dwóch bitów). Wówczas śledzenie liczby krawędzi pomiędzy częściami podziału oraz kontrolę, czy podział jest połowieniem, można wykonać w czasie $O(|V|)$. Oczywiście aktualizację znalezionego minimum wykonujemy tylko wtedy, gdy aktualny podział jest połowieniem. W rezultacie otrzymujemy algorytm o złożoności $O(|V|2^{|V|})$, czyli tylko trochę gorszej od złożoności algorytmu wzorcowego. Dla większych testów nie mieścił się on już w założonych ograniczeniach czasowych — pozwalał uzyskać około 80% punktów.

Implementacja znajduje się w plikach `pods3.cpp` i `pods6.pas`.

Rozwiązania niepoprawne

Większość rozwiązań niepoprawnych polega na losowaniu dużego zbioru połowień i próbie „złożenia” z nich możliwie dobrego rozwiązania, niekoniecznie optymalnego. Ze względu na stosunkowo małą przestrzeń stanów, niektóre z algorytmów mogą trafić na rozwiązanie optymalne i podać dobrą odpowiedź.

Algorytm 4.1. Losujemy wiele podziałów i dla każdego z nich sprawdzamy, czy jest to połowienie. Jeśli tak, to obliczamy liczbę krawędzi pomiędzy połówkami grafu. Wypisujemy ten spośród wylosowanych podziałów, który ma najmniejszą liczbę krawędzi. Jest to algorytm bardzo podobny do algorytmu 3.2 — zamiast sprawdzać przestrzeń podziałów w uporządkowany sposób, testujemy ją losowo.

Takie rozwiązanie pozwalało uzyskać 0–20% punktów. Jego implementacja znajduje się w pliku `podb1.cpp`.

Algorytm 4.2. To rozwiązanie opiera się na obserwacji, że fragment grafu tworzący połówkę w optymalnym połowieniu powinien być jak najbardziej zagęszczony, czyli jak najbardziej przypominać klikę. Dlatego w algorytmie najpierw losuje się wierzchołek, a następnie dołącza do niego inne wierzchołki zachłannie tak, aby w każdym ruchu maksymalizować liczbę krawędzi pomiędzy nowo dodanym a poprzednio wybranymi. Jeśli istnieje kilka możliwości optymalnych ruchów, to algorytm wybiera losowo jedną z nich. Zauważmy, że w algorytmie zakłada się, że jeśli graf jest spójny, to co najmniej jedna część połowienia jest spójna. Tak być nie musi, na co kontrprzykładami są wszystkie testy oznaczone literką `b`. Stąd algorytm, jako błędny, nie pozwala zdobyć żadnych punktów.

Implementacja znajduje się w pliku `podb2.cpp`.

Algorytm 4.3. Jest to drobna modyfikacja algorytmu 4.2. Tym razem, zamiast zachłannie budować gęsty podgraf, dobieramy wierzchołki tak, aby w każdym kroku minimalizować liczbę krawędzi pomiędzy wierzchołkami wybranymi i pozostałymi. Podstawową wadą tego algorytmu jest to, że ma tendencję do dobierania najpierw wierzchołków o małych stopniach, a dopiero potem o większych, co może już na początku przekreślić optymalność rozwiązania. Większość kontrprzykładów na heurystykę 4.2 jest również kontrprzykładami na ten pomysł, ale w trzech przypadkach to testy z literką `e` obnażają jego błędy. Implementacja znajduje się w pliku `podb3.cpp`. Program nie dostaje żadnych punktów.

Algorytm 4.4. W algorytmie losuje się połowienie i poprawia się je dopóty, dopóki jest to możliwe. Dla bieżącego połowienia sprawdza się, czy istnieją dwa wierzchołki z różnych połówek, których wymiana zmniejsza liczbę krawędzi pomiędzy połówkami. Jeśli tak, to wierzchołki zostają wymienione, w przeciwnym razie uznaje się rozwiązanie za lokalnie optymalne i losuje kolejne. Każdą poprawkę realizujemy w czasie $O(|V|^2)$, więc przy każdym losowaniu wykonujemy operacje o złożoności $O(|E||V|^2)$. Jako wyjście wypisujemy najlepsze znalezione połowienie. Dość dobrymi kontrprzykładami, dla których ta heurystyka zawodzi, są dopełnienia prostych grafów takich jak cykl, ścieżka itp. oraz rzadkie grafy strukturalne, najlepiej o dużej liczbie wierzchołków o stopniach parzystych. Algorytm ten pozwalał uzyskać około 20% punktów.

Implementacja znajduje się w pliku `podb4.cpp`.

Algorytm 4.5. Jest to drobna modyfikacja algorytmu 4.4. Zamiast wybierać dowolną parę wierzchołków, wybieramy zawsze tę, której zamiana najbardziej zmniejsza liczbę krawędzi między połówkami. W rzeczywistości zmniejsza to losowość algorytmu 4.4, co powoduje, że heurystyka ta daje nieznacznie gorsze wyniki.

Implementacja znajduje się w pliku `podb5.cpp`. Algorytm pozwalał uzyskać około 20% punktów.

Testy

Przygotowano następujące testy:

Nazwa	n	Opis
pod1abcde.in	2, 18, 20	mała grupa poprawnościowa
pod2abcd.in	18, 20	mała grupa poprawnościowa
pod3abcde.in	18, 20	mała grupa poprawnościowa
pod4abc.in	22, 24	średnia grupa poprawnościowa
pod5abc.in	24	średnia grupa poprawnościowa
pod6abc.in	24	średnia grupa poprawnościowo-wydajnościowa
pod7abcde.in	24	średnia grupa poprawnościowo-wydajnościowa
pod8abcde.in	26	duża grupa poprawnościowo-wydajnościowa
pod9abcd.in	26	duża grupa poprawnościowo-wydajnościowa
pod10abc.in	26	duża grupa poprawnościowo-wydajnościowa

Testy `a` są ogólnymi testami poprawnościowymi, sprawdzającymi błędy implementacyjne. Nie są kontrprzykładami na żadną konkretną heurystykę. Są w zasadzie strukturalnymi testami z elementami losowości, raczej z grafami rzadkimi.

Testy `b` stanowią kontrprzykłady dla rozwiązań niepoprawnych typu 2, jak i innych wymyślonych przez zawodników. Są „złośliwymi" testami z małą liczbą połowień optymalnych, przy czym we wszystkich połowieniach optymalnych mimo spójności całego grafu, podgrafy indukowane przez połówki są niespójne. Sprawia to, że heurystyka 2 generuje niepoprawne rozwiązania. Testy te stanowią raczej grafy rzadkie.

Testy `c` i `d` są testami poprawnościowo-wydajnościowymi. Są tworzone na zasadzie znalezienia dopełnienia pewnego prostego grafu, np. cyklu, ścieżki, gwiazdy itp. Są to grafy gęste, dzięki czemu testują szybkość działania programu. Ponadto takie grafy często stanowią dobre kontrprzykłady na heurystyki typu 4 i 5.

Testy `e` stanowią (oprócz 1*e*) kontrprzykłady dla heurystyki typu 3. Są to zazwyczaj grafy, w których wierzchołki o małych stopniach nie mogą należeć do tej samej połówki grafu. W momencie, gdy jest klika wierzchołków o małych stopniach, a reszta ma duże stopnie, heurystyki typu 3 mają tendencję do dobierania na początku wierzchołków o małych stopniach, co z reguły nie jest optymalne.

Testy są podzielone na 10 grup. Grupy od 1. do 3. zawierają tylko testy z $n \leqslant 20$ — powinny je przejść również rozwiązania nieefektywne. Na testach od 4. do 10. rozwiązania efektywne mieściły się w $\frac{1}{7}$–$\frac{1}{4}$ limitu czasowego, w zależności od implementacji i języka programowania. Rozwiązania nieefektywne tracą punkty za przekroczenie połowy limitu czasowego lub nie kończą działania w określonym czasie. Grupy od 4. do 7. zawierają testy o $n = 22, 24$, zaś grupy od 8. do 10. dostępne tylko dla algorytmów efektywnych $n = 26$.

W praktyce okazało się, że na ocenę rozwiązań miały wpływ także drobne różnice implementacyjne, wybór języka programowania itp.

Jakub Radoszewski — Treść zadania, Opracowanie
Anna Niewiarowska — Program

OI, Etap III, dzień pierwszy, 02.04.2008

Trójkąty

Mamy danych n parami różnych punktów na płaszczyźnie ($n \geqslant 3$). Istnieje $\frac{n \cdot (n-1) \cdot (n-2)}{6}$ trójkątów, których wierzchołkami są pewne parami różne spośród tych punktów (wliczając trójkąty zdegenerowane, tzn. takie, których wierzchołki są współliniowe).

Chcemy obliczyć sumę powierzchni wszystkich trójkątów o wierzchołkach w danych punktach.

Fragmenty płaszczyzny należące do wielu trójkątów liczymy wielokrotnie. Przyjmujemy, że powierzchnia trójkątów zdegenerowanych jest równa zero.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia współrzędne danych punktów na płaszczyźnie,*
- *wyznaczy sumę powierzchni wszystkich trójkątów o wierzchołkach w danych punktach,*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu wejścia znajduje się jedna liczba całkowita n ($3 \leqslant n \leqslant 3\,000$), oznaczająca liczbę wybranych punktów. Kolejne n wierszy zawiera po dwie liczby całkowite x_i oraz y_i ($0 \leqslant x_i, y_i \leqslant 10\,000$), oddzielone pojedynczym odstępem i oznaczające współrzędne i-tego punktu (dla $i = 1, 2, \ldots, n$). Żadna para (uporządkowana) współrzędnych na wejściu nie powtarza się.

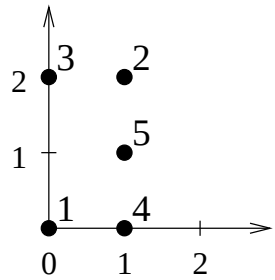
Wyjście

W pierwszym i jedynym wierszu wyjścia powinna się znajdować jedna liczba rzeczywista, równa sumie powierzchni wszystkich trójkątów o wierzchołkach w danych punktach. Wynik powinien być wypisany z dokładnie jedną cyfrą po kropce dziesiętnej i nie powinien się różnić od faktycznej wartości o więcej niż 0.1.

Przykład

Dla danych wejściowych:

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



poprawnym wynikiem jest:

```
7.0
```

Rozwiązanie

Rozwiązanie $O(n^3)$

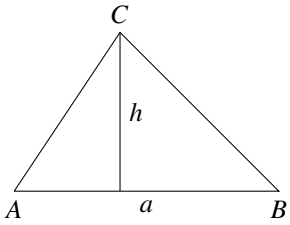
Zadanie polega na wyznaczeniu sumy pól wszystkich trójkątów, których wierzchołki pochodzą z zadanego zbioru n punktów. Najprostsze rozwiązanie to przejrzenie wszystkich trójkątów i zsumowanie ich pól — jego złożoność to $O(n^3)$. Pole trójkąta można policzyć na kilka sposobów.

Metoda I

Bodaj najbardziej znanym wzorem na pole trójkąta jest

$$S = \frac{ah}{2},$$

czyli połowa iloczynu długości podstawy i wysokości trójkąta. Podstawą trójkąta może przy tym być dowolny z jego boków.



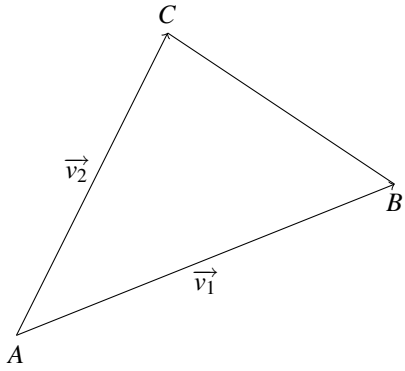
Metoda II

Aby policzyć pole trójkąta, można wybrać dwa jego boki i wyznaczyć połowę wartości bezwzględnej iloczynu wektorowego wektorów odpowiadających tym bokom:

$$S = \frac{|\overrightarrow{v_1} \times \overrightarrow{v_2}|}{2}.$$

Przypomnijmy, że wartość iloczynu wektorowego $\overrightarrow{v_1} = [x_1, y_1]$ oraz $\overrightarrow{v_2} = [x_2, y_2]$ to

$$\overrightarrow{v_1} \times \overrightarrow{v_2} = x_1 \cdot y_2 - x_2 \cdot y_1.$$

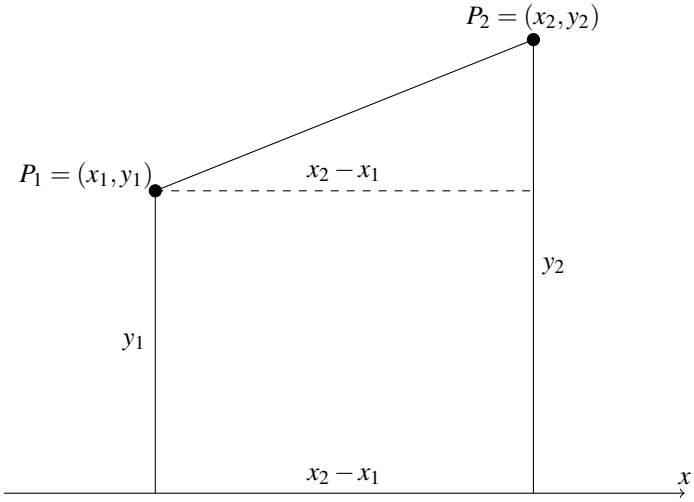


Metoda III

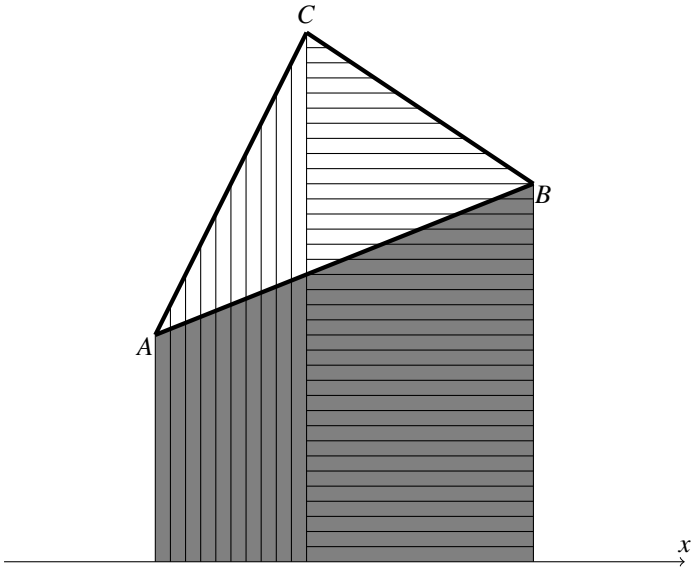
Pole trójkąta, podobnie jak i pole dowolnego innego wielokąta, można wyznaczyć za pomocą *metody trapezów*, czyli obchodząc boki trójkąta w kolejności zgodnej z kierunkiem ruchu wskazówek zegara i dodając do wyniku składowe odpowiadające kolejnym bokom. Składową dla boku łączącego punkty $P_1 = (x_1, y_1)$ i $P_2 = (x_2, y_2)$ jest „pole" trapezu ograniczonego tym bokiem, osią OX oraz liniami rzutów punktów P_1 i P_2 na oś OX. Wartość nazwana umownie „polem" wyraża się wzorem

$$\frac{(x_2 - x_1)(y_1 + y_2)}{2}$$

i może być ujemna (na przykład, jeżeli $x_1 > x_2$ oraz $y_1, y_2 > 0$).



Działanie metody trapezów dla całej figury ($\triangle ABC$) jest przedstawione na kolejnym rysunku:



Obchodząc ten trójkąt w kolejności zgodnej z kierunkiem ruchu wskazówek zegara, uzyskujemy „ścieżkę": $A \to C \to B \to A$. Składowe boków AC i CB (pola trapezów zakreskowanych na rysunku odpowiednio pionowo i poziomo) są dodatnie, natomiast składowa boku BA (zanegowane pole szarego trapezu na rysunku) jest ujemna. Zsumowanie wszystkich składowych daje więc w wyniku pole białego, zakreskowanego obszaru z rysunku, czyli faktycznie pole trójkąta $\triangle ABC$.

Jak to przyspieszyć?

Trzeba poszukać rozwiązania efektywniejszego niż działające w czasie $O(n^3)$, jeśli chcemy zmieścić się w limicie czasowym dla największych danych. Skoro sumowanie pól wszystkich trójkątów jest zbyt czasochłonne, to może powinniśmy spróbować zliczać pola trójkątów nie pojedynczo, a większymi partiami?

Możemy, na przykład, rozważyć wszystkie trójkąty zawierające jako bok określony odcinek AB. Wszystkich możliwych odcinków utworzonych z n punktów jest $O(n^2)$, więc daje to pewne szanse na rozwiązanie o lepszej złożoności niż poprzednie, jeśli tylko poradzimy sobie szybko z każdą grupą. Rzeczywiście, właśnie na tym pomyśle opierają się wszystkie rozwiązania wzorcowe niniejszego zadania[1].

Dla każdego *odcinka skierowanego* $\overrightarrow{AB}$, czyli wektora zaczepionego w pewnych dwóch spośród zadanych punktów, będziemy wyliczać jego *wagę* W_{AB}, czyli sumę zależnych od niego składników wchodzących w skład końcowego wyniku. Waga ta będzie miała

[1] Rozwiązania te różnią się przede wszystkim zastosowanym wzorem na pole trójkąta. Ponieważ każdy z Czytelników ma zapewne swój ulubiony, więc w dalszym ciągu opisu przedstawiamy „równolegle" rozwiązania wzorcowe dla każdej z trzech wcześniej przytoczonych metod wyznaczania pola trójkąta.

różną wartość i znaczenie w zależności od tego, z jakiej metody obliczania pola trójkąta zamierzamy skorzystać:

Metoda I — wagą odcinka skierowanego będzie suma pól trójkątów, w których jest on jednym z boków;

Metoda II — wagą wektora $\overrightarrow{AB}$ jest suma pól trójkątów, w których wektor $\overrightarrow{AB}$ pojawia się przy liczeniu pola za pomocą iloczynu wektorowego;

Metoda III — wagą wektora jest suma (po odpowiednich trójkątach) pól trapezów, które są ograniczone tym wektorem, osią OX i liniami rzutów punktów końcowych wektora na oś OX, z uwzględnieniem znaku; innymi słowy, będzie to iloczyn pola ze znakiem trapezu wyznaczonego przez ten wektor oraz liczby trójkątów, na których (odpowiednio skierowanym) obwodzie ten wektor występuje.

Zauważmy, że jeśli zsumujemy wagi W_{AB} dla wszystkich możliwych wektorów $A \neq B$, to pole każdego trójkąta policzymy kilkakrotnie. Dokładniej, w przypadku:

Metoda I — obwód każdego trójkąta składa się z trzech boków, z których każdy można skierować na dwa sposoby, a zatem pole trójkąta zostanie uwzględnione sześciokrotnie;

Metoda II — na sześć sposobów można wybrać parę boków skierowanych, czyli wektorów zaczepionych w jednym wierzchołku trójkąta (zauważ, że trzy spośród tych sześciu par będą różniły się od pozostałych trzech jedynie kolejnością wektorów);

Metoda III — skierowany zgodnie z kierunkiem ruchu wskazówek zegara obwód trójkąta składa się z 3 wektorów, stąd pole trójkąta zostanie wliczone do sumy końcowej trzykrotnie.

Skoro wiemy już, jak na podstawie sum wag wszystkich wektorów uzyskać wynik końcowy, to możemy zastanowić się, jak wyznaczyć W_{AB} w każdej z metod. Oznaczmy przez $P_1, P_2, \ldots, P_{n-2}$ wszystkie punkty dane w zadaniu, oprócz punktów A i B.

Metoda I — szukaną wagą odcinka skierowanego AB jest suma pól trójkątów:

$$W_{AB} = S(\triangle ABP_1) + S(\triangle ABP_2) + \ldots + S(\triangle ABP_{n-2}).$$

Wybierając w każdym z nich jako podstawę bok AB, dostajemy następujący wzór:

$$W_{AB} = \frac{ah_1}{2} + \frac{ah_2}{2} + \ldots + \frac{ah_{n-2}}{2} = \frac{a}{2}(h_1 + h_2 + \ldots + h_{n-2}),$$

gdzie $a = |AB|$, natomiast $h_1, h_2, \ldots, h_{n-2}$ oznaczają wysokości trójkątów $\triangle ABP_1, \triangle ABP_2, \ldots, \triangle ABP_{n-2}$ opuszczone na podstawę AB — inaczej mówiąc, są to odległości punktów $P_1, P_2, \ldots, P_{n-2}$ od prostej $pr(AB)$.

Jeżeli równanie tej prostej zapiszemy jako $ax + by + c = 0$, to odległość punktu $P_i = (x_i, y_i)$ od $pr(AB)$ wyraża się znanym wzorem:

$$h_i = d(P_i, pr(AB)) = \frac{|ax_i + by_i + c|}{\sqrt{a^2 + b^2}}.$$

Gdyby dla wszystkich punktów P_i wyrażenie $ax_i + by_i + c$ było dodatnie, to moglibyśmy bardzo ładnie uprościć wzór na W_{AB}, gdyż wówczas suma $h_1 + h_2 + \ldots + h_{n-2}$ wynosiłoby:

$$\frac{a(x_1 + \ldots + x_{n-2}) + b(y_1 + \ldots + y_{n-2}) + c}{\sqrt{a^2 + b^2}}. \tag{1}$$

Chociaż nie możemy liczyć na powyższy przypadek, to możemy jednak podzielić wszystkie punkty P_i na leżące po jednej stronie prostej $pr(AB)$ — gdzie rzeczywiście $ax_i + by_i + c \geqslant 0$, oraz na położone po drugiej stronie — gdzie $ax_i + by_i + c < 0$ (punkty leżące na prostej są i tak dla nas nieistotne, gdyż tworzą z A i B wyłącznie trójkąty zdegenerowane o zerowym polu). To spostrzeżenie pozwala przedstawić wagę wektora AB jako różnicę dwóch ułamków postaci analogicznej jak (1).

Metoda II — wagą odcinka skierowanego AB jest, tak samo jak w metodzie I, suma:

$$W_{AB} = S(\triangle ABP_1) + S(\triangle ABP_2) + \ldots + S(\triangle ABP_{n-2}).$$

Wyznaczając pole trójkąta $\triangle ABP_i$ w oparciu o skierowaną parę boków $\overrightarrow{AB}$ i $\overrightarrow{AP_i}$, dostajemy następujący wzór na W_{AB}:

$$W_{AB} = \frac{|\overrightarrow{AB} \times \overrightarrow{AP_1}|}{2} + \frac{|\overrightarrow{AB} \times \overrightarrow{AP_2}|}{2} + \ldots + \frac{|\overrightarrow{AB} \times \overrightarrow{AP_{n-2}}|}{2}.$$

Iloczyn wektorowy jest liniowy ze względu na każdy z dwóch argumentów. W szczególności, dla dowolnych wektorów $\vec{u}$, $\vec{v}$ i $\vec{w}$ mamy:

$$\vec{u} \times (\vec{v} + \vec{w}) = \vec{u} \times \vec{v} + \vec{u} \times \vec{w}.$$

Ponieważ we wzorze na W_{AB} występują wartości bezwzględne iloczynów wektorowych, aby skorzystać z powyższej własności, podzielimy punkty P_i na dwie grupy ze względu na znak iloczynu $\overrightarrow{AB} \times \overrightarrow{AP_i}$. Zauważmy, że jest on dodatni dla punktów P_i leżących na lewo od wektora $\overrightarrow{AB}$, czyli tworzących z nim kąt wypukły, natomiast niedodatni dla pozostałych punktów (punkty leżące na prostej $pr(AB)$ są, jak w metodzie I, nieistotne). Dla każdej z grup wystarczy wyznaczyć wartość postaci:

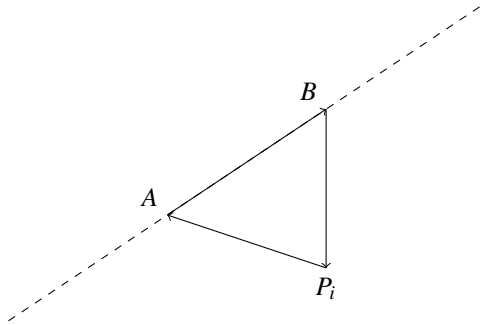
$$W_{AB} = \frac{1}{2}\overrightarrow{AB} \times (\overrightarrow{AP_{i_1}} + \overrightarrow{AP_{i_2}} + \ldots + \overrightarrow{AP_{i_k}}). \tag{2}$$

Wagę AB dostajemy jako różnicę tych wartości.

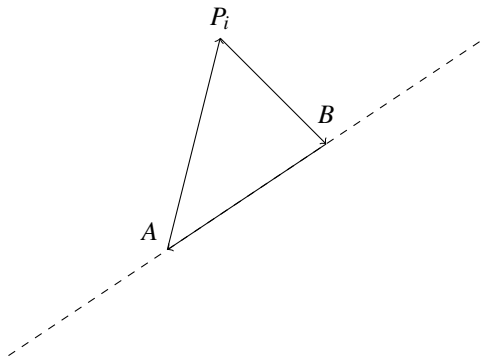
Metoda III — wagą odcinka skierowanego $\overrightarrow{AB}$ dla punktów $A = (x_A, y_A)$ i $B = (x_B, y_B)$ jest wartość:

$$W_{AB} = m \cdot \frac{(x_B - x_A)(y_A + y_B)}{2}, \tag{3}$$

gdzie m oznacza liczbę trójkątów $\triangle ABP_i$, na których obwodzie, skierowanym zgodnie z kierunkiem ruchu wskazówek zegara, występuje odcinek $\overrightarrow{AB}$. Zauważmy, że aby trójkąt $\triangle ABP_i$ miał tę własność, punkt P_i musi leżeć na prawo od wektora $\overrightarrow{AB}$, czyli wektor $\overrightarrow{BP_i}$ musi tworzyć z nim kąt skierowany wklęsły:



Dla punktów P_i położonych na lewo od wektora $\overrightarrow{AB}$, trójkąt $\triangle ABP_i$ nie zawiera na swoim (prawoskrętnym) obwodzie wektora $\overrightarrow{AB}$, lecz wektor $\overrightarrow{BA}$:



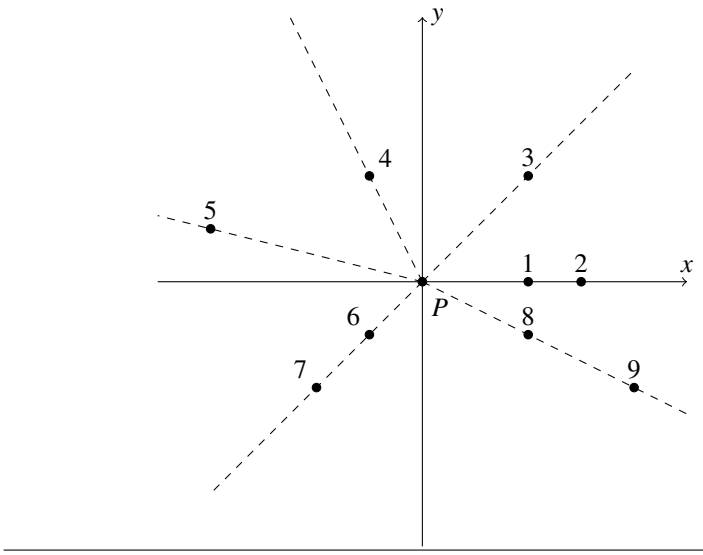
Punkty P_i leżące na prostej $pr(AB)$ nie mają, tak samo jak poprzednio, wpływu na wagę odcinka $\overrightarrow{AB}$. Parametr m ze wzoru (3) jest więc równy liczbie punktów P_i, które leżą na prawo od wektora $\overrightarrow{AB}$.

Analiza przeprowadzona dla wszystkich trzech metod wykazuje, że niezależnie od obranego wzoru liczenia pól trójkątów, musimy umieć dla wektora $\overrightarrow{AB}$ efektywnie podzielić wszystkie punkty P_i na leżące na prawo i na lewo od niego[2]. Dla każdej z tych grup musimy także wyznaczyć określoną wartość mającą wpływ na wagę wektora $\overrightarrow{AB}$ — w kolejnych metodach jest to: suma odciętych i rzędnych punktów z grupy (metoda I), suma wektorów o początku w A i końcach w punktach z grupy (metoda II) czy po prostu liczba punktów w grupie (metoda III). Do efektywnego rozwiązania zadania może więc doprowadzić wybranie takiej kolejności przeglądania wszystkich wektorów $\overrightarrow{AB}$, żeby podział punktów na leżące na lewo i na prawo od wektora był łatwy do przeprowadzenia lub zaktualizowania na podstawie podziału dla poprzednio rozważanego wektora.

[2]W przypadku metody I mówiliśmy raczej o punktach położonych nad oraz pod prostą $pr(AB)$. Zauważmy jednak, że punkty położone nad i pod prostą $pr(AB)$ odpowiadają punktom leżącym na lewo i na prawo od wektora $\overrightarrow{AB}$.

Zamiatanie kątowe

Dobrym kierunkiem poszukiwania właściwej kolejności analizowania wektorów wydaje się być rozważenie punktów *posortowanych kątowo (biegunowo)*. Niech $Z = \{P_1, P_2, \ldots, P_n\}$ oznacza zbiór wszystkich zadanych punktów. W porządku kątowym określamy środek — może to być dowolny punkt $P = (x, y)$ ze zbioru Z. Następnie sortujemy pozostałe punkty $Q \in Z$ niemalejąco względem kątów skierowanych, jakie tworzą z półprostą poziomą o początku w punkcie P, czyli względem $\angle P'PQ$, gdzie $P' = (x + \delta, y)$, $\delta > 0$. Kolejność punktów tworzących z półprostą taki sam kąt może być dowolna. Dla ustalenia uwagi możemy przyjąć, że w przypadku jednakowych kątów wcześniejszy jest punkt położony bliżej środka P.



Rys. 1: Przykład posortowania kątowego dziewięciu punktów wokół punktu P umieszczonego w środku układu współrzędnych (numery punktów odpowiadają kolejności w posortowanej sekwencji).

Punkty ze zbioru Z będziemy sortować kątowo kolejno wokół $P_1, P_2, \ldots, P_n$. Uporządkowanie Z wokół P_i pozwoli nam przeanalizować wszystkie wektory $\overrightarrow{P_iP_j}$ dla $j \neq i$ w kolejności zgodnej z porządkiem kątowym P_j i określić ich wagi zgodnie z metodą I, II lub III. Algorytm będzie zatem pewną formą klasycznej *metody zamiatania*, z tą tylko różnicą, że miotłą będzie półprosta obracająca się wokół punktu P_i.

Zamiatanie dla punktu P_i rozpoczynamy od półprostej poziomej p o początku w P_i, skierowanej ku rosnącym wartościom odciętej. Uzupełnienie p do prostej, czyli półprostą o początku w P_i skierowaną ku malejącym wartościom odciętej, nazwiemy q — oznaczenie to przyda nam się później, przy opisie algorytmu. Dla półprostej p łatwo jest wyznaczyć wszystkie punkty leżące na lewo (zbiór L) i na prawo od niej (zbiór P):

$$\begin{aligned} L &= \{P_j : |\angle P'P_iP_j| \in (0^\circ, 180^\circ)\} \\ P &= \{P_j : |\angle P'P_iP_j| \in (180^\circ, 360^\circ)\}, \end{aligned}$$

gdzie P' jest dowolnym, różnym od P_i punktem półprostej p.

Potrzebne do wyliczenia wagi wektora $\overrightarrow{P_iP'}$ wartości dla zbioru L (odpowiednio P) są wówczas równe:

Metoda I — sumom postaci:

$$\sum_{P_j=(x_j,y_j)\in L} x_j \quad \text{(odpowiednio} \quad \sum_{P_j=(x_j,y_j)\in P} x_j\text{)}$$

$$\sum_{P_j=(x_j,y_j)\in L} y_j \quad \text{(odpowiednio} \quad \sum_{P_j=(x_j,y_j)\in P} y_j\text{)}$$

Metoda II — sumie postaci:

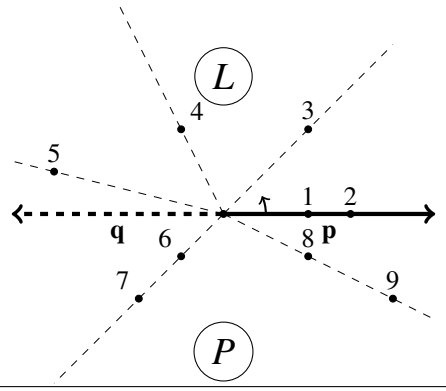
$$\sum_{P_j\in L} \overrightarrow{P_iP_j} \quad \text{(odpowiednio} \quad \sum_{P_j\in P} \overrightarrow{P_iP_j}\text{)}$$

Metoda III — rozmiarowi zbioru $|L|$ (odpowiednio $|P|$).

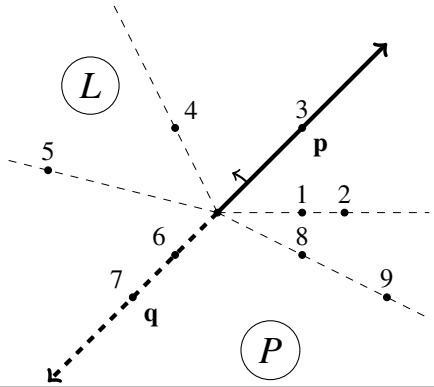
Znając powyższe zbiory i wartości dla początkowego położenia miotły, możemy przystąpić do zamiatania. Wykonamy półprostą p obrót o 360° wokół środka P_i, powiedzmy przeciwnie do kierunku ruchu wskazówek zegara. W trakcie obrotu na miotle p będą pojawiać się punkty zbioru Z w porządku kątowym względem środka P_i. Gdy na p znajdzie się punkt $P_j \in Z$, będziemy wyznaczać wagę odcinka skierowanego $\overrightarrow{P_iP_j}$. Po rozważeniu P_j punkt ten przerzucimy ze zbioru L do P, aktualizując jednocześnie interesujące nas wartości związane z L i P. Jeśli w trakcie obrotu miotły, na półprostej q znajdzie się jakiś punkt $P_k \in Z$, to punkt ten przenosimy z kolei ze zbioru P do L, także uaktualniając odpowiednie wartości służące do wyliczania wag.

Aby sprawnie wykonać obrót miotłą p, odnotowując wszystkie wystąpienia punktów ze zbioru Z na półprostej p lub q, dla każdej z tych półprostych będziemy pamiętać ostatni punkt ze zbioru Z, przez który ona przechodziła. Oznaczmy te punkty odpowiednio Q i R, a ich następniki w porządku kątowym — odpowiednio Q' i R'. Obrót prostej będziemy wykonywać zawsze o mniejszy z kątów: pomiędzy P_iQ i P_iQ' lub pomiędzy P_iR i P_iR'.

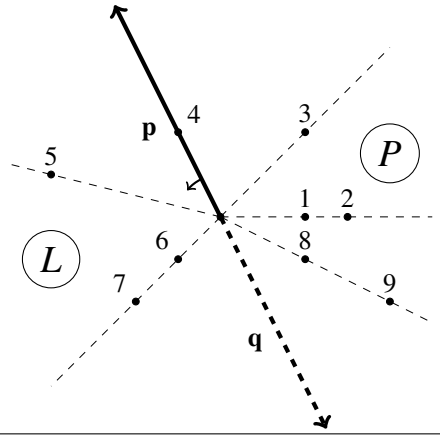
Rozważmy konkretny przykład, który pozwoli lepiej zrozumieć mechanizm opisanego zamiatania kątowego. Na rys. 2–6 przedstawiamy, jak kształtują się zbiory L i P przy kilku kolejnych obrotach półprostych p i q dla zbioru punktów z rys. 1.



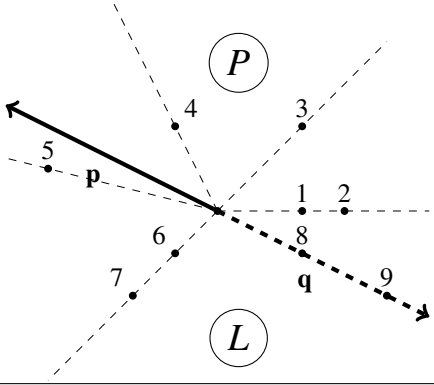
Rys. 2: Na samym początku mamy $L = \{3,4,5\}$, $P = \{6,7,8,9\}$. Tuż po tym, jak półprosta p zacznie się obracać, do zbioru P dołączą punkty o numerach 1 i 2.



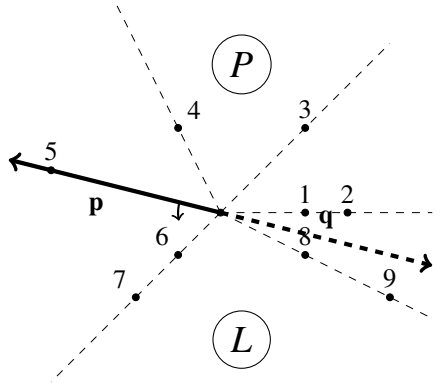
Rys. 3: W kolejnym kroku p napotyka na punkt 3, a q, równocześnie, na punkty 6 i 7. Wszystkie te punkty wskutek tego zdarzenia zmieniają swoje położenia w ramach zbiorów L i P. Dokładniej, zaraz po tym, jak półprosta zacznie się obracać, będzie $L = \{4,5,6,7\}$, a $P = \{1,2,3,8,9\}$.



Rys. 4: Po tym, jak p napotka punkt 4, nowe postaci zbiorów L i P to: $L = \{5,6,7\}$ i $P = \{1,2,3,4,8,9\}$.



Rys. 5: Przed tym, jak p napotka punkt 5, jej przedłużenie, czyli q, napotka punkty 8 oraz 9, które zostaną wówczas przerzucone z L do P. Nowe postaci zbiorów to: $L = \{5,6,7,8,9\}$, $P = \{1,2,3,4\}$.



Rys. 6: Tuż po napotkaniu przez p punktu 5 mamy $L = \{6,7,8,9\}$, $P = \{1,2,3,4,5\}$.

„Arytmetyka na kątach" — uwaga na pułapki!

Teoretycznie już wszystko wiemy — jak posortować punkty, jak je potem przetwarzać i jak wyznaczyć wynik końcowy (nawet na trzy sposoby). Pozostaje bardzo praktyczne pytanie: jak porównywać kąty i jak właściwie posortować punkty kątowo. Można spróbować się do tego zabrać na (co najmniej) dwa sposoby.

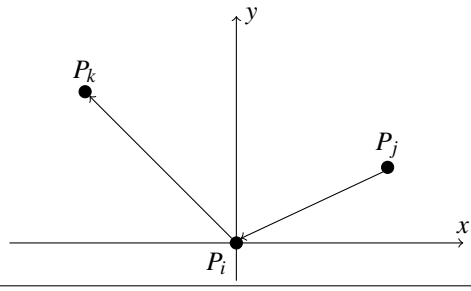
Pierwszy sposób to przypisanie punktom kątów, jakie tworzą one z wyjściową półprostą p, i sortowanie (porównywanie) względem tych kątów. Wartość kąta pomiędzy p i P_iQ możemy wyznaczyć za pomocą funkcji odwrotnych do funkcji trygonometrycznych (na przykład arcsin, arccos, ...). W obliczeniach komputerowych, szczególnie rozwiązując problemy geometryczne, staramy się jednak *unikać stosowania liczb zmiennoprzecinkowych*, gdy tylko da się zastąpić je obliczeniami na liczbach całkowitych. W obliczeniach na zmiennoprzecinkowych wartościach kątów, szczególnie z wykorzystaniem tak złożonych funkcji, jak odwrotne funkcje trygonometryczne, trudno bowiem uniknąć błędów zaokrągleń. Ich konsekwencją mogłaby być na przykład niewłaściwa kolejność w posortowanym ciągu punktów, które są „prawie współliniowe" z P_i. Taka pomyłka mogłaby oczywiście skończyć się złym wynikiem końcowym.

Spróbujmy więc obejść się bez funkcji trygonometrycznych (i ich odwrotności) i obliczeń zmiennoprzecinkowych. Możemy zaimplementować własną funkcję porównującą dwa punkty, tzn. zwracającą dla punktów P_j i P_k prawdę wtedy i tylko wtedy, gdy P_j powinien wystąpić w porządku przed P_k. Naturalne jest zastosowanie do tego celu iloczynu wektorowego, którego znak pozwala badać, czy przechodząc przez wspólny koniec dwóch odcinków skręcamy w lewo, w prawo, czy też idziemy prosto[3]. Na rysunku 7 możemy znaleźć przykład takiego porównania punktów.

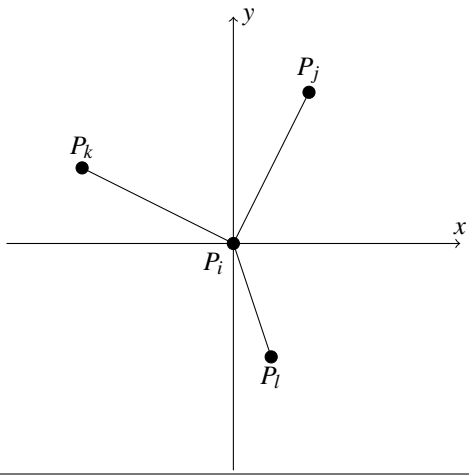
Reguła skrętu w prawo kryje jednak w sobie pułapki. Wyznacza ona bowiem porządek, w którym mogą wystąpić cykle (czyli tak naprawdę nie jest to porządek!). Przykład tej sytuacji przedstawiony jest na rysunku 8.

Jeśli tylko dostrzeżemy powyższy problem, to łatwo sobie z nim poradzić. Wystarczy w pierwszej kolejności zbadać, do której połowy układu współrzędnych należą rozważane punkty P_j i P_k. Jeśli oba należą do górnej lub oba należą do dolnej, to do ich porównania

[3]Dokładniejszy opis takiego testu można znaleźć w książce [20].



Rys. 7: Punkt P_j znajduje się w porządku kątowym względem środka P_i przed punktem P_k, gdyż idąc ścieżką $P_j \to P_i \to P_k$, skręcamy w prawo.



Rys. 8: Zgodnie z regułą, P_j jest przed P_k, P_k przed P_l, a P_l przed P_j, czyli rzeczywiście mamy cykl.

możemy zastosować iloczyn wektorowy. Jeśli natomiast należą do różnych półpłaszczyzn, to od razu decydujemy, że punkt z górnej półpłaszczyzny jest wcześniejszy w porządku od punktu z dolnej półpłaszczyzny.

To nie koniec pułapek kryjących się w sortowaniu kątowym. Kolejnym przypadkiem, w którym bardzo łatwo o błąd, są punkty leżące na osi poziomej, czyli na granicy półpłaszczyzny górnej i dolnej. By uniknąć problemów z określeniem ich położenia i porządku, najlepiej punkty postaci $(x,0)$ dla $x \geqslant 0$ zaliczyć do górnej półpłaszczyzny, natomiast pozostałe — do dolnej.

W ogólnym przypadku trzeba jeszcze zwrócić uwagę, w jakiej kolejności powinny występować punkty leżące na tej samej półprostej wychodzącej ze środka sortowania. Na szczęście w naszym przypadku nie jest to istotne.

Liczne przykłady implementacji sortowania kątowego można znaleźć w podręcznikach i innych opracowaniach. Tutaj ją pominiemy, polecając ją Czytelnikowi jako ćwiczenie. Z racji mnożących się pułapek i trudności w wychwyceniu ewentualnych błędów jest to ćwiczenie *bardzo pouczające*!

Złożoność rozwiązania

Najwyższa pora oszacować złożoność przedstawionych rozwiązań i ocenić ich efektywność. Posortowanie kątowe n punktów ma złożoność czasową taką samą, jak i każde inne sortowanie, gdyż potrafimy już porównać dwa punkty w czasie stałym. Stąd etap ten wymaga czasu $O(n \log n)$, oczywiście o ile tylko zastosujemy jakiś efektywny algorytm sortowania, np. przez scalanie. Koszt czasowy pojedynczego zamiatania kątowego jest, jak łatwo zauważyć, liniowy względem n. Złożoność czasowa całego algorytmu, który składa się z n takich faz, to zatem $O(n^2 \log n)$.

Czy jest to najefektywniejsza metoda rozwiązania naszego problemu? Okazuje się, że nie. Otóż istnieje bardzo skomplikowany algorytm jednoczesnego sortowania kątowego względem wszystkich punktów o złożoności czasowej $O(n^2)$ [4]. Nie był on jednakże brany pod uwagę przy sprawdzaniu efektywności rozwiązań niniejszego zadania, właśnie ze względu na duże skomplikowanie, ale także i niewielką różnicę rzędu złożoności czasowej w stosunku do rozwiązania wzorcowego.

Warto jednakże wspomnieć o ciekawym, a prostym pomyśle na usprawnienie opisanych rozwiązań wzorcowych. Nie poprawia on co prawda rzędu złożoności czasowej, ale polepsza stałą, co częściowo (bądź całkowicie) zapobiega wielokrotnemu wyliczaniu pól wszystkich trójkątów. Otóż po rozważeniu danego punktu jako środka sortowania P_i, można ten punkt po prostu wyrzucić ze zbioru Z. Dzięki temu, w kolejnych iteracjach maleje liczba punktów koniecznych do rozważenia, a trójkąty postaci $\triangle P_i P_j P_k$ dla $i < j < k$ są liczone tylko w iteracji algorytmu odpowiadającej P_i. To oznacza, że w przypadku metod I i II, pole każdego trójkąta jest w nowej wersji algorytmu wliczane zaledwie dwukrotnie, natomiast w przypadku metody III jest uwzględniane dokładnie raz!

Implementacje i inne rozwiązania

W rozwiązaniach wzorcowych zostały zaimplementowane metody oparte o dwa z opisanych sposobów wyznaczania pola trójkąta. W plikach `tro.cpp`, `tro0.pas` i `tro2.java` znajdują się implementacje rozwiązania wzorcowego oparte o metodę II, natomiast w pliku `tro1.java` — oparte o metodę III. Wobec wcześniejszych uwag na temat arytmetyki zmiennoprzecinkowej, nie powinno nikogo dziwić, że rozwiązanie oparte o metodę I nie zostało w ogóle zaimplementowane. W metodzie tej bowiem wielokrotnie zachodzi potrzeba operowania na liczbach rzeczywistych (pojawiają się w nim ułamki — na przykład przy wyznaczaniu parametrów prostej A, B i C — czy pierwiastki kwadratowe), co może powodować problemy z dokładnością.

Ciekawym spostrzeżeniem dotyczącym obliczeń w metodach II i III, o którym dotychczas nie wspominaliśmy, jest postać liczbowa rozważanych tam wag. Okazuje się, że pole trójkąta lub trapezu o wierzchołkach w punktach o całkowitoliczbowych współrzędnych musi być całkowitą wielokrotnością $\frac{1}{2}$. Dzięki temu, wszystkie obliczenia w rozwiązaniach opartych o metody II oraz III mogą być wykonywane na liczbach całkowitych, a dopiero na samym końcu trzeba wynik podzielić przez 2. To spostrzeżenie wyjaśnia także, dlaczego w zadaniu wystarczy wypisać wynik z dokładnością do zaledwie jednej cyfry po przecinku.

[4] Więcej szczegółów można znaleźć w artykule [35].

Testy

Rozwiązania zawodników były sprawdzane na 11 zestawach testów. Rozwiązania o złożoności $O(n^3)$ mieściły się w limicie czasowym tylko dla testów 1–3.

Nazwa	n	Opis
tro1a.in	10	prosty test poprawnościowy, losowy
tro1b.in	3	test poprawnościowy w kształcie małego trójkąta, odpowiedź niecałkowita
tro2a.in	10	prosty test poprawnościowy, dużo punktów na jednej prostej
tro2b.in	30	prosty test poprawnościowy, odpowiedź to 0.0
tro2c.in	5	prosty test poprawnościowy
tro2d.in	3	test poprawnościowy w kształcie małego trójkąta, odpowiedź niecałkowita
tro3a.in	20	prosty test poprawnościowy, losowy
tro3b.in	100	mały test w postaci prostokąta punktów kratowych, dużo trójek współliniowych punktów
tro3c.in	3	test poprawnościowy w kształcie małego trójkąta, odpowiedź niecałkowita
tro4.in	1 300	test średniej wielkości, losowy
tro5.in	1 500	test średniej wielkości, losowy
tro6.in	1 750	test duży, losowy
tro7.in	1 900	test duży, losowy
tro8a.in	2 000	test duży, losowy
tro8b.in	1 500	średni test w postaci prostokąta punktów kratowych, dużo trójek współliniowych punktów
tro9a.in	2 500	test duży, losowy
tro9b.in	2 000	duży test w postaci prostokąta punktów kratowych, dużo trójek współliniowych punktów
tro10.in	3 000	test duży, losowy
tro11a.in	3 000	test duży, losowy
tro11b.in	3 000	duży test w postaci prostokąta punktów kratowych, dużo trójek współliniowych punktów

Marcin Kubica
Treść zadania, Opracowanie

Jakub Pawlewicz
Program

OI, Etap III, dzień pierwszy, 02.04.2008

Kupno gruntu

Bajtazar planuje kupno działki przemysłowej. Jego majątek jest wyceniany na k bajtalarów. Tyle też zamierza on przeznaczyć na zakup gruntu. Jednak znalezienie działki, która kosztuje dokładnie k bajtalarów, jest kłopotliwe. W związku z tym, Bajtazar jest gotów kupić ewentualnie droższą działkę. Dodatkowe fundusze może uzyskać przez zaciągnięcie kredytu. Maksymalny rozmiar kredytu, jaki może mu udzielić Bajtocki Bank Kredytowy, wynosi tyle, ile majątek Bajtazara, czyli k bajtalarów. Innymi słowy, Bajtazar chciałby przeznaczyć na kupno działki kwotę w wysokości od k bajtalarów do 2k bajtalarów włącznie.

Teren, na którym Bajtazar zamierza kupić działkę, ma kształt kwadratu o boku długości n metrów. Aktualni właściciele ziemi wyznaczyli różne ceny w przeliczeniu na metr kwadratowy. Bajtazar przeprowadził dokładny wywiad i sporządził mapę cenową tego terenu. Mapa ta opisuje cenę każdego kwadratu o rozmiarze metr na metr. Takich kwadratów jest dokładnie n^2. Teraz pozostaje wyznaczyć wymarzoną działkę. Musi ona mieć kształt prostokąta, złożonego wyłącznie z całych kwadratów jednostkowych. Bajtazar zaczął szukać na mapie odpowiedniej działki, ale mimo wzmożonych wysiłków nie był w stanie znaleźć właściwego prostokąta. Pomóż Bajtazarowi.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia liczby k i n oraz mapę cenową terenu,*
- *wyznaczy działkę o cenie z przedziału $[k, 2k]$ lub stwierdzi, że taka działka nie istnieje,*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu wejścia znajdują się dwie liczby całkowite k i n oddzielone pojedynczym odstępem, $1 \leqslant k \leqslant 1\,000\,000\,000$, $1 \leqslant n \leqslant 2\,000$. Każdy z następnych n wierszy zawiera n liczb całkowitych nieujemnych, pooddzielanych pojedynczymi odstępami. Liczba i-ta w wierszu numer $j+1$ określa cenę jednego kwadratu metr na metr, którego współrzędne w terenie to (i, j). Cena jednego metra nie przekracza 2 000 000 000 bajtalarów.

Wyjście

Jeżeli nie istnieje działka o cenie z przedziału $[k, 2k]$, to program powinien wypisać jeden wiersz zawierający słowo `NIE`*. W przeciwnym przypadku powinien wypisać jeden wiersz zawierający cztery liczby całkowite dodatnie x_1, y_1, x_2, y_2 pooddzielane pojedynczymi odstępami, określające współrzędne prostokąta. Para (x_1, y_1) oznacza lewy górny róg prostokąta, a para (x_2, y_2) — prawy dolny róg prostokąta. Wtedy taki prostokąt określony jest przez zbiór*

współrzędnych kwadratów: $\{(x,y) \mid x_1 \leqslant x \leqslant x_2 \text{ i } y_1 \leqslant y \leqslant y_2\}$. *Suma cen kwadratów c leżących wewnątrz wskazanego prostokąta powinna spełniać nierówności* $k \leqslant c \leqslant 2k$. *Jeżeli jest wiele działek spełniających wymagany warunek, to należy wypisać dowolną z nich.*

Przykład

Dla danych wejściowych:

```
4 3
1 1 1
1 9 1
1 1 1
```

poprawnym wynikiem jest:

```
NIE
```

a dla danych wejściowych:

```
8 4
1 2 1 3
25 1 2 1
4 20 3 3
3 30 12 2
```

poprawnym wynikiem jest:

```
2 1 4 2
```

1	*2*	*1*	*3*
25	*1*	*2*	*1*
4	*20*	*3*	*3*
3	*30*	*12*	*2*

Cenowa mapa terenu i wyznaczona działka w drugim przykładzie

Rozwiązanie

Rozwiązania nieoptymalne

Dla uproszczenia, kwadraty jednostkowe, na które podzielona jest mapa, będziemy nazywać *polami*, a sumę cen pól zawartych w prostokącie — *ceną prostokąta*. Zadanie polega na znalezieniu prostokąta, którego cena mieści się w przedziale od k do $2k$.

Proste rozwiązania

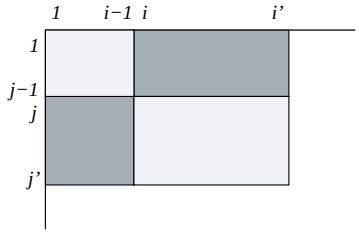
Najprostsze rozwiązanie polega na przejrzeniu wszystkich możliwych prostokątów i wyznaczeniu ceny każdego z nich przez proste zsumowanie cen pól. Jednak wszystkich prostokątów jest $O(n^4)$ i każdy zawiera średnio $O(n^2)$ pól. Prosty program złożony z sześciu zagnieżdżonych pętli **for** działa więc w czasie rzędu $O(n^6)$. Rozwiązanie takie zostało zaimplementowane w programach `kups3.cpp` i `kups6.java`.

W poszukiwaniu efektywniejszego podejścia możemy zajrzeć do różnych archiwów zadań — także z Olimpiady Informatycznej. Można znaleźć w nich wiele problemów opartych na schemacie podobnym jak „Kupno gruntu". Zazwyczaj jest w nich dana prostokątna „mapa" podzielona na jednostkowe pola, którym są przypisane pewne nieujemne

wagi, i należy znaleźć prostokąt o sumarycznej wadze spełniającej określone warunki (patrz, na przykład, zadania „Artemis" z XVI IOI [12] oraz „Piramida" z XVIII IOI [14]). Istnieją pewne standardowe techniki, które można zastosować przy rozwiązywaniu tego typu zadań. Jedna z nich polega na wstępnym przetworzeniu mapy tak, aby potem móc w stałym czasie określać dla dowolnego prostokąta sumę wag zawartych w nim pól. (Została ona opisana także w zadaniu „Mapa gęstości" w [8].)

Oznaczmy przez $c_{i,j}$ cenę pola o współrzędnych (i, j), a przez $K_{i,j}^{i',j'}$ cenę prostokąta, którego lewy górny róg ma współrzędne (i, j), a prawy dolny (i', j'):

$$K_{i,j}^{i',j'} = \sum_{x=i}^{i'} \sum_{y=j}^{j'} c_{x,y}.$$



Rys. 1: Wyznaczanie ceny prostokąta

Jeżeli znamy wartości $K_{1,1}^{x,y}$ dla wszystkich $x = 1, 2, \ldots, n$, $y = 1, 2, \ldots, n$, to możemy łatwo wyznaczyć każdą inną wartość $K_{i,j}^{i',j'}$, korzystając z tożsamości (patrz też rys. 1):

$$K_{i,j}^{i',j'} = K_{1,1}^{i',j'} - K_{1,1}^{i-1,j'} - K_{1,1}^{i',j-1} + K_{1,1}^{i-1,j-1}.$$

Z kolei, aby wyznaczyć wartości $K_{1,1}^{x,y}$, wystarczy przejrzeć kolejne wiersze mapy, idąc z góry na dół, i skorzystać z tożsamości:

$$K_{1,1}^{x,y} = K_{1,1}^{x-1,y} + K_{1,1}^{x,y-1} - K_{1,1}^{x-1,y-1} + c_{x,y}$$

lub sumować na bieżąco wagi pól w danym wierszu i skorzystać z tożsamości:

$$K_{1,1}^{x,y} = K_{1,1}^{x,y-1} + (c_{1,y} + c_{2,y} + \cdots + c_{x,y}).$$

Widzimy więc, że możemy wyznaczyć wszystkie wartości $K_{1,1}^{x,y}$ w czasie i pamięci $O(n^2)$. Mając te wartości, wystarczy przejrzeć wszystkie możliwe prostokąty (jest ich $O(n^4)$) i sprawdzić ich ceny (każdą obliczamy w czasie $O(1)$). W ten sposób dostajemy algorytm działający w czasie rzędu $O(n^4)$ i pamięci rzędu $O(n^2)$. Jest on zaimplementowany w plikach `kups2.cpp` i `kups5.java`.

Poszukiwanie w przedziale

Jak dotąd przedstawiliśmy rozwiązania, w których analizujemy ceny wszystkich prostokątów. Nie wykorzystaliśmy faktu, że szukana suma musi być liczbą z podanego przedziału — najwyższa pora uwzględnić ten fakt.

Obserwacja 1. Załóżmy, że mamy dwa prostokąty: jeden większy, a drugi mniejszy, zawarty w większym. Ponieważ ceny pól są nieujemne, więc cena większego prostokąta nie może być mniejsza niż cena mniejszego prostokąta. Inaczej mówiąc, jeśli $1 \leqslant i_1 \leqslant i_2 \leqslant i_3 \leqslant i_4 \leqslant n$, oraz $1 \leqslant j_1 \leqslant j_2 \leqslant j_3 \leqslant j_4 \leqslant n$, to:

$$K_{i_2,j_2}^{i_3,j_3} \leqslant K_{i_1,j_1}^{i_4,j_4}.$$

Korzystając z tej obserwacji, możemy pominąć niektóre prostokąty. Jeśli trafimy na prostokąt o cenie większej niż $2k$, to możemy pominąć wszystkie większe, zawierające go prostokąty. Analogicznie, jeśli mamy prostokąt o cenie mniejszej niż k, to możemy pominąć wszystkie zawarte w nim prostokąty.

Ustalmy na chwilę pewne wartości współrzędnych pionowych $1 \leqslant j \leqslant j' \leqslant n$. Możemy przejrzeć wszystkie prostokąty o górnej krawędzi na wysokości j i dolnej na wysokości j' tzw. metodą „gąsienicy". Prostokąty rozważamy w kolejności od lewej do prawej, przy czym, jak zobaczymy, wystarczy, że sprawdzimy tylko niektóre z nich. Zaczynamy od prostokąta o lewym górnym rogu w $(1, j)$ i prawym dolnym rogu w $(1, j')$.

Powiedzmy, że lewy górny róg aktualnie rozważanego prostokąta ma współrzędne (i, j), a prawy dolny — (i', j'). W zależności od wartości $K_{i,j}^{i',j'}$ rozróżniamy trzy przypadki:

- Jeśli $K_{i,j}^{i',j'} < k$, to znaczy, że ani nasz prostokąt, ani żaden w nim zawarty nie spełnia warunków zadania. W szczególności, możemy pominąć prostokąty o prawym dolnym rogu w (i', j') i lewym górnym na prawo od (i, j). Możemy więc powiększyć nasz prostokąt, zwiększając i' o 1.

- Jeśli $K_{i,j}^{i',j'} > 2k$, to znaczy, że ani nasz prostokąt, ani żaden zawierający go większy prostokąt nie spełniają warunków zadania. W szczególności, możemy pominąć prostokąty o lewym górnym rogu w (i, j) i prawym dolnym na prawo od (i', j'). Możemy więc zmniejszyć nasz prostokąt, zwiększając i o 1.

- Jeżeli żaden z poprzednich przypadków nie zachodzi, to znaczy, że $k \leqslant K_{i,j}^{i',j'} \leqslant 2k$ i nasz prostokąt spełnia warunki zadania.

Badany prostokąt jest raz węższy, raz szerszy — jak gąsienica pełznąca z lewa na prawo. Ile kroków może wykonać taka „gąsienica"? Co najwyżej $2n$, gdyż w każdym kroku zwiększamy o 1 współrzędną lewego lub prawego boku prostokąta. Jeżeli weźmiemy pod uwagę, że możliwych par j i j' jest $O(n^2)$, otrzymujemy algorytm działający w czasie $O(n^3)$ i pamięci $O(n^2)$. Rozwiązanie takie jest zaimplementowane w plikach `kups1.cpp` i `kups4.java`.

Opisany algorytm możemy stosować wszędzie tam, gdzie cena szukanego prostokąta powinna należeć do zadanego przedziału. Jednak w naszym przypadku nie jest to dowolny przedział, ale przedział postaci: od k do $2k$. Wykorzystując ten fakt, można skonstruować jeszcze szybsze rozwiązanie.

Rozwiązanie optymalne

Zacznijmy od kilku prostych obserwacji.

Obserwacja 2. Jeżeli, dla pewnych i i j, $c_{i,j} > 2k$, to pole (i, j) nie może być zawarte w żadnym prostokącie spełniającym warunki zadania.

Jest to dosyć oczywisty fakt: cena dowolnego prostokąta zawierającego pole (i, j) musi być większa niż $2k$. W związku z tym pola o cenach większych niż $2k$ nazwiemy *zabronionymi*. Pozostałe pola nazwiemy *dozwolonymi*. Cały prostokąt nazwiemy *dozwolonym*, jeśli wszystkie pola w nim zawarte są dozwolone. Dla uproszczenia przyjmiemy, że wszystkie pola leżące poza mapą są zabronione.

Fakt 1. *Jeżeli prostokąt jest dozwolony i jego cena jest nie mniejsza niż k, to istnieje zawarty w nim prostokąt, którego cena jest między k a 2k.*

Dowód: Dowód przeprowadzimy przez indukcję ze względu na wielkość prostokąta. Oznaczmy przez P dozwolony prostokąt o cenie $S \geqslant k$.

1. Jeśli prostokąt P składa się tylko z jednego pola, to musi ono być dozwolone, czyli $S \leqslant 2k$. Tym samym P stanowi szukane rozwiązanie.

2. Załóżmy, że prostokąt P składa się z więcej niż jednego pola. Jeżeli $S \leqslant 2k$, to stanowi on szukane rozwiązanie.

   Załóżmy przeciwnie, że $S > 2k$. Wówczas można podzielić P, w pionie lub w poziomie, na dwa mniejsze prostokąty P_1 i P_2. Oznaczmy ich ceny, odpowiednio, przez S_1 i S_2. Bez straty ogólności możemy założyć, że $S_1 \leqslant S_2$. Mamy więc $S_1 + S_2 = S > 2k$. Tak więc $S_2 > k$. Z założenia indukcyjnego, prostokąt P_2 zawiera szukane rozwiązanie.

■

Pokazany fakt pozwala nam zredukować problem z zadania do znalezienia *prostokąta dozwolonego o maksymalnej cenie*:

- Jeśli jego cena jest mniejsza od k, to prostokąt będący rozwiązaniem nie istnieje.
- Jeśli zaś jego cena jest większa lub równa k, to zawiera on szukany prostokąt. Dowód Faktu 1 stanowi jednocześnie przepis na konstrukcję rozwiązania zadania w obrębie znalezionego maksymalnego prostokąta. Jeżeli w każdym kroku będziemy dzielić dany prostokąt mniej więcej w połowie, to po co najwyżej $O(\log n)$ krokach otrzymamy rozwiązanie.

Maksymalny dozwolony prostokąt

Zastanówmy się, jak znaleźć prostokąt dozwolony o maksymalnej cenie. Jak w przypadku rozwiązań nieoptymalnych, okazuje się, że pomocne może być poszperanie w olimpijskich archiwach — rozwiązanie tego problemu zostało opisane w rozdziale „Działka” w [9]. Opiszemy je jednak również tutaj.

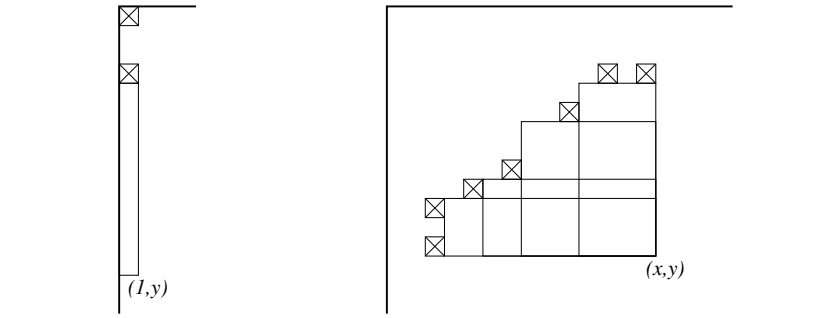
Rozwiązanie polega na sukcesywnym przeglądaniu prostokątów dozwolonych, które mogą być maksymalne ze względu na cenę. Zauważmy, że pewne prostokąty możemy pominąć — jeśli mamy dwa dozwolone prostokąty A i B, gdzie A zawiera się w B, to wystarczy rozważyć B (A nie może mieć ceny większej niż B). Przypomnijmy, że naszą mapę

zorientowaliśmy tak, że lewy górny róg ma współrzędne $(0,0)$, pierwsza współrzędna (x) rośnie w prawo wzdłuż osi poziomej, a druga (y) — w dół wzdłuż osi pionowej. Oznaczmy przez $M_{x,y}$ zbiór lewych górnych rogów prostokątów dozwolonych, których prawy dolny róg ma współrzędne (x,y). Dodatkowo będziemy wymagać, by prostokątów umieszczonych w zbiorze $M_{x,y}$ nie dało się rozszerzyć w górę lub w lewo do prostokąta dozwolonego — takie prostokąty nazwiemy *nierozszerzalnymi*. Punkty należące do zbiorów $M_{x,y}$ będziemy utożsamiać z prostokątami, które one (wraz z punktem (x,y)) wyznaczają, i dla uproszczenia będziemy mówić o prostokątach należących do $M_{x,y}$.

Oczywiście dla każdego pola zabronionego (x,y) (w tym dla każdego pola spoza mapy) mamy $M_{x,y} = \emptyset$. Rozważmy pole dozwolone (x,y). Jeśli $x = 1$, to:

$$M_{1,y} = \{(1, \max\{j : 0 \leqslant j \leqslant y,\ (1,j) \text{ jest zabronione}\} + 1)\}.$$

Dla $x > 1$ zbiór $M_{x,y}$ składa się z prostokątów o takich lewych górnych rogach (x',y'), że zarówno wśród $(x',y'-1),(x'+1,y'-1),\ldots,(x,y'-1)$, jak i wśród $(x'-1,y'),(x'-1,y'+1),\ldots,(x'-1,y)$ muszą występować pola zabronione. Ilustruje to rysunek 2.



Rys. 2: Prostokąty tworzące zbiory $M_{1,y}$ i $M_{x,y}$. Pola zabronione oznaczono przekreślonymi kwadracikami.

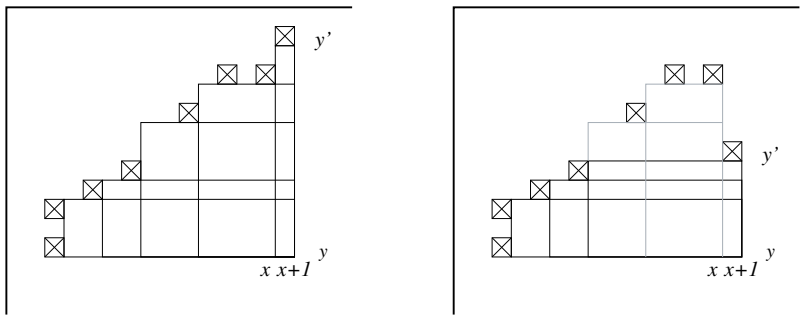
Niech $M_{x,y} = \{(x_1,y_1),(x_2,y_2),\ldots,(x_k,y_k)\}$, przy czym $x_1 \leqslant x_2 \leqslant \cdots \leqslant x_k$. Z tego, że prostokąty ze zbioru $M_{x,y}$ są nierozszerzalne, wynika, że $x_1 < x_2 < \cdots < x_k$ oraz $y_1 > y_2 > \cdots > y_k$. Zastanówmy się teraz, jak zmieni się zbiór $M_{x,y}$, gdy zwiększymy x o jeden, czyli jak mają się do siebie $M_{x,y}$ i $M_{x+1,y}$? Zależy to od tego, na jakiej wysokości znajduje się najniższe pole zabronione leżące nad polem $(x+1,y)$ — oznaczmy je przez $(x+1,y')$:

$$y' = \max\{j : 0 \leqslant j \leqslant y,\ (x+1,j) \text{ jest zabronione}\}.$$

Możliwe są cztery przypadki (dwa najważniejsze z nich przedstawiono na rys. 3):

- jeśli $y'+1 < y_k$, to $M_{x+1,y} = M_{x,y} \cup \{(x+1,y'+1)\}$,
- jeśli $y'+1 = y_k$, to $M_{x+1,y} = M_{x,y}$,
- jeśli $y_k < y'+1 \leqslant y$, to:

$$M_{x+1,y} = \{(x_i,y_i) : y_i > y', i = 1,2,\ldots,k\} \cup \{(\min\{x_i : 1 \leqslant i \leqslant k,\ y_i \leqslant y'+1\}, y'+1)\},$$



Rys. 3: Zależność między $M_{x,y}$ i $M_{x+1,y}$.

- jeśli $y' = y$, to $M_{x+1,y} = \emptyset$.

Zbiory $M_{x,y}$ konstruujemy wiersz po wierszu, dla kolejnych wartości y. Zauważmy, że poszukując prostokąta dozwolonego o maksymalnej cenie, nie musimy brać pod uwagę wszystkich prostokątów ze zbioru $M_{x,y}$. Jeśli punkt $(x',y') \in M_{x,y} \cap M_{x+1,y}$, to wyznaczony przez niego prostokąt ze zbioru $M_{x,y}$ zawiera się w całości w wyznaczonym przez ten punkt prostokącie ze zbioru $M_{x+1,y}$. Jeśli więc będziemy konstruować kolejne zbiory $M_{x,y}$ (dla danego y) w kolejności od lewej do prawej (dla $x = 1,2,\ldots$), to wystarczy, że będziemy uwzględniać tylko ceny prostokątów o prawym dolnym rogu w (x,y) i lewych górnych rogach ze zbioru $M_{x,y} \setminus M_{x+1,y}$ (czyli „odpadających" przy przejściu od zbioru $M_{x,y}$ do zbioru $M_{x+1,y}$).

Implementacja

Zastanówmy się, jakiej struktury danych użyć do reprezentowania zbiorów $M_{x,y}$, tak aby móc je efektywnie przeglądać. Zauważmy, że przekształcając $M_{x,y}$ w $M_{x+1,y}$, usuwamy pewną liczbę najwyżej położonych punktów i ewentualnie dodajemy jeden, położony wyżej niż wszystkie pozostałe. Odpowiednią strukturą będzie więc stos — pola $(x_1,y_1),(x_2,y_2),\ldots,(x_k,y_k)$ będą na nim ułożone w kolejności od spodu do wierzchu. Oto pseudokod programu, który znajduje prostokąt dozwolony o maksymalnej cenie:

```
1:  s := −1;  { maksymalna dotychczas uzyskana cena }
2:  w := NULL;  { najlepszy znaleziony prostokąt dozwolony }
3:  for y := 1 to n do begin
4:    M := ∅;  { stos }
5:    for x := 1 to n+1 do begin
6:      y' := max{j : 0 ⩽ j ⩽ y, (x, j) jest zabronione};
7:      x' := x;
8:      while (M ≠ ∅) and (M.top.y < y' + 1) do begin
9:        if K^{x−1,y}_{M.top} > s then begin
10:         s := K^{x−1,y}_{M.top};
11:         w := prostokąt o rogach w M.top i (x − 1, y);
12:       end;
13:       x' := M.top.x;
```

```
14:          M.pop;
15:        end;
16:        if (M = ∅) ∨ (M.top.y > y' + 1) then M.push(x', y' + 1);
17:      end;
18:    end;
19:    return w;
```

Złożoność rozwiązania

Jaka jest złożoność czasowa przedstawionego rozwiązania? Zawartość wewnętrznej pętli **for** (linie 6–16) jest powtarzana $O(n^2)$ razy. Pokażemy, że jest ona wykonywana w stałym czasie (zamortyzowanym). Musimy przyjrzeć się dokładniej dwóm instrukcjom: obliczeniu wartości y' (linia 6) i pętli **while** (linie 8–15).

Linia 6 — maksimum. Nie musimy obliczać najbliższych pól zabronionych za każdym razem od nowa. Dysponując wartościami wyznaczonymi dla poprzedniego wiersza $(y-1)$, wystarczy poprawić je jedynie w tych kolumnach, gdzie w wierszu y pojawiło się pole zabronione. W ten sposób, potrzebne maksimum obliczamy tak naprawdę w czasie stałym.

Linie 8–15 — pętla while. Zauważmy, że w każdym kroku pętli ze stosu jest zdejmowany jeden element. Jednak elementy są wkładane na stos poza pętlą **while**, a każdy włożony element może być zdjęty tylko raz. To oznacza, że sumaryczna liczba wykonań tej pętli w czasie rozważania jednego wiersza jest rzędu $O(n)$, czyli pętla jest wykonywana w stałym czasie zamortyzowanym.

Widzimy więc, że przedstawiona procedura działa w czasie $O(n^2)$. Dodatkowe operacje, pozwalające na podstawie prostokąta dozwolonego o maksymalnej cenie wyznaczyć poszukiwaną działkę o cenie z przedziału $[k,2k]$, zajmują czas $O(n^2+\log n)$. Tak więc, cały algorytm działa w czasie $O(n^2)$ i pamięci $O(n)$. Jest to optymalne rozwiązanie, gdyż samo wczytanie danych wymaga wykonania n^2 operacji. Zostało ono zaimplementowane w plikach `kup.cpp` i `kup1.java`.

Testy

Testy zostały tak dobrane, że programy o złożoności czasowej $O(n^6)$ nie dostają żadnych punktów. Rozwiązania działające w czasie $O(n^4)$ uzyskują 20 %, a działające w czasie $O(n^3)$ — 40 % punktów.
W poniższej tabeli przedstawiono charakterystykę testów.

Nazwa	n	Opis
kup0 .in	3	test przykładowy, brak rozwiązania
kup0a.in	4	test przykładowy
kup1a.in	5	test losowy, poprawnościowy
kup1b.in	50	test strukturalny, wydajnościowy

Nazwa	n	Opis
kup1c.in	10	test losowy, poprawnościowy, brak rozwiązania
kup2a.in	15	test strukturalny, poprawnościowy
kup2b.in	60	test losowy, wydajnościowy i poprawnościowy
kup3a.in	200	test losowy, poprawnościowy
kup3b.in	200	test losowy, wydajnościowy i poprawnościowy
kup4a.in	500	test losowy, poprawnościowy
kup4b.in	500	test losowy, wydajnościowy i poprawnościowy
kup5 .in	1000	test losowy, wydajnościowy i poprawnościowy
kup6 .in	2000	test losowy, wydajnościowy i poprawnościowy
kup7 .in	500	test z polami zabronionymi na obu przekątnych, wydajnościowy
kup8 .in	2000	test z polami zabronionymi na obu przekątnych, wydajnościowy
kup9a.in	500	test z polami zabronionymi w kształcie konturu „karo”, wydajnościowy
kup9b.in	500	test losowy, wydajnościowy, brak rozwiązania
kup10a.in	2000	test z polami zabronionymi w kształcie konturu „karo”, wydajnościowy
kup10b.in	2000	test losowy, wydajnościowy, brak rozwiązania

Marcin Kubica
Treść zadania, Opracowanie

Jakub Radoszewski
Program

OI, Etap III, dzień drugi, 03.04.2008

Stacja

W Bajtocji zakończył się pierwszy etap reformy[1] *sieci kolejowej. Owa sieć składa się z dwukierunkowych odcinków torów łączących stacje kolejowe. Żadne dwie stacje nie są połączone więcej niż jednym odcinkiem torów. Ponadto wiadomo, że z każdej stacji kolejowej da się dojechać do każdej innej dokładnie jedną trasą. Trasa może być złożona z kilku odcinków torów, ale nigdy nie przechodzi przez żadną stację więcej niż raz.*

Celem drugiego etapu reformy jest zaplanowanie połączeń kolejowych. Bajtazar liczy na to, że mu w tym pomożesz. Aby uprościć zadanie, Bajtazar postanowił, że:

- *jedna ze stacji stanie się wielkim węzłem kolejowym i otrzyma nazwę Bitowice,*
- *ze wszystkich pozostałych stacji uruchomione zostaną połączenia kolejowe do Bitowic i z powrotem,*
- *każdy pociąg będzie jeździł między Bitowicami i drugą stacją końcową po jedynej możliwej trasie, zatrzymując się na wszystkich mijanych stacjach.*

Pozostaje pytanie o to, która stacja powinna zostać Bitowicami. Postanowiono, że system połączeń powinien być tak zaplanowany, by średni koszt przejazdu między dwiema różnymi stacjami kolejowymi był minimalny. W Bajtocji obowiązują wyłącznie bilety jednorazowe w cenie 1 *bajtalara, upoważniające do przejazdu jednym połączeniem na dowolną odległość. Tak więc koszt przejazdu między dwiema konkretnymi stacjami to minimalna liczba połączeń, jakie trzeba wykorzystać, aby przejechać z jednej stacji do drugiej.*

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis sieci kolejowej w Bajtocji,*
- *wyznaczy stację, która powinna zostać Bitowicami,*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu wejścia zapisana jest jedna liczba całkowita n *(*$2 \leqslant n \leqslant 1\,000\,000$*) — jest to liczba stacji kolejowych. Stacje kolejowe są ponumerowane od* 1 *do* n*. Stacje łączy* $n-1$ *odcinków torów kolejowych. Są one opisane w kolejnych* $n-1$ *wierszach, po jednym w wierszu. W każdym z nich są zapisane dwie dodatnie liczby całkowite* a *oraz* b *(*$1 \leqslant a < b \leqslant n$*), oddzielone pojedynczym odstępem i oznaczające numery stacji, które łączy dany odcinek torów.*

[1] *Został on opisany w zadaniu* **Koleje** *z III etapu XIV OI [14]. Znajomość zadania Koleje nie jest jednak w najmniejszym stopniu potrzebna do rozwiązania niniejszego zadania.*

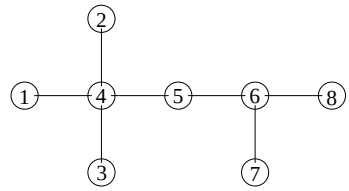
Wyjście

W pierwszym i jedynym wierszu wyjścia Twój program powinien wypisać jedną liczbę całkowitą — optymalną lokalizację stacji Bitowice. Jeżeli istnieje więcej niż jedna optymalna odpowiedź, Twój program powinien wypisać dowolną z nich.

Przykład

Dla danych wejściowych:

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



poprawnym wynikiem jest:

```
7
```

Kółka na rysunku reprezentują stacje (liczby w kółkach to numery stacji), a krawędzie to odcinki torów. Możliwymi optymalnymi lokalizacjami Bitowic są stacje 7 oraz 8. W przypadku wyboru dowolnej z nich, średni koszt przejazdu między różnymi stacjami będzie równy $\frac{36}{28} \approx 1.2857$ *(w przykładzie jest* 28 *par nieuporządkowanych różnych stacji).*

Rozwiązanie

Analiza problemu i pierwsze pomysły na rozwiązanie

Zadanie polega na takim umiejscowieniu Bitowic, aby średni koszt przejazdu między dwiema stacjami był jak najmniejszy. Mamy n stacji, a więc jest $\frac{n \cdot (n-1)}{2}$ (nieuporządkowanych) par różnych stacji. Dla uproszczenia rozważań, zamiast minimalizować średni koszt, możemy minimalizować sumę kosztów przejazdu między wszystkimi parami różnych stacji. Koszt przejazdu między każdymi dwiema stacjami wynosi:

- 1 — jeśli istnieje między nimi bezpośrednie połączenie,
- 2 — jeśli bezpośrednie połączenie nie istnieje, to zawsze można dojechać z jednej stacji do drugiej z jedną przesiadką (np. w Bitowicach).

Naszym celem jest zatem takie umiejscowienie Bitowic, aby było jak najwięcej par stacji, między którymi istnieje bezpośrednie połączenie.

Sieć połączeń kolejowych Bajtocji jest *spójna* (między każdymi dwiema stacjami istnieje połączenie) i mamy $n-1$ odcinków torów łączących n stacji — oznacza to, że sieć połączeń kolejowych tworzy *drzewo*. Wyobraźmy sobie, że za korzeń tego drzewa wybieramy właśnie Bitowice. Po ukorzenieniu drzewa możemy mówić, że jedne wierzchołki są przodkami (lub potomkami) innych. Zauważmy, że koszt przejazdu między dwiema stacjami:

- jest równy 1, gdy jedna ze stacji jest przodkiem drugiej;
- w pozostałych przypadkach koszt wynosi 2.

Algorytm $O(n^4)$. Proste rozwiązanie może więc polegać na sprawdzeniu każdej lokalizacji Bitowic i przeszukaniu drzewa dla każdej pary stacji, w celu określenia ich wzajemnego położenia (czy jedna jest przodkiem drugiej). Niestety, jest to strasznie nieefektywny algorytm, działający w czasie $O(n^4)$. Jego implementacja znajduje się w pliku `stas2.cpp`.

Pierwszy algorytm $O(n^3)$. Oznaczmy przez $\delta(v,w)$ odległość między wierzchołkami v i w. Zauważmy, że w jest przodkiem v wtedy i tylko wtedy, gdy:

$$\delta(v, Bitowice) = \delta(v,w) + \delta(w, Bitowice)$$

Spostrzeżenie to można wykorzystać, konstruując następujący algorytm. Wystarczy raz obliczyć odległości między wszystkimi parami wierzchołków (za pomocą algorytmu Floyda-Warshalla[1], w czasie $O(n^3)$). Następnie dla każdej lokalizacji Bitowic można rozważyć każdą parę stacji, sprawdzając dla nich w czasie stałym powyższą równość. Daje to rozwiązanie działające w czasie $O(n^3)$ i wymagające pamięci rzędu $O(n^2)$, co niestety nadal nie jest akceptowalne przy podanym w zadaniu ograniczeniu na n.

Postawmy problem trochę inaczej

Poszukując dalszych ulepszeń, zastanówmy się, ile jest takich par różnych stacji, że koszt przejazdu między nimi jest równy 1? Zamiast liczyć, jak dotychczas, pary nieuporządkowane, możemy równie dobrze liczyć pary uporządkowane wierzchołków (v,w), gdzie $v \neq w$ i w jest przodkiem v. Ile jest takich par dla ustalonego v? Otóż dokładnie tyle, jaka jest odległość v od korzenia, czyli od Bitowic. Naszym celem jest więc takie umiejscowienie Bitowic, aby:

suma odległości Bitowic od wszystkich wierzchołków była jak największa.

To oznacza, że chcemy zmaksymalizować następującą sumę:

$$\sum_{v=1}^{n} \delta(Bitowice, v).$$

Drugi algorytm $O(n^3)$. Każdy składnik sumy potrafimy obliczyć w czasie $O(n)$, przeszukując sieć połączeń. Niestety, składników jest n, więc dla danego położenia Bitowic wartość sumy zostanie obliczona w czasie $O(n^2)$. Testowanie wszystkich lokalizacji Bitowic, w celu wyznaczenia optymalnej, doprowadza więc do algorytmu działającego w czasie $O(n^3)$ i pamięci $O(n)$. Jego implementacja znajduje się w pliku `stas3.cpp`.

[1] Opis tego algorytmu można znaleźć w podstawowych podręcznikach z algorytmiki, na przykład w [15] i [20].

Algorytm $O(n^2)$. Chwila namysłu pozwala nam wprowadzić do rozwiązania małą poprawkę, dającą w efekcie sporą oszczędność czasu. Żeby wyznaczyć wartość podanej sumy dla ustalonego położenia Bitowic, wystarczy tylko raz przeszukać sieć połączeń (wszerz lub w głąb) i w trakcie tego przeszukiwania wyznaczyć najkrótsze odległości od Bitowic do wszystkich wierzchołków — można to zrobić w czasie liniowym. Sprawdzenie wszystkich możliwych lokalizacji Bitowic możemy w ten sposób wykonać w czasie $O(n^2)$ i pamięci $O(n)$. Takie rozwiązanie zostało zaimplementowane w pliku `stas0.cpp`.

Algorytm $O(n^2)$ z poprawką. Kolejny moment refleksji i dochodzimy do wniosku, że nie musimy sprawdzać wszystkich możliwych lokalizacji Bitowic. Następujący fakt pozwala nam ograniczyć się w rozważaniach tylko do liści.

Fakt 1. *Optymalna lokalizacja Bitowic musi znajdować się w liściu drzewa, które tworzy sieć połączeń.*

Dowód: Załóżmy przeciwnie, że optymalna lokalizacja Bitowic znajduje się w wierzchołku wewnętrznym. Tak więc Bitowice znajdują się w *punkcie artykulacji* — wierzchołku, którego usunięcie podzieliłoby sieć połączeń na kilka (przynajmniej dwie) spójnych składowych. Oznaczmy przez $m_1, m_2, \ldots, m_k$ liczby wierzchołków w tych składowych. Bez straty ogólności możemy przyjąć, że $m_1 \leqslant m_2 \leqslant \cdots \leqslant m_k$. Przesuwając Bitowice o jedną krawędź w stronę pierwszej (tj. najmniejszej) z tych składowych, zwiększylibyśmy sumę odległości z Bitowic do wszystkich pozostałych wierzchołków o $m_2 + m_3 + \cdots + m_k - m_1 + 1 > 0$ (zagadka dla Czytelnika: dlaczego zachodzi ta nierówność?). Jest to jednak sprzeczne z założeniem, że ulokowaliśmy Bitowice optymalnie. ■

Rozwiązania korzystające z tej obserwacji zostały zaimplementowane w plikach `stas1.cpp` i `stas4.pas`. Niestety, liczba liści w drzewie może być liniowa ze względu na liczbę wszystkich wierzchołków. Tak więc czas działania tych rozwiązań nadal musimy szacować na $O(n^2)$.

Kompleksowe badanie lokalizacji opłaca się — algorytm liniowy

Spróbujmy podejść do problemu bardziej kompleksowo i wysnuć wnioski, które można wykorzystać przy szacowaniu jakości wielu lokalizacji Bitowic jednocześnie. Wartością, którą można stosunkowo szybko obliczyć dla wszystkich wierzchołków drzewa (przekonamy się, że w czasie liniowym) i która wystarcza do wyznaczenia optymalnej lokalizacji Bitowic, jest

suma odległości od danego wierzchołka do wszystkich pozostałych.

Wówczas problem sprowadza się do wybrania wierzchołka o największej sumie odległości i uzyskujemy rozwiązanie działające w czasie $O(n)$.

Wyznaczmy na korzeń drzewa stanowiącego sieć połączeń dowolny wierzchołek, na przykład ten o numerze 1. Dwukrotnie przeglądając rekurencyjnie drzewo połączeń,

obliczamy dla każdego wierzchołka poszukiwaną sumę odległości do wszystkich pozostałych wierzchołków.
Najpierw, dla każdego wierzchołka v wyznaczamy:

- l_v — liczbę wierzchołków w poddrzewie o korzeniu w v oraz
- s_v — sumę odległości od v do wierzchołków w poddrzewie o korzeniu w v.

Wartości te obliczamy, idąc w górę drzewa, od liści do korzenia. Dla liści jest to proste:

$$l_v = 1, \;\; s_v = 0.$$

Jeśli v nie jest liściem, to niech $v_1, v_2, \ldots, v_k$ będą jego następnikami (synami). Wówczas wartości $l_{v_1}, \ldots, l_{v_k}$ oraz $s_{v_1}, \ldots, s_{v_k}$ zostały już obliczone i zachodzą zależności:

$$l_v = l_{v_1} + l_{v_2} + \cdots + l_{v_k} + 1$$

oraz

$$\begin{aligned} s_v &= s_{v_1} + l_{v_1} + s_{v_2} + l_{v_2} + \cdots + s_{v_k} + l_{v_k} = \\ &= s_{v_1} + s_{v_2} + \cdots + s_{v_k} + l_v - 1. \end{aligned}$$

Po wyznaczeniu wartości l_v i s_v powtórnie przechodzimy rekurencyjnie drzewo. Idąc od korzenia do liści, obliczamy dla każdego wierzchołka v:

- r_v — sumę odległości od niego do wszystkich pozostałych wierzchołków.

Dla korzenia ($v = 1$) jest łatwo:

$$r_v = s_v.$$

Jeśli v nie jest korzeniem, to niech o będzie poprzednikiem (ojcem) v. Wartość r_o została już obliczona. Zastanówmy się, o ile r_v różni się od r_o? Gdy przesuwamy się z o do v, to odległość do wszystkich wierzchołków w poddrzewie o korzeniu w v zmniejsza się o 1, natomiast odległość do wszystkich pozostałych wierzchołków (wliczając o) zwiększa się o 1. Tak więc

$$r_v - r_o = -l_v + (n - l_v) = n - 2 \cdot l_v$$

czyli

$$r_v = r_o + n - 2 \cdot l_v.$$

Potrafimy więc obliczyć wszystkie wartości r_v, a tym samym wyznaczyć optymalną lokalizację Bitowic, w liniowym czasie i pamięci. Jest to rozwiązanie wzorcowe, którego implementacja znajduje się w plikach `sta.cpp`, `sta0.pas` i `sta1.java`.

Testy

Zostało przygotowanych 10 grup testów. Większość z nich to testy losowe. Są one wystarczające do sprawdzenia zarówno poprawności, jak i wydajności rozwiązań. W każdym z nich drzewo połączeń ma około $\frac{n}{2}$ liści, a wartości wyniku są raczej przypadkowe. Dodatkowo, zadbano, by w testowych sieciach połączeń liczba optymalnych lokalizacji Bitowic była niewielka.

Rozwiązania działające w czasie $O(n^4)$ i $O(n^3)$ otrzymywały 20% punktów. Rozwiązania działające w czasie $O(n^2)$ otrzymywały 40%–50% punktów, w zależności od tego, czy uwzględniały spostrzeżenie przedstawione w Fakcie 1, czy też nie. Rozwiązania działające w czasie liniowym otrzymywały, oczywiście, 100% punktów.

Poniższa tabelka zawiera zbiorcze zestawienie testów. Liczbę stacji oznaczono w niej przez n, a liczbę liści — przez l.

Nazwa	n	l	**Opis**
sta0.in	8	5	test przykładowy z treści zadania
sta1a.in	15	9	test losowy
sta1b.in	2	2	jedna krawędź
sta2.in	100	54	test losowy
sta3.in	700	343	test losowy
sta4.in	2 000	1 000	test losowy
sta5.in	500 000	7	bardzo długa ścieżka z kilkoma dodatkami
sta6.in	40 000	19 963	test losowy
sta7.in	100 000	49 973	test losowy
sta8a.in	250 000	124 919	test losowy
sta8b.in	200 001	100 000	gwiazda o ramionach długości 2
sta9a.in	500 000	250 092	test losowy
sta9b.in	800 001	4	gwiazda o ramionach długości 200 000
sta10a.in	1 000 000	499 693	test losowy
sta10b.in	1 000 000	7	bardzo długa ścieżka z kilkoma dodatkami

Jakub Radoszewski
Treść zadania, Opracowanie

Wojciech Tyczyński
Program

OI, Etap III, dzień drugi, 03.04.2008

Permutacja

Multizbiorem nazywamy obiekt matematyczny podobny do zbioru, w którym jednak ten sam element może występować wielokrotnie. Podobnie jak w przypadku zbioru, także dla multizbioru wszystkie elementy można ustawić w ciąg i to zazwyczaj na wiele sposobów; każde takie ustawienie nazywamy **permutacją** *multizbioru. Dla przykładu, permutacjami multizbioru* $\{1,1,2,3,3,3,7,8\}$ *są między innymi* $(2,3,1,3,3,7,1,8)$ *oraz* $(8,7,3,3,3,2,1,1)$.

Powiemy, że jedna permutacja danego multizbioru jest mniejsza (w porządku leksykograficznym) od drugiej, jeżeli na pierwszej pozycji, na której te permutacje się różnią, pierwsza z nich zawiera element mniejszy niż druga. Wszystkie permutacje **danego multizbioru** *można ponumerować (zaczynając od jedynki) w kolejności od najmniejszej do największej.*

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis permutacji pewnego multizbioru oraz dodatnią liczbę* m,
- *wyznaczy numer wczytanej permutacji w porządku leksykograficznym, a dokładniej jego resztę z dzielenia przez* m,
- *wypisze wynik na standardowe wyjście.*

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby całkowite n *oraz* m $(1 \leqslant n \leqslant 300\,000,\ 2 \leqslant m \leqslant 1\,000\,000\,000)$, *oddzielone pojedynczym odstępem. Oznaczają one odpowiednio liczbę elementów multizbioru oraz ... liczbę* m. *Drugi wiersz wejścia zawiera* n *dodatnich liczb całkowitych* a_i $(1 \leqslant a_i \leqslant 300\,000)$, *pooddzielanych pojedynczymi odstępami i oznaczających kolejne elementy danej permutacji multizbioru.*

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać jedną liczbę całkowitą, oznaczającą resztę z dzielenia przez m *numeru podanej permutacji w porządku leksykograficznym.*

Przykład

Dla danych wejściowych:

```
4 1000
2 1 10 2
```

poprawnym wynikiem jest:

```
5
```

Wszystkie permutacje mniejsze od zadanej to (w kolejności leksykograficznej): $(1,2,2,10)$, $(1,2,10,2)$, $(1,10,2,2)$ oraz $(2,1,2,10)$.

Rozwiązanie

Ile tego jest?

Zadania polega na wyznaczeniu numeru zadanej permutacji $a = a_1, a_2, \ldots, a_n$ pewnego multizbioru w porządku leksykograficznym. Na początek zastanówmy się, ile w ogóle jest różnych permutacji multizbioru $Z = \{a_1, a_2, \ldots, a_n\}$, czyli jak duży może być wynik.

Niech $z = \max\{a_1, \ldots, a_n\}$, wówczas $Z \subseteq \{1, 2, \ldots, z\}$ (w tym zapisie traktujemy Z jako zbiór). Przez $l_1, \ldots, l_z$ oznaczmy krotności (liczby wystąpień) liczb $1, \ldots, z$ w multizbiorze Z. Liczba wszystkich permutacji n-elementowego *zbioru* to $n!$, czyli dla przypomnienia:

$$\begin{aligned} 0! &= 1, \\ n! &= n \cdot (n-1)! \quad \text{dla } n \geqslant 1. \end{aligned}$$

Multizbiór Z możemy przekształcić w zbiór, numerując wystąpienia poszczególnych liczb w Z, czyli zastępując kolejne wystąpienia elementu $i \in \{1, \ldots, z\}$ w Z parami uporządkowanymi:

$$(i,1),\ (i,2),\ \ldots,\ (i,l_i).$$

Utworzony w ten sposób zbiór oznaczmy przez Z'. Liczba permutacji Z' to oczywiście $n!$.

Ilu permutacjom zbioru Z' odpowiada jedna permutacja p multizbioru Z? Wystąpienia liczby 1 w p można ponumerować od 1 do l_1 na $l_1!$ sposobów, co za każdym razem da w rezultacie inną permutację Z'. Podobnie, wystąpienia kolejnych liczb $2, \ldots, z$ można ponumerować odpowiednio na $l_2!$, $\ldots$, $l_z!$ sposobów, co daje łączną liczbę różnych permutacji Z', jakie odpowiadają p, równą $l_1! \cdot l_2! \cdot \ldots \cdot l_z!$. Zatem liczba różnych permutacji multizbioru Z to

$$\frac{n!}{l_1! \cdot l_2! \cdot \ldots \cdot l_z!}. \tag{1}$$

Jak się za to zabrać?

Numer permutacji a w porządku leksykograficznym możemy wyznaczyć wprost — wystarczy przejrzeć wszystkie permutacje i zliczyć te, które są mniejsze od niej. Musimy przy tym zadbać, by żadnej nie policzyć wielokrotnie. Realizacja tego pomysłu wymaga przejrzenia $O(n!)$ permutacji i wykonania dla każdej z nich porównania z a w czasie $O(n)$. Daje to rozwiązanie o złożoności czasowej $O(n \cdot n!)$, które zaimplementowaliśmy w pliku `pers0.cpp`.

Dla $n = 10$ liczba elementarnych operacji takiego prostego algorytmu może wynieść około 30 milionów. To być może jeszcze nie nadużyje cierpliwości sprawdzających, ale dla $n = 11, 12, \ldots$ Rzut oka na ograniczenia z zadania wystarcza, aby zauważyć, że musimy wymyślić coś znacznie, ale to znacznie szybszego.

Poszukiwanie bardziej efektywnego rozwiązania rozpoczniemy od wymyślenia sposobu systematycznego przeanalizowania wszystkich permutacji mniejszych od zadanej permutacji

a. Dowolna permutacja b mniejsza od a jest na pewnym prefiksie (tj. początkowym fragmencie) długości $i-1$ zgodna z a, następnie znajduje się w niej element $b_i < a_i$, po którym następuje całkiem już dowolny ciąg elementów.

Definicja 1. Powiemy, że permutacja b jest *kategorii* i względem permutacji a, jeżeli

$$(\forall j < i)\ (b_j = a_j) \text{ oraz } b_i < a_i.$$

W ten sposób uzasadniliśmy następujący fakt:

Fakt 1. *Każda permutacja multizbioru Z mniejsza leksykograficznie od a należy do dokładnie jednej kategorii $i \in \{1,\ldots,n\}$ względem a.*

Skupmy się więc na odrębnym zliczeniu permutacji należących do każdej z kategorii $1,2,\ldots,n$. Warto przy tym zauważyć jeszcze następujący fakt:

Fakt 2. *Permutacje przynależne do kategorii i możemy utożsamiać z permutacjami kategorii* 1 *względem permutacji*

$$p_i = a_i, a_{i+1}, \ldots, a_n$$

multizbioru

$$Z_i = \{a_i, a_{i+1}, \ldots, a_n\}.$$

Permutacje kategorii i

Niech $l_{i,1}, l_{i,2}, \ldots, l_{i,z}$ oznaczają krotności liczb $1,2,\ldots,z$ w multizbiorze Z_i (dla $1 \leqslant i \leqslant n$). Wówczas każda permutacja b należąca do kategorii 1 względem p_i zaczyna się od pewnego elementu $j < a_i$, takiego że $l_{i,j} > 0$; po nim następuje jakakolwiek permutacja multizbioru $\{a_i, a_{i+1}, \ldots, a_n\} \setminus \{j\}$. Z tego wynika, że liczba wszystkich permutacji b kategorii i względem a wyraża się wzorem

$$K_i = \sum_{j<a_i, l_{i,j}>0} \frac{(n-i)!}{l_{i,1}! \cdot \ldots \cdot l_{i,j-1}! \cdot (l_{i,j}-1)! \cdot l_{i,j+1}! \cdot \ldots \cdot l_{i,z}!}. \tag{2}$$

Delikatnie rzecz ujmując, powyższy wzór na K_i wygląda koszmarnie. Już pobieżna analiza pozwala zauważyć, że wyliczanie sumy $K_1 + K_2 + \ldots + K_n \pmod m$ poprzez bezpośrednie zastosowanie tego wzoru będzie wymagać czasu $\Omega(n \cdot z^2)$. Choć jest już lepiej niż poprzednio, to jednak powinniśmy jeszcze znacznie poprawić tę metodę.

Natychmiastowe usprawnienie otrzymamy, przekształcając nieco wzór (2). Zauważmy mianowicie, że iloczyn z mianownika jest w gruncie rzeczy równy

$$(l_{i,1}! \cdot l_{i,2}! \cdot \ldots \cdot l_{i,z}!)/l_{i,j}.$$

Wykorzystując tę równość, a także wyciągając ułamek niezależny od j przed znak sumy, przekształcamy wzór na K_i do postaci:

$$K_i = \frac{(n-i)!}{l_{i,1}! \cdot \ldots \cdot l_{i,z}!} \cdot \sum_{j<a_i, l_{i,j}>0} l_{i,j}.$$

Zauważmy wreszcie, że w nowej postaci wzoru warunek $l_{i,j} > 0$ jest zbędny, więc ostatecznie uzyskujemy istotnie uproszczony wzór:

$$K_i = \frac{(n-i)!}{l_{i,1}! \cdot \ldots \cdot l_{i,z}!} \cdot \sum_{j=1}^{a_i-1} l_{i,j}. \tag{3}$$

Zliczanie permutacji kategoriami

Rozwiązanie wzorcowe oprzemy na wzorze (3). Na jego podstawie będziemy wyznaczać kolejno wartości $K_1, K_2, \ldots, K_n$. Postaramy się przy tym korzystać zawsze z wcześniej wyznaczonych wartości K_i, gdyż bezpośrednie zastosowanie wzoru nadal wygląda na procedurę zbyt czasochłonną dla podanych w zadaniu ograniczeń. Pierwszym krokiem do sukcesu jest znalezienie związku między wartościami $l_{i,*}$ a $l_{i+1,*}$. Korzystając z zależności $Z_{i+1} = Z_i \setminus \{a_i\}$, widzimy, że jest on bardzo prosty:

$$l_{i+1,k} = \begin{cases} l_{i,k} - 1 & \text{jeżeli } k = a_i, \\ l_{i,k} & \text{w przeciwnym przypadku.} \end{cases} \tag{4}$$

Następnie zauważmy, że wzór (3) składa się z dwóch niezależnych czynników, które możemy wyznaczyć niezależnie: ułamka

$$U_i = \frac{(n-i)!}{l_{i,1}! \cdot \ldots \cdot l_{i,z}!}$$

i sumy

$$S_i = \sum_{j=1}^{a_i-1} l_{i,j}.$$

Ułamki U_i

Zależność (4) między wartościami $l_{i,*}$ a $l_{i+1,*}$ pozwala następująco przekształcić wzór na U_{i+1}:

$$\begin{aligned} U_{i+1} &= \frac{(n-(i+1))!}{l_{i+1,1}! \cdot \ldots \cdot l_{i+1,z}!} \\ &= \frac{(n-i)!/(n-i-1)}{l_{i,1}! \cdot \ldots \cdot l_{i,z}!/l_{i,a_i}} \\ &= \frac{l_{i,a_i}}{n-i-1} \cdot U_i \end{aligned}$$

Po tym przekształceniu widzimy, że wystarczy tylko U_1 obliczyć wprost ze wzoru:

$$U_1 = \frac{(n-1)!}{l_{1,1}! \cdot \ldots \cdot l_{1,z}!} = \frac{(n-1)!}{l_1! \cdot \ldots \cdot l_z!},$$

po czym każde kolejne U_{i+1} możemy policzyć, mnożąc U_i przez $l_{i,a_i}/(n-i-1)$. Pomijając na razie problem pojawiających się w obliczeniach dużych liczb, daje to już całkiem efektywną metodę, wymagającą wykonania łącznie $O(n+z)$ mnożeń i dzieleń (zauważmy, że $l_1 + \ldots + l_z = n$).

Uwaga 1. Jakkolwiek ułamki U_i mogą być niecałkowite, to iloczyny $S_i \cdot U_i$ już całkowite być muszą, a zatem przy zachowaniu w algorytmie odpowiedniej kolejności mnożeń i dzieleń można przy wszystkich obliczeniach pozostać w dziedzinie liczb całkowitych.

Sumy S_i

Wyznaczanie kolejnych wartości S_i może okazać się prostsze, gdy przyjrzymy się, jakie jest ich znaczenie. Przypomnijmy:

$$Z_i = \{a_i, a_{i+1}, \ldots, a_n\} \text{ oraz } Z_{i+1} = \{a_{i+1}, \ldots, a_n\}.$$

Wartość S_i to liczba elementów mniejszych od a_i w multizbiorze Z_i. Analogicznie S_{i+1} to liczba elementów mniejszych od a_{i+1} w multizbiorze Z_{i+1}. Jeśli umieścimy zbiór Z_i w strukturze, dzięki której łatwo odpowiedzieć na pytanie o liczbę elementów mniejszych od dowolnej zadanej wartości x, to aby wyznaczyć S_{i+1} musimy:

- usunąć element a_i ze zbioru — dostajemy automatycznie strukturę ze zbiorem Z_{i+1};
- zapytać o liczbę elementów mniejszych od a_{i+1} w strukturze.

Do implementacji powyższej struktury świetnie nadaje się statyczne drzewo binarne, zwane *drzewem licznikowym* albo *przedziałowym*. Ma ono z liści, które odpowiadają kolejno (od lewej) wartościom $1, 2, \ldots, z$. Gdy rozważamy zbiór Z_i, to w j-tym liściu mamy wartość $l_{i,j}$, czyli liczbę wystąpień elementu j w zbiorze Z_i. W każdym węźle wewnętrznym v drzewa przechowujemy sumę wartości z wszystkich liści poddrzewa ukorzenionego w v, czyli liczbę elementów ze zbioru $[j_1, j_2] \cap Z_i$, gdzie j_1 jest najmniejszym, a j_2 największym liściem w poddrzewie v[1]. Koszt czasowy konstrukcji, a zarazem rozmiar takiego drzewa dla ciągu z-elementowego to $O(z)$. Natomiast każdą z potrzebnych operacji możemy z jego pomocą wykonać w czasie $O(\log z)$. To daje łączną złożoność czasową wyznaczenia wszystkich sum S_i rzędu $O(z + n \cdot \log z)$.

Działania modulo

Mogłoby się wydawać, że opracowanie efektywnych metod wyznaczania ułamków U_i oraz sum S_i to już koniec zadania. Byłoby tak, gdyby wszystkie liczby, z którymi mamy do czynienia, były niewielkie i całkowite. Niestety liczby K_i nie są małe a w obliczeniach pojawiają się ułamki U_i.

Ponieważ ostatecznie musimy podać wynik modulo m, więc jedyne, czego nam brakuje, to umiejętność wykonywania modulo m wszystkich działań pośrednich, czyli głównie mnożenia i dzielenia[2]. Przypomnijmy, że w działaniach modulo m ostatecznie wszystkie liczby sprowadzamy do reszt z dzielenia całkowitego przez m, czyli do wartości $\{0, 1, \ldots, m-1\}$. Zapis $a \bmod m$ oznacza właśnie resztę z dzielenia a przez m. Piszemy także, że $a \equiv b \pmod{m}$, jeżeli reszty z dzielenia a przez m oraz b przez m są takie same.

[1] Więcej o statycznych drzewach przedziałowych można przeczytać np. w opisie rozwiązania zadania Tetris 3D z książeczki XIII Olimpiady Informatycznej [13].

[2] Więcej o działaniu modulo i o przystawaniu (kongruencji) modulo można przeczytać w podstawowych podręcznikach z matematyki dyskretnej: [31] i [32], a także w [20].

Samo mnożenie nie stwarza problemów, gdyż, jak łatwo się przekonać, zachodzi równość:

$$(a \cdot b \cdot c) \bmod m = \big(((a \cdot b) \bmod m) \cdot c\big) \bmod m.$$

Oznacza to, że po każdym mnożeniu wynik można „zredukować" do reszty z dzielenia przez m. Dużo gorzej wygląda sytuacja z dzieleniem. Aby przekonać się o tym, wystarczy przyjrzeć się kilku przykładom:

- ponieważ $7 \cdot 5 = 35 \equiv 11 \pmod{12}$, to $11/5 \equiv 7 \pmod{12}$;
- jako że $2 \cdot 4 = 8$, ale także $8 \cdot 4 = 32 \equiv 8 \pmod{12}$, zatem wynik dzielenia $8/4$ na resztach modulo 12 nie jest jednoznaczny;
- nie istnieje żadna liczba naturalna x, dla której zachodziłoby $x \cdot 6 \equiv 1 \pmod{12}$, dlatego dzielenie $1/6$ nie jest wykonalne na resztach modulo 12.

Widać, że sytuacja wygląda raczej niewesoło — trudno wykonać dzielenie, jeśli szukany iloraz nie istnieje albo jest niejednoznaczny. Nawet w przypadku, gdy wynik jest dobrze określony, to na pierwszy rzut oka nie bardzo widać, jak go wyznaczyć.

Ponieważ dzielenie modulo m okazuje się być takie kłopotliwe, to spróbujmy sprowadzić je do mnożenia. Dzielenie modulo, podobnie jak zwykłe dzielenie, jest operacją odwrotną do mnożenia i dzielenie przez x można interpretować jako mnożenie przez odwrotność x, czyli x^{-1}:

$$x \cdot x^{-1} \equiv 1 \bmod m,$$

$$a/x \equiv a \cdot x^{-1} \bmod m.$$

Dzięki powyższej zamianie dzielenie nie staje się automatycznie prostsze: wartość x^{-1} nie musi istnieć i nie musi być wyznaczona jednoznacznie. Jednak w pewnych przypadkach jest łatwo — okazuje się, że dla x względnie pierwszego z m odwrotność x modulo m zawsze istnieje i potrafimy ją efektywnie wyznaczyć.

Fakt 3. *Jeżeli x jest względnie pierwsze z m, to x^{-1} mod m jest określone jednoznacznie.*

Dowód tego faktu, wykorzystujący tzw. *rozszerzony algorytm Euklidesa*, można znaleźć w Dodatku na końcu niniejszego opisu rozwiązania. Jest on opisany także w książce [20]. Złożoność czasowa tego algorytmu zastosowanego do liczb $x < m$ oraz m to $O(\log m)$.

Jeszcze jeden drobny kłopot

Gdyby m było dużą liczbą pierwszą, to w trakcie obliczeń mielibyśmy do czynienia tylko z liczbami względnie pierwszymi z m i problem dzielenia modulo m byłby już rozwiązany. Niestety, nie możemy na to liczyć.

Niech

$$m = p_1^{\alpha_1} \cdot \ldots \cdot p_k^{\alpha_k}$$

będzie rozkładem m na czynniki pierwsze ($p_i \neq p_j$ dla $i \neq j$). Widzimy, że kłopoty przy liczeniu odwrotności mogą sprawić tylko liczby podzielne przez którąś z liczb p_i. Aby sobie

z nimi poradzić, wydzielmy z rozważanych liczb część podzielną przez liczby p_i. W tym celu wszystkie liczby w algorytmie będziemy reprezentować w postaci

$$v \cdot p_1^{a_1} \cdot \ldots \cdot p_k^{a_k},$$

gdzie $NWD(v,m) = 1$ (zwróć uwagę, że liczby pojawiające się w wykładnikach to a_i, a nie α_i). Wszystkie potrzebne nam operacje wykonujemy na tak zapisanych liczbach następująco:

- *Mnożenie* liczb wykonujemy poprzez wymnożenie współczynników v i dodanie do siebie odpowiednich wykładników potęg a_i. Koszt czasowy mnożenia to po prostu $O(k)$.

$$\left(v \cdot p_1^{a_1} \cdot \ldots \cdot p_k^{a_k}\right) \cdot \left(v' \cdot p_1^{a'_1} \cdot \ldots \cdot p_k^{a'_k}\right) = (v \cdot v' \bmod m) \cdot p_1^{a_1+a'_1} \cdot \ldots \cdot p_k^{a_k+a'_k}$$

- *Dzielenie* dwóch liczb wykonujemy, dzieląc modulo m współczynnik v pierwszej liczby przez współczynnik drugiej (co jest wykonalne, gdyż oba są względnie pierwsze z m) i odejmując wykładniki a_i drugiej liczby od wykładników pierwszej. Koszt czasowy dzielenia to zatem $O(k + \log m)$, gdzie drugi składnik wynika z konieczności użycia algorytmu Euklidesa do policzenia odwrotności modulo m.

$$\left(v \cdot p_1^{a_1} \cdot \ldots \cdot p_k^{a_k}\right) / \left(v' \cdot p_1^{a'_1} \cdot \ldots \cdot p_k^{a'_k}\right) = (v \cdot v'^{-1} \bmod m) \cdot p_1^{a_1-a'_1} \cdot \ldots \cdot p_k^{a_k-a'_k}$$

- *Przekształcenie liczby całkowitej x do postaci* $(v, a_1, \ldots, a_k)$ wykonujemy, dzieląc x kolejno przez $p_1, \ldots, p_k$. Sumarycznie operacji dzielenia wykonamy najwyżej $O(\log(n+m))$, gdyż dla liczby x suma wykładników a_i nie przekracza logarytmu z x, a będziemy mieć do czynienia tylko z x wielkości $O(\max(n,m))$. Uwzględniając, że musimy także sprawdzić każdą wartość p_i, widzimy, że złożoność czasowa tej operacji wynosi $O(k + \log(n+m))$.

- *Przekształcenie liczby zapisanej w postaci* $(v, a_1, \ldots, a_k)$ na zapisaną „normalnie" resztę modulo m wykonujemy za pomocą szybkiego potęgowania binarnego modulo m. Obliczenie tą metodą $x^y \bmod m$ wymaga $O(\log y)$ mnożeń liczb rzędu $O(m)$. Największą liczbą, jaka może pojawić się w trakcie obliczeń w naszym rozwiązaniu jest, jak łatwo sprawdzić, $n!$. Stąd mamy

$$a_i \leqslant \log_{p_i} n! \leqslant \log_2 n! = \log(1 \cdot 2 \cdot \ldots \cdot n) = \log 1 + \log 2 + \ldots + \log n \leqslant n \cdot \log n.$$

  To oznacza, że koszt czasowy całej operacji zamiany wynosi $O(k \cdot \log(n \cdot \log n)) = O(k \log n)$.

Z przeprowadzonych rozważań wynika, że każdą z potrzebnych operacji potrafimy wykonać w czasie $O(\log m \cdot \log n)$ (zauważmy, że $k = O(\log m)$). Biorąc pod uwagę ograniczenia z zadania, oraz to, że każda z tych operacji jest wykonywana w algorytmie $O(n+z)$ razy, daje to efektywne rozwiązanie problemu.

Opisane rozwiązanie wzorcowe zostało zaimplementowane w plikach `per2.cpp` oraz `per4.java`.

Alternatywna wersja rozwiązania wzorcowego

Ostatni krok rozwiązania wzorcowego, czyli sposób wykonywania działań na resztach z dzielenia przez m, można zaimplementować także trochę inaczej, z wykorzystaniem konstruktywnej wersji Chińskiego twierdzenia o resztach.

Twierdzenie 1 (Chińskie twierdzenie o resztach). *Niech $y_1, \ldots, y_k$ będą dowolnymi liczbami całkowitymi, a $n_1, \ldots, n_k$ — liczbami całkowitymi parami względnie pierwszymi. Wówczas układ*

$$\begin{aligned} x &\equiv y_1 \pmod{n_1} \\ x &\equiv y_2 \pmod{n_2} \\ &\ldots \\ x &\equiv y_k \pmod{n_k} \end{aligned} \tag{5}$$

spełnia dokładnie jedna reszta x_0 modulo $N = n_1 n_2 \cdot \ldots \cdot n_k$.

Dowód: Podobnie jak w dowodzie faktu o odwrotności modulo, i tym razem rozpoczniemy od wykazania jednoznaczności x_0. Jeżeli więc x_1 również spełnia układ kongruencji (5), to

$$(\forall\ 1 \leqslant i \leqslant k)\ (x_0 \equiv y_i \equiv x_1 \pmod{n_i}).$$

Stąd

$$(\forall\ 1 \leqslant i \leqslant k)\ (n_i \mid (x_0 - x_1)),$$

a więc

$$NWW(n_1, \ldots, n_k) \mid (x_0 - x_1),$$

gdzie NWW oznacza najmniejszą wspólną wielokrotność. Ponieważ liczby n_i są parami względnie pierwsze, więc $NWW(n_1, \ldots, n_k) = n_1 \cdot \ldots \cdot n_k$ i powyższa podzielność implikuje już żądaną jednoznaczność, gdyż jest równoważna

$$x_0 \equiv x_1 \pmod{N}.$$

Pokażmy teraz, że reszta x_0 istnieje. Jest kilka niekonstruktywnych dowodów tego faktu (wykazują istnienie x_0, ale nie wskazują, jak je wyliczyć). Na szczęście znamy także dowód konstruktywny. Zawiera on jawny wzór na x_0:

$$x_0 = \sum_{i=1}^{k} \left(\frac{N}{n_i}\right) \cdot \left(\left(\frac{N}{n_i}\right)^{-1} \bmod n_i\right) \cdot y_i \tag{6}$$

Nie bardzo widać, skąd ten wzór się wziął, ale na szczęście sprawdzenie, że jest poprawny, nie jest trudne. Rozważmy konkretną wartość $j \in \{1, \ldots, k\}$ i pokażmy, że

$$x_0 \equiv y_j \pmod{n_j}.$$

Zauważmy, że dla $i \neq j$ składniki sumy (6) są modulo n_j równe 0, gdyż $n_j \mid (N/n_i)$. Pozostaje zatem przyjrzeć się jedynie j-emu składnikowi sumy (6), czyli:

$$\left(\frac{N}{n_j}\right) \cdot \left(\left(\frac{N}{n_j}\right)^{-1} \bmod n_j\right) \cdot y_j.$$

Czynnik

$$\frac{N}{n_j} = n_1 \cdot \ldots \cdot n_{j-1} \cdot n_{j+1} \cdot \ldots \cdot n_k$$

jest względnie pierwszy z n_j, więc istnieje $(N/n_j)^{-1} \bmod n_j$ — jego odwrotność modulo n_j. Oczywiście z definicji odwrotności prawdziwa jest kongruencja

$$\left(\frac{N}{n_j}\right) \cdot \left(\left(\frac{N}{n_j}\right)^{-1} \bmod n_j\right) \equiv 1 \pmod{n_j}.$$

To pozwala zauważyć, że reszta z dzielenia j-ego wyrazu sumy (6) przez n_j wynosi

$$\left(\frac{N}{n_j}\right) \cdot \left(\left(\frac{N}{n_j}\right)^{-1} \bmod n_j\right) \cdot y_j \equiv y_j \pmod{n_j}.$$

Jest to także reszta z dzielenia x_0 przez n_j, co kończy dowód poprawności wzoru (6). ■

Jak można wykorzystać Chińskie twierdzenie o resztach do efektywnego wykonywania działań na resztach modulo m? Jak poprzednio, przedstawimy liczbę m w postaci

$$m = p_1^{\alpha_1} \cdot \ldots \cdot p_k^{\alpha_k},$$

gdzie $NWD(p_i, p_j) = 1$ dla $i \neq j$. Przyjmując $n_i = p_i^{\alpha_i}$, dostajemy układ $n_1, \ldots, n_k$, w którym liczby z przedziału $[0, m-1]$ możemy reprezentować za pomocą reszt. Każdą z reszt modulo n_i możemy z kolei zapisać, jak poprzednio, w postaci $v_i \cdot p_i^{a_i}$, gdzie $v_i \in \{0, \ldots, p_i^{\alpha_i} - 1\}$ jest względnie pierwsze z p_i.

To oznacza, że reprezentacja liczby x ma wówczas postać wektora o $2k$ elementach: $(v_1, \ldots, v_k, a_1, \ldots, a_k)$. Operacje na tak zapisanych liczbach wykonujemy analogicznie jak poprzednio. Na przykład dla reszt x i y, o reprezentacjach:

$$x \to (v_1, \ldots, v_k, a_1, \ldots, a_k),$$

$$y \to (v'_1, \ldots, v'_k, a'_1, \ldots, a'_k),$$

reprezentacja ich iloczynu jest równa

$$(x \cdot y) \to ((v_1 \cdot v'_1) \bmod p_1^{\alpha_1}, \ldots, (v_k \cdot v'_k) \bmod p_k^{\alpha_k}, a_1 + a'_1, \ldots, a_k + a'_k).$$

Dzielenie i przekształcanie liczby ze „zwykłego” zapisu do omawianej reprezentacji odbywa się analogicznie, jak poprzednio. Natomiast przekształcenie odwrotne jest możliwe dzięki Chińskiemu twierdzeniu o resztach, konkretnie dzięki wzorowi (6).

Można by się zapytać, po co właściwie zamieszczony został opis tej wersji sposobu reprezentacji reszt modulo m, skoro wydaje się ona bardzo podobna do tej z rozwiązania wzorcowego, a przy tym jeszcze nieco bardziej skomplikowana. Pierwszym powodem jest fakt, że przy zamianie nowej reprezentacji liczb na resztę modulo m nie potrzeba szybkiego potęgowania modulo, gdyż jeżeli którykolwiek z wykładników a_i jest większy od odpowiadającego wykładnika α_i, to $x \equiv 0 \pmod{p_i^{\alpha_i}}$, więc nie musimy się przejmować dużymi wartościami a_i. Drugim zaś powodem było właśnie to, że Chińskie twierdzenie o resztach jest interesujące samo w sobie...

Po implementacje tej wersji rozwiązania wzorcowego odsyłamy do plików `per.cpp`, `per1.pas` oraz `per3.java`.

Rozwiązania wolne

Prawie każdy krok rozumowania prowadzący do rozwiązania wzorcowego może zostać zrealizowany w inny, mniej efektywny sposób, co prowadzi do całej gamy rozwiązań wolniejszych, które na zawodach uzyskiwały od 10% do około 55% punktów. Zainteresowanych Czytelników odsyłamy do implementacji w plikach `pers1.cpp`-`pers6.pas`.

Testy

Rozwiązania zawodników były sprawdzane na zestawie 15 testów. W większości z nich konkretne permutacje były generowane w sposób całkowicie losowy; w niektórych (testy 3, 4, 5, 8, 9 i 10) permutacje składają się z niedużej liczby szczytów, czyli fragmentów rosnąco-malejących.

Nazwa	n	m	z	Opis
per1.in	8	$2^2 \cdot 7 \cdot 11$	5	test poprawnościowy
per2.in	10	$2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13$	456	test poprawnościowy
per3.in	73	90089	995	kilka szczytów
per4.in	81	$9221 \cdot 56477$	163913	kilka szczytów
per5.in	87	2^{19}	289633	kilka szczytów
per6.in	728	$53 \cdot 193 \cdot 293$	4999	test losowy
per7.in	2741	$2^4 \cdot 3^2 \cdot 5 \cdot 7^3$	99973	test losowy
per8.in	5363	$2 \cdot 3^2 \cdot 7 \cdot 29 \cdot 293$	29481	kilka szczytów
per9.in	8179	$11 \cdot 17 \cdot 19 \cdot 29 \cdot 41$	299840	kilka szczytów
per10.in	67528	$97 \cdot 107 \cdot 94597$	148545	kilka szczytów
per11.in	100000	$28669 \cdot 28687$	9841	test losowy
per12.in	213819	$103 \cdot 163 \cdot 167 \cdot 173$	6152	test losowy
per13.in	284315	975261919	150000	test losowy
per14.in	300000	$2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23$	299997	test losowy
per15.in	300000	$2^9 \cdot 5^9$	300000	test losowy

Dodatek — dowód Faktu 3

Dowód rozpoczniemy nietypowo — pokażemy, że liczba x względnie pierwsza z m ma najwyżej jedną odwrotność modulo m, a dopiero później uzasadnimy, dlaczego ją w ogóle ma. Załóżmy, że y i y' spełniają równość modulo:

$$x \cdot y \equiv 1 \equiv x \cdot y' \pmod{m}.$$

Wówczas także

$$m \mid xy - xy' = x(y - y').$$

Ponieważ x i m są względnie pierwsze, to ta podzielność oznacza, że

$$m \mid y - y',$$

czyli że

$$y' \equiv y \pmod{m}.$$

A zatem faktycznie odwrotność x modulo m musi być wyznaczona jednoznacznie.

Pokażmy teraz, że taka odwrotność istnieje. W tym celu udowodnimy, że dla dowolnych liczb całkowitych a i b równanie (zwykłe, nie modulo):

$$au + bv = NWD(a,b) \tag{7}$$

ma zawsze rozwiązanie, w którym u i v są całkowite. Zastosowanie tej własności dla x i m będzie oznaczało, że istnieją liczby całkowite u i v takie, że

$$xu + mv = NWD(x,m) = 1,$$

czyli

$$xu \equiv 1 \pmod{m}.$$

Liczba u będzie więc szukaną odwrotnością x modulo m.

Wróćmy do równania (7). Przypomnijmy, że $NWD(a,b)$ można wyznaczyć za pomocą algorytmu Euklidesa, w którym w każdym kroku redukujemy problem, korzystając z równości:

$$NWD(a,b) = \begin{cases} NWD(b, a \bmod b) & \text{dla } b \neq 0, \\ a & \text{dla } b = 0. \end{cases} \tag{8}$$

Niech

$$(a_1,b_1),\ (a_2,b_2),\ \ldots,\ (a_t,b_t)$$

będą kolejnymi parami liczb, które pojawiają się w trakcie działania algorytmu Euklidesa wywołanego dla liczb a i b. Muszą wówczas zachodzić podstawowe zależności:

$$a_1 = a,\ b_1 = b,\ b_i > 0 \text{ dla } 1 \leqslant i < t,\ b_t = 0$$

oraz

$$NWD(a_1,b_1) = NWD(a_2,b_2) = \ldots = NWD(a_t,b_t) = a_t.$$

Zauważmy, że dla liczb a_t i b_t bardzo łatwo znaleźć rozwiązanie równania (7):

$$a_t u_t + b_t v_t = NWD(a_t,b_t) = a_t, \tag{9}$$

czyli

$$u_t = 1,\ v_t = 0. \tag{10}$$

Z kolei korzystając z rozwiązania u_{i+1}, v_{i+1} dla pary a_{i+1}, b_{i+1}, można stosunkowo prosto wyznaczyć rozwiązanie u_i, v_i dla pary a_i, b_i. Otóż w równaniu

$$a_{i+1}u_{i+1} + b_{i+1}v_{i+1} = NWD(a_{i+1},b_{i+1}) \tag{11}$$

możemy podstawić:

$$NWD(a_{i+1},b_{i+1}) := NWD(a_i,b_i),\ a_{i+1} := b_i \text{ oraz } b_{i+1} := a_i \bmod b_i$$

i otrzymujemy

$$b_i \cdot u_{i+1} + (a_i \bmod b_i) \cdot v_{i+1} = NWD(a_i, b_i). \tag{12}$$

Ponieważ

$$a_i \bmod b_i = a_i - \left\lfloor \frac{a_i}{b_i} \right\rfloor \cdot b_i,$$

to (12) możemy zapisać w postaci

$$b_i \cdot u_{i+1} + \left(a_i - \left\lfloor \frac{a_i}{b_i} \right\rfloor \cdot b_i\right) \cdot v_{i+1} = NWD(a_i, b_i),$$

czyli

$$a_i \cdot v_{i+1} + b_i \cdot \left(u_{i+1} - \left\lfloor \frac{a_i}{b_i} \right\rfloor v_{i+1}\right) = NWD(a_i, b_i).$$

To oznacza, że:

$$u_i = v_{i+1},\ v_i = u_{i+1} - \left\lfloor \frac{a_i}{b_i} \right\rfloor v_{i+1}, \tag{13}$$

są poszukiwanym rozwiązaniem dla a_i oraz b_i. Korzystając z rozwiązania (10) i zależności (13), potrafimy w t krokach znaleźć rozwiązanie dla pary $a = a_1$ i $b = b_1$. To kończy dowód.

Przykład 1. Prześledźmy działanie algorytmu podczas wyznaczania odwrotności 28^{-1} mod 51. W tym celu rozwiążemy równanie

$$28u + 51v = 1,$$

obliczając $NWD(28,51) = 1$ algorytmem Euklidesa. Kolumny a_i oraz b_i poniższej tabeli zawierają argumenty kolejnych wywołań procedury NWD w algorytmie. Kolumny u_i oraz v_i zawierają rozwiązania równań

$$a_i u_i + b_i v_i = 1$$

wyznaczone za pomocą wzorów (10) oraz (13). Strzałki pokazują kolejność wyliczania wartości poszczególnych parametrów.

wywołania rekurencyjne	$\mathbf{a_i}$	$\mathbf{b_i}$	**powroty z wywołań**	$\mathbf{u_i}$	$\mathbf{v_i}$
↓	28	51		−20	11
	51	28	↑	11	−20
	28	23		−9	11
	23	5		2	−9
	5	3		−1	2
	3	2		1	−1
	2	1		0	1
	1	0		1	0

Szukaną odwrotnością 28 modulo 51 jest więc

$$u_1 \bmod b_1 = -20 \bmod 51 = 31.$$

Możemy dodatkowo sprawdzić, że faktycznie:

$$28 \cdot 31 = 868 \equiv 1 \pmod{51}.$$

XX Międzynarodowa Olimpiada Informatyczna,

Kair, Egipt 2008

Tomasz Kulczyński, Jakub Radoszewski
Tłumaczenie

IOI 2008, dzień pierwszy, 18.08.2008

Drukarka

Musisz wydrukować N słów. Twoja drukarka to stary model, który obsługuje się, składając słowo z czcionek — odlanych z metalu małych elementów, z których każdy zawiera jedną literę. Do tak ułożonego wzoru przykłada się kartkę papieru, aby wydrukować złożone słowo. Drukarka, którą posiadasz, pozwala na wykonanie każdej z następujących operacji:

- *dodania jednej litery na końcu słowa znajdującego się aktualnie w drukarce;*
- *usunięcia ostatniej litery słowa znajdującego się aktualnie w drukarce;*
- *wydrukowania słowa znajdującego się aktualnie w drukarce.*

Początkowo drukarka jest pusta; nie zawiera żadnych czcionek. Po zakończeniu pracy nie musisz czyścić drukarki — możesz pozostawić w niej dowolne czcionki. Ponadto, słowa możesz drukować w dowolnej kolejności.

Ponieważ każda z operacji wymaga pewnego czasu, chcesz zminimalizować łączną liczbę wykonanych operacji.

Zadanie

Napisz program, który mając danych N słów, wyznaczy minimalną liczbę operacji potrzebnych do ich wydrukowania w dowolnej kolejności i wypisze jedną taką sekwencję operacji.

Ograniczenia

$1 \leqslant N \leqslant 25\,000$ *— liczba słów, które chcesz wydrukować.*

Wejście

Twój program powinien wczytać ze standardowego wejścia następujące dane:

- *Pierwszy wiersz zawierający liczbę całkowitą N — liczbę słów do wydrukowania.*
- *Kolejnych N wierszy, z których każdy zawiera jedno słowo. Każde słowo składa się wyłącznie z małych liter (*`'a'` *–* `'z'`*) i ma długość od 1 do 20 (włącznie). Wszystkie słowa są różne.*

Wyjście

Twój program powinien wypisać na standardowe wyjście następujący wynik:

- *Pierwszy wiersz powinien zawierać liczbę całkowitą M oznaczającą minimalną liczbę operacji potrzebnych do wydrukowania danych N słów.*

- *Każdy z kolejnych M wierszy powinien zawierać jeden znak. Znaki te opisują sekwencję wykonanych operacji. Każda z nich musi być zapisana następująco:*
  - *dodanie małej litery jest reprezentowane przez nią samą,*
  - *usunięcie ostatniej litery jest reprezentowane przez znak '-' (minus, kod ASCII 45),*
  - *wydrukowanie aktualnego słowa jest reprezentowane przez znak 'P' (wielka litera P).*

Punktacja

W testach wartych łącznie 40 punktów, wartość N nie przekracza 18.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
3	20
print	t
the	h
poem	e
	P
	-
	-
	-
	p
	o
	e
	m
	P
	-
	-
	-
	r
	i
	n
	t
	P

Tomasz Kulczyński, Jakub Radoszewski
Tłumaczenie

IOI 2008, dzień pierwszy, 18.08.2008

Ryby

Szecherezada powiedziała Ci, że bardzo daleko stąd, na środku pustyni znajduje się jezioro. Na początku w jeziorze było F ryb. Spośród najdroższych klejnotów na Ziemi wybrano K różnych rodzajów i każdej spośród F ryb dano dokładnie jeden klejnot do połknięcia. Zauważ, że ponieważ K może być mniejsze niż F, to dwie lub więcej ryb mogło połknąć klejnoty tego samego rodzaju.

Z upływem czasu jedne ryby zjadały inne ryby. Jedna ryba może zjeść drugą tylko wtedy, gdy jest co najmniej dwukrotnie dłuższa od niej (ryba A może zjeść rybę B wtedy i tylko wtedy, gdy $L_A \geqslant 2 \cdot L_B$). Nie istnieje żadna reguła mówiąca, kiedy ryba decyduje się na konsumpcję. Jedne ryby mogą decydować się na zjedzenie wielu mniejszych ryb jedna po drugiej, podczas gdy inne mogą postanowić nie jeść żadnych ryb, nawet jeżeli mogą to uczynić. Kiedy ryba zjada mniejszą rybę, jej długość nie zmienia się, a klejnoty zawarte w żołądku mniejszej ryby trafiają nieuszkodzone do żołądka większej ryby.

Szecherezada powiedziała Ci, że jeżeli zdołasz znaleźć opisane jezioro, to będziesz mógł wyłowić z niego jedną rybę i zachować dla siebie wszystkie klejnoty z jej żołądka. Zamierzasz spróbować szczęścia, lecz przed wyruszeniem w długą podróż chciałbyś się dowiedzieć, ile różnych kombinacji klejnotów możesz uzyskać, jeżeli złowisz jedną rybę.

Zadanie

Napisz program, który mając dane długości wszystkich ryb i rodzaje połkniętych przez nie początkowo klejnotów, wyznaczy liczbę różnych kombinacji klejnotów, które mogą znaleźć się w żołądku jakiejkolwiek ryby, modulo dana liczba całkowita M. Kombinacje są zdefiniowane wyłącznie poprzez liczby klejnotów każdego z K rodzajów. Uporządkowanie klejnotów jest nieistotne, a każde dwa klejnoty tego samego rodzaju są nierozróżnialne.

Ograniczenia

$1 \leqslant F \leqslant 500\,000$ — *początkowa liczba ryb w jeziorze;*
$1 \leqslant K \leqslant F$ — *liczba różnych rodzajów klejnotów;*
$2 \leqslant M \leqslant 30\,000$;
$1 \leqslant L_X \leqslant 1\,000\,000\,000$ — *długość ryby X.*

Wejście

Twój program powinien wczytać ze standardowego wejścia następujące dane:

- *Pierwszy wiersz zawierający liczbę całkowitą F — początkową liczbę ryb w jeziorze.*
- *Drugi wiersz zawierający liczbę całkowitą K — liczbę rodzajów klejnotów. Rodzaje klejnotów są oznaczone liczbami całkowitymi od 1 do K włącznie.*

- *Trzeci wiersz zawierający liczbę całkowitą M.*
- *Każdy z kolejnych F wierszy opisuje jedną rybę za pomocą dwóch liczb całkowitych, oddzielonych pojedynczym odstępem i oznaczających długość ryby oraz rodzaj klejnotu połkniętego początkowo przez tę rybę.*

UWAGA: *We wszystkich testach wykorzystywanych do oceny występuje co najmniej jeden klejnot każdego spośród K rodzajów.*

Wyjście

Twój program powinien wypisać na standardowe wyjście jeden wiersz zawierający jedną liczbę całkowitą między 0 a $M-1$ (włącznie): liczbę różnych możliwych kombinacji klejnotów modulo M.
Zauważ, że wartość M nie ma żadnego znaczenia w rozwiązaniu zadania, poza uproszczeniem obliczeń.

Punktacja

W pewnej liczbie testów, wartych łącznie 70 punktów, K nie przekracza 7 000.
Ponadto, w pewnych spośród tych testów, wartych łącznie 25 punktów, K nie przekracza 20.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`5` `3` `7` `2 2` `5 1` `8 3` `4 1` `2 3`	`4`

Jest 11 możliwych kombinacji, więc powinieneś wypisać 11 modulo 7, czyli 4.

Możliwymi kombinacjami są: [1] [1,2] [1,2,3] [1,2,3,3] [1,3] [1,3,3] [2] [2,3] [2,3,3] [3] oraz [3,3]. (Dla każdej kombinacji wymieniono listę klejnotów, które zawiera. Dla przykładu, [2,3,3] to kombinacja składająca się z jednego klejnotu rodzaju 2 i dwóch klejnotów rodzaju 3.) Te kombinacje mogą zostać osiągnięte na następujące sposoby:

- *[1]: Możliwe, że złowisz drugą (lub czwartą) rybę, zanim zje ona jakąkolwiek inną rybę.*
- *[1,2]: Jeżeli druga ryba zje pierwszą, to będzie miała klejnot rodzaju 1 (ten, który początkowo połknęła) oraz klejnot rodzaju 2 (z żołądka pierwszej ryby).*
- *[1,2,3]: Jeden z możliwych sposobów uzyskania tej kombinacji to: czwarta ryba zjada pierwszą, po czym trzecia ryba zjada czwartą. Jeżeli wówczas złowisz trzecią rybę, to będzie miała w żołądku po jednym klejnocie każdego rodzaju.*

- *[1,2,3,3]: Czwarta zjada pierwszą, trzecia zjada czwartą, trzecia zjada piątą, łowisz trzecią rybę.*
- *[1,3]: Trzecia zjada czwartą, łowisz ją.*
- *[1,3,3]: Trzecia zjada piątą, trzecia zjada czwartą, łowisz ją.*
- *[2]: Łowisz pierwszą rybę.*
- *[2,3]: Trzecia zjada pierwszą, łowisz ją.*
- *[2,3,3]: Trzecia zjada pierwszą, trzecia zjada piątą, łowisz ją.*
- *[3]: Łowisz trzecią rybę.*
- *[3,3]: Trzecia zjada piątą, łowisz ją.*

Tomasz Kulczyński, Jakub Radoszewski
Tłumaczenie

IOI 2008, dzień pierwszy, 18.08.2008

Wyspy

Właśnie odwiedzasz park, w którym znajduje się N wysp. Kiedy budowano park, z każdej wyspy został poprowadzony jeden most — długość mostu wychodzącego z wyspy i oznaczamy przez L_i. Łącznie w parku jest N mostów. Każdym mostem można poruszać się w obu kierunkach (choć każdy most miał swój początek na jednej wyspie). Ponadto, dowolne dwie wyspy są połączone promem pływającym pomiędzy nimi w tę i z powrotem.

Ponieważ wolisz spacerować niż pływać promami, chcesz zmaksymalizować sumę długości mostów, którymi przejdziesz, stosując się do ograniczeń podanych poniżej.

- *Możesz rozpocząć zwiedzanie parku od dowolnej wyspy.*
- *Nie możesz odwiedzić żadnej wyspy więcej niż raz.*
- *W każdej chwili możesz przemieścić się z wyspy S, na której aktualnie się znajdujesz, na inną wyspę D, której nie odwiedzałeś wcześniej. Ruch z S do D możesz wykonać na jeden ze sposobów:*
  - *Idąc: Możliwe jedynie wtedy, gdy istnieje most łączący obie wyspy. W tym przypadku długość mostu zostaje dodana do łącznej długości pokonanej trasy.*
  - *Promem: Możesz wybrać ten wariant tylko wtedy, jeśli D nie jest osiągalne z S przy użyciu dowolnej kombinacji mostów i/lub wcześniej użytych promów. (Przy sprawdzaniu osiągalności należy wziąć pod uwagę wszystkie ścieżki, także te prowadzące przez wyspy wcześniej odwiedzone.)*

Zauważ, że nie musisz odwiedzić wszystkich wysp, a przejście wszystkimi mostami może okazać się niemożliwe.

Zadanie

Napisz program, który mając dany opis N mostów wraz z ich długościami, wyznaczy maksymalną możliwą do przejścia odległość, zgodnie z zasadami opisanymi powyżej.

Ograniczenia

$2 \leqslant N \leqslant 1\,000\,000$ — *liczba wysp w parku;*
$1 \leqslant L_i \leqslant 100\,000\,000$ — *długość mostu o numerze i.*

Wejście

Twój program powinien wczytać ze standardowego wejścia następujące dane:

- *Pierwszy wiersz zawierający liczbę całkowitą N — liczbę wysp w parku. Wyspy są ponumerowane liczbami od 1 do N włącznie.*

- *Każdy z kolejnych N wierszy opisuje most — i-ty z nich zawiera dwie liczby całkowite oddzielone pojedynczym odstępem, opisujące most wychodzący z wyspy o numerze i. Pierwsza z tych liczb to numer wyspy po drugiej stronie mostu, a druga to długość L_i tego mostu. Możesz założyć, że każdy most jest rozpięty pomiędzy dwiema różnymi wyspami.*

Wyjście

Twój program powinien wypisać na standardowe wyjście jeden wiersz zawierający jedną liczbę całkowitą — maksymalną możliwą do przejścia odległość.
UWAGA 1: *Dla niektórych testów odpowiedź nie zmieści się w 32-bitowym typie całkowitym, możesz potrzebować typów* `int64` *w Pascalu lub* `long long` *w C/C++, aby uzyskać pełną liczbę punktów za to zadanie.*
UWAGA 2: *Podczas uruchamiania programów napisanych w Pascalu w środowisku zawodów, liczby 64-bitowe są wczytywane ze standardowego wejścia znacząco wolniej niż liczby 32-bitowe, nawet gdy wczytywane wartości i tak mieszczą się w 32 bitach. Zalecamy wczytywanie danych do typów 32-bitowych.*

Punktacja

W pewnych testach wartych 40 punktów, N nie przekracza 4 000.

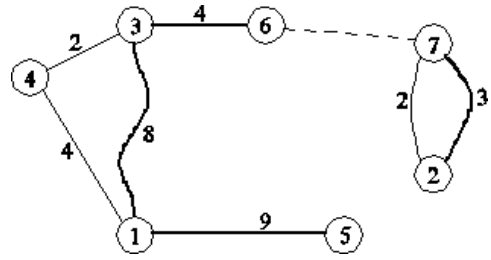
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
24
```



N = 7 mostów z przykładu to (1-3), (2-7), (3-4), (4-1), (5-1), (6-3) i (7-2). Zauważmy, że istnieją dwa różne mosty pomiędzy wyspami 2 i 7.
Oto jedna z dróg realizujących maksymalną odległość, jaką można przejść:

- *Zacznij od wyspy 5.*
- *Przejdź mostem długości 9 na wyspę 1.*
- *Przejdź mostem długości 8 na wyspę 3.*

- *Przejdź mostem długości 4 na wyspę 6.*
- *Przepłyń promem z wyspy 6 na wyspę 7.*
- *Przejdź mostem długości 3 na wyspę 2.*

Na końcu podróży jesteś na wyspie 2, a pokonana pieszo odległość wynosi $9+8+4+3=24$*.*

Jedyną nieodwiedzoną wyspą jest 4. Zauważmy, że nie można jej już odwiedzić po pokonaniu powyższej trasy. Dokładniej:

- *Nie jesteś w stanie odwiedzić jej pieszo, gdyż nie ma mostu łączącego wyspę 2 (gdzie obecnie jesteś) z wyspą 4.*
- *Nie możesz przepłynąć na nią promem, gdyż wyspa 4 jest osiągalna z wyspy 2, na której aktualnie jesteś. Możliwa droga do niej to: przejście mostem (2-7), potem użytym już poprzednio promem z wyspy 7 na wyspę 6, następnie mostem (6-3) i ostatecznie mostem (3-4).*

Tomasz Kulczyński, Jakub Radoszewski
Tłumaczenie

IOI 2008, dzień drugi, 20.08.2008

Liniowy ogród

Ramzes II wrócił właśnie ze zwycięskiej bitwy. Aby uczcić swe zwycięstwo, postanowił założyć wspaniały ogród. W ogrodzie ma się znajdować długa rabata, ciągnąca się od jego pałacu w Luksorze aż do świątyni Karnak. Na rabacie będą rosły wyłącznie kwiaty lotosu i papirusu[1]*, jako że symbolizują one odpowiednio Górny i Dolny Egipt.*

Na rabacie musi być dokładnie N roślin. Ponadto, rabata musi być zbalansowana: w każdym jej spójnym fragmencie liczby roślin lotosu i papirusu mogą różnić się co najwyżej o 2.

Każdy możliwy układ roślin na rabacie można przedstawić jako napis złożony z liter `'L'` *(lotos) i* `'P'` *(papirus). Na przykład dla $N = 5$ istnieje 14 możliwych zbalansowanych układów rabaty. Są to, w kolejności alfabetycznej:* `LLPLP`, `LLPPL`, `LPLLP`, `LPLPL`, `LPLPP`, `LPPLL`, `LPPLP`, `PLLPL`, `PLLPP`, `PLPLL`, `PLPLP`, `PLPPL`, `PPLLP` *i* `PPLPL`.

Wszystkie opisy zbalansowanych rabat o ustalonej długości można uporządkować alfabetycznie, a następnie ponumerować, poczynając od 1. Na przykład dla $N = 5$ rabatą numer 12 jest `PLPPL`.

Zadanie

Napisz program, który mając daną liczbę roślin N oraz napis reprezentujący pewną zbalansowaną rabatę, wyznaczy numer przypisany tej rabacie modulo pewna podana liczba całkowita M.
Zauważ, że wartość M nie ma żadnego znaczenia w rozwiązaniu zadania, poza uproszczeniem obliczeń.

Ograniczenia

$1 \leqslant N \leqslant 1\,000\,000$;
$7 \leqslant M \leqslant 10\,000\,000$.

Wejście

Twój program powinien wczytać ze standardowego wejścia następujące dane:

- *Pierwszy wiersz zawierający liczbę całkowitą N — liczbę roślin na rabacie.*
- *Drugi wiersz zawierający liczbę całkowitą M.*
- *Trzeci wiersz zawierający napis złożony z N znaków* `'L'` *(lotos) lub* `'P'` *(papirus) reprezentujący pewien zbalansowany układ roślin na rabacie.*

[1] Chodzi tu o ciborę papirusową, z której wyrabiano papirus (przyp. tłum.).

Wyjście

Twój program powinien wypisać na standardowe wyjście jeden wiersz zawierający jedną liczbę całkowitą między 0 *a* $M-1$ *(włącznie) — numer przypisany układowi roślin na rabacie podanemu na wejściu, modulo* M*.*

Punktacja

W testach wartych łącznie 40 punktów, N *nie przekracza 40.*

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`5` `7` `PLPPL`	`5`

Numerem przypisanym `PLPPL` *jest 12, a zatem wynikiem jest 12 modulo 7, czyli 5.*

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`12` `10000` `LPLLPLPPLPLL`	`39`

Tomasz Kulczyński, Jakub Radoszewski
Tłumaczenie

IOI 2008, dzień drugi, 20.08.2008

Podstawa piramidy

Poproszono Cię o wyznaczenie największej podstawy nowo budowanej piramidy, przy której dostępny budżet inwestycji nie zostanie przekroczony. Aby pomóc Ci podjąć decyzję, dostarczono Ci raport o dostępnych fragmentach gruntu, który to grunt został dla wygody przedstawiony jako krata złożona z $M \times N$ kwadratowych pól. Podstawa piramidy musi być kwadratem o bokach równoległych do boków kraty.

Raport wykazał istnienie P (być może nachodzących na siebie) przeszkód, które zostały opisane jako prostokąty na kracie o bokach równoległych do jej boków. Żeby można było zbudować piramidę, wszystkie pola pokryte przez jej podstawę muszą zostać oczyszczone z przeszkód. Koszt usunięcia i-tej przeszkody to C_i. Przeszkody można usuwać wyłącznie w całości, to znaczy nie można usunąć jedynie fragmentu jakiejś przeszkody. Ponadto, usunięcie przeszkody nie wpływa na inne przeszkody, nawet jeśli na nią nachodzą.

Zadanie

Napisz program, który mając dane wymiary kraty M oraz N, opis P przeszkód, koszt usunięcia każdej z przeszkód oraz budżet B, którym dysponujesz, wyznaczy największą możliwą długość boku piramidy, dla której istnieje lokalizacja piramidy o łącznym koszcie usunięcia przeszkód nieprzekraczającym B.

Ograniczenia i punktacja

Twój program będzie oceniany na trzech rozłącznych zbiorach testów. W każdym z nich spełnione są następujące ograniczenia:
$1 \leqslant M, N \leqslant 1\,000\,000$ — wymiary kraty;
$1 \leqslant C_i \leqslant 7\,000$ — koszt usunięcia i-tej przeszkody;
$1 \leqslant X_{i1} \leqslant X_{i2} \leqslant M$ — współrzędne X najbardziej lewego i najbardziej prawego pola i-tej przeszkody;
$1 \leqslant Y_{i1} \leqslant Y_{i2} \leqslant N$ — współrzędne Y najniższego i najwyższego pola i-tej przeszkody.

W pierwszym zbiorze testów wartych 35 punktów:
$B = 0$ — Twój budżet (nie możesz usuwać żadnych przeszkód);
$1 \leqslant P \leqslant 1\,000$ — liczba przeszkód na kracie.

W drugim zbiorze testów wartych 35 punktów:
$0 < B \leqslant 2\,000\,000\,000$ — Twój budżet;
$1 \leqslant P \leqslant 30\,000$ — liczba przeszkód na kracie.

W trzecim zbiorze testów wartych 30 punktów:
$B = 0$ — Twój budżet (nie możesz usuwać żadnych przeszkód);
$1 \leqslant P \leqslant 400\,000$ — liczba przeszkód na kracie.

Wejście

Twój program powinien wczytać ze standardowego wejścia następujące dane:

- *Pierwszy wiersz zawierający dwie liczby całkowite oddzielone pojedynczym odstępem, reprezentujące odpowiednio M oraz N.*
- *Drugi wiersz zawierający liczbę całkowitą B — maksymalny koszt, na jaki możesz sobie pozwolić (tzn. Twój budżet).*
- *Trzeci wiersz zawierający liczbę całkowitą P — liczbę przeszkód opisanych w raporcie.*
- *Każdy z kolejnych P wierszy opisuje przeszkodę — i-ty z nich opisuje i-tą przeszkodę. Opis każdej przeszkody składa się z pięciu liczb całkowitych: X_{i1}, Y_{i1}, X_{i2}, Y_{i2} oraz C_i pooddzielanych pojedynczymi odstępami. Reprezentują one odpowiednio: współrzędne lewego dolnego pola przeszkody, współrzędne prawego górnego pola przeszkody i koszt usunięcia przeszkody. Lewe dolne pole kraty ma współrzędne $(1,1)$, natomiast prawe górne — współrzędne (M,N).*

Wyjście

Twój program powinien wypisać na standardowe wyjście jeden wiersz zawierający jedną liczbę całkowitą — maksymalną możliwą długość boku podstawy piramidy, która może zostać wybudowana. Jeżeli nie jest możliwe wybudowanie żadnej piramidy, Twój program powinien wypisać 0.

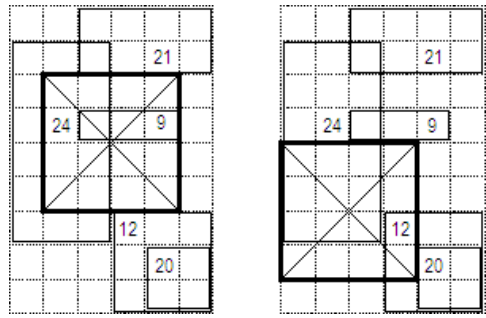
Przykład

Dla danych wejściowych:

```
6 9
42
5
4 1 6 3 12
3 6 5 6 9
1 3 3 8 24
3 8 6 9 21
5 1 6 2 20
```

poprawnym wynikiem jest:

```
4
```



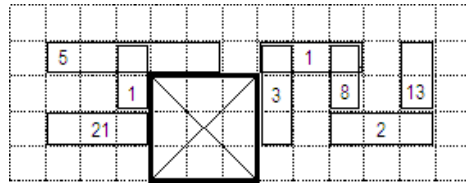
Rysunek prezentuje dwie możliwe lokalizacje podstawy piramidy; w obydwu z nich długość boku podstawy wynosi 4.

Dla danych wejściowych:

```
13 5
0
8
8 4 10 4 1
4 3 4 4 1
10 2 12 2 2
8 2 8 4 3
2 4 6 4 5
10 3 10 4 8
12 3 12 4 13
2 2 4 2 21
```

poprawnym wynikiem jest:

```
3
```



Rysunek prezentuje jedyną możliwą lokalizację podstawy piramidy o boku równym 3.

Tomasz Kulczyński, Jakub Radoszewski
Tłumaczenie

IOI 2008, dzień drugi, 20.08.2008

Teleporty

Startujesz w konkursie polegającym na przejechaniu Egiptu z zachodu na wschód po zadanej prostoliniowej trasie. Początkowo znajdujesz się na zachodnim końcu trasy. Zasady konkursu mówią, że zawsze musisz poruszać się po tej trasie i zawsze na wschód.

Na trasie znajduje się N teleportów. Teleport ma dwa końce. Kiedy tylko dotrzesz do jednego z końców teleportu, przenosi Cię on natychmiast na swój drugi koniec. (Zauważ, że zależnie od tego, na który koniec dotarłeś, teleport może przenieść Cię na wschód lub na zachód od wcześniejszej pozycji). Po teleportacji musisz kontynuować swoją wędrówkę na wschód wzdłuż trasy. Nie da się ominąć teleportu znajdującego się na drodze. Żadne końce teleportów nie znajdują się w tym samym miejscu. Końce teleportów leżą ściśle pomiędzy początkiem a końcem trasy.

Za każdą teleportację dostajesz 1 punkt. Celem konkursu jest zdobycie tak wielu punktów, jak to możliwe. W celu zmaksymalizowania liczby punktów możesz przed rozpoczęciem swojej wędrówki dołożyć maksymalnie M nowych teleportów na trasie. Za używanie nowych teleportów również dostajesz punkty.

Możesz umieścić końce nowych teleportów gdziekolwiek chcesz (także w punktach o współrzędnych niecałkowitych), o ile nie będą kolidowały z już istniejącymi końcami. Oznacza to, że pozycje końców wszystkich $N + M$ teleportów muszą być różne. Nowe końce również muszą leżeć ściśle pomiędzy początkiem a końcem trasy.

Zauważ, że jest zagwarantowane, iż niezależnie od układu nowo dodanych teleportów, będzie zawsze można dotrzeć do końca trasy.

Zadanie

Napisz program, który mając dane pozycje końców N teleportów oraz liczbę M nowych teleportów, które możesz dołożyć, obliczy maksymalną liczbę punktów, jakie można zdobyć.

Ograniczenia

$1 \leqslant N \leqslant 1\,000\,000$ — liczba teleportów znajdujących się początkowo na trasie;
$1 \leqslant M \leqslant 1\,000\,000$ — maksymalna liczba teleportów, które możesz dodać;
$1 \leqslant W_X < E_X \leqslant 2\,000\,000$ — odległości od początku trasy do zachodniego i wschodniego końca teleportu X.

Wejście

Twój program powinien wczytać ze standardowego wejścia następujące dane:

- *Pierwszy wiersz zawierający liczbę całkowitą N — liczbę teleportów początkowo rozmieszczonych na trasie.*

- *Drugi wiersz zawierający liczbę całkowitą M — maksymalną liczbę nowych teleportów, które możesz dodać.*
- *Każdy z kolejnych N wierszy opisuje jeden teleport — i-ty z nich opisuje i-ty teleport. Opis każdego teleportu składa się z dwóch liczb całkowitych W_i oraz E_i oddzielonych pojedynczym odstępem. Reprezentują one odpowiednio odległości od początku trasy do zachodniego i wschodniego końca teleportu.*

Żadne dwa końce teleportów podane na wejściu nie pokrywają się. Trasa, którą będziesz podróżować, zaczyna się na pozycji 0, a kończy na pozycji $2\,000\,001$.

Wyjście

Twój program powinien wypisać na standardowe wyjście jeden wiersz zawierający jedną liczbę całkowitą — maksymalną liczbę punktów, jaką możesz uzyskać.

Punktacja

W testach wartych 30 punktów, $N \leqslant 500$ i $M \leqslant 500$.

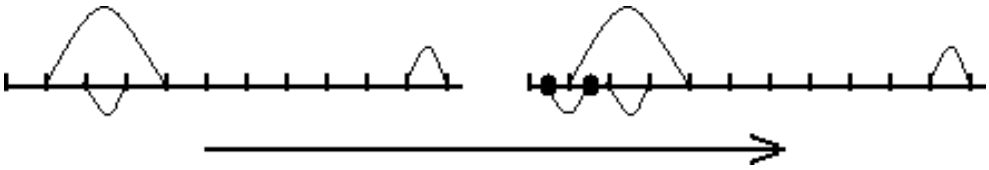
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
6
```



Pierwszy rysunek przedstawia trasę z trzema początkowymi teleportami. Na drugim rysunku widać tę samą trasę z dodanym teleportem o końcach na pozycjach $0,5$ i $1,5$.

Po dodaniu nowego teleportu tak, jak pokazano na rysunku, Twoja podróż wyglądałaby następująco:

- *Zaczynasz na pozycji 0, poruszając się na wschód.*
- *Docierasz do końca teleportu na pozycji $0,5$ i teleportujesz się na pozycję $1,5$ (zyskujesz 1 punkt).*
- *Kontynuujesz wędrówkę na wschód i docierasz do końca teleportu na pozycji 2; teleportujesz się na pozycję 3 (masz 2 punkty).*

- *Docierasz do końca teleportu na pozycji 4 i teleportujesz się na pozycję 1 (masz 3 punkty).*
- *Docierasz do końca teleportu na pozycji 1,5 i teleportujesz się na pozycję 0,5 (masz 4 punkty).*
- *Docierasz do końca teleportu na pozycji 1 i teleportujesz się na pozycję 4 (masz 5 punktów).*
- *Docierasz do końca teleportu na pozycji 10 i teleportujesz się na pozycję 11 (masz 6 punktów).*
- *Kontynuujesz podróż, aż dotrzesz do końca trasy, z łączną notą 6 punktów.*

Dla danych wejściowych:

```
3
3
5 7
6 10
1999999 2000000
```

poprawnym wynikiem jest:

```
12
```

XIV Bałtycka Olimpiada Informatyczna,

Gdynia 2008

Tomasz Kulczyński, Jakub Łącki, Jakub Radoszewski
Tłumaczenie

BOI 2008, dzień próbny, 18.04.2008

Mafia

Policja Bajtlandii dostała anonimową wiadomość o tym, że lokalni bossowie mafii planują duży transport z portu do jednego z sekretnych magazynów, położonego na wsi. Oficerowie znają datę transportu. Wiedzą też, że przestępcy będą poruszać się po sieci autostrad.

Sieć ta składa się z dwukierunkowych połączeń, z których każde łączy bezpośrednio dwa różne punkty poboru opłat. Punkt poboru opłat może być połączony z wieloma innymi punktami. Pojazdy mogą wjeżdżać na autostradę i opuszczać ją tylko w punktach poboru opłat. Transport wjedzie na autostradę w punkcie poboru opłat położonym koło portu, a zjedzie z niej w punkcie w pobliżu magazynu (nie będzie on zjeżdżał z autostrad nigdzie po drodze między tymi punktami). W wybranych punktach poboru opłat mają zostać rozlokowane szwadrony policji. Kiedy transport wjedzie do punktu obstawionego przez władze, zostanie natychmiast ujęty.

Najprościej byłoby więc ustawić patrol albo w punkcie początkowym transportu, albo w końcowym. Jednakże, kontrolowanie każdego punktu wiąże się z pewnym kosztem, który może być różny dla różnych punktów. Policjanci chcieliby uniknąć zbyt dużych kosztów operacji, więc planują znaleźć **minimalny zbiór kontrolny** *punktów poboru opłat, który spełniałby następujące warunki:*

- *każda trasa z portu do magazynu musi przechodzić przez co najmniej jeden punkt ze zbioru;*
- *koszt pilnowania tego zbioru punktów (tzn. suma kosztów obstawienia poszczególnych punktów z tego zbioru) jest minimalna.*

Możesz założyć, że istnieje co najmniej jeden sposób dostania się z portu do magazynu po sieci autostrad.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis sieci autostrad, koszty pilnowania punktów poboru opłat oraz miejsca wjazdu i zjazdu transportu z autostrady,*
- *znajdzie minimalny zbiór kontrolny punktów poboru opłat,*
- *wypisze ten zbiór na standardowe wyjście.*

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby całkowite n i m ($2 \leqslant n \leqslant 200$, $1 \leqslant m \leqslant 20\,000$) — liczbę punktów poboru opłat i liczbę odcinków autostrad. Punkty poboru opłat są ponumerowane od 1 do n.

Drugi wiersz zawiera dwie liczby całkowite a i b ($1 \leqslant a,b \leqslant n$, $a \neq b$) oznaczające numery punktów poboru opłat położonych w bezpośrednim sąsiedztwie, odpowiednio, portu i magazynu.

Kolejnych n wierszy zawiera opisy kosztów kontrolowania. W i-tym spośród tych wierszy (dla $1 \leqslant i \leqslant n$) znajduje się jedna liczba całkowita, nie większa niż 10 000 000 — koszt pilnowania przez policję i-tego punktu poboru opłat.

Kolejnych m wierszy zawiera opis sieci autostrad. W j-tym z nich (dla $1 \leqslant j \leqslant m$) są zapisane dwie liczby całkowite x i y ($1 \leqslant x < y \leqslant n$), oznaczające, że w sieci jest bezpośrednie połączenie pomiędzy punktami opłat o numerach x i y. Każdy istniejący odcinek autostrady jest wymieniony na wejściu dokładnie raz.

Dodatkowo, w testach, za które można uzyskać w sumie około 40% punktów, $n \leqslant 20$.

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać numery punktów poboru opłat, które tworzą minimalny zbiór kontrolny. Liczby powinny być podane w kolejności rosnącej i oddzielone pojedynczymi odstępami. Jeśli istnieje więcej niż jeden minimalny zbiór kontrolny, to Twój program powinien wypisać dowolny z nich.

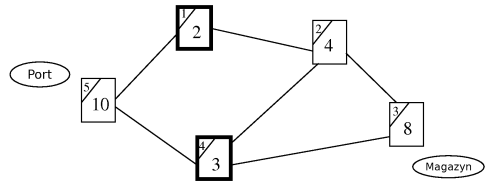
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
1 4
```



Rysunek przedstawia sieć autostrad z numerami punktów poboru opłat (w lewych górnych rogach) i kosztami pilnowania. Punkty o numerach 1 i 4 tworzą minimalny zbiór kontrolny o koszcie 5.

Tomasz Kulczyński, Jakub Łącki, Jakub Radoszewski
Tłumaczenie

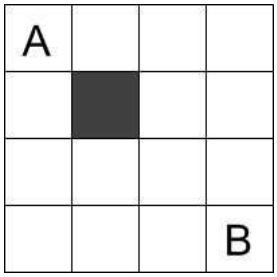
BOI 2008, dzień pierwszy, 19.04.2008

Gra

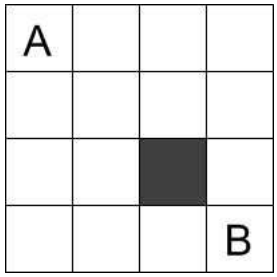
Dwaj gracze, A i B, grają w grę na kwadratowej planszy o rozmiarach $n \times n$. Pola planszy dzielą się na białe i czarne. Gra toczy się jedynie na białych polach. Każdy gracz ma swój pionek, początkowo ustawiony na **polu startowym** *danego gracza — jednym z białych pól planszy. Pola startowe graczy A i B są różne.*

W każdym ruchu gracz przesuwa swój pionek na jedno z białych pól sąsiadujących z jego pozycją (w górę, w dół, w lewo lub w prawo). Jeśli gracz przesunie pionek na pole zajmowane w tej samej chwili przez pionek przeciwnika, to wykonuje dodatkowy ruch (w ten sposób przeskakuje nad pionkiem przeciwnika). W takim przypadku kierunek drugiego ruchu może być różny od kierunku pierwszego ruchu.

Gracze poruszają się na zmianę, przy czym pierwszy ruch wykonuje gracz A. Celem gry jest przesunięcie własnego pionka na pole startowe przeciwnika. Gracz, który dokona tego wcześniej, wygrywa. Gracz wygrywa również w przypadku, gdy jego ruch składa się z dwóch posunięć i jedynie przechodzi on przez pole startowe przeciwnika (w sytuacji, gdy przeciwnik stoi na swoim polu startowym). Dla danej sytuacji chcemy stwierdzić, który z graczy ma strategię wygrywającą (mówimy, że gracz ma strategię wygrywającą, jeśli może wygrać grę niezależnie od ruchów przeciwnika).



Rysunek 1. *Jeśli gracz A poruszy się trzykrotnie w prawo w swoich trzech pierwszych ruchach, gracz B może odpowiedzieć na każdy ruch posunięciem w górę. Wtedy w trzecim ruchu gracz B wejdzie na pole zajmowane przez gracza A i wykona dodatkowy ruch. To pozwoli mu wcześniej dotrzeć do pola startowego gracza A i wygrać.*



Rysunek 2. *Gracz A może zacząć od ruchów w prawo i w dół. Później, zależnie od ruchów gracza B, może pójść w dół albo w prawo — tak, by uniknąć spotkania z pionkiem gracza B. W ten sposób A szybciej dotrze do pola startowego B, wygrywając grę. Widać więc, że w pokazanej sytuacji A ma strategię wygrywającą.*

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis planszy oraz położenie pól startowych obu zawodników,*
- *wyznaczy, który zawodnik ma strategią wygrywającą,*

- *wypisze wynik na standardowe wyjście.*

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę całkowitą t, oznaczającą liczbę przypadków testowych do rozpatrzenia ($1 \leqslant t \leqslant 10$). Po nim następuje kolejno t opisów testów. Każdy z testów jest opisany w następujący sposób. Pierwszy wiersz zawiera jedną liczbę całkowitą n ($2 \leqslant n \leqslant 300$), oznaczającą długość boku planszy. Kolejnych n wierszy zawiera opis planszy. Każdy z nich składa się z n znaków (bez odstępów pomiędzy nimi). Każdy ze znaków to: `'.'` *(białe pole),* `'#'` *(czarne pole),* `'A'` *(pole startowe gracza A) lub* `'B'` *(pole startowe gracza B).*

Możesz założyć, że istnieje ścieżka składająca się z białych pól, łącząca pola startowe graczy A i B.

Dodatkowo, w testach pozwalających uzyskać 60% punktów, zachodzi $n \leqslant 150$, zaś w testach pozwalających uzyskać 40% punktów, zachodzi $n \leqslant 40$.

Wyjście

Dla każdego zestawu testowego na wyjściu powinien znaleźć się dokładnie jeden wiersz zawierający pojedynczy znak `'A'` *lub* `'B'`*, oznaczający gracza, który posiada strategię wygrywającą.*

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`2` `4` `A...` `.#..` `....` `...B` `4` `A...` `....` `..#.` `...B`	`B` `A`

Tomasz Kulczyński, Jakub Łącki, Jakub Radoszewski
Tłumaczenie

BOI 2008, dzień pierwszy, 19.04.2008

Magiczne kamienie

W Krainie Czarów można znaleźć słynne kamienie Xi-n-k. Kamień tego rodzaju to nic innego, jak granitowa tabliczka z zamieszczonym na środku napisem złożonym z liter `X` *oraz* `I`*. Na tabliczce znajduje się w sumie n liter. Jest też nie więcej niż k miejsc, gdzie litery* `X` *i* `I` *sąsiadują ze sobą.*

Kamienie nie mają określonej góry i dołu, można je obracać „do góry nogami". Przykładowo, poniższe dwa rysunki przedstawiają ten sam kamień:

`IXXIIXXX` `XXXIIXXI`

Rys. 1: *Ten sam kamień pokazany na dwa różne sposoby. Jest to kamień rodzaju Xi-8-3, ale również Xi-8-4 (a także wszystkich innych rodzajów Xi-8-k dla* $k \geqslant 3$*).*

W Krainie Czarów nie ma dwóch takich samych kamieni, czyli takich, na których jest ten sam napis, nawet jeżeli dopuścimy obracanie kamieni „do góry nogami".

Jeżeli napis na jakimś kamieniu możemy odczytać na dwa różne sposoby (drugi sposób uzyskujemy, obracając kamień „do góry nogami"), to **kanoniczną reprezentacją** *kamienia nazwiemy ten z dwóch odczytanych napisów, który jest wcześniejszy alfabetycznie*[1]*.*

Jeśli napis na kamieniu brzmi tak samo czytany „normalnie" i „do góry nogami", to ma jedną reprezentację i jest to reprezentacja kanoniczna.

Przykład: *Jest dokładnie 6 kamieni Xi-3-2. Ich reprezentacje kanoniczne, wypisane w kolejności alfabetycznej, to:* `III`*,* `IIX`*,* `IXI`*,* `IXX`*,* `XIX` *i* `XXX`*.*

Alicja jest znanym w Krainie Czarów specjalistą od Xi-n-k kamieni. Chce ona stworzyć alfabetyczny spis reprezentacji kanonicznych wszystkich kamieni rodzaju Xi-n-k (dla pewnych konkretnych liczb n *i* k*). Co powinno się znaleźć w spisie na* i*-tej pozycji, dla danego* i*?*

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia liczby* n*,* k *oraz* i*,*
- *wyznaczy* i*-tą (w kolejności alfabetycznej) kanoniczną reprezentację Xi-*n*-*k *kamieni,*
- *wypisze wynik na standardowe wyjście.*

[1] Mówimy, że napis A jest alfabetycznie wcześniejszy/mniejszy od napisu B (takiej samej długości), jeśli na pierwszej pozycji od lewej strony, na której A i B się różnią, w napisie A znajduje się litera `I`, a w B — litera `X`.

Wejście

W pierwszym i jedynym wierszu wejścia znajdują się trzy liczby całkowite n, k *oraz* i ($0 \leqslant k < n \leqslant 60$, $0 < i < 10^{18}$), *pooddzielane pojedynczymi odstępami.*

Wyjście

W pierwszym i jedynym wierszu wyjścia Twój program powinien wypisać i*-tą w kolejności alfabetycznej kanoniczną reprezentację Xi-*n*-*k *kamieni.*

Jeżeli liczba Xi-n*-*k *kamieni jest mniejsza niż* i*, program powinien wypisać jeden wiersz z napisem* `NO SUCH STONE`.

Przykład

Dla danych wejściowych:
```
3 2 5
```
poprawnym wynikiem jest:
```
XIX
```

natomiast dla danych:
```
3 2 7
```
poprawnym wynikiem jest:
```
NO SUCH STONE
```

Tomasz Kulczyński, Jakub Łącki, Jakub Radoszewski
Tłumaczenie

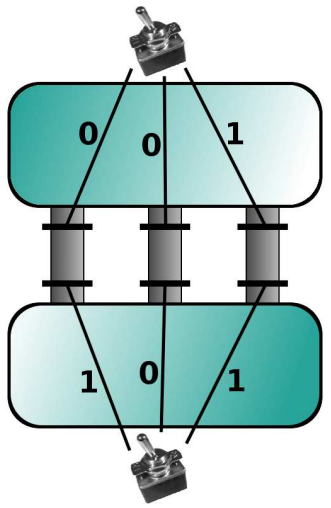
BOI 2008, dzień pierwszy, 19.04.2008

Śluzy

Lata pracy w inżynierii oprogramowania mogą spowodować ciężkie komplikacje psychiczne. Dlatego też programista Bajtazar postanowił porzucić dotychczasowe zajęcie i zaczął się przyglądać losowo wybranym ofertom zatrudnienia. Najbardziej zainteresowała go oferta pracy przy hodowli ryb. „Fajna sprawa," — pomyślał Bajtazar — „w końcu rybki to całkiem przyjemne stworzonka". Bajtazar złożył podanie o interesującą go posadę, przeszedł pozytywnie proces rekrutacji i dzisiaj po raz pierwszy przyszedł do nowej pracy.

Nowy szef już zdążył przydzielić Bajtazarowi pierwsze zadanie. Musi on odizolować jeden zbiornik wodny od drugiego. Bajtazar przyjrzał się już sposobowi, w jaki zbiorniki są połączone, i oto co wywnioskował.

Dwa zbiorniki wodne są połączone pewną liczbą kanałów. Na każdym kanale znajdują się dwie śluzy. Kanał jest otwarty wtedy i tylko wtedy, gdy obie znajdujące się na nim śluzy są otwarte. Śluzy można otwierać i zamykać za pomocą przełączników. Jeden przełącznik może być podłączony do wielu śluz, ale jedna śluza jest zawsze podłączona do dokładnie jednego przełącznika. Może się tak zdarzyć, że dwie śluzy znajdujące się na jednym kanale są podłączone do jednego przełącznika, a także że jakiś przełącznik nie jest podłączony do żadnej śluzy.



Rys. 1: *Przykład zawierający trzy kanały i dwa przełączniki.*

Przełącznik może sterować określoną śluzą w jeden z poniższych sposobów:

- *śluza jest otwarta, jeżeli przełącznik jest włączony, a zamknięta, jeżeli przełącznik jest wyłączony,*
- *śluza jest zamknięta, jeżeli przełącznik jest włączony, a otwarta, jeżeli przełącznik jest wyłączony.*

Bajtazar poeksperymentował trochę z przełącznikami i doszedł do wniosku, że do wykonania zadania może mu się przydać doświadczenie programistyczne. Napisz program, który na podstawie opisu połączeń śluz i przełączników sprawdzi, czy da się zamknąć wszystkie kanały, a jeżeli jest to możliwe, to poda stany wszystkich przełączników w konfiguracji, powodującej zamknięcie wszystkich kanałów.

Wejście

Pierwszy wiersz standardowego wejścia zawiera dwie liczby całkowite n ($1 \leqslant n \leqslant 250\,000$) oraz m ($1 \leqslant m \leqslant 500\,000$), oznaczające odpowiednio liczbę kanałów i liczbę przełączników. Przełączniki są ponumerowane od 1 do m. Dodatkowo, w testach, za które można uzyskać co najmniej 30% punktów, n nie przekracza 40, a m jest nie większe niż 20.

Kolejnych n wierszy zawiera opisy kanałów; każdy kanał jest opisany w osobnym wierszu za pomocą czterech liczb całkowitych: a, s_a, b, s_b. Liczby a i b reprezentują przełączniki ($1 \leqslant a,b \leqslant m$), które sterują śluzami danego kanału. Liczby s_a i s_b mogą przyjmować wartości 0 lub 1; opisują one sposób sterowania śluzami: $s_i = 0$ oznacza, że śluza jest zamknięta wtedy i tylko wtedy, gdy i-ty przełącznik jest wyłączony, natomiast $s_i = 1$ oznacza, że śluza jest zamknięta wtedy i tylko wtedy, kiedy i-ty przełącznik jest włączony.

Wyjście

Jeżeli da się zamknąć wszystkie kanały, to na standardowe wyjście powinno zostać wypisanych m wierszy. W i-tym z tych wierszy powinno być 0, jeżeli i-ty przełącznik powinien być wyłączony, a 1 — jeżeli powinien być włączony. Jeżeli istnieje więcej niż jedno poprawne rozwiązanie, Twój program powinien wypisać którekolwiek z nich.

Jeżeli nie jest możliwe zamknięcie wszystkich kanałów, to Twój program powinien wypisać jeden wiersz, zawierający jedno słowo `IMPOSSIBLE`.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`3 2` `1 0 2 1` `1 0 2 0` `1 1 2 1`	`0` `1`

natomiast dla danych wejściowych:	*poprawnym wynikiem jest:*
`2 1` `1 0 1 0` `1 1 1 1`	`IMPOSSIBLE`

Pierwszy z przykładów odpowiada rysunkowi z treści zadania.

Tomasz Kulczyński, Jakub Łącki, Jakub Radoszewski
Tłumaczenie

BOI 2008, dzień drugi, 21.04.2008

Mapka w kratkę

Mapa Bajtocji, wraz z naniesionymi południkami i równoleżnikami, tworzy prostokąt o wymiarach $n \times m$ (n to jego wysokość, a m — szerokość). Równoleżniki są oznaczone na mapie poziomymi liniami, ponumerowanymi od 0 do n, zaś południki to linie pionowe o numerach od 0 do m (patrz także rysunek).

Prognozowanie pogody w Bajtocji jest zadaniem niełatwym. Dla każdego obszaru jednostkowego (tj. leżącego pomiędzy sąsiednimi południkami i równoleżnikami) wiemy, jak długo bajtocki komputer meteorologiczny oblicza prognozę. Uwarunkowania topograficzne powodują, że czas ten może być różny dla różnych obszarów. Dotychczas prognoza pogody przygotowywana była w najprostszy możliwy sposób: jeden komputer obliczał prognozę kolejno dla wszystkich obszarów jednostkowych. Obliczenie prognozy pogody dla całej Bajtocji zajmowało więc tyle czasu, ile wynosi suma wszystkich czasów obliczeń dla poszczególnych obszarów.

Zostałeś poproszony o opracowanie nowego systemu przeznaczonego na komputer wieloprocesorowy. Aby rozdzielić obliczenia pomiędzy procesory, teren Bajtocji należy podzielić, wybierając r równoleżników i s południków. Wybrane linie podzielą całość na $(r+1)(s+1)$ mniejszych prostokątów. Każdy z procesorów będzie obliczał pogodę dla jednego prostokąta z tego podziału, przetwarzając kolejno jego obszary jednostkowe. Wszystkie procesory będą pracować jednocześnie. Cała prognoza pogody będzie gotowa, gdy tylko ostatni procesor zakończy swoją pracę.

Twoim zadaniem jest wyznaczenie minimalnego czasu obliczenia prognozy pogody dla całej Bajtocji, podzielonej na mniejsze obszary przez wybór r równoleżników i s południków.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia rozmiary mapy Bajtocji, liczby południków i równoleżników oraz czasy wyznaczania prognozy pogody dla poszczególnych obszarów jednostkowych,*
- *wyznaczy minimalny czas potrzebny na obliczenie prognozy pogody,*
- *wypisze znaleziony wynik na standardowe wyjście.*

Wejście

Pierwszy wiersz wejścia zawiera cztery liczby całkowite n, m, r i s, pooddzielane pojedynczymi odstępami ($1 \leqslant r < n \leqslant 18$, $1 \leqslant s < m \leqslant 18$). Kolejnych n wierszy zawiera czasy obliczenia prognozy pogody dla poszczególnych obszarów jednostkowych. Liczba j-ta w wierszu $(i+1)$-szym oznacza $c_{i,j}$ — czas potrzebny do obliczenia prognozy pogody dla obszaru pomiędzy $(i-1)$-szym a i-tym równoleżnikiem oraz pomiędzy $(j-1)$-szym a j-tym południkiem ($1 \leqslant i \leqslant n$, $1 \leqslant j \leqslant m$, $0 \leqslant c_{i,j} \leqslant 2\ 000\ 000$).

Dodatkowo, w testach, za które można uzyskać sumarycznie 40% punktów, n i m nie przekraczają 10.

Wyjście

Twój program powinien wypisać dokładnie jeden wiersz zawierający jedną liczbą całkowitą — optymalny czas obliczenia prognozy pogody.

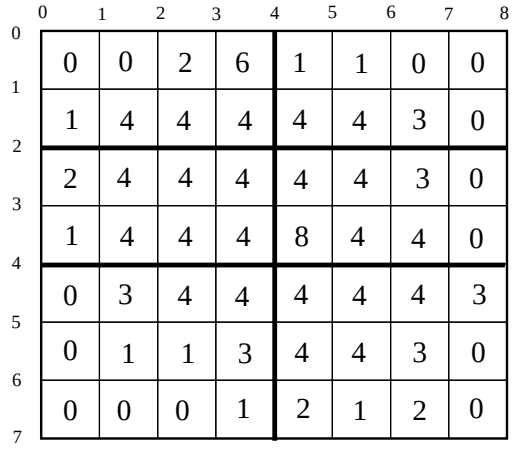
Przykład

Dla danych wejściowych:

```
7 8 2 1
0 0 2 6 1 1 0 0
1 4 4 4 4 4 3 0
2 4 4 4 4 4 3 0
1 4 4 4 8 4 4 0
0 3 4 4 4 4 4 3
0 1 1 3 4 4 3 0
0 0 0 1 2 1 2 0
```

poprawnym wynikiem jest:

```
31
```



Drugi i czwarty równoleżnik oraz czwarty południk dzielą Bajtocję na 6 obszarów, dla których czasy obliczenia prognozy pogody to: 21, 13, 27, 27, 17, 31. Czas obliczenia prognozy dla całej Bajtocji to 31.

Tomasz Kulczyński, Jakub Łącki, Jakub Radoszewski
Tłumaczenie

BOI 2008, dzień drugi, 21.04.2008

Rękawiczki

W ciemnej piwnicy domu znanego chemika, profesora Bajkwasa, znajdują się dwie szuflady pełne rękawiczek — jedna z lewymi, a druga z prawymi rękawiczkami. W każdej z nich są rękawiczki w n różnych kolorach. Profesor wie, ile rękawiczek każdego koloru znajduje się w każdej z szuflad (liczba rękawiczek danego koloru może być różna w różnych szufladach). Jest także pewien, że możliwe jest znalezienie pary rękawiczek tego samego koloru.

Eksperyment Bajkwasa może się powieść tylko wtedy, gdy profesor użyje rękawiczek tego samego koloru (nie ma znaczenia którego), więc przed każdym doświadczeniem idzie on do piwnicy i bierze rękawiczki z szuflad, mając nadzieję, że znajdzie się wśród nich co najmniej jedna para w tym samym kolorze. W piwnicy jest bardzo ciemno i nie ma możliwości rozpoznania koloru żadnej rękawiczki bez wychodzenia z piwnicy. Profesor nie chce chodzić do piwnicy więcej niż raz (gdyby nie było pary rękawiczek jednego koloru), nie lubi też przynosić niepotrzebnie wielu rękawiczek do laboratorium.

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia liczbę kolorów, a następnie liczby rękawiczek w każdym z kolorów w każdej z szuflad,*
- *obliczy najmniejszą łączną liczbę rękawiczek, które profesor musi przynieść z piwnicy do laboratorium, aby mieć pewność, że jest wśród nich co najmniej jedna para rękawiczek tego samego koloru (należy określić liczbę rękawiczek wziętych z każdej szuflady),*
- *wypisze wynik na standardowe wyjście.*

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę całkowitą n ($1 \leqslant n \leqslant 20$) oznaczającą liczbę różnych kolorów. Kolory są ponumerowane od 1 do n. Drugi wiersz wejścia zawiera n liczb całkowitych $0 \leqslant a_1, a_2, \ldots, a_n \leqslant 10^8$, gdzie a_i to liczba rękawiczek koloru i w szufladzie z lewymi rękawiczkami. Wreszcie, trzeci wiersz wejścia zawiera n liczb całkowitych $0 \leqslant b_1, b_2, \ldots, b_n \leqslant 10^8$, gdzie b_i to liczba rękawiczek koloru i w szufladzie z prawymi rękawiczkami.

Dodatkowo, w testach, za które można uzyskać 40% punktów, $n \leqslant 4$ i $a_i, b_i \leqslant 10$.

Wyjście

Pierwszy wiersz wyjścia powinien zawierać jedną liczbę całkowitą — liczbę rękawiczek, które profesor musi wyjąć z szuflady z lewymi rękawiczkami. Drugi wiersz wyjścia także powinien zawierać jedną liczbę całkowitą — liczbę rękawiczek, które profesor musi wyjąć z szuflady

z prawymi rękawiczkami. Suma tych dwóch liczb powinna być najmniejsza z możliwych. Jeśli istnieje wiele poprawnych wyników, Twój program powinien wypisać dowolny z nich.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`4` `0 7 1 6` `1 5 0 6`	`2` `8`

Tomasz Kulczyński, Jakub Łącki, Jakub Radoszewski
Tłumaczenie

BOI 2008, dzień drugi, 21.04.2008

Wybory

W Bajtocji niedawno odbyły się wybory do parlamentu. Teraz, kiedy wyniki zostały już ogłoszone, partie polityczne przymierzają się do utworzenia koalicji.

Każda partia uzyskała pewną liczbę miejsc w parlamencie. **Koalicją** *nazywamy taki podzbiór zbioru wszystkich partii, który dysponuje łącznie więcej niż połową miejsc w parlamencie. Wskazane jest, aby koalicja miała łącznie tak wiele miejsc, jak to tylko możliwe, gdyż umożliwia to przegłosowanie ustaw nawet w przypadku, gdy niektórzy posłowie z koalicji są nieobecni na posiedzeniu parlamentu.*

Koalicję nazywamy **redundantną**, *jeżeli można z niej usunąć jakąś partię, a pozostałe w koalicji partie będą wciąż posiadały ponad połowę miejsc w parlamencie. Oczywiście taka „zbędna" partia nie miałaby praktycznie żadnej władzy — pozostali członkowie koalicji mogliby przegłosowywać ustawy niezależnie od jej opinii.*

Zadanie

Napisz program, który:

- *wczyta wyniki wyborów ze standardowego wejścia,*
- *znajdzie nieredundantną koalicję, która dysponuje największą możliwą liczbą głosów w parlamencie,*
- *wypisze opis znalezionej koalicji na standardowe wyjście.*

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą n ($1 \leqslant n \leqslant 300$) — liczbę partii, które brały udział w wyborach. Partie są ponumerowane od 1 do n.

Drugi wiersz zawiera n nieujemnych liczb całkowitych $a_1, \ldots, a_n$, pooddzielanych pojedynczymi odstępami, gdzie a_i oznacza liczbę miejsc zdobytych w wyborach przez i-tą partię. Możesz założyć, że łączna liczba miejsc w parlamencie jest dodatnią liczbą nie większą niż 100 000.

Dodatkowo, w testach, za które można uzyskać łącznie 40% punktów, liczba partii nie przekracza 20.

Wyjście

Pierwszy wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą k — liczbę partii tworzących nieredundantną koalicję, która dysponuje największą możliwą liczbą głosów.

Drugi wiersz wyjścia powinien zawierać k różnych liczb całkowitych, pooddzielanych pojedynczymi odstępami i oznaczających numery partii tworzących powyższą koalicję.

Jeżeli istnieje więcej niż jedna nieredundantna koalicja posiadająca maksymalną liczbę miejsc w parlamencie, Twój program powinien wypisać dowolną z nich. Partie tworzące koalicję mogą zostać wypisane w dowolnej kolejności.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`4` `1 3 2 4`	`2` `2 4`

XV Olimpiada Informatyczna Krajów Europy Środkowej,

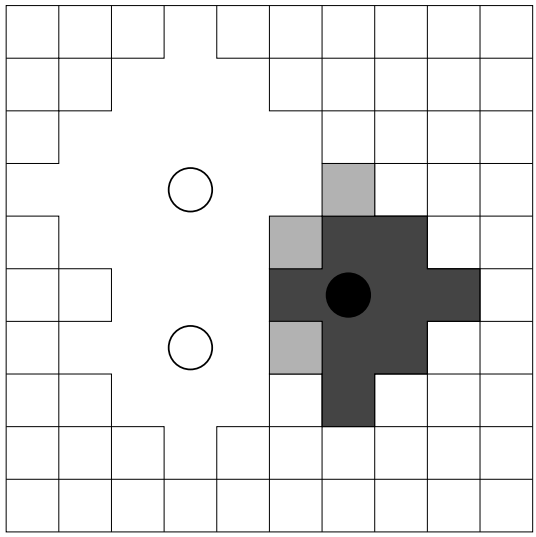
Drezno, Niemcy 2008

Tomasz Kulczyński, Jakub Radoszewski
Tłumaczenie

CEOI 2008, dzień pierwszy, 08.07.2008

Dominacja

Bugtopia to prostokątny obszar złożony z $W \times H$ kwadratów, zamieszkany przez białe i czarne robaki. Każdy kwadrat jest zasiedlony albo wyłącznie przez białe robaki (wtedy nazywamy go „białym polem"), albo wyłącznie przez czarne robaki (wtedy nazywamy go „czarnym polem"), albo w ogóle nie jest zamieszkany. Różnie ubarwione robaki są dość agresywne w stosunku do siebie i każdy z gatunków dąży do samodzielnego zawładnięcia Bugtopią. W tym celu robaki mogą poruszać się po planszy — pojedynczym ruchem robaka nazywamy przejście na kwadrat sąsiadujący z aktualnym w pionie lub poziomie. Robaki z jednego pola są w stanie zaatakować inne pola tylko wtedy, gdy mogą do nich dojść, nie przekraczając ustalonej z góry liczby kroków, którą nazwiemy „zasięgiem" robaka. „Zasięg" zależy od pola macierzystego robaka — różne pola zapewniają robakom różne warunki życia, więc robaki pochodzące z różnych pól mogą mieć różny zasięg, ale wszystkie robaki pochodzące z tego samego pola mają taki sam zasięg. Mówimy, że kwadrat Bugtopii jest **zdominowany przez białe robaki**, *jeśli może być zaatakowany z większej liczby pól białych niż czarnych. Podobnie, kwadrat Bugtopii jest* **zdominowany przez czarne robaki**, *jeśli może być zaatakowany z większej liczby pól czarnych niż białych. Kwadrat nazywamy* **neutralnym**, *jeśli nie może być zaatakowany wcale lub może być zaatakowany z jednakowej liczby pól białych i czarnych.*



Rys. 1: *Na mapie Bugtopii widzimy dwa białe pola (zaznaczone białymi kółkami) o zasięgach 3 oraz 2, a także jedno czarne pole (oznaczone czarnym kółkiem) o zasięgu 2. 30 pól jest zdominowanych przez białe robaki, a 9 pól — przez czarne robaki. Trzy pola zamalowane na szaro to pola neutralne, które mogą być zaatakowane, ale nie są zdominowane przez żaden z rodzajów robaków.*

Na podstawie rozmiaru planszy, pozycji, kolorów i zasięgów zamieszkanych pól, oblicz i wypisz całkowitą liczbę pól zdominowanych przez każdy z rodzajów robaków.

Wejście

Pierwszy wiersz zawiera dwie liczby całkowite W i H, oznaczające rozmiary Bugtopii ($1 \leqslant W,H \leqslant 1\,000\,000\,000$). Kolejny wiersz zawiera jedną liczbę całkowitą N ($0 \leqslant N \leqslant 3\,000$), oznaczającą liczbę zamieszkanych pól.

Każdy z kolejnych N wierszy zawiera opis zamieszkanego kwadratu. Wiersz o numerze $i+2$ zawiera wartości c_i, x_i ($0 \leqslant x_i < W$), y_i ($0 \leqslant y_i < H$) oraz r_i ($0 \leqslant r_i < 500\,000\,000$), pooddzielane pojedynczymi odstępami. Oznaczają one odpowiednio kolor pola, jego współrzędne oraz zasięg. Kolor pola jest jedną z liter 'W' (dla pola białego) i 'B' (dla pola czarnego). Wszystkie pola, które mogą zostać zaatakowane z zamieszkanego pola, zawsze leżą w granicach Bugtopii. Lewe dolne pole planszy ma współrzędne $(0,0)$, a prawe górne — współrzędne $(W-1,H-1)$.

W co najmniej 30% testów zachodzi silniejsze ograniczenie: $W,H \leqslant 2\,000$.

Wyjście

Wypisz jeden wiersz z dwiema liczbami całkowitymi oddzielonymi pojedynczym odstępem: liczbą pól zdominowanych przez białe robaki oraz liczbą pól zdominowanych przez czarne robaki.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`10 10` `3` `W 3 6 3` `B 6 4 2` `W 3 3 2`	`30 9`

Przykład jest odwzorowaniem rysunku z treści zadania.

Tomasz Kulczyński, Jakub Radoszewski
Tłumaczenie

CEOI 2008, dzień pierwszy, 08.07.2008

Informacja

Tajna agencja wywiadowcza (tak tajna, że nawet nie wymienimy tu jej nazwy) zatrudnia **agentów** *z całego świata. Od czasu do czasu kwatera główna musi rozesłać jakąś wiadomość do wszystkich agentów. Z oczywistych względów, transmisja musi być tak bezpieczna, jak tylko się da.*

Szefowie agencji nie ufają komunikacji elektronicznej, więc przesyłają wiadomości poprzez wyznaczone osoby kontaktowe (w skrócie nazywane **kontaktami***). Agentów i kontakty zorganizowano w rozległą sieć. Każdy kontakt przekazuje informacje tylko między określoną parą agentów i to wyłącznie w jednym kierunku. Może jednak istnieć więcej niż jeden kontakt przekazujący informacje między określoną parą agentów w każdym z kierunków.*

Wiadomości z kwatery głównej rozsyła „oficer komunikacyjny". Wykorzystuje on pewne spośród swoich kontaktów, by przekazać wiadomość do wybranych agentów polowych. Następnie, agenci, którzy otrzymali wiadomość, wykorzystują swoje kontakty, by przekazać wiadomość do innych agentów i tak dalej, aż w końcu dotrze ona do wszystkich agentów. W celu zmniejszenia ryzyka, liczba przekazań wiadomości przez kontakty powinna być minimalna (tzn. żaden agent nie powinien otrzymać wiadomości dwukrotnie). Agenci nie przesyłają zatem wiadomości za pośrednictwem wszystkich swoich kontaktów, lecz podporządkowują się **schematowi transmisyjnemu** *zaplanowanemu dla tej wiadomości — zawiera on informację, z których kontaktów powinni korzystać do jej przekazania.*

Niedawno agencja dowiedziała się, że pewne kontakty w niecny sposób wykorzystują poufne informacje. Z tego powodu postanowiono wysyłać każdą wiadomość podzieloną na dwie **części**. *Są one bezużyteczne, jeśli nie są przeczytane naraz. Teraz obie części powinny zostać rozesłane tak, by żaden kontakt nie został wykorzystany dwukrotnie — w ten sposób żaden kontakt nigdy nie dostanie obu części (i tym samym całej) wiadomości. Niemniej jednak ważne jest, aby każdy agent w końcu otrzymał obie części wiadomości. Agencja zastanawia się teraz, jak stworzyć* **poprawny** *schemat transmisyjny dla każdej części wiadomości, który spełnia powyższe warunki.*

Zadanie

Napisz program, który wyznaczy poprawny schemat transmisyjny dla każdej części wiadomości, mając daną sieć agentów i kontaktów. Może się zdarzyć, że nie będą istniały takie dwa schematy.

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby całkowite N ($2 \leqslant N \leqslant 2\,000$) — liczbę agentów, oraz M ($1 \leqslant M \leqslant 1\,000\,000$) — liczbę kontaktów. Oficer komunikacyjny z kwatery głównej ma numer 1, pozostali agenci są ponumerowani od 2 do N; kontakty mają numery od 1 do M. W i-tym z kolejnych M wierszy są zapisane dwie liczby całkowite v_i oraz w_i ($v_i \neq w_i$), oznaczające, że kontakt i przekazuje wiadomości od agenta v_i do agenta w_i.

Wyjście

Jeżeli nie istnieje poszukiwana para schematów transmisyjnych, to wyjście powinno składać się z jednego wiersza zawierającego słowo `NONE`.

W przeciwnym przypadku wyjście powinno składać się z dwóch wierszy. Każdy wiersz powinien opisywać poprawny schemat transmisyjny dla jednej części wiadomości — pierwszy wiersz dla pierwszej części, drugi wiersz dla drugiej części. Opis to lista kontaktów (numerów pooddzielanych pojedynczymi odstępami) wykorzystanych do przekazania wiadomości.

Jeżeli istnieje więcej niż jedno rozwiązanie, wypisz dowolne z nich.

Przykład

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

Wyjaśnienie

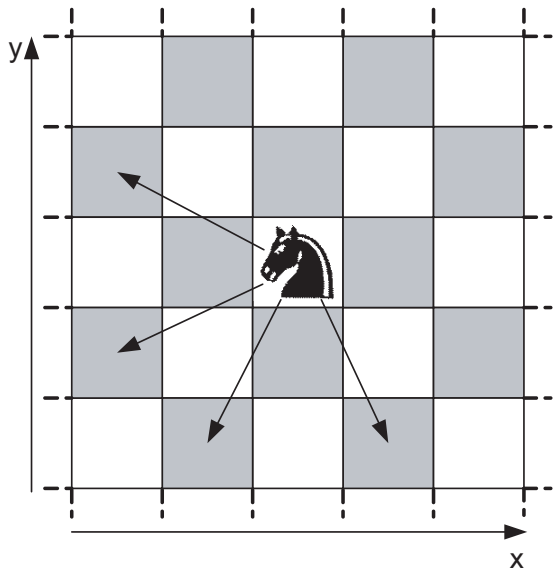
Pierwsza część wiadomości zostaje przekazana za pomocą kontaktów 1, 3 i 5, tzn. z kwatery głównej do agenta 2 (przez kontakt 1), od agenta 2 do agenta 3 (przez kontakt 3) i również od agenta 2 do agenta 4 (przez kontakt 5). Druga część wiadomości zostaje przekazana za pomocą kontaktów 2, 4 i 6, tzn. z kwatery głównej do agenta 3 (przez kontakt 2), od agenta 3 do agenta 2 (przez kontakt 4) i od agenta 2 do agenta 4 (przez kontakt 6).

Tomasz Kulczyński, Jakub Radoszewski
Tłumaczenie

CEOI 2008, dzień pierwszy, 08.07.2008

Skoczki

Alicja i Bob grają w ciekawą grę. Początkowo, na szachownicy o rozmiarze $N \times N$ stoi K czarnych skoczków. Gracze zaczynają wykonywać ruchy na przemian. W każdym ruchu gracz przesuwa każdego ze skoczków, który ma możliwość ruchu. Prawidłowe jest każde z następujących czterech posunięć, o ile nie powoduje ono opuszczenia szachownicy przez skoczka:



Skoczek, którego nie można prawidłowo przesunąć, pozostaje na swojej pozycji. Przegrywa gracz, który nie jest w stanie wykonać posunięcia żadnym skoczkiem. Podczas gry kilka skoczków może jednocześnie stać na tym samym polu.

Masz dane pozycje skoczków na szachownicy. Alicja zaczyna grę. Sprawdź, czy może ona wygrać, zakładając, że obaj gracze grają optymalnie. Jeśli Alicja może wygrać, to wypisz wszystkie posunięcia skoczkami, które powinna wykonać w pierwszym ruchu. Przyjmij, że w pozycji początkowej każdy ze skoczków ma możliwość ruchu i że żadne dwa skoczki nie stoją na tym samym polu.

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby całkowite K $(1 \leqslant K \leqslant 200\,000)$ i N $(1 \leqslant N \leqslant 1\,000\,000)$ oddzielone pojedynczym odstępem. Po nim następuje K wierszy opisujących pozycje początkowe skoczków.

Wiersz o numerze $i+1$ zawiera dwie liczby całkowite x_i i y_i $(1 \leqslant x_i, y_i \leqslant N)$ oddzielone pojedynczym odstępem, będące współrzędnymi skoczka o numerze i.

Wyjście

Jeśli Alicja nie może wygrać, wyjście powinno składać się z jednego wiersza zawierającego słowo `NO`*. W przeciwnym przypadku należy wypisać* $K + 1$ *wierszy. Pierwszy powinien zawierać jedno słowo:* `YES`*. W wierszu* $(i + 1)$*-szym powinny znaleźć się współrzędne* x'_i, y'_i*, oznaczające pole, na które Alicja powinna przesunąć skoczka o numerze* i*, aby zapewnić sobie zwycięstwo.*

Przykład

Dla danych wejściowych:

```
2 3
2 3
3 2
```

poprawnym wynikiem jest:

```
YES
1 1
1 1
```

natomiast dla danych:

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

poprawnym wynikiem jest:

```
NO
```

Tomasz Kulczyński, Jakub Radoszewski
Tłumaczenie

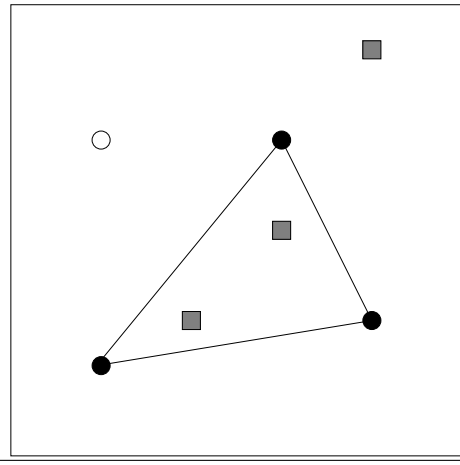
CEOI 2008, dzień drugi, 10.07.2008

Płot

Pewnego ranka sadownik Franek udał się obejrzeć swoje jabłonie i zauważył, że jedna z nich została w nocy ścięta. Oznacza to dla niego stratę 111 € — *tyle pieniędzy zarobiłby średnio, sprzedając jabłka z jednego drzewa. Aby uniknąć dalszych strat, postanowił zainwestować i ogrodzić sad płotem.*

Płot będzie się składać ze słupków połączonych siatką. Siatkę na płot Franek ma za darmo, musi jednak zakupić słupki, płacąc 20 € za sztukę. Na dodatek słupki może umieścić tylko w znajdujących się już w sadzie dziurach. Może więc okazać się, że nie będzie mu się opłacało ogrodzić wszystkich drzew, a nawet że nie będzie warto w ogóle stawiać płotu.

Sad jest kwadratem o rozmiarze 1 000 × 1 000 m^2. Jeżeli spojrzeć z lotu ptaka, lewy dolny róg ma współrzędne (0, 0), a prawy górny (1 000, 1 000).



Rys. 1: *W pokazanym sadzie są cztery dziury na słupki (zaznaczone kółkami) i trzy drzewa (narysowane jako kwadraty). Frankowi najbardziej opłaca się kupić trzy słupki i wstawić je do dziur zaznaczonych na czarno, połączyć je siatką i lewą górną dziurę pozostawić pustą. Koszt takiej inwestycji to* $3 \cdot 20$ € $+ 1 \cdot 111$ € $= 171$ €*, gdyż należy zakupić trzy słupki i pogodzić się ze stratą zysku z drzewa, które nie mieści się wewnątrz ogrodzenia.*

Napisz program, który wczyta pozycje dziur oraz drzew w sadzie Franka i wypisze minimalny koszt inwestycji (zakupu słupków i strat zysku z nieogrodzonych jabłoni). Możesz zaniedbać rzeczywiste rozmiary drzew i dziur, uznając, że drzewa i dziury są punktami na planie sadu.

Wejście

Pierwszy wiersz zawiera dwie liczby całkowite N *i* M *(*$3 \leqslant N \leqslant 100, 1 \leqslant M \leqslant 100$*)* — N *jest liczbą wywierconych dziur, a* M *jest liczbą drzew. W kolejnych* N *wierszach są podane*

pozycje dziur, a w kolejnych M wierszach — pozycje drzew. Każda pozycja to para liczb całkowitych (x,y) zapisanych w jednym wierszu i oddzielonych odstępem ($0 \leqslant x,y \leqslant 1\,000$). Możesz założyć, że żadne dwie pozycje (dziur lub drzew) nie są takie same oraz żadne trzy pozycje nie leżą na jednej prostej.

Wyjście

Wypisz jeden wiersz zawierający jedną liczbę całkowitą: minimalny koszt inwestycji Franka. Jeśli Franek kupi P słupków i nie uda mu się ogrodzić T drzew, to koszt wyniesie $20P+111T$.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
4 3 800 300 200 200 200 700 600 700 400 300 600 500 800 900	171

Przykład odpowiada rysunkowi z treści zadania.

Tomasz Kulczyński, Jakub Radoszewski
Tłumaczenie

CEOI 2008, dzień drugi, 10.07.2008

Wąż

Podczas wycieczki do Sächsische Schweiz w trakcie CEOI niewidzialny magiczny wąż szepcze do Ciebie: „Dam Ci punkty, jeżeli odgadniesz moją długość". Nie widzisz węża, więc początkowo nie wiesz nic o jego położeniu. Słyszysz głos węża ponownie: „Oddalam się od Ciebie wzdłuż linii prostej, która ma początek w miejscu, w którym stoisz. Pozostaję jednakże w zasięgu Twojego słuchu". Wiesz, że potrafisz usłyszeć głos węża z odległości 12 122 jednostek. Decydujesz się zatem podzielić prostą na 12 122 jednostek ponumerowanych od 0 do 12 121, gdzie 0 jest Twoim położeniem.

Teraz wąż przemawia do Ciebie ponownie: „Możesz mi zadawać pytania — po dwa w jednej rundzie. Wszystkie pytania muszą mieć następującą postać:

> *Czy leżysz na jednostce U?"*

Na każde pytanie wąż obiecuje dać jedną z następujących odpowiedzi:

> *Tak, część mojego ciała leży na jednostce U.*
>
> *ALBO*
>
> *Nie, nie dotarłem jeszcze do jednostki U.*
>
> *ALBO*
>
> *Nie, już oddaliłem się poza jednostkę U* ☺.

Niestety, tuż przed odpowiedzią na kolejną rundę pytań wąż może przemieścić się **do przodu** *dokładnie o K jednostek. Na dodatek jego decyzje o wykonaniu ruchu* **nie** *muszą być zgodne z jakimkolwiek schematem; może on specjalnie poruszać się wyjątkowo złośliwie.*

Twoim celem jest oszacowanie długości węża, na podstawie otrzymanych od niego informacji. Dokładne wyznaczenie tej długości może nie być możliwe, ale uzyskasz punkty, jeżeli pomylisz się **co najwyżej o K jednostek**.

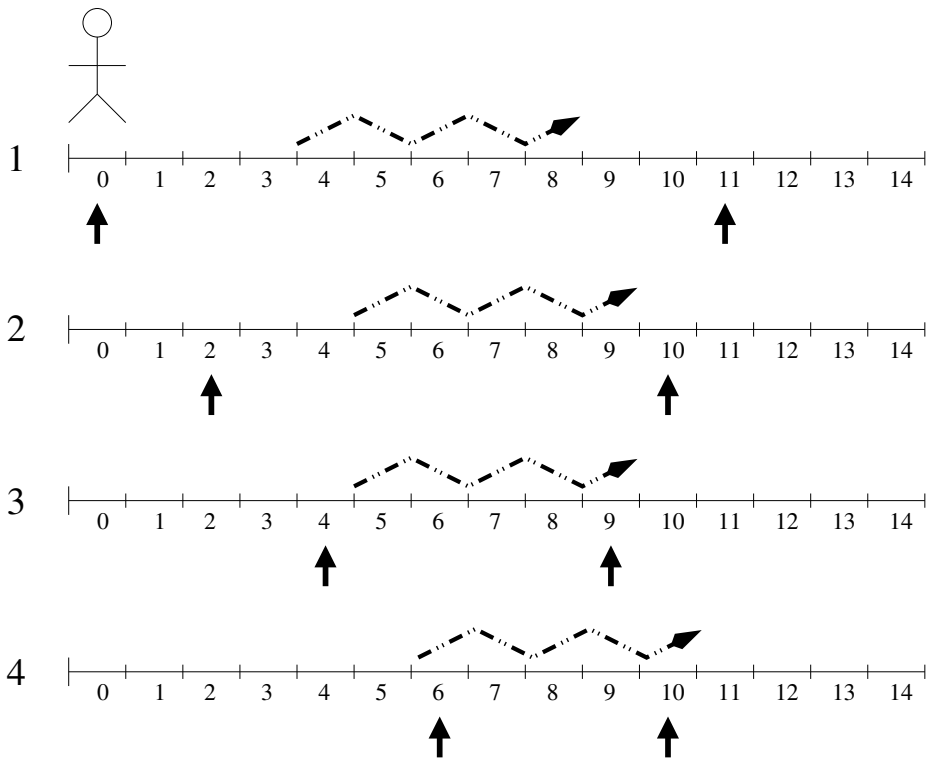
Przykład

Na rysunku widzimy węża o długości 5. Wiesz, że jego prędkość K jest równa 1. Nie wiesz natomiast, że początkowo rozciąga się on od jednostki 4. do jednostki 8. oraz że postanowił przemieszczać się przy co drugiej rundzie pytań.

Teraz przyszła pora na zadawanie pytań. W pierwszej rundzie pytasz węża o jednostki 0 i 11. Odpowiada on, że oddalił się już poza jednostkę 0 i jeszcze nie dotarł do jednostki 11.

Oczywiście masz za mało informacji — zaczynasz kolejną rundę pytań. Tym razem pytasz o jednostki 2 i 10. Wąż przemieszcza się o 1 jednostkę i odpowiada, że przemieścił się już poza jednostkę 2 i nie dotarł jeszcze do jednostki 10. W tym momencie jesteś pewien, że długość węża jest liczbą między 1 a 7 i że jego ciało rozciąga się gdzieś między jednostkami 3 i 9 (włącznie).

Zgodnie ze swoim planem, wąż nie przemieszcza się przed odpowiedzią na trzecią rundę pytań, w której pytasz o jednostki 4 oraz 9. Odpowiada, że część jego ciała leży na jednostce



Rys. 1: *Możliwy przebieg zabawy z wężem długości 5.*

9 i przemieścił się poza jednostkę 4. Możesz teraz stwierdzić, że głowa węża znajduje się maksymalnie na 10. jednostce (pamiętaj, że nie wiesz, kiedy wąż się przesuwa); możesz także wydedukować, że długość węża jest liczbą pomiędzy 1 a 6.

Ponieważ wąż może przemieszczać się z prędkością 1 jednostki, postanawiasz spytać w czwartej rundzie, czy leży na jednostkach 6 oraz 10. Wąż przemieszcza się o 1 jednostkę i odpowiada, że owszem, leży na jednostkach 6 oraz 10. Wnioskujesz, że wąż może mieć długość 5 albo 6. (Wąż może leżeć na pozycjach: od 5 do 10, od 6 do 10 oraz od 6 do 11. Nie jest możliwe, żeby wąż w całości leżał na pozycjach od 5 do 11, gdyż wówczas w poprzednim zapytaniu odpowiedziałby, że część jego ciała leży na jednostce 4.) Znasz już dostatecznie dobre oszacowanie długości węża i możesz podać odpowiedź, co kończy zabawę.

Interakcja

Twój program musi używać biblioteki, aby komunikować się z wężem. Biblioteka dostarcza następujących funkcjonalności:

- `function get_speed() : longint` / `int get_speed()` — *zwraca prędkość węża* K*, czyli odległość, o którą wąż może przesunąć się przed każdą rundą pytań.*

- `procedure ask_snake(U1, U2 : longint, var A1 : char, var A2 : char)` / `void ask_snake(int U1, int U2, char *A1, char *A2)` — *wywołując tę funkcję, zadajesz wężowi dwa pytania w jednej rundzie;* `U1` *to jednostka, o którą pytasz w pierwszym pytaniu, a* `U2` *— jednostka, o którą pytasz w drugim pytaniu.* `A1` *stanowi odpowiedź na pierwsze pytanie i jest równe:*
  - *'*`s`*', jeżeli wąż leży na jednostce* `U1`*;*
  - *'*`f`*' (od* ***front****), jeżeli wąż nie dotarł jeszcze do jednostki* `U1`*.*
  - *'*`b`*' (od* ***behind****), jeżeli wąż przesunął się już poza jednostkę* `U1`*.*

  Wartość `A2` *zawiera analogiczną odpowiedź na drugie pytanie.*
- `procedure tell_length(L : longint)` / `void tell_length(int L)` — *wywołując tę funkcję, przekazujesz wężowi swoje oszacowanie jego długości* `L`*.*

Twój program musi wywoływać funkcje biblioteczne w następującej kolejności:

1. *Najpierw musisz użyć* `get_speed`*. Ta funkcja może zostać wywołana tylko raz.*
2. *Następnie używasz* `ask_snake` *tyle razy, ile potrzebujesz do zebrania informacji (być może 0 razy).*
3. *Na koniec wywołujesz funkcję* `tell_length`*, aby przekazać oszacowanie długości węża. To wywołanie kończy Twój program.*

Twój program nie może czytać z plików ani pisać do nich i nie może wykorzystywać standardowego wejścia/wyjścia ani próbować odwoływać się do pamięci spoza programu.

Programowanie i instrukcje kompilacji

Jeżeli wysyłasz kod źródłowy w Pascalu, to musi on zawierać instrukcję:

```
uses snakelib;
```

Do kompilacji swojego programu użyj instrukcji:

```
ppc386 -O2 -dCONTEST -XS snake.pas
```

Jeżeli wysyłasz kod źródłowy w C/C++, to musi on zawierać wiersz:

```
#include "snakelib.h"
```

Do kompilacji swojego programu użyj jednej z instrukcji:

```
gcc -std=c99 -O2 -DCONTEST -static snakelib.c snake.c -o snake
g++ -std=c++98 -O2 -DCONTEST -static snakelib.c snake.cpp -o snake
```

System oceniający używa opcji kompilacji wymienionych na ***Overview sheet*** *— zwróć na nie uwagę.*

Przykładowa biblioteka i przykładowy program

Do eksperymentów ze swoim rozwiązaniem możesz wykorzystać przykładową bibliotekę, której źródło znajduje się w plikach `snakelib.pas`, `snakelib.c` *i* `snakelib.h` *w Twoim katalogu domowym. Możesz ją modyfikować, ale lepiej nie zmieniaj jej interfejsu.*

Dostarczona biblioteka działa następująco:

- *Po wywołaniu* `get_speed`, *biblioteka otwiera plik* `snake.in` *i wczytuje dane węża. Plik* `snake.in` *ma następujący format. W jedynym wierszu zawiera trzy oddzielone pojedynczymi odstępami liczby całkowite:* ***Len***, ***Tail*** *i* ***K*** *— oznaczają one odpowiednio rzeczywistą długość węża, najbliższą Ciebie jednostkę, na której leży wąż, i stałą określającą możliwą prędkość węża. Liczby te są sprawdzane pod kątem zgodności z ograniczeniami podanymi na końcu treści zadania.*
- *Kiedy zostaje wywołana funkcja* `ask_snake`, *biblioteka odpowiada zgodnie z powyższą specyfikacją i przy co drugiej rundzie pytań wąż przesuwa się.*
- *Biblioteka kończy program, kiedy zostaje wywołana funkcja* `tell_length`, *i sprawdza, czy podane przez Ciebie oszacowanie długości jest dostatecznie dobre.*
- *Biblioteka wypisuje błąd i kończy program, jeśli jakaś funkcja zostaje wywołana niezgodnie z podaną kolejnością.*
- *Biblioteka zapisuje informacje diagnostyczne w pliku* `snake.log`.

Otrzymałeś także dwa proste programy (`snake.pas` *i* `snake.cpp`*) oraz przykładowy plik wejściowy (*`snake.in`*), które prezentują sposób korzystania z biblioteki.*

Przykładowy przebieg

Wywołanie funkcji	**Zwracane wartości i wyjaśnienia**
`get_speed();`	*Zwraca* `1`. *Wąż może przesunąć się o 1 jednostkę tuż przed rundą pytań.*
Pascal `ask_snake(0, 11, A1, A2);` *C lub C++* `ask_snake(0, 11, &A1, &A2);`	`A1='b', A2='f'`. *Pytasz węża o jednostki 0 i 11, a on odpowiada, że przesunął się już poza jednostkę 0 i że nie dotarł jeszcze do jednostki 11.*
Pascal `ask_snake(2, 10, A1, A2);` *C lub C++* `ask_snake(2, 10, &A1, &A2);`	`A1='b', A2='f'`. *Pytasz węża o jednostki 2 i 10, a on odpowiada, że przesunął się już poza jednostkę 2 i że nie dotarł jeszcze do jednostki 10.*

Pascal `ask_snake(4, 9, A1, A2);` *C lub C++* `ask_snake(4, 9, &A1, &A2);`	`A1='b', A2='s'`. *Pytasz węża o jednostki 4 i 9, a on odpowiada, że przesunął się już poza jednostkę 4 i że leży na jednostce 9.*
Pascal `ask_snake(6, 10, A1, A2);` *C lub C++* `ask_snake(6, 10, &A1, &A2);`	`A1='s', A2='s'`. *Pytasz węża o jednostki 6 i 10, a on odpowiada, że leży na jednostkach 6 i 10.*
`tell_length(6);`	*Zgadujesz, że długość węża to 6 jednostek. Ponieważ faktyczna długość to 5, to Twoje oszacowanie jest wystarczająco dobre.*

Ograniczenia

Prędkość węża K *spełnia ograniczenia:* $0 \leqslant K \leqslant 10$. *W zasięgu Twojego słuchu jest* $12\,122$ *jednostek, ponumerowanych od* 0 *do* $12\,121$. *Wąż przemieszcza się jedynie do przodu i nigdy nie wychodzi poza zasięg słuchu. Jego długość* Len *spełnia ograniczenia:* $1 \leqslant Len \leqslant 12\,122$.

Punktacja

Uzyskasz wszystkie punkty za dany test, jeżeli Twoje oszacowanie będzie różnić się co najwyżej o K *jednostek od rzeczywistej długości węża, a Twój program wywoła funkcję* `ask_snake` *co najwyżej 13 razy. Jeżeli twój program wykona więcej niż 13 wywołań, ale mniej niż 42 wywołania i podasz właściwe oszacowanie, to wąż przyzna Ci za ten test jedynie połowę punktów. W przeciwnym przypadku nie otrzymasz żadnych punktów. Jeżeli Twój program podczas wykonywania testu zachowa się niezgodnie z podanymi zasadami, to uzyskasz za ten test 0 punktów.*

Tomasz Kulczyński, Jakub Radoszewski
Tłumaczenie

CEOI 2008, dzień drugi, 10.07.2008

Wybór zleceń i wynajem maszyn

Stolarz Sam otrzymał N zleceń. W trakcie ich przeglądania zdał sobie sprawę, że brakuje mu M maszyn niezbędnych do wykonania tych zleceń. Nie wszystkie zlecenia wymagają wszystkich brakujących maszyn, ale do każdego ze zleceń potrzeba co najmniej jednej z nich.

Aby ukończyć zamówienie, Sam musi albo kupić, albo wynająć każdą z maszyn, których to zamówienie wymaga. Ponieważ różne zlecenia wiążą się z różnym nakładem pracy na poszczególnych maszynach, koszt wynajmu maszyny może zależeć od tego, które ze zleceń ma być na niej wykonane. Natomiast koszt zakupu maszyny nie zależy od zleceń, do których ma być ona wykorzystana — raz zakupioną maszynę Sam może wykorzystywać do realizacji dowolnej liczby zleceń bez żadnych dodatkowych kosztów.

Jeśli Sam uzna któreś ze zleceń za zbyt kosztowne do wykonania, może je odrzucić. Zaoszczędzi w ten sposób koszty, ale nie będzie także miał zysku z tego zlecenia.

Pomóż Samowi zdecydować, które zlecenia odrzucić, które maszyny kupić, a które wynająć, aby zmaksymalizować całkowity zysk.

Przykład

$N = 2$, $M = 3$

Zlecenie	*Zysk Sama za wykonanie zlecenia*
O_1	*100*
O_2	*100*

Maszyna	*Cena zakupu*
M_1	*50*
M_2	*80*
M_3	*110*

Zlecenie	*Maszyna wymagana do ukończenia zlecenia*	*Cena najmu maszyny do tego zlecenia*
O_1	M_1	*30*
	M_2	*20*
O_2	M_1	*40*
	M_3	*80*

Istnieją dwa rozwiązania o maksymalnym zysku, równym 50:

- *Odrzucenie O_2, wykonanie O_1, wynajęcie obu maszyn M_1 i M_2.*
- *Wykonanie obu zleceń O_1 i O_2, zakup M_1, wynajęcie M_2 i M_3.*

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby całkowite N ($1 \leqslant N \leqslant 1\,200$) i M ($1 \leqslant M \leqslant 1\,200$).

Kolejnych N bloków zawiera opisy zleceń. Każdy blok ma następującą strukturę: pierwszy wiersz bloku o numerze i zawiera dwie liczby całkowite, wartość zysku v_i $(1 \leqslant v_i \leqslant 5\,000)$ *ze zlecenia* O_i *oraz liczbę maszyn* m_i $(1 \leqslant m_i \leqslant M)$ *potrzebnych do ukończenia* O_i*. Każdy z kolejnych* m_i *wierszy zawiera numer maszyny* j $(1 \leqslant j \leqslant M)$ *potrzebnej do ukończenia* O_i *oraz koszt wynajmu* r_{ij} $(1 \leqslant r_{ij} \leqslant 20\,000)$ *tej maszyny do tego zlecenia.*

Każdy z M wierszy następujących po bloku opisującym ostatnie ze zleceń zawiera jedną liczbę całkowitą: cenę zakupu s_i $(1 \leqslant s_i \leqslant 20\,000)$ *maszyny o numerze i.*

Wyjście

Wyjście powinno zawierać dokładnie jedną liczbę całkowitą: maksymalny możliwy do uzyskania przez Sama zysk.

Przykład

Dla danych wejściowych:

```
2 3
100 2
1 30
2 20
100 2
1 40
3 80
50
80
110
```

poprawnym wynikiem jest:

```
50
```

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] *XIV Olimpiada Informatyczna 2006/2007*. Warszawa, 2007.

[15] A. V. Aho, J. E. Hopcroft, J. D. Ullman. *Projektowanie i analiza algorytmów komputerowych*. PWN, Warszawa, 1983.

[16] L. Banachowski, A. Kreczmar. *Elementy analizy algorytmów*. WNT, Warszawa, 1982.

[17] L. Banachowski, K. Diks, W. Rytter. *Algorytmy i struktury danych*. WNT, Warszawa, 1996.

[18] J. Bentley. *Perełki oprogramowania*. WNT, Warszawa, 1992.

[19] I. N. Bronsztejn, K. A. Siemiendiajew. *Matematyka. Poradnik encyklopedyczny*. PWN, Warszawa, wydanie XIV, 1997.

[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest. *Wprowadzenie do algorytmów*. WNT, Warszawa, 1997.

[21] D. Harel. *Algorytmika. Rzecz o istocie informatyki*. WNT, Warszawa, 1992.

[22] J. E. Hopcroft, J. D. Ullman. *Wprowadzenie do teorii automatów, języków i obliczeń*. PWN, Warszawa, 1994.

[23] W. Lipski. *Kombinatoryka dla programistów*. WNT, Warszawa, 2004.

[24] E. M. Reingold, J. Nievergelt, N. Deo. *Algorytmy kombinatoryczne*. WNT, Warszawa, 1985.

[25] R. Sedgewick. *Algorytmy w C++. Grafy*. RM, 2003.

[26] M. M. Sysło. *Algorytmy*. WSiP, Warszawa, 1998.

[27] M. M. Sysło. *Piramidy, szyszki i inne konstrukcje algorytmiczne*. WSiP, Warszawa, 1998.

[28] M. M. Sysło, N. Deo, J. S. Kowalik. *Algorytmy optymalizacji dyskretnej z programami w języku Pascal*. PWN, Warszawa, 1993.

[29] R. J. Wilson. *Wprowadzenie do teorii grafów*. PWN, Warszawa, 1998.

[30] N. Wirth. *Algorytmy + struktury danych = programy*. WNT, Warszawa, 1999.

[31] Ronald L. Graham, Donald E. Knuth, Oren Patashnik. *Matematyka konkretna*. PWN, Warszawa, 1996.

[32] K. A. Ross, C. R. B. Wright. *Matematyka dyskretna*. PWN, Warszawa, 1996.

[33] Donald E. Knuth. *Sztuka programowania*. WNT, Warszawa, 2002.

[34] Steven S. Skiena, Miguel A. Revilla. *Wyzwania programistyczne*. WSiP, Warszawa, 2004.

[35] Bernard Chazelle, Leo J. Guibas, D. T. Lee. The power of geometric duality. *BIT*, 25(1):76–90, 1985.

Niniejsza publikacja stanowi wyczerpujące źródło informacji o zawodach XV Olimpiady Informatycznej, przeprowadzonych w roku szkolnym 2007/2008. 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 XX Międzynarodowej Olimpiady Informatycznej, XIV Bałtyckiej Olimpiady Informatycznej oraz XV Olimpiady Informatycznej Krajów Europy Środkowej.

XV 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

ISBN 978–83–922946–4–1