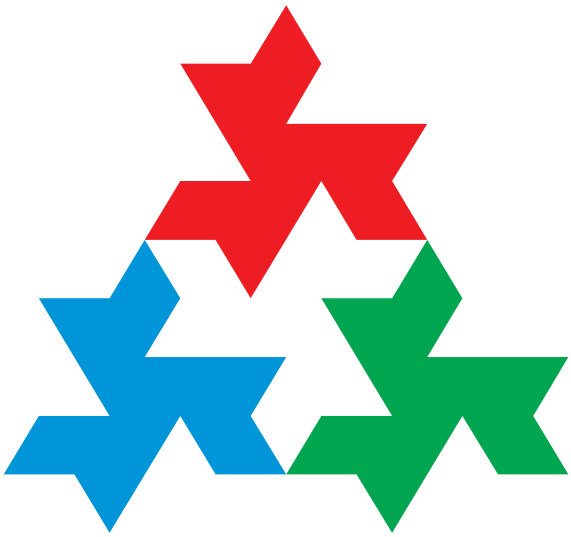
MINISTERSTWO EDUKACJI NARODOWEJ
FUNDACJA ROZWOJU INFORMATYKI
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ



XX OLIMPIADA INFORMATYCZNA
2012/2013

Olimpiada Informatyczna jest organizowana przy współudziale

WARSZAWA, 2013

Autorzy tekstów:

Krzysztof Diks
Bartłomiej Dudek
Tomasz Idziaszek
Wiktor Kuropatwa
Jan Kanty Milczek
Jakub Pachocki
Marcin Pilipczuk
Karol Pokorski
Adam Polak
Jakub Radoszewski
Wojciech Rytter
Bartosz Tarnawski
Jacek Tomasiewicz
Tomasz Waleń
Jakub Wojtaszczyk

Autorzy programów:

Michał Adamczyk
Igor Adamski
Marcin Andrychowicz
Mateusz Baranowski
Maciej Borsz
Dawid Dąbrowski
Lech Duraj
Przemysław Horban
Krzysztof Kiljan
Wiktor Kuropatwa
Maciej Matraszek
Karol Pokorski
Adam Polak
Bartosz Tarnawski

Opracowanie i redakcja:

Bartłomiej Gajewski
Tomasz Idziaszek
Marcin Kubica
Jakub Radoszewski

Tłumaczenie treści zadań:

Dawid Dąbrowski
Jakub Łącki
Jakub Radoszewski

Skład:

Bartłomiej Gajewski

Pozycja dotowana przez Ministerstwo Edukacji Narodowej.

Druk książki został sfinansowany przez

ISBN 978-83-64292-00-2

Spis treści

Sprawozdanie z przebiegu XX Olimpiady Informatycznej *5*

Regulamin Olimpiady Informatycznej *35*

Zasady organizacji zawodów *43*

Zasady organizacji zawodów II i III stopnia *49*

Zawody I stopnia – opracowania zadań **53**

Cennik *55*

Gobeliny *61*

Multidrink *69*

Taksówki *77*

Usuwanka *81*

Zawody II stopnia – opracowania zadań **85**

Spacer *87*

Inspektor *91*

Łuk triumfalny *99*

Konewka *105*

Morskie opowieści *113*

Zawody III stopnia – opracowania zadań **119**

Gra Tower Defense *121*

Bajtokomputer *129*

Labirynt *135*

Łańcuch kolorowy *141*

Gdzie jest jedynka? *145*

Laser *151*

Polaryzacja *157*

XXV Międzynarodowa Olimpiada Informatyczna – treści zadań **165**

Czas snu *167*

Historia sztuki *169*

Wombaty *171*

Jaskinia *174*

Robociki *177*

Gra *179*

XIX Bałtycka Olimpiada Informatyczna – treści zadań **183**

Kulki *185*

Liczby antypalindromiczne *187*

Rury *188*

Urodziny Bajtyny *190*

Ślady *192*

Vim *194*

Literatura **197**

Krzysztof Diks

Wstęp

Drodzy Czytelnicy,

już po raz dwudziesty oddajemy Wam do rąk niebieską książeczkę zawierającą opis przebiegu kolejnej edycji Olimpiady Informatycznej i wzorcowe rozwiązania zadań, z którymi zmierzyli się uczestnicy tych zawodów. Dwadzieścia lat w informatyce to szmat czasu. Olimpiada Informatyczna została powołana w grudniu 1993 roku, a pierwsza edycja Olimpiady miała miejsce w pierwszym półroczu 1994 roku. W zawodach pierwszego stopnia wystartowało wówczas 528 uczniów, którzy mieli do rozwiązania 3 zadania. Do zawodów II stopnia zakwalifikowało się 64 uczniów, którzy na drodze do finału musieli zmierzyć się z 2 zadaniami. W finale wystąpiło 33 zawodników, a do rozwiązania były 3 zadania. Zwycięzcą I Olimpiady Informatycznej został Michał Wala z I LO im. J. Kasprowicza w Raciborzu. Zwycięzca w nagrodę otrzymał komputer 486SX ufundowany przez IBM Polska. Nikt wtedy jeszcze nie przypuszczał, ale wszyscy o tym marzyli, że Olimpiada Informatyczna stanie się jedną z najlepszych olimpiad przedmiotowych w Polsce, o uczestników której zabiegać będą najlepsze uczelnie informatyczne w kraju, a potem czołowe firmy IT z całego świata. Marzyliśmy też, żeby reprezentanci Polski z sukcesami rywalizowali z rówieśnikami z innych krajów. Tak też się stało. W ciągu dwudziestu lat w zawodach Olimpiady wystartowało ponad 11 000 uczniów. Reprezentanci Polski w zawodach Międzynarodowej Olimpiady Informatycznej zdobyli dotychczas 89 medali, ustępując w liczbie zdobytych medali tylko Chińczykom. Wśród 89 medali są 32 medale złote, 32 medale srebrne i 25 medali brązowych. Biorąc pod uwagę kolory medali, Polska jest (nieoficjalnie) klasyfikowana na czwartym miejscu w świecie po Chinach, Rosji i Stanach Zjednoczonych. Tak wielkie sukcesy Olimpiady Informatycznej nie byłyby możliwe bez jej twórców, członków pierwszego Komitetu Głównego Olimpiady Informatycznej, którzy ukształtowali ramy organizacyjne zawodów oraz wytyczyli kierunki rozwoju, z których czerpiemy do dzisiaj. W tym miejscu warto przedstawić raz jeszcze skład tego Komitetu:

prof. dr hab. Jacek Błażewicz (Politechnika Poznańska)
prof. dr hab. Jan Madey (Uniwersytet Warszawski)
prof. dr hab. Andrzej W. Mostowski (Uniwersytet Gdański)
prof. dr hab. Wojciech Rytter (Uniwersytet Warszawski)
prof. dr hab. Maciej M. Sysło (Uniwersytet Wrocławski)
prof. dr hab. inż. Stanisław Waligórski (Uniwersytet Warszawski)
dr Piotr Chrząstowski-Wachtel (Uniwersytet Warszawski)
dr Andrzej Walat (OEIiZK w Warszawie)
dr Bolesław Wojdyło (Uniwersytet Mikołaja Kopernika w Toruniu)
mgr Jerzy Dałek (Ministerstwo Edukacji Narodowej)
mgr Krzysztof J. Święcicki (Ministerstwo Edukacji Narodowej)
Tadeusz Kuran (OEIiZK w Warszawie)
mgr Krystyna Kominek (II LO im. St. Batorego w Warszawie)

Wiele z tych osób do dzisiaj dba o najwyższą jakość i pomyślność Olimpiady. Dziękując twórcom Olimpiady i podkreślając jej sukcesy, należy sobie życzyć, żeby jej kolejne lata były jeszcze lepsze, a w zawodach brało udział coraz więcej uczniów, którzy realizowaliby w praktyce hasło Olimpiady Informatycznej: „Informatyka specjalnością młodych Polaków".

XX Olimpiada Informatyczna także zaznaczyła się w historii. Na XXV Międzynarodowej Olimpiadzie Informatycznej nasi reprezentanci zdobyli dwa medale srebrne i dwa brązowe. W XX Olimpiadzie Krajów Europy Środkowej zdobyliśmy dwa medale srebrne. Polscy reprezentanci całkowicie zdominowali zawody XIX Bałtyckiej Olimpiady Informatycznej, zdobywając cztery medale złote i dwa medale srebrne. Sukcesem organizacyjnym jest wybór Jakuba Łąckiego na członka Komitetu Naukowego Międzynarodowej Olimpiady Informatycznej.

Sukcesy Olimpiady Informatycznej nie byłyby możliwe bez jej wypróbowanych przyjaciół: partnerów organizacyjnych, sponsorów, nauczycieli i przede wszystkim wszystkich zaangażowanych w jej bezpośrednią organizację. Dziękuję,

Krzysztof Diks
Przewodniczący Komitetu Głównego Olimpiady Informatycznej

Sprawozdanie z przebiegu XX Olimpiady Informatycznej w roku szkolnym 2012/2013

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). Organizatorem XX Olimpiady Informatycznej jest Fundacja Rozwoju Informatyki.

ORGANIZACJA ZAWODÓW

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.

9 października 2012 roku rozesłano do 3296 szkół i zespołów szkół młodzieżowych ponadgimnazjalnych pismo oraz plakaty informujące o rozpoczęciu XX Olimpiady. Zawody I stopnia rozpoczęły się 15 października 2012 roku. Ostatecznym terminem nadsyłania prac konkursowych był 12 listopada 2012 roku.

Zawody II i III stopnia były dwudniowymi sesjami stacjonarnymi, poprzedzonymi jednodniowymi sesjami próbnymi. Zawody II stopnia odbyły się w siedmiu okręgach: Białymstoku, Krakowie, Poznaniu, Sopocie, Toruniu, Warszawie i Wrocławiu w dniach 12–14 lutego 2013 roku, natomiast zawody III stopnia odbyły się w Warszawie na Wydziale Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego, w dniach 12–14 marca 2013 roku.

Uroczystość zakończenia XX Olimpiady Informatycznej, wraz z obchodami XX–lecia, odbyła się 15 marca 2013 roku na Zamku Królewskim w Warszawie.

SKŁAD OSOBOWY KOMITETÓW OLIMPIADY

Komitet Główny

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

zastępcy przewodniczącego:
dr Przemysława Kanarek (Uniwersytet Wrocławski)
prof. dr hab. Paweł Idziak (Uniwersytet Jagielloński)

sekretarz naukowy:
dr Marcin Kubica (Uniwersytet Warszawski)

kierownik Jury:
dr Jakub Radoszewski (Uniwersytet Warszawski)

kierownik techniczny:
mgr Szymon Acedański (Uniwersytet Warszawski)

kierownik organizacyjny:
Tadeusz Kuran

członkowie:
dr Piotr Chrząstowski-Wachtel (Uniwersytet Warszawski)
prof. dr hab. inż. Zbigniew Czech (Politechnika Śląska)
dr hab. inż. Piotr Formanowicz, prof. PP (Politechnika Poznańska)
mgr Anna Beata Kwiatkowska (Gimnazjum i Liceum Akademickie w Toruniu)
prof. dr hab. Krzysztof Loryś (Uniwersytet Wrocławski)
prof. dr hab. Jan Madey (Uniwersytet Warszawski)
prof. dr hab. Wojciech Rytter (Uniwersytet Warszawski)
dr hab. Krzysztof Stencel, prof. UW (Uniwersytet Warszawski)
prof. dr hab. Maciej Sysło (Uniwersytet Wrocławski)
dr Maciej Ślusarek (Uniwersytet Jagielloński)
mgr Krzysztof J. Święcicki (współtwórca OI)
dr Tomasz Waleń (Uniwersytet Warszawski)
dr hab. inż. Stanisław Waligórski, prof. UW (Uniwersytet Warszawski)

sekretarz Komitetu Głównego:
mgr Monika Kozłowska-Zając (OEIiZK w Warszawie)

Komitet Główny ma siedzibę w Ośrodku Edukacji Informatycznej i Zastosowań Komputerów w Warszawie przy ul. Nowogrodzkiej 73.

Komitet Główny odbył 4 posiedzenia.

Komitety okręgowe

Komitet Okręgowy w Warszawie
przewodniczący:
dr Jakub Pawlewicz (Uniwersytet Warszawski)
zastępca przewodniczącego:
dr Adam Malinowski (Uniwersytet Warszawski)
sekretarz:
mgr Monika Kozłowska-Zając (OEIiZK w Warszawie)
członkowie:
mgr Szymon Acedański (Uniwersytet Warszawski)
prof. dr hab. Krzysztof Diks (Uniwersytet Warszawski)
dr Jakub Radoszewski (Uniwersytet Warszawski)

Siedzibą Komitetu Okręgowego jest Ośrodek Edukacji Informatycznej i Zastosowań Komputerów w Warszawie, ul. Nowogrodzka 73, Warszawa.

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 hab. Tomasz Jurdziński (Uniwersytet Wrocławski)
dr Przemysława Kanarek (Uniwersytet Wrocławski)
dr Witold Karczewski (Uniwersytet Wrocławski)

Siedzibą Komitetu Okręgowego jest Instytut Informatyki Uniwersytetu Wrocławskiego, ul. Joliot-Curie 15, Wrocław.

Komitet Okręgowy w Toruniu
przewodniczący:
prof. dr hab. Edward Ochmański (Uniwersytet Mikołaja Kopernika w Toruniu)
zastępca przewodniczącego:
dr Bartosz Ziemkiewicz (Uniwersytet Mikołaja Kopernika w Toruniu)
sekretarz:
dr Kamila Barylska (Uniwersytet Mikołaja Kopernika w Toruniu)
członkowie:
dr Łukasz Mikulski (Uniwersytet Mikołaja Kopernika w Toruniu)
dr Marcin Piątkowski (Uniwersytet Mikołaja Kopernika w Toruniu)

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

Górnośląski Komitet Okręgowy
przewodniczący:
prof. dr hab. inż. Zbigniew Czech (Politechnika Śląska w Gliwicach)
zastępca przewodniczącego:
dr inż. Krzysztof Simiński (Politechnika Śląska w Gliwicach)
członkowie:
dr inż. Jacek Widuch (Politechnika Śląska w Gliwicach)
mgr inż. Tomasz Drosik (Politechnika Śląska w Gliwicach)
mgr inż. Dariusz Myszor (Politechnika Śląska w Gliwicach)

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

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

członkowie:
dr Iwona Cieślik (Uniwersytet Jagielloński)
mgr Grzegorz Gutowski (Uniwersytet Jagielloński)
Marek Wróbel (student Uniwersytetu Jagiellońskiego)

Siedzibą Komitetu Okręgowego jest Katedra Algorytmiki Uniwersytetu Jagiellońskiego, ul. Łojasiewicza 6, Kraków.
(Strona internetowa Komitetu Okręgowego: `www.tcs.uj.edu.pl/OI/`.)

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 inż. Piotr Błajdo (Wyższa Szkoła Informatyki i Zarządzania w Rzeszowie)
dr inż. Maksymilian Knap (Wyższa Szkoła Informatyki i Zarządzania w Rzeszowie)
mgr Czesław Wal (Wyższa Szkoła Informatyki i Zarządzania w Rzeszowie)
mgr inż. Dominik Wojtaszek (Wyższa Szkoła Informatyki i Zarządzania w Rzeszowie)

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

Komitet Okręgowy w Poznaniu
przewodniczący:
dr inż. Szymon Wąsik (Politechnika Poznańska)
zastępca przewodniczącego:
mgr inż. Hanna Ćwiek (Politechnika Poznańska)
sekretarz:
mgr inż. Bartosz Zgrzeba (Politechnika Poznańska)
członkowie:
dr inż. Maciej Miłostan (Politechnika Poznańska)
mgr inż. Andrzej Stroiński (Politechnika Poznańska)

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

Pomorski Komitet Okręgowy
przewodniczący:
prof. dr hab. inż. Marek Kubale (Politechnika Gdańska)
zastępca przewodniczącego:
dr hab. Andrzej Szepietowski (Uniwersytet Gdański)

sekretarz:
dr inż. Krzysztof Ocetkiewicz (Politechnika Gdańska)
członkowie:
dr inż. Dariusz Dereniowski (Politechnika Gdańska)
dr inż. Adrian Kosowski (Politechnika Gdańska)
dr inż. Michał Małafiejski (Politechnika Gdańska)
mgr inż. Ryszard Szubartowski (III Liceum Ogólnokształcące im. Marynarki Wojennej RP w Gdyni)
dr Paweł Żyliński (Uniwersytet Gdański)

Siedzibą Komitetu Okręgowego jest Politechnika Gdańska, Wydział Elektroniki, Telekomunikacji i Informatyki, ul. Gabriela Narutowicza 11/12, Gdańsk Wrzeszcz.

Jury Olimpiady Informatycznej

W pracach Jury, które nadzorował Krzysztof Diks, a którymi kierowali Szymon Acedański i Jakub Radoszewski, brali udział pracownicy, doktoranci i studenci Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego, Katedry Algorytmiki, Wydziału Matematyki i Informatyki Uniwersytetu Jagiellońskiego i Wydziału Matematyki i Informatyki Uniwersytetu Wrocławskiego:

Michał Adamczyk
Igor Adamski
Marcin Andrychowicz
Mateusz Baranowski
Maciej Borsz
Dawid Dąbrowski
Maciej Dębski
Bartłomiej Dudek
dr Lech Duraj
Przemysław Horban
Adam Karczmarz
Krzysztof Kiljan
Wiktor Kuropatwa
Krzysztof Leszczyński
Maciej Matraszek
Mirosław Michalski
Jan Kanty Milczek
Jakub Pachocki
Karol Pokorski
Adam Polak
Bartosz Tarnawski

ZAWODY I STOPNIA

Zawody I stopnia XX Olimpiady Informatycznej odbyły się w dniach 15 października – 12 listopada 2012 roku. Wzięło w nich udział 975 zawodników. Decyzją Komitetu Głównego zdyskwalifikowano 28 zawodników. Powodem dyskwalifikacji była niesamodzielność rozwiązań zadań konkursowych. Sklasyfikowano 947 zawodników.

Decyzją Komitetu Głównego do zawodów zostało dopuszczonych 38 uczniów gimnazjów. Pochodzili oni z następujących szkół:

szkoła	miejscowość	liczba uczniów
Gimnazjum nr 16 w Zespole Szkół Ogólnokształcących nr 7	Szczecin	6
Gimnazjum nr 24 z Oddziałami Dwujęzycznymi	Gdynia	5
Gimnazjum nr 50 w Zespole Szkół Ogólnokształcących nr 6	Bydgoszcz	4
Gimnazjum z Oddziałami Dwujęzycznymi nr 42	Warszawa	4
Gimnazjum nr 58 z Oddziałami Dwujęzycznymi w Zespole Szkół nr 15	Warszawa	3
Gimnazjum nr 13 im. Stanisława Staszica w Zespole Szkół nr 82	Warszawa	3
Gimnazjum nr 49 z Oddziałami Dwujęzycznymi w Zespole Szkół nr 14	Wrocław	3
Społeczne Gimnazjum nr 2 STO	Białystok	1
Gimnazjum w Dąbiu	Dąbie	1
Gimnazjum nr 24 w Zespole Szkół nr 7	Lublin	1
Gimnazjum nr 2 im. Jana Pawła II	Luboń	1
Gimnazjum w Ogrodzieńcu	Ogrodzieniec	1
Publiczne Gimnazjum nr 23 w Zespole Szkół Ogólnokształcących nr 6	Radom	1
Gimnazjum nr 5 w Zespole Szkół nr 5	Ropczyce	1
Gimnazjum nr 1 im. Jana Pawła II	Sejny	1
Gimnazjum nr 33 z Oddziałami Dwujęzycznymi im. Stefana Batorego w Zespole Szkół nr 66	Warszawa	1
Gimnazjum nr 77 w Zespole Szkół nr 51 im. Ignacego Domeyki	Warszawa	1

Kolejność województw pod względem liczby zawodników była następująca:

mazowieckie	194 zawodników	zachodniopomorskie	39
małopolskie	120	podkarpackie	36
dolnośląskie	93	lubelskie	35
pomorskie	82	łódzkie	25
kujawsko-pomorskie	73	świętokrzyskie	17
śląskie	70	warmińsko-mazurskie	17
podlaskie	67	opolskie	12
wielkopolskie	65	lubuskie	2

W zawodach I stopnia najliczniej reprezentowane były szkoły:

szkoła	miejscowość	liczba uczniów
Zespół Szkół nr 82 (XIV Liceum Ogólnokształcące im. Stanisława Staszica oraz Gimnazjum nr 13 im. Stanisława Staszica)	Warszawa	79

V Liceum Ogólnokształcące im. Augusta Witkowskiego	Kraków	66
I Liceum Ogólnokształcące im. Adama Mickiewicza	Białystok	51
III Liceum Ogólnokształcące im. Marynarki Wojennej RP oraz Gimnazjum nr 24 z Oddziałami Dwujęzycznymi	Gdynia	48
I Liceum Ogólnokształcące im. Tadeusza Kościuszki	Legnica	38
Zespół Szkół Ogólnokształcących nr 7 (XIII Liceum Ogólnokształcące oraz Gimnazjum nr 16)	Szczecin	33
Zespół Szkół Ogólnokształcących nr 6 (VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich oraz Gimnazjum nr 50)	Bydgoszcz	31
Zespół Szkół nr 14 (Liceum Ogólnokształcące nr 14 im. Polonii Belgijskiej oraz Gimnazjum nr 49 z Oddziałami Dwujęzycznymi)	Wrocław	31
VIII Liceum Ogólnokształcące im. Adama Mickiewicza	Poznań	28
Zespół Szkół Ogólnokształcących nr 6 (VI Liceum Ogólnokształcące im. Jana Kochanowskiego oraz Publiczne Gimnazjum nr 23)	Radom	22
Zespół Szkół Uniwersytetu Mikołaja Kopernika (Liceum Akademickie)	Toruń	21
Liceum Ogólnokształcące nr III im. Adama Mickiewicza	Wrocław	17
I Liceum Ogólnokształcące im. Stanisława Staszica	Lublin	15
V Liceum Ogólnokształcące	Bielsko-Biała	13
Zespół Szkół Ogólnokształcących nr 2	Tarnów	13
Zespół Szkół nr 15 (VIII Liceum Ogólnokształcące im. Władysława IV i Gimnazjum nr 58 z Oddziałami Dwujęzycznymi im. Władysława IV)	Warszawa	13
VIII Liceum Ogólnokształcące z Oddziałami Dwujęzycznymi im. Marii Skłodowskiej-Curie	Katowice	11
IV Liceum Ogólnokształcące im. Marii Skłodowskiej-Curie	Olsztyn	10
XVIII Liceum Ogólnokształcące im. Jana Zamoyskiego	Warszawa	9
V Liceum Ogólnokształcące im. Stefana Żeromskiego	Gdańsk	8
III Liceum Ogólnokształcące im. św. Jana Kantego	Poznań	7
Publiczne Liceum Ogólnokształcące nr II z Oddziałami Dwujęzycznymi	Opole	6
I Liceum Ogólnokształcące im. Bolesława Chrobrego	Piotrków Trybunalski	6
Publiczne Liceum Ogólnokształcące Sióstr Prezentek	Rzeszów	6
I Liceum Ogólnokształcące im. Bolesława Prusa	Siedlce	6
Zespół Szkół Zawodowych	Brodnica	5

IX Liceum Ogólnokształcące im. Cypriana Kamila Norwida	Częstochowa	5
Zespół Szkół Informatycznych im. gen. Józefa Hauke Bosaka	Kielce	5
II Liceum Ogólnokształcące im. Króla Jana III Sobieskiego	Kraków	5
I Liceum Ogólnokształcące im. Mikołaja Kopernika	Krosno	5
Zespół Szkół Technicznych	Rybnik	5
III Liceum Ogólnokształcące im. Krzysztofa Kamila Baczyńskiego	Białystok	4
III Liceum Ogólnokształcące im. Stefana Żeromskiego	Bielsko-Biała	4
Zespół Szkół Elektronicznych im. Wojska Polskiego	Bydgoszcz	4
Zespół Szkół Techniczno-Informatycznych	Gliwice	4
VIII Liceum Ogólnokształcące im. Stanisława Wyspiańskiego	Kraków	4
Zespół Szkół Łączności im. Obrońców Poczty Polskiej w Gdańsku	Kraków	4
Zespół Szkół Ogólnokształcących (I Liceum Ogólnokształcące im. Tadeusza Kościuszki)	Łomża	4
Zespół Szkół Ogólnokształcących nr 4 (IV Liceum Ogólnokształcące im. Tadeusza Kościuszki)	Toruń	4
Gimnazjum z Oddziałami Dwujęzycznymi nr 42	Warszawa	4
XXVII Liceum Ogólnokształcące im. Tadeusza Czackiego	Warszawa	4
Zespół Szkół nr 51 im. Ignacego Domeyki (CXXII Liceum Ogólnokształcące oraz Gimnazjum nr 77)	Warszawa	4
Zespół Szkół Technicznych	Wodzisław Śląski	4
Zespół Szkół Politechnicznych im. Bohaterów Monte Cassino	Września	4
I Liceum Ogólnokształcące im. Juliusza Słowackiego	Chorzów	3
I Liceum Ogólnokształcące im. Edwarda Dembowskiego	Gliwice	3
III Liceum Ogólnokształcące im. Mikołaja Kopernika	Kalisz	3
Zespół Szkół Sióstr Nazaretanek im. św. Jadwigi Królowej	Kielce	3
Zespół Szkół im. Mikołaja Kopernika	Konin	3
II Liceum Ogólnokształcące im. Marii Skłodowskiej-Curie	Końskie	3
II Liceum Ogólnokształcące im. Hetmana Jana Zamoyskiego	Lublin	3
I Liceum Ogólnokształcące im. Mikołaja Kopernika	Łódź	3
II Liceum Ogólnokształcące im. Mikołaja Kopernika	Mielec	3
III Liceum Ogólnokształcące im. Mikołaja Kopernika	Olsztyn	3

I Liceum Ogólnokształcące im. ks. Adama Jerzego Czartoryskiego	Puławy	3
Zespół Szkół Elektryczno-Elektronicznych im. prof. Janusza Groszkowskiego	Radomsko	3
Zespół Szkół Elektronicznych	Rzeszów	3
I Liceum Ogólnokształcące im. Bolesława Krzywoustego	Słupsk	3
Zespół Szkół nr 3 im. Jana III Sobieskiego	Szczytno	3
IV Liceum Ogólnokształcące im. Jana Pawła II	Tarnów	3
XXVIII Liceum Ogólnokształcące im. Jana Kochanowskiego	Warszawa	3
I Liceum Ogólnokształcące im. Stefana Żeromskiego	Zawiercie	3

Najliczniej reprezentowane były następujące miejscowości:

Warszawa	138 zawodników	Nowy Sącz	6
Kraków	87	Brodnica	5
Białystok	58	Krosno	5
Gdynia	51	Łódź	5
Wrocław	49	Słupsk	5
Poznań	40	Konin	4
Legnica	39	Końskie	4
Bydgoszcz	37	Łomża	4
Szczecin	35	Mielec	4
Radom	29	Radomsko	4
Lublin	26	Włocławek	4
Toruń	26	Wodzisław Śląski	4
Tarnów	20	Września	4
Bielsko-Biała	18	Chorzów	3
Rzeszów	14	Kalisz	3
Gdańsk	13	Kartuzy	3
Olsztyn	13	Lębork	3
Katowice	11	Ostrów Mazowiecka	3
Kielce	10	Ostrów Wielkopolski	3
Gliwice	9	Piła	3
Piotrków Trybunalski	9	Przemyśl	3
Siedlce	9	Puławy	3
Częstochowa	8	Szczytno	3
Rybnik	8	Zawiercie	3
Opole	7		

Zawodnicy uczęszczali do następujących klas:

do klasy I gimnazjum	1 uczeń
do klasy II gimnazjum	9 uczniów
do klasy III gimnazjum	28

do klasy I szkoły ponadgimnazjalnej	170
do klasy II szkoły ponadgimnazjalnej	363
do klasy III szkoły ponadgimnazjalnej	344
do klasy IV szkoły ponadgimnazjalnej	32

W zawodach I stopnia zawodnicy mieli do rozwiązania pięć zadań:

- „Cennik" autorstwa Jakuba Radoszewskiego
- „Gobeliny" autorstwa Jakuba Wojtaszczyka
- „Multidrink" autorstwa Jakuba Radoszewskiego i Wojciecha Ryttera
- „Taksówki" autorstwa Krzysztofa Diksa i Wojciecha Ryttera
- „Usuwanka" autorstwa Jakuba Pachockiego i Wojciecha Ryttera

każde oceniane maksymalnie po 100 punktów.

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

- **CEN** – Cennik

	CEN	
	liczba zawodników	czyli
100 pkt.	60	6,34%
75–99 pkt.	4	0,42%
50–74 pkt.	13	1,37%
1–49 pkt.	261	27,56%
0 pkt.	192	20,28%
brak rozwiązania	417	44,03%

- **GOB** – Gobeliny

	GOB	
	liczba zawodników	czyli
100 pkt.	2	0,21%
75–99 pkt.	2	0,21%
50–74 pkt.	5	0,53%
1–49 pkt.	18	1,90%
0 pkt.	92	9,72%
brak rozwiązania	828	87,43%

- **MUL** – Multidrink

	MUL	
	liczba zawodników	czyli
100 pkt.	101	10,67%
75–99 pkt.	42	4,44%
50–74 pkt.	18	1,90%
1–49 pkt.	88	9,29%
0 pkt.	56	5,91%
brak rozwiązania	642	67,79%

- **TAK** – Taksówki

	TAK	
	liczba zawodników	czyli
100 pkt.	283	29,88%
75–99 pkt.	100	10,56%
50–74 pkt.	67	7,08%
1–49 pkt.	258	27,24%
0 pkt.	203	21,44%
brak rozwiązania	36	3,80%

- **USU** – Usuwanka

	USU	
	liczba zawodników	czyli
100 pkt.	262	27,67%
75–99 pkt.	17	1,80%
50–74 pkt.	30	3,17%
1–49 pkt.	181	19,11%
0 pkt.	149	15,73%
brak rozwiązania	308	32,52%

W sumie za wszystkie 5 zadań konkursowych:

SUMA	liczba zawodników	czyli
500 pkt.	1	0,11%
375–499 pkt.	37	3,91%
250–374 pkt.	108	11,40%
125–249 pkt.	238	25,13%
1–124 pkt.	389	41,08%
0 pkt.	174	18,37%

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

ZAWODY II STOPNIA

Do zawodów II stopnia, które odbyły się w dniach 12–14 lutego 2013 roku w sześciu stałych okręgach i Białymstoku, zakwalifikowano 384 zawodników, którzy osiągnęli w zawodach I stopnia wynik nie mniejszy niż 128 pkt.

Zawodnicy zostali przydzieleni do okręgów w następującej liczbie:

Białystok	50 zawodników	Toruń	31
Kraków	109	Warszawa	85
Poznań	17	Wrocław	40
Sopot	52		

Dwunastu zawodników nie stawiło się na zawody, w zawodach wzięło więc udział 372 zawodników.

Zawodnicy uczęszczali do szkół w następujących województwach:

mazowieckie	78 zawodników	zachodniopomorskie	13
małopolskie	59	podkarpackie	11
podlaskie	49	lubelskie	7
dolnośląskie	37	łódzkie	6
pomorskie	33	warmińsko-mazurskie	4
kujawsko-pomorskie	31	świętokrzyskie	3
śląskie	25	lubuskie	1 zawodnik
wielkopolskie	14	opolskie	1

W zawodach II stopnia najliczniej reprezentowane były szkoły:

szkoła	miejscowość	liczba uczniów
I Liceum Ogólnokształcące im. Adama Mickiewicza	Białystok	44
V Liceum Ogólnokształcące im. Augusta Witkowskiego	Kraków	43
Zespół Szkół nr 82 (XIV Liceum Ogólnokształcące im. Stanisława Staszica oraz Gimnazjum nr 13 im. Stanisława Staszica)	Warszawa	38
III Liceum Ogólnokształcące im. Marynarki Wojennej RP oraz Gimnazjum nr 24 z Oddziałami Dwujęzycznymi	Gdynia	26
Zespół Szkół nr 14 (Liceum Ogólnokształcące nr 14 im. Polonii Belgijskiej oraz Gimnazjum nr 49 z Oddziałami Dwujęzycznymi)	Wrocław	22
Zespół Szkół Ogólnokształcących nr 6 (VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich oraz Gimnazjum nr 50)	Bydgoszcz	16
Zespół Szkół Ogólnokształcących nr 6 (VI Liceum Ogólnokształcące im. Jana Kochanowskiego)	Radom	15
Zespół Szkół Ogólnokształcących nr 7 (XIII Liceum Ogólnokształcące oraz Gimnazjum nr 16)	Szczecin	13
Zespół Szkół Uniwersytetu Mikołaja Kopernika (Liceum Akademickie)	Toruń	12
VIII Liceum Ogólnokształcące im. Adama Mickiewicza	Poznań	10
Zespół Szkół nr 15 (VIII Liceum Ogólnokształcące im. Władysława IV i Gimnazjum nr 58 z Oddziałami Dwujęzycznymi im. Władysława IV)	Warszawa	9
V Liceum Ogólnokształcące	Bielsko-Biała	8
VIII Liceum Ogólnokształcące z Oddziałami Dwujęzycznymi im. Marii Skłodowskiej-Curie	Katowice	7
I Liceum Ogólnokształcące im. Tadeusza Kościuszki	Legnica	6
V Liceum Ogólnokształcące im. Stefana Żeromskiego	Gdańsk	5
I Liceum Ogólnokształcące im. Stanisława Staszica	Lublin	5
Liceum Ogólnokształcące nr III im. Adama Mickiewicza	Wrocław	5
II Liceum Ogólnokształcące im. Króla Jana III Sobieskiego	Kraków	3
Zespół Szkół Ogólnokształcących (I Liceum Ogólnokształcące im. Tadeusza Kościuszki)	Łomża	3
II Liceum Ogólnokształcące im. Mikołaja Kopernika	Mielec	3
IV Liceum Ogólnokształcące im. Marii Skłodowskiej-Curie	Olsztyn	3

Zespół Szkół Ogólnokształcących nr 2 (II Liceum Ogólnokształcące)	Tarnów	3
Zespół Szkół nr 51 im. Ignacego Domeyki (CXXII Liceum Ogólnokształcące oraz Gimnazjum nr 77)	Warszawa	3

Najliczniej reprezentowane były miejscowości:

Warszawa	56 zawodników	Katowice	7
Kraków	47	Legnica	7
Białystok	44	Tarnów	7
Wrocław	28	Gdańsk	6
Gdynia	26	Lublin	5
Bydgoszcz	18	Łomża	3
Radom	16	Łódź	3
Poznań	13	Mielec	3
Szczecin	13	Nowy Sącz	3
Toruń	13	Olsztyn	3
Bielsko-Biała	10	Rzeszów	3

12 lutego odbyła się sesja próbna, podczas której zawodnicy rozwiązywali nieliczące się do ogólnej klasyfikacji zadanie „Spacer" autorstwa Wojciecha Ryttera. W dniach konkursowych zawodnicy rozwiązywali następujące zadania:

- w pierwszym dniu zawodów (13 lutego):
  - „Inspektor" autorstwa Jakuba Wojtaszczyka
  - „Łuk triumfalny" autorstwa Jacka Tomasiewicza
- w drugim dniu zawodów (14 lutego):
  - „Konewka" autorstwa Bartłomieja Dudka
  - „Morskie opowieści" autorstwa Wiktora Kuropatwy

każde oceniane maksymalnie po 100 punktów.

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

- **SPA – próbne** – Spacer

	SPA – próbne	
	liczba zawodników	czyli
100 pkt.	4	1,08%
75–99 pkt.	3	0,81%
50–74 pkt.	7	1,88%
1–49 pkt.	145	38,98%
0 pkt.	157	42,20%
brak rozwiązania	56	15,05%

- **INS** – Inspektor

	INS	
	liczba zawodników	czyli
100 pkt.	7	1,88%
75–99 pkt.	1	0,27%
50–74 pkt.	0	0%
1–49 pkt.	16	4,30%
0 pkt.	266	71,51%
brak rozwiązania	82	22,04%

- **LUK** – Łuk triumfalny

	LUK	
	liczba zawodników	czyli
100 pkt.	44	11,83%
75–99 pkt.	17	4,57%
50–74 pkt.	5	1,35%
1–49 pkt.	115	30,91%
0 pkt.	168	45,16%
brak rozwiązania	23	6,18%

- **KON** – Konewka

	KON	
	liczba zawodników	czyli
100 pkt.	29	7,80%
75–99 pkt.	5	1,34%
50–74 pkt.	148	39,78%
1–49 pkt.	122	32,80%
0 pkt.	45	12,10%
brak rozwiązania	23	6,18%

- **MOR** – Morskie opowieści

	MOR	
	liczba zawodników	czyli
100 pkt.	51	13,71%
75–99 pkt.	13	3,50%
50–74 pkt.	52	13,98%
1–49 pkt.	47	12,63%
0 pkt.	161	43,28%
brak rozwiązania	48	12,90%

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

SUMA	liczba zawodników	czyli
400 pkt.	4	1,08%
300–399 pkt.	10	2,69%
200–299 pkt.	42	11,29%
100–199 pkt.	78	20,97%
1–99 pkt.	199	53,49%
0 pkt.	39	10,48%

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

ZAWODY III STOPNIA

Zawody III stopnia odbyły się na Wydziale Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego w dniach od 12 do 14 marca 2013 roku. Do zawodów III stopnia zakwalifikowano 96 najlepszych uczestników zawodów II stopnia, którzy uzyskali wynik nie mniejszy niż 140 pkt.

Zawodnicy uczęszczali do szkół w następujących województwach:

mazowieckie	26 zawodników	śląskie	4
kujawsko-pomorskie	11	zachodniopomorskie	3
małopolskie	11	łódzkie	2
podlaskie	11	warmińsko-mazurskie	2
dolnośląskie	10	lubelskie	1 zawodnik
pomorskie	10	wielkopolskie	1
podkarpackie	4		

Niżej wymienione szkoły miały w finale więcej niż jednego zawodnika:

szkoła	miejscowość	liczba uczniów
Zespół Szkół nr 82 (XIV Liceum Ogólnokształcące im. Stanisława Staszica)	Warszawa	17
V Liceum Ogólnokształcące im. Augusta Witkowskiego	Kraków	10
I Liceum Ogólnokształcące im. Adama Mickiewicza	Białystok	8
III Liceum Ogólnokształcące im. Marynarki Wojennej RP oraz Gimnazjum nr 24 z Oddziałami Dwujęzycznymi	Gdynia	8
Zespół Szkół Ogólnokształcących nr 6 (VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich)	Bydgoszcz	7
Zespół Szkół Ogólnokształcących nr 6 (VI Liceum Ogólnokształcące im. Jana Kochanowskiego)	Radom	6
Zespół Szkół nr 14 (Liceum Ogólnokształcące nr 14 im. Polonii Belgijskiej oraz Gimnazjum nr 49 z Oddziałami Dwujęzycznymi)	Wrocław	6
Zespół Szkół Uniwersytetu Mikołaja Kopernika (Liceum Akademickie)	Toruń	4
I Liceum Ogólnokształcące im. Tadeusza Kościuszki	Legnica	3
Zespół Szkół Ogólnokształcących nr 7 (XIII Liceum Ogólnokształcące)	Szczecin	3
II Liceum Ogólnokształcące im. Mikołaja Kopernika	Mielec	2
IV Liceum Ogólnokształcące im. Marii Skłodowskiej-Curie	Olsztyn	2

12 marca odbyła się sesja próbna, podczas której zawodnicy rozwiązywali nieliczące się do ogólnej klasyfikacji zadania „Gra Tower Defense” autorstwa Marcina Pilipczuka oraz „Izolator” autorstwa Zbigniewa Czecha. W dniach konkursowych zawodnicy rozwiązywali następujące zadania:

- w pierwszym dniu zawodów (13 marca):
  - „Bajtokomputer” autorstwa Jacka Tomasiewicza
  - „Labirynt” autorstwa Tomasza Idziaszka

  - „Łańcuch kolorowy” autorstwa Tomasza Walenia
- w drugim dniu zawodów (14 marca):
  - „Gdzie jest jedynka?” autorstwa Jakuba Pachockiego
  - „Laser” autorstwa Karola Pokorskiego
  - „Polaryzacja” autorstwa Jana Kantego Milczka

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:

- **GRA – próbne** – Gra Tower Defense

	GRA – próbne	
	liczba zawodników	czyli
100 pkt.	42	4,37%
75–99 pkt.	2	0,21%
50–74 pkt.	2	0,21%
1–49 pkt.	15	1,56%
0 pkt.	22	2,29%
brak rozwiązania	13	1,35%

- **IZO – próbne** – Izolator

	IZO – próbne	
	liczba zawodników	czyli
100 pkt.	95	98,96%
75–99 pkt.	0	0%
50–74 pkt.	0	0%
1–49 pkt.	0	0%
0 pkt.	0	0%
brak rozwiązania	1	1,04%

- **BAJ** – Bajtokomputer

	BAJ	
	liczba zawodników	czyli
100 pkt.	61	63,54%
75–99 pkt.	5	5,21%
50–74 pkt.	11	11,46%
1–49 pkt.	11	11,46%
0 pkt.	6	6,25%
brak rozwiązania	2	2,08%

- **LAB** – Labirynt

	LAB	
	liczba zawodników	czyli
100 pkt.	4	4,17%
75–99 pkt.	0	0%
50–74 pkt.	1	1,04%
1–49 pkt.	3	3,12%
0 pkt.	36	37,50%
brak rozwiązania	52	54,17%

- **LAN** – Łańcuch kolorowy

	LAN	
	liczba zawodników	czyli
100 pkt.	42	43,75%
75–99 pkt.	44	45,83%
50–74 pkt.	6	6,25%
1–49 pkt.	3	3,13%
0 pkt.	1	1,04%
brak rozwiązania	0	0%

- **GDZ** – Gdzie jest jedynka?

	GDZ	
	liczba zawodników	czyli
100 pkt.	15	15,63%
75–99 pkt.	8	8,33%
50–74 pkt.	8	8,33%
1–49 pkt.	42	43,75%
0 pkt.	2	2,08%
brak rozwiązania	21	21,88%

- **LAS** – Laser

	LAS	
	liczba zawodników	czyli
100 pkt.	13	13,54%
75–99 pkt.	6	6,25%
50–74 pkt.	6	6,25%
1–49 pkt.	12	12,50%
0 pkt.	19	19,79%
brak rozwiązania	40	41,67%

- **POL** – Polaryzacja

	POL	
	liczba zawodników	czyli
100 pkt.	2	2,08%
75–99 pkt.	0	0%
50–74 pkt.	4	4,17%
1–49 pkt.	17	17,71%
0 pkt.	39	40,62%
brak rozwiązania	34	35,42%

W sumie za wszystkie 6 zadań konkursowych rozkład wyników zawodników był następujący:

SUMA	liczba zawodników	czyli
600 pkt.	0	0%
450–599 pkt.	2	2,08%
300–449 pkt.	25	26,04%
150–299 pkt.	51	53,13%
1–149 pkt.	18	18,75%
0 pkt.	0	0%

15 marca 2013 roku, na Zamku Królewskim w Warszawie, ogłoszono wyniki finału XX Olimpiady Informatycznej 2012/2013 i rozdano nagrody ufundowane przez: Ministerstwo Edukacji Narodowej, Asseco Poland SA, Olimpiadę Informatyczną, Wydawnictwa Naukowe PWN i Wydawnictwo „Delta".

Poniżej zestawiono listę wszystkich laureatów i wyróżnionych finalistów:

(1) **Błażej Magnowski**, 3 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia, 491 pkt., laureat I miejsca

(2) **Stanisław Dobrowolski**, 3 klasa, Zespół Szkół nr 82 (XIV Liceum Ogólnokształcące im. Stanisława Staszica), Warszawa, 451 pkt., laureat I miejsca

(3) **Marek Sommer**, 3 klasa, Zespół Szkół nr 82 (XIV Liceum Ogólnokształcące im. Stanisława Staszica), Warszawa, 430 pkt., laureat I miejsca

(4) **Karol Farbiś**, 3 klasa, Zespół Szkół Ogólnokształcących nr 6 (VI Liceum Ogólnokształcące im. Jana Kochanowskiego), Radom, 421 pkt., laureat II miejsca

(4) **Tomasz Syposz**, 3 klasa, Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej, Wrocław, 421 pkt., laureat II miejsca

(6) **Paweł Nowak**, 3 klasa, Zespół Szkół Ogólnokształcących nr 7 (XIII Liceum Ogólnokształcące), Szczecin, 416 pkt., laureat II miejsca

(7) **Igor Kotrasiński**, 3 klasa, Zespół Szkół nr 82 (XIV Liceum Ogólnokształcące im. Stanisława Staszica), Warszawa, 400 pkt., laureat II miejsca

(8) **Krzysztof Pszeniczny**, 3 klasa, Gimnazjum i Liceum im. Jana Pawła II Sióstr Prezentek (Liceum), Rzeszów, 399 pkt., laureat II miejsca

(9) **Jarosław Kwiecień**, 3 klasa, Gimnazjum nr 49 z Oddziałami Dwujęzycznymi, Wrocław, 390 pkt., laureat III miejsca

(10) **Adam Czapliński**, 3 klasa, IV Liceum Ogólnokształcące im. Marii Skłodowskiej-Curie, Olsztyn, 380 pkt., laureat III miejsca

(10) **Kamil Dębowski**, 3 klasa, I Liceum Ogólnokształcące im. Marii Konopnickiej, Suwałki, 380 pkt., laureat III miejsca

(12) **Kamil Rychlewicz**, 2 klasa, I Liceum Ogólnokształcące im. Mikołaja Kopernika, Łódź, 373 pkt., laureat III miejsca

(13) **Jan Ludziejewski**, 1 klasa, Zespół Szkół nr 15 (VIII Liceum Ogólnokształcące im. Władysława IV), Warszawa, 368 pkt., laureat III miejsca

(14) **Michał Zieliński**, 2 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków, 357 pkt., laureat III miejsca

(15) **Karol Kaszuba**, 2 klasa, Zespół Szkół nr 82 (XIV Liceum Ogólnokształcące im. Stanisława Staszica), Warszawa, 354 pkt., laureat III miejsca

(16) **Przemysław Jakub Kozłowski**, 1 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza, Białystok, 351 pkt., laureat III miejsca

(17) **Ewelina Krakowiak**, 3 klasa, Zespół Szkół Ogólnokształcących nr 6 (VI Liceum Ogólnokształcące im. Jana Kochanowskiego), Radom, 346 pkt., laureatka III miejsca

(18) **Stanisław Purgał**, 3 klasa, Zespół Szkół nr 82 (XIV Liceum Ogólnokształcące im. Stanisława Staszica), Warszawa, 334 pkt., laureat III miejsca

(19) **Daniel Danielski**, 3 klasa, Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej, Wrocław, 315 pkt., laureat III miejsca

(20) **Grzegorz Świrski**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków, 313 pkt., laureat III miejsca

(21) **Rafał Stefański**, 3 klasa, Zespół Szkół nr 82 (XIV Liceum Ogólnokształcące im. Stanisława Staszica), Warszawa, 311 pkt., laureat III miejsca

(22) **Stanisław Barzowski**, 2 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia, 305 pkt., laureat III miejsca

(23) **Paweł Nałęcz-Jawecki**, 3 klasa, Zespół Szkół nr 82 (XIV Liceum Ogólnokształcące im. Stanisława Staszica), Warszawa, 304 pkt., laureat III miejsca

(24) **Michał Kowalczyk**, 3 klasa, Zespół Szkół Ogólnokształcących nr 6 (VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich), Bydgoszcz, 301 pkt., laureat III miejsca

(25) **Grzegorz Fabiański**, 3 klasa, Zespół Szkół nr 82 (XIV Liceum Ogólnokształcące im. Stanisława Staszica), Warszawa, 300 pkt., laureat III miejsca

(25) **Szymon Łukasz**, 2 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków, 300 pkt., laureat III miejsca

(25) **Konrad Paluszek**, 1 klasa, Zespół Szkół nr 82 (XIV Liceum Ogólnokształcące im. Stanisława Staszica), Warszawa, 300 pkt., laureat III miejsca

(28) **Bartosz Kostka**, 3 klasa, Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej, Wrocław, 286 pkt., laureat III miejsca

(29) **Patryk Czajka**, 3 klasa, Zespół Szkół nr 82 (XIV Liceum Ogólnokształcące im. Stanisława Staszica), Warszawa, 280 pkt., laureat III miejsca

(29) **Michał Glapa**, 2 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków, 280 pkt., laureat III miejsca

(29) **Michał Łuszczyk**, 2 klasa, IV Liceum Ogólnokształcące im. Jana Pawła II w Zespole Szkół Ogólnokształcących nr 1, Tarnów, 280 pkt., laureat III miejsca

(32) **Jakub Staroń**, 2 klasa, V Liceum Ogólnokształcące, Bielsko-Biała, 276 pkt., finalista z wyróżnieniem

(33) **Michał Cylwik**, 3 klasa, Zespół Szkół Ogólnokształcących nr 7 (XIII Liceum Ogólnokształcące), Szczecin, 273 pkt., finalista z wyróżnieniem

(34) **Kamil Żyła**, 3 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia, 271 pkt., finalista z wyróżnieniem

(35) **Marek Sokołowski**, 2 klasa, Zespół Szkół Ogólnokształcących (I Liceum Ogólnokształcące im. Tadeusza Kościuszki), Łomża, 270 pkt., finalista z wyróżnieniem

(36) **Szymon Stankiewicz**, 3 klasa, Zespół Szkół Ogólnokształcących nr 6 (VI Liceum Ogólnokształcące im. Jana Kochanowskiego), Radom, 268 pkt., finalista z wyróżnieniem

(37) **Albert Gutowski**, 2 klasa, I Liceum Ogólnokształcące im. Stanisława Staszica, Lublin, 267 pkt., finalista z wyróżnieniem

(38) **Paweł Burzyński**, 2 klasa, Gimnazjum nr 24 z Oddziałami Dwujęzycznymi, Gdynia, 261 pkt., finalista z wyróżnieniem

(39) **Łukasz Łabęcki**, 3 klasa, Zespół Szkół Ogólnokształcących nr 6 (VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich), Bydgoszcz, 258 pkt., finalista z wyróżnieniem

(40) **Marcin Kostrzewa**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków, 254 pkt., finalista z wyróżnieniem

(41) **Aleksander Łukasiewicz**, 3 klasa, I Liceum Ogólnokształcące im. Tadeusza Kościuszki, Legnica, 240 pkt., finalista z wyróżnieniem

(42) **Aleksander Matusiak**, 3 klasa, Zespół Szkół nr 82 (XIV Liceum Ogólnokształcące im. Stanisława Staszica), Warszawa, 239 pkt., finalista z wyróżnieniem

(43) **Michał Kowalewski**, 2 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia, 235 pkt., finalista z wyróżnieniem

(44) **Adam Trzaskowski**, 2 klasa, Zespół Szkół nr 82 (XIV Liceum Ogólnokształcące im. Stanisława Staszica), Warszawa, 234 pkt., finalista z wyróżnieniem

(45) **Marcin Kudla**, 3 klasa, Zespół Szkół Ponadgimnazjalnych nr 1 (Liceum Ogólnokształcące im. Noblistów Polskich), Rydułtowy, 231 pkt., finalista z wyróżnieniem

(46) **Monika Olchowik**, 2 klasa, Zespół Szkół nr 51 im. Ignacego Domeyki (CXXII Liceum Ogólnokształcące), Warszawa, 230 pkt., finalistka z wyróżnieniem

(46) **Konrad Sikorski**, 3 klasa, Zespół Szkół Ogólnokształcących nr 6 (VI Liceum Ogólnokształcące im. Jana Kochanowskiego), Radom, 230 pkt., finalista z wyróżnieniem

(46) **Magdalena Szarkowska**, 2 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza, Białystok, 230 pkt., finalistka z wyróżnieniem

(49) **Sebastian Jaszczur**, 2 klasa, Zespół Szkół Ogólnokształcących nr 6 (VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich), Bydgoszcz, 229 pkt., finalista z wyróżnieniem

(50) **Jan Derbisz**, 2 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków, 228 pkt., finalista z wyróżnieniem

(51) **Kamil Niziński**, 2 klasa, Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej, Wrocław, 225 pkt., finalista z wyróżnieniem

(52) **Leszek Kania**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków, 221 pkt., finalista z wyróżnieniem

(52) **Adrian Naruszko**, 1 klasa, Zespół Szkół Ogólnokształcących nr 6 (VI Liceum Ogólnokształcące im. Jana Kochanowskiego), Radom, 221 pkt., finalista z wyróżnieniem

(52) **Paweł Wegner**, 2 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia, 221 pkt., finalista z wyróżnieniem

(55) **Jakub Machaj**, 3 klasa, Zespół Szkół Uniwersytetu Mikołaja Kopernika (Liceum Akademickie), Toruń, 219 pkt., finalista z wyróżnieniem

(56) **Jakub Łabaj**, 2 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków, 217 pkt., finalista z wyróżnieniem

(57) **Piotr Pietrzak**, 3 klasa, Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej, Wrocław, 216 pkt., finalista z wyróżnieniem

(58) **Marcin Gregorczyk**, 2 klasa, IV Liceum Ogólnokształcące im. Marii Skłodowskiej-Curie, Olsztyn, 211 pkt., finalista z wyróżnieniem

(59) **Mateusz Puczel**, 2 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza, Białystok, 208 pkt., finalista z wyróżnieniem

(60) **Wojciech Janczewski**, 3 klasa, I Liceum Ogólnokształcące im. Tadeusza Kościuszki, Legnica, 201 pkt., finalista z wyróżnieniem

(60) **Bartosz Łukasiewicz**, 2 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia, 201 pkt., finalista z wyróżnieniem

(62) **Piotr Dulikowski**, 3 klasa, XXXI Liceum Ogólnokształcące im. Ludwika Zamenhoffa, Łódź, 200 pkt., finalista z wyróżnieniem

(62) **Piotr Gawryluk**, 3 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza, Białystok, 200 pkt., finalista z wyróżnieniem

(62) **Mateusz Nowak**, 2 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków, 200 pkt., finalista z wyróżnieniem

Lista pozostałych finalistów w kolejności alfabetycznej:

- **Adrian Akerman**, 1 klasa, Zespół Szkół Katolickich im. św. Jadwigi Królowej (Uniwersyteckie Katolickie Liceum Ogólnokształcące), Tczew
- **Grzegorz Araminowicz**, 3 klasa, I Liceum Ogólnokształcące im. Mikołaja Kopernika, Gdańsk
- **Kamil Braun**, 3 klasa, I Liceum Ogólnokształcące im. Mikołaja Kopernika, Krosno
- **Filip Chmielewski**, 3 klasa, Zespół Szkół Ogólnokształcących nr 7 (XIII Liceum Ogólnokształcące), Szczecin
- **Mateusz Chołołowicz**, 2 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza, Białystok
- **Jakub Cisło**, 2 klasa, II Liceum Ogólnokształcące im. Mikołaja Kopernika, Mielec
- **Piotr Domański**, 3 klasa, VIII Liceum Ogólnokształcące z Oddziałami Dwujęzycznymi im. Marii Skłodowskiej-Curie, Katowice
- **Bartosz Dziewoński**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków
- **Wojciech Jabłoński**, 2 klasa, Zespół Szkół nr 82 (XIV Liceum Ogólnokształcące im. Stanisława Staszica), Warszawa
- **Piotr Jarosz**, 3 klasa, I Liceum Ogólnokształcące im. Zygmunta Krasińskiego, Ciechanów
- **Jarosław Jedynak**, 3 klasa, III Liceum Ogólnokształcące im. Stefana Żeromskiego, Bielsko-Biała
- **Eryk Kijewski**, 3 klasa, Gimnazjum nr 1 im. Jana Pawła II, Sejny
- **Wojciech Kordalski**, 2 klasa, Zespół Szkół Ogólnokształcących nr 6 (VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich), Bydgoszcz
- **Dawid Kuczma**, 3 klasa, Zespół Szkół Uniwersytetu Mikołaja Kopernika (Liceum Akademickie), Toruń
- **Mateusz Ledzianowski**, 3 klasa, VIII Liceum Ogólnokształcące im. Adama Mickiewicza, Poznań
- **Michał Majewski**, 3 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza, Białystok
- **Tomasz Miotk**, 3 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia
- **Marek Mystkowski**, 2 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza, Białystok
- **Bartłomiej Najdecki**, 2 klasa, I Liceum Ogólnokształcące im. Tadeusza Kościuszki, Legnica
- **Kacper Oreszczuk**, 3 klasa, Zespół Szkół Ogólnokształcących nr 6 (VI Liceum Ogólnokształcące im. Jana Kochanowskiego), Radom
- **Patryk Osmólski**, 3 klasa, Zespół Szkół Ogólnokształcących nr 6 (VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich), Bydgoszcz

- **Piotr Michał Padlewski**, 3 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza, Białystok
- **Tomasz Rewak**, 3 klasa, II Liceum Ogólnokształcące im. Stanisława Wyspiańskiego, Legnica
- **Marek Rusinowski**, 3 klasa, II Liceum Ogólnokształcące im. Mikołaja Kopernika, Mielec
- **Karol Sarnecki**, 3 klasa, Zespół Szkół nr 82 (XIV Liceum Ogólnokształcące im. Stanisława Staszica), Warszawa
- **Jakub Stanecki**, 2 klasa, Zespół Szkół nr 82 (XIV Liceum Ogólnokształcące im. Stanisława Staszica), Warszawa
- **Jakub Supeł**, 2 klasa, Zespół Szkół nr 82 (XIV Liceum Ogólnokształcące im. Stanisława Staszica), Warszawa
- **Dariusz Szałkowski**, 3 klasa, Zespół Szkół Ogólnokształcących nr 6 (VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich), Bydgoszcz
- **Dawid Wegner**, 1 klasa, Zespół Szkół Ogólnokształcących nr 6 (VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich), Bydgoszcz
- **Arkadiusz Wróbel**, 3 klasa, Zespół Szkół nr 82 (XIV Liceum Ogólnokształcące im. Stanisława Staszica), Warszawa
- **Aleksander Zendel**, 2 klasa, Zespół Szkół Uniwersytetu Mikołaja Kopernika (Liceum Akademickie), Toruń
- **Wojciech Zieliński**, 3 klasa, Zespół Szkół Uniwersytetu Mikołaja Kopernika (Liceum Akademickie), Toruń

Komitet Główny Olimpiady Informatycznej przyznał następujące nagrody rzeczowe:

(1) puchar przechodni ufundowany przez Olimpiadę Informatyczną wręczono zwycięzcy XX Olimpiady Błażejowi Magnowskiemu,

(2) puchar ufundowany przez Olimpiadę Informatyczną wręczono zwycięzcy XX Olimpiady Błażejowi Magnowskiemu,

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

(4) laptopy (3 szt.) ufundowane przez Asseco Poland SA przyznano laureatom I miejsca,

(5) aparaty fotograficzne (5 szt.) ufundowane przez Asseco Poland SA przyznano laureatom II miejsca,

(6) dyski twarde zewnętrzne (23 szt.) ufundowane przez Asseco Poland SA przyznano laureatom III miejsca,

(7) 32-GB pamięci USB (33 szt.) ufundowane przez Ministerstwo Edukacji Narodowej przyznano wyróżnionym finalistom,

(8) książki ufundowane przez PWN przyznano wszystkim finalistom. Lista propozycji książkowych została wysłana e-mailem do finalistów. Po dokonaniu wyboru, książki zostały przesłane na adresy domowe finalistów.

(9) roczną prenumeratę miesięcznika „Delta” przyznano wszystkim laureatom.

Komitet Główny powołał reprezentacje na:

- **Międzynarodową Olimpiadę Informatyczną IOI'2013**, która odbędzie się w Australii w terminie 7–12 lipca 2013 roku, w składzie:

  (1) Błażej Magnowski
  (2) Stanisław Dobrowolski
  (3) Marek Sommer
  (4) laureat II miejsca, który zdobędzie najlepszy wynik na Bałtyckiej Olimpiadzie Informatycznej BOI'2013

  rezerwowi: dwaj następni laureaci II miejsca w kolejności wyników na BOI'2013.

  Dogrywka o czwarte miejsce w reprezentacji na IOI'2013 była spowodowana usterką w sprawdzaniu jednego z zadań finałowych.

- **Olimpiadę Informatyczną Krajów Europy Środkowej CEOI'2013**, która odbędzie się w Chorwacji w terminie 13–19 października 2013 roku:

  (1) Jarosław Kwiecień
  (2) Kamil Rychlewicz
  (3) Jan Ludziejewski
  (4) Michał Zieliński

  rezerwowi:

  (5) Karol Kaszuba
  (6) Przemysław Jakub Kozłowski

  Na CEOI'2013 pojadą zawodnicy, którzy nie uczęszczają do klas maturalnych, w kolejności rankingowej, ponieważ olimpiada odbędzie się w nowym roku szkolnym.

- **Bałtycką Olimpiadę Informatyczną BOI'2013**, która odbędzie się w Niemczech w terminie 8–12 kwietnia 2013 roku:

  (1) Błażej Magnowski
  (2) Stanisław Dobrowolski
  (3) Marek Sommer
  (4) Karol Farbiś
  (5) Tomasz Syposz
  (6) Paweł Nowak

  zawodnicy dodatkowi:

  (7) Igor Kotrasiński
  (8) Krzysztof Pszeniczny

Komitet Główny podjął następujące uchwały o udziale młodzieży w obozach:

- w Obozie Naukowo-Treningowym im. A. Kreczmara w Kielnarowej k. Rzeszowa wezmą udział laureaci i finaliści Olimpiady, którzy nie uczęszczają w tym roku szkolnym do programowo najwyższej klasy szkoły ponadgimnazjalnej,

- w Obozie Czesko-Polsko-Słowackim, który odbędzie się w Warszawie, wezmą udział reprezentanci na Międzynarodową Olimpiadę Informatyczną, wraz z zawodnikami rezerwowymi.

Sekretariat wystawił łącznie 31 zaświadczeń o uzyskaniu tytułu laureata, 33 zaświadczenia o uzyskaniu tytułu wyróżnionego finalisty oraz 32 zaświadczenia o uzyskaniu tytułu finalisty XX Olimpiady Informatycznej.

Komitet Główny wyróżnił dyplomami, za wkład pracy w przygotowanie finalistów Olimpiady Informatycznej, wszystkich podanych przez zawodników opiekunów naukowych:

- Michał Adamczyk (student Uniwersytetu Warszawskiego)
  - Patryk Czajka – laureat III miejsca
  - Rafał Stefański – laureat III miejsca
- Marcin Andrychowicz (student Uniwersytetu Warszawskiego)
  - Karol Farbiś – laureat II miejsca
  - Ewelina Krakowiak – laureatka III miejsca
  - Adrian Naruszko – finalista z wyróżnieniem
  - Konrad Sikorski – finalista z wyróżnieniem
  - Szymon Stankiewicz – finalista z wyróżnieniem
- Grzegorz Andrzejczak (Politechnika Łódzka)
  - Kamil Rychlewicz – laureat III miejsca
- Iwona Bujnowska (I Liceum Ogólnokształcące im. Adama Mickiewicza w Białymstoku)
  - Przemysław Jakub Kozłowski – laureat III miejsca
  - Piotr Gawryluk – finalista z wyróżnieniem
  - Magdalena Szarkowska – finalistka z wyróżnieniem
  - Michał Majewski – finalista
  - Marek Mystkowski – finalista
- Ireneusz Bujnowski (I Liceum Ogólnokształcące im. Adama Mickiewicza w Białymstoku)
  - Przemysław Jakub Kozłowski – laureat III miejsca
  - Piotr Gawryluk – finalista z wyróżnieniem
  - Mateusz Puczel – finalista z wyróżnieniem
  - Marek Sokołowski – finalista z wyróżnieniem
  - Magdalena Szarkowska – finalistka z wyróżnieniem
  - Mateusz Chołołowicz – finalista
  - Michał Majewski – finalista
  - Marek Mystkowski – finalista
  - Piotr Michał Padlewski – finalista
- Magda Burakowska (Zespół Szkół Ogólnokształcących nr 2 w Olsztynie)
  - Adam Czapliński – laureat III miejsca
  - Marcin Gregorczyk – finalista z wyróżnieniem

- Marek Cygan (Uniwersytet Warszawski)
  - Michał Kowalczyk – laureat III miejsca
  - Łukasz Łabęcki – finalista z wyróżnieniem
  - Wojciech Kordalski – finalista
- Mariusz Długoszewski (I Liceum Ogólnokształcące im. Zygmunta Krasińskiego w Ciechanowie)
  - Piotr Jarosz – finalista
- Czesław Drozdowski (XIII Liceum Ogólnokształcące w Szczecinie)
  - Paweł Nowak – laureat II miejsca
  - Michał Cylwik – finalista z wyróżnieniem
  - Filip Chmielewski – finalista
- Bartłomiej Dudek (student Uniwersytetu Wrocławskiego)
  - Tomasz Syposz – laureat II miejsca
  - Daniel Danielski – laureat III miejsca
  - Bartosz Kostka – laureat III miejsca
  - Jarosław Kwiecień – laureat III miejsca
  - Kamil Niziński – finalista z wyróżnieniem
  - Piotr Pietrzak – finalista z wyróżnieniem
- Lech Duraj (Uniwersytet Jagielloński)
  - Michał Zieliński – laureat III miejsca
  - Bartosz Dziewoński – finalista
- Andrzej Dyrek (V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie)
  - Michał Glapa – laureat III miejsca
  - Szymon Łukasz – laureat III miejsca
  - Michał Zieliński – laureat III miejsca
  - Jan Derbisz – finalista z wyróżnieniem
  - Jakub Łabaj – finalista z wyróżnieniem
  - Mateusz Nowak – finalista z wyróżnieniem
  - Bartosz Dziewoński – finalista
- Barbara Fandrejewska (Uniwersyteckie Katolickie Liceum Ogólnokształcące w Tczewie)
  - Adrian Akerman – finalista
- Marek Gałaszewski (I Liceum Ogólnokształcące im. Marii Konopnickiej w Suwałkach)
  - Kamil Dębowski – laureat III miejsca
- Alina Gościniak (VIII Liceum Ogólnokształcące im. Adama Mickiewicza w Poznaniu)
  - Mateusz Ledzianowski – finalista
- Eugeniusz Gwóźdź (Liceum Ogólnokształcące im. Noblistów Polskich w Rydułtowach)
  - Marcin Kudla – finalista z wyróżnieniem

- Grzegorz Herman (Uniwersytet Jagielloński)
  - Michał Glapa – laureat III miejsca
  - Szymon Łukasz – laureat III miejsca
  - Jan Derbisz – finalista z wyróżnieniem
  - Jakub Łabaj – finalista z wyróżnieniem
  - Mateusz Nowak – finalista z wyróżnieniem
- Krzysztof Hyżyk (Zespół Szkół Ogólnokształcących nr 6 w Bydgoszczy)
  - Sebastian Jaszczur – finalista z wyróżnieniem
- Wiesław Jakubiec (III Liceum Ogólnokształcące im. Stefana Żeromskiego w Bielsku-Białej)
  - Jarosław Jedynak – finalista
- Wiktor Janas (Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej we Wrocławiu)
  - Bartosz Kostka – laureat III miejsca
- Henryk Kawka (Zespół Szkół nr 7 – Gimnazjum nr 24 w Lublinie)
  - Albert Gutowski – finalista z wyróżnieniem
- Karol Konaszyński (student Uniwersytetu Wrocławskiego)
  - Tomasz Syposz – laureat II miejsca
  - Kamil Niziński – finalista z wyróżnieniem
  - Piotr Pietrzak – finalista z wyróżnieniem
- Paweł Kura (student Uniwersytetu Warszawskiego)
  - Piotr Domański – finalista
- Wiktor Kuropatwa (student Uniwersytetu Jagiellońskiego)
  - Michał Glapa – laureat III miejsca
- Anna Kwiatkowska (Gimnazjum i Liceum Akademickie w Toruniu)
  - Aleksander Zendel – finalista
  - Wojciech Zieliński – finalista
- Romualda Laskowska (I Liceum Ogólnokształcące im. Tadeusza Kościuszki w Legnicy)
  - Wojciech Janczewski – finalista z wyróżnieniem
  - Aleksander Łukasiewicz – finalista z wyróżnieniem
  - Bartłomiej Najdecki – finalista
- Krzysztof Loryś (Uniwersytet Wrocławski)
  - Daniel Danielski – laureat III miejsca
- Piotr Łowicki (Zespół Szkół Ogólnokształcących – I Liceum Ogólnokształcące im. Tadeusza Kościuszki w Łomży)
  - Marek Sokołowski – finalista z wyróżnieniem
- Jan Marcinkowski (student Uniwersytetu Wrocławskiego)
  - Aleksander Łukasiewicz – finalista z wyróżnieniem
- Paweł Mateja (I Liceum Ogólnokształcące im. Mikołaja Kopernika w Łodzi)
  - Kamil Rychlewicz – laureat III miejsca

- Dawid Matla (Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej we Wrocławiu)
  - Tomasz Syposz – laureat II miejsca
  - Bartosz Kostka – laureat III miejsca
  - Jarosław Kwiecień – laureat III miejsca
  - Kamil Niziński – finalista z wyróżnieniem
- Maciej Matraszek (student Uniwersytetu Warszawskiego)
  - Patryk Czajka – laureat III miejsca
  - Grzegorz Fabiański – laureat III miejsca
  - Rafał Stefański – laureat III miejsca
- Mirosław Mortka (VI Liceum Ogólnokształcące im. Jana Kochanowskiego w Radomiu)
  - Karol Farbiś – laureat II miejsca
  - Ewelina Krakowiak – laureatka III miejsca
  - Adrian Naruszko – finalista z wyróżnieniem
  - Konrad Sikorski – finalista z wyróżnieniem
  - Szymon Stankiewicz – finalista z wyróżnieniem
  - Kacper Oreszczuk – finalista
- Wojciech Nadara (student Uniwersytetu Warszawskiego)
  - Patryk Czajka – laureat III miejsca
- Andrzej Nowak (XIV Liceum Ogólnokształcące im. Stanisława Staszica w Warszawie)
  - Karol Kaszuba – laureat III miejsca
- Rafał Nowak (Uniwersytet Wrocławski)
  - Tomasz Syposz – laureat II miejsca
  - Bartosz Kostka – laureat III miejsca
  - Jarosław Kwiecień – laureat III miejsca
  - Kamil Niziński – finalista z wyróżnieniem
- Beata Padlewska (Zespół Szkół Ogólnokształcących nr 9 w Białymstoku)
  - Piotr Michał Padlewski – finalista
- Marcin Panasiuk (absolwent Uniwersytetu Wrocławskiego)
  - Wojciech Janczewski – finalista z wyróżnieniem
- Anna Piekarska (studentka Uniwersytetu Wrocławskiego)
  - Jarosław Kwiecień – laureat III miejsca
- Małgorzata Piekarska (Zespół Szkół Ogólnokształcących nr 6 w Bydgoszczy)
  - Michał Kowalczyk – laureat III miejsca
  - Sebastian Jaszczur – finalista z wyróżnieniem
  - Łukasz Łabęcki – finalista z wyróżnieniem
  - Wojciech Kordalski – finalista
  - Patryk Osmólski – finalista
  - Dawid Wegner – finalista

- Mirosław Pietrzycki (I Liceum Ogólnokształcące im. Stanisława Staszica w Lublinie)
  - Albert Gutowski – finalista z wyróżnieniem
- Andrzej Piotrowski (Zespół Szkół Ogólnokształcących w Krośnie)
  - Kamil Braun – finalista
- Karol Pokorski (student Uniwersytetu Wrocławskiego)
  - Kamil Niziński – finalista z wyróżnieniem
  - Paweł Wegner – finalista z wyróżnieniem
- Damian Rusak (student Uniwersytetu Wrocławskiego)
  - Bartosz Kostka – laureat III miejsca
  - Kamil Niziński – finalista z wyróżnieniem
- Piotr Smulewicz (student Uniwersytetu Warszawskiego)
  - Jan Ludziejewski – laureat III miejsca
- Hanna Stachera (XIV Liceum Ogólnokształcące im. Stanisława Staszica w Warszawie)
  - Marek Sommer – laureat I miejsca
  - Grzegorz Fabiański – laureat III miejsca
  - Rafał Stefański – laureat III miejsca
- Władysław Strejczek (IV Liceum Ogólnokształcące im. Jana Pawła II w Tarnowie)
  - Michał Łuszczyk – laureat III miejsca
- Agata Suścicka (II Liceum Ogólnokształcące im. Stanisława Wyspiańskiego w Legnicy)
  - Tomasz Rewak – finalista
- Piotr Suwara (student Uniwersytetu Warszawskiego)
  - Grzegorz Fabiański – laureat III miejsca
- Bartosz Szreder (doktorant Uniwersytetu Warszawskiego)
  - Patryk Czajka – laureat III miejsca
  - Karol Kaszuba – laureat III miejsca
  - Jan Ludziejewski – laureat III miejsca
  - Monika Olchowik – finalistka z wyróżnieniem
  - Bartłomiej Najdecki – finalista
- Ryszard Szubartowski (III Liceum Ogólnokształcące im. Marynarki Wojennej RP w Gdyni, Stowarzyszenie TALENT)
  - Błażej Magnowski – laureat I miejsca
  - Stanisław Barzowski – laureat III miejsca
  - Paweł Burzyński – finalista z wyróżnieniem
  - Michał Kowalewski – finalista z wyróżnieniem
  - Bartosz Łukasiewicz – finalista z wyróżnieniem
  - Paweł Wegner – finalista z wyróżnieniem
  - Kamil Żyła – finalista z wyróżnieniem
  - Tomasz Miotk – finalista

- Stanisław Szulc (I Liceum Ogólnokształcące im. Zygmunta Krasińskiego w Ciechanowie)
  - Piotr Jarosz – finalista
- Joanna Śmigielska (XIV Liceum Ogólnokształcące im. Stanisława Staszica w Warszawie)
  - Marek Sommer – laureat I miejsca
  - Stanisław Dobrowolski – laureat I miejsca
  - Igor Kotrasiński – laureat II miejsca
  - Patryk Czajka – laureat III miejsca
  - Paweł Nałęcz-Jawecki – laureat III miejsca
  - Konrad Paluszek – laureat III miejsca
  - Rafał Stefański – laureat III miejsca
  - Aleksander Matusiak – finalista z wyróżnieniem
  - Wojciech Jabłoński – finalista
  - Arkadiusz Wróbel – finalista
- Bartosz Tarnawski (student Uniwersytetu Warszawskiego)
  - Piotr Domański – finalista
- Wojciech Tomalczyk (III Liceum Ogólnokształcące im. Marynarki Wojennej RP w Gdyni)
  - Paweł Burzyński – finalista z wyróżnieniem
- Jacek Tomasiewicz (student Uniwersytetu Warszawskiego)
  - Przemysław Jakub Kozłowski – laureat III miejsca
  - Piotr Gawryluk – finalista z wyróżnieniem
  - Mateusz Puczel – finalista z wyróżnieniem
  - Magdalena Szarkowska – finalistka z wyróżnieniem
  - Mateusz Chołołowicz – finalista
  - Michał Majewski – finalista
  - Marek Mystkowski – finalista
  - Piotr Michał Padlewski – finalista
- Szymon Wąsik (Politechnika Poznańska)
  - Mateusz Ledzianowski – finalista
- Grzegorz Witek (II Liceum Ogólnokształcące im. Mikołaja Kopernika w Mielcu)
  - Jakub Cisło – finalista

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

Warszawa, 15 lipca 2013 roku

Regulamin Olimpiady Informatycznej

§1 WSTĘP

Olimpiada Informatyczna, zwana dalej Olimpiadą, jest olimpiadą przedmiotową powołaną przez Instytut Informatyki Uniwersytetu Wrocławskiego 10 grudnia 1993 roku. Olimpiada działa zgodnie z Rozporządzeniem Ministra Edukacji Narodowej i Sportu z 29 stycznia 2002 roku w sprawie organizacji oraz sposobu przeprowadzenia konkursów, turniejów i olimpiad (Dz. U. Nr 13, poz. 125). Organizatorem Olimpiady Informatycznej jest Fundacja Rozwoju Informatyki, zwana dalej Organizatorem. W organizacji Olimpiady Fundacja Rozwoju Informatyki współdziała z Wydziałem Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego, Instytutem Informatyki Uniwersytetu Wrocławskiego, Katedrą Algorytmiki Uniwersytetu Jagiellońskiego, Wydziałem Matematyki i Informatyki Uniwersytetu im. Mikołaja Kopernika w Toruniu, Instytutem Informatyki Wydziału Automatyki, Elektroniki i Informatyki Politechniki Śląskiej w Gliwicach, Wydziałem Informatyki Politechniki Poznańskiej, a także z innymi środowiskami akademickimi, zawodowymi i oświatowymi działającymi w sprawach edukacji informatycznej.

§2 CELE OLIMPIADY I SPOSOBY ICH OSIĄGANIA

(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.

Cele Olimpiady są osiągane poprzez:

- organizację olimpiady przedmiotowej z informatyki dla uczniów szkół ponadgimnazjalnych;
- organizowanie corocznych obozów naukowych dla wyróżniających się uczestników olimpiad;

- organizowanie warsztatów treningowych dla nauczycieli zainteresowanych przygotowywaniem uczniów do udziału w olimpiadach;
- przygotowywanie i publikowanie materiałów edukacyjnych dla uczniów zainteresowanych udziałem w olimpiadach i ich nauczycieli.

§3 ORGANIZACJA OLIMPIADY

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

(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; miejsce i sposób przekazania określone są w „Zasadach organizacji zawodów" danej edycji Olimpiady, 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.

(8) 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.

(9) 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, czas działania tego programu oraz ilość wymaganej przez program pamięci. Jeśli rozwiązaniem zadania jest plik z danymi, wówczas ocenia się poprawność danych. Za każde zadanie zawodnik może zdobyć maksymalnie 100 punktów, gdzie 100 jest sumą maksymalnych liczb punktów za poszczególne testy (lub dane z wynikami) dla tego zadania. Oceną rozwiązań zawodnika jest suma punktów za poszczególne zadania. Oceny rozwiązań zawodników są podstawą utworzenia listy rankingowej zawodników po zawodach każdego stopnia.

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

(11) Rozwiązania zespołowe, niesamodzielne, niezgodne z zasadami lub takie, co do których nie można ustalić autorstwa, nie będą oceniane. W przypadku uznania przez Komitet Główny pracy za niesamodzielną lub zespołową zawodnicy mogą zostać zdyskwalifikowani.

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

(13) W szczególnie rażących wypadkach łamania Regulaminu lub Zasad, Komitet Główny może zdyskwalifikować zawodnika.

(14) 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.

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

(16) 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.

(17) Na podstawie analizy rozwiązań zadań w zawodach III stopnia i listy rankingowej Komitet Główny przyznaje tytuły laureatów Olimpiady Informatycznej: I stopnia (prace na poziomie złotych medalistów Międzynarodowej Olimpiady Informatycznej), II stopnia (prace na poziomie srebrnych medalistów Międzynarodowej Olimpiady Informatycznej), III stopnia (prace na poziomie brązowych medalistów Międzynarodowej Olimpiady Informatycznej) i nagradza ich medalami, odpowiednio, złotymi, srebrnymi i brązowymi. Liczba laureatów nie przekracza połowy uczestników zawodów finałowych.

(18) W przypadku bardzo wysokiego poziomu finałów Komitet Główny może dodatkowo wyróżnić uczniów niebędących laureatami.

(19) Zwycięzcą Olimpiady Informatycznej zostaje osoba, która osiągnęła najlepszy wynik w zawodach finałowych.

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

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

(4) Komitet powołuje i rozwiązuje komitety okręgowe Olimpiady.

(5) Komitet:

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

   (b) udziela wyjaśnień w sprawach dotyczących Olimpiady;

   (c) zatwierdza listy rankingowe oraz listy laureatów i wyróżnionych uczestników;

   (d) przyznaje uprawnienia i nagrody rzeczowe wyróżniającym się uczestnikom Olimpiady;

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

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

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

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

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

(10) Komitet dysponuje funduszem Olimpiady za zgodą Organizatora i za pośrednictwem kierownika organizacyjnego Olimpiady.

(11) Komitet przyjmuje plan finansowy Olimpiady na przyszły rok na ostatnim posiedzeniu w roku poprzedzającym.

(12) Komitet przyjmuje sprawozdanie finansowe z przebiegu Olimpiady na ostatnim posiedzeniu w roku, na dzień 30 listopada.

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

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

(15) Kierownik Jury w porozumieniu z przewodniczącym powołuje i odwołuje członków Jury Olimpiady, które jest odpowiedzialne za opracowanie i sprawdzanie zadań.

(16) Kierownik techniczny odpowiada za stronę techniczną przeprowadzenia zawodów.

(17) 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.

(18) 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.

(19) 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 Główny.

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

§6 PRZEBIEG OLIMPIADY

(1) Komitet Główny rozsyła do szkół wymienionych w § 3.3 oraz kuratoriów oświaty i koordynatorów edukacji informatycznej informację o przebiegu danej edycji Olimpiady.

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

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

§7 UPRAWNIENIA I NAGRODY

(1) 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 punktach 1. i 2. przysługują na zasadach określonych w rozporządzeniu MEN z 30 kwietnia 2007 r. w sprawie warunków i sposobu oceniania, klasyfikowania i promowania uczniów i słuchaczy oraz przeprowadzania sprawdzianów i egzaminów w szkołach publicznych (Dz. U. z 2007 r. Nr 83, poz. 562, §§ 20 i 60).

(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 27 lipca 2005 r. „Prawo o szkolnictwie wyższym", na zasadach zawartych w tych uchwałach (Dz. U. Nr 164, poz. 1365).

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

(6) Komitet Główny może nagrodzić opiekunów, których praca przy przygotowaniu uczestnika Olimpiady zostanie oceniona przez ten Komitet jako wyróżniająca.

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

(8) Komitet Główny może przyznawać wyróżniającym się aktywnością członkom Komitetu Głównego i komitetów okręgowych nagrody pieniężne z funduszu Olimpiady.

(9) Osobom, które wniosły szczególnie duży wkład w rozwój Olimpiady Informatycznej, Komitet Główny może przyznać honorowy tytuł „Zasłużony dla Olimpiady Informatycznej".

§8 FINANSOWANIE OLIMPIADY

Komitet Główny finansuje działania Olimpiady zgodnie z umową podpisaną przez Ministerstwo Edukacji Narodowej i Fundację Rozwoju Informatyki. Komitet Główny 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 Główny zatwierdza sprawozdanie merytoryczne i przedstawia je Organizatorowi celem przedłożenia Ministerstwu Edukacji Narodowej.

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

Zasady organizacji zawodów XX Olimpiady Informatycznej w roku szkolnym 2012/2013

§1 WSTĘP

Olimpiada Informatyczna, zwana dalej Olimpiadą, jest olimpiadą przedmiotową powołaną przez Instytut Informatyki Uniwersytetu Wrocławskiego 10 grudnia 1993 roku. Olimpiada działa zgodnie z Rozporządzeniem Ministra Edukacji Narodowej i Sportu z 29 stycznia 2002 roku w sprawie organizacji oraz sposobu przeprowadzenia konkursów, turniejów i olimpiad (Dz. U. 02.13.125). Organizatorem Olimpiady Informatycznej jest Fundacja Rozwoju Informatyki. W organizacji Olimpiady Fundacja Rozwoju Informatyki współdziała z Wydziałem Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego, Instytutem Informatyki Uniwersytetu Wrocławskiego, Katedrą Algorytmiki Uniwersytetu Jagiellońskiego, Wydziałem Matematyki i Informatyki Uniwersytetu im. Mikołaja Kopernika w Toruniu, Instytutem Informatyki Wydziału Automatyki, Elektroniki i Informatyki Politechniki Śląskiej w Gliwicach, Wydziałem Informatyki Politechniki Poznańskiej, a także z innymi środowiskami akademickimi, zawodowymi i oświatowymi działającymi w sprawach edukacji informatycznej oraz firmą Asseco Poland SA. Partnerami Olimpiady są OFEK i OEIiZK.

§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. W Olimpiadzie mogą również uczestniczyć – za zgodą Komitetu Głównego – uczniowie szkół podstawowych i gimnazjów.

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

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

(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 380 uczestników, których rozwiązania zadań I stopnia zostaną ocenione najwyżej; do zawodów III stopnia – 80 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 – 15 października–12 listopada 2012 roku

  ogłoszenie wyników w witrynie Olimpiady – 7 grudnia 2012 roku godz. 20.00

  rozesłanie pocztą materiałów Olimpiady (w tym związanych z zawodami II stopnia) do wszystkich uczestników Olimpiady – 14 grudnia 2012 roku
- zawody II stopnia – 12–14 lutego 2013 roku

  ogłoszenie wyników w witrynie Olimpiady – 22 lutego 2013 roku godz. 20.00
- zawody III stopnia – 12–16 marca 2013 roku

§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 lub takie, co do których nie można ustalić autorstwa, nie będą oceniane. W przypadku uznania przez Komitet Główny pracy za niesamodzielną lub zespołową zawodnicy mogą zostać zdyskwalifikowani.

(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 powinny być podane w komentarzu na początku każdego programu.

(5) Nazwy plików z programami w postaci źródłowej muszą mieć następujące rozszerzenia zależne od użytego języka programowania:

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

(6) Podczas oceniania skompilowane programy będą wykonywane w wirtualnym środowisku uruchomieniowym modelującym zachowanie 32-bitowego procesora serii Intel Pentium 4, pod kontrolą systemu operacyjnego Linux. Ma to na celu uniezależnienie mierzonego czasu działania programu od modelu komputera, na którym odbywa się sprawdzanie. Daje także zawodnikom możliwość wygodnego testowania efektywności działania programów w warunkach oceny. Przygotowane środowisko jest dostępne, wraz z opisem działania, w witrynie Olimpiady, na stronie *Środowisko testowe* w dziale „Dla zawodników" (zarówno dla systemu Linux, jak i Windows).

(7) W uzasadnionych przypadkach Komitet Główny zastrzega sobie prawo do oceny rozwiązań w rzeczywistym środowisku systemu operacyjnego Linux.

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

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

§4 ZAWODY I STOPNIA

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

- poprzez System Internetowy Olimpiady, zwany dalej SIO, o adresie: `http://sio.mimuw.edu.pl`, do 12 listopada 2012 roku do godz. 12.00 (południe). Komitet Główny nie ponosi odpowiedzialności za brak możliwości przekazania rozwiązań przez Internet w sytuacji nadmiernego obciążenia lub awarii SIO. Odbiór przesyłki zostanie potwierdzony przez SIO 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 rozwiązań i związanej z tym rejestracji będą dokładnie podane w SIO.
- pocztą, jedną 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 12 listopada 2012 roku (decyduje data stempla pocztowego). Uczestnik ma obowiązek zachować dowód nadania przesyłki do czasu otrzymania wyników oceny. Nawet w przypadku wysyłania rozwiązań pocztą, każdy uczestnik musi

założyć sobie konto w SIO. **Zarejestrowana nazwa użytkownika musi być zawarta w przesyłce.**

Rozwiązania dostarczane w inny sposób nie będą przyjmowane. W przypadku jednoczesnego zgłoszenia rozwiązania danego zadania przez SIO i listem poleconym, ocenie podlega jedynie rozwiązanie wysłane listem poleconym.

(2) Uczestnik korzystający z poczty zwykłej przysyła:

- nośnik (CD lub DVD) zawierający:
  - spis zawartości nośnika oraz nazwę użytkownika z SIO w pliku nazwanym `SPIS.TXT`;
  - do każdego rozwiązanego zadania – program źródłowy lub plik z danymi.

  Na nośniku nie powinno być żadnych podkatalogów.
  W przypadku braku możliwości odczytania nośnika z rozwiązaniami, nieodczytane rozwiązania nie będą brane pod uwagę.
- wypełniony dokument zgłoszeniowy (dostępny w witrynie internetowej Olimpiady).

(3) W trakcie rozwiązywania zadań można korzystać z dowolnej literatury oraz ogólnodostępnych kodów źródłowych. Należy wówczas podać w rozwiązaniu, w komentarzu, odnośnik do wykorzystanej literatury lub kodu.

(4) Podczas korzystania z SIO zawodnik postępuje zgodnie z instrukcjami umieszczonymi w tej witrynie. W szczególności, warunkiem koniecznym do kwalifikacji zawodnika do dalszych etapów jest podanie lub aktualizacja w SIO wszystkich wymaganych danych osobowych.

(5) Każdy uczestnik powinien założyć w SIO dokładnie jedno konto. Zawodnicy korzystający z więcej niż jednego konta mogą zostać zdyskwalifikowani.

(6) Rozwiązanie każdego zadania można zgłosić w SIO co najwyżej 10 razy. Spośród tych zgłoszeń oceniane jest jedynie najpóźniejsze poprawnie kompilujące się rozwiązanie. Po wyczerpaniu tego limitu kolejne rozwiązanie może zostać zgłoszone już tylko zwykłą pocztą.

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

(8) Poprzez SIO udostępniane są narzędzia do sprawdzania rozwiązań pod względem formalnym. Szczegóły dotyczące sposobu postępowania będą dokładnie podane w witrynie.

(9) Od 26.11.2012 roku poprzez SIO każdy zawodnik będzie mógł zapoznać się ze wstępną oceną swojej pracy.

(10) Do 30.11.2012 roku (włącznie) poprzez SIO każdy zawodnik będzie mógł zgłaszać uwagi do wstępnej oceny swoich rozwiązań. Reklamacji nie podlega jednak dobór testów, limitów czasowych, kompilatorów i sposobu oceny.

(11) Reklamacje złożone po 30.11.2012 roku 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.

(4) Zawody II i III stopnia są przeprowadzane za pomocą SIO.

(5) W czasie trwania zawodów nie można korzystać z żadnych książek ani innych pomocy takich jak: dyski, kalkulatory, notatki itp.

(6) Tryb przeprowadzenia zawodów II i III stopnia jest opisany szczegółowo w „Zasadach organizacji zawodów II i III stopnia".

§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 punktach 1. i 2. przysługują na zasadach określonych w rozporządzeniu MEN z 30 kwietnia 2007 r. w sprawie warunków i sposobu oceniania, klasyfikowania i promowania uczniów i słuchaczy oraz przeprowadzania sprawdzianów i egzaminów w szkołach publicznych (Dz. U. z 2007 roku Nr 83, poz. 562, §§ 20 i 60).

(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 roku Nr 164, poz. 1365).

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

(6) Komitet Główny ustala skład reprezentacji Polski na XXV Międzynarodową Olimpiadę Informatyczną w 2013 roku na podstawie wyników Olimpiady oraz regulaminu tej Olimpiady Międzynarodowej.

(7) Komitet Główny może nagrodzić opiekunów, których praca przy przygotowaniu uczestnika Olimpiady zostanie oceniona przez ten Komitet jako wyróżniająca.

(8) Wyznaczeni przez Komitet Główny reprezentanci Polski na olimpiady międzynarodowe zostaną zaproszeni do nieodpłatnego udziału w XIV Obozie Naukowo-Treningowym im. Antoniego Kreczmara, który odbędzie się w czasie wakacji 2013 roku. Do nieodpłatnego udziału w Obozie Komitet Główny może zaprosić także innych finalistów, którzy nie są w ostatniej programowo klasie swojej szkoły, w zależności od uzyskanych wyników.

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

§7 PRZEPISY KOŃCOWE

(1) Komitet Główny zawiadamia wszystkich uczestników zawodów I i II stopnia o ich wynikach. Wszyscy uczestnicy zawodów I stopnia zostaną zawiadomieni o swoich wynikach zwykłą pocztą, a poprzez SIO będą mogli zapoznać się ze szczegółowym raportem ze sprawdzania ich rozwiązań.

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

(3) 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`

Zasady organizacji zawodów II i III stopnia XX Olimpiady Informatycznej

(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 okręgowy lub 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 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 przedłużony o tyle, ile trwało usunięcie awarii. Awarie sprzętu należy zgłaszać dyżurującym członkom Komisji Regulaminowej.

(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) Podczas każdej sesji:

   (a) W trakcie pierwszych 60 minut nie wolno opuszczać przydzielonej sali zawodów. Zawodnicy spóźnieni więcej niż godzinę nie będą w tym dniu dopuszczeni do zawodów.

   (b) W trakcie pierwszych 90 minut każdej sesji uczestnik może zadawać pytania dotyczące treści zadań, w ustalony przez Jury sposób, na które otrzymuje

jedną z odpowiedzi: *tak*, *nie*, *niepoprawne pytanie*, *odpowiedź wynika z treści zadania* lub *bez odpowiedzi*. Pytania techniczne można zadawać podczas całej sesji zawodów.

(c) W SIO znajdą się też publiczne odpowiedzi na pytania zawodników. Odpowiedzi te mogą zawierać ważne informacje dotyczące toczących się zawodów, więc wszyscy uczestnicy zawodów proszeni są o regularne zapoznawanie się z ukazującymi się odpowiedziami.

(d) Jakikolwiek inny sposób komunikowania się z członkami Jury co do treści i sposobów rozwiązywania zadań jest niedopuszczalny.

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

(f) Każdy zawodnik ma prawo drukować wyniki swojej pracy w sposób opisany w Ustaleniach technicznych.

(g) Każdy zawodnik powinien umieścić ostateczne rozwiązania zadań w SIO, za pomocą przeglądarki lub za pomocą skryptu do wysyłania rozwiązań `submit`. Skrypt `submit` działa także w przypadku awarii sieci, wówczas rozwiązanie zostaje automatycznie dostarczone do SIO, gdy komputer odzyska łączność z siecią. Tylko zgłoszone w podany sposób rozwiązania zostaną ocenione.

(h) 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 liczą się do końcowej klasyfikacji). Te same testy przykładowe są używane do wstępnego sprawdzenia za pomocą skryptu do weryfikacji rozwiązań na komputerze zawodnika (skryptu „ocen").

(i) Podczas zawodów III stopnia, w przypadku niektórych zadań, wskazanych przez Komitet Główny, zawodnicy będą mogli poznać wynik punktowy swoich trzech wybranych zgłoszeń. Przez ostatnie 30 minut zawodów ta opcja nie będzie dostępna.

(j) Rozwiązanie każdego zadania można zgłosić co najwyżej 10 razy. Spośród tych zgłoszeń oceniane jest jedynie najpóźniejsze poprawnie kompilujące się rozwiązanie.

(10) Każdy program zawodnika powinien mieć na początku komentarz zawierający imię i nazwisko autora.

(11) W sprawach spornych decyzje podejmuje Jury Odwoławcze, złożone z jurora niezaangażowanego w daną kwestię i wyznaczonego członka Komitetu Głównego lub kierownika danego regionu podczas II etapu. Decyzje w sprawach o wielkiej wadze (np. dyskwalifikacji zawodników) Jury Odwoławcze podejmuje w porozumieniu z przewodniczącym Komitetu Głównego.

(12) Każdego dnia zawodów, po około dwóch godzinach od zakończenia sesji, zawodnicy otrzymają raporty oceny swoich prac na wybranym zestawie testów. Od tego momentu, przez pół godziny będzie czas na reklamację tej oceny, a w szczególności na reklamację wyboru rozwiązania, które ma podlegać ocenie.

(13) Od 14.02.2013 roku od godz. 20.00 do 18.02.2013 roku do godz. 20.00 poprzez SIO każdy zawodnik będzie mógł zapoznać się z pełną oceną swoich rozwiązań z zawodów II stopnia i zgłaszać uwagi do tej oceny. Reklamacji nie podlega jednak dobór testów, limitów czasowych, kompilatorów i sposobu oceny.

Zawody I stopnia

opracowania zadań

Jakub Radoszewski
Treść zadania, Opracowanie

Dawid Dąbrowski, Karol Pokorski
Program

Dostępna pamięć: 64 MB.

OI, etap I, 15.10–12.11.2012

Cennik

Najpopularniejszym środkiem transportu w Bajtocji od zawsze była kolej. Spośród n miast tego kraju, m par miast jest połączonych odcinkami torów należącymi do Bajtockich Kolei Państwowych (BKP). Odcinki nie krzyżują się nigdzie poza miastami i mogą prowadzić tunelami lub mostami. Koszt przejazdu między dwoma miastami połączonymi bezpośrednio odcinkiem torów jest zawsze taki sam i wynosi a bajtalarów.

Obecnie sytuacja na rynku usług komunikacyjnych w Bajtocji uległa zmianie. Pojawiła się konkurencja dla BKP – powstały Bajtockie Linie Lotnicze (BLL). BLL zamierzają uruchomić połączenia lotnicze między niektórymi parami miast. Ponieważ jazda bajtocką koleją jest bardzo wygodna, zarząd BLL postanowił uruchamiać połączenia lotnicze tylko między takimi parami miast, dla których **nie** *istnieje bezpośrednie połączenie kolejowe. Ze względów ekonomicznych, BLL utworzy połączenia lotnicze jedynie między tymi parami miast, dla których najtańsze połączenie kolejowe wymaga dokładnie jednej przesiadki. Koszt biletu lotniczego na jedno połączenie będzie stały i równy b bajtalarów.*

Żeby pomóc mieszkańcom Bajtocji w planowaniu podróży, Ministerstwo Transportu Bajtocji (MTB) postanowiło stworzyć cenniki zawierające koszty najtańszych tras między poszczególnymi miastami kraju. Trasę rozumiemy tu jako sekwencję złożoną z dowolnej liczby pojedynczych połączeń kolejowych lub lotniczych. Zadanie stworzenia cenników przypadło w udziale Bajtazarowi, który pracuje jako urzędnik w MTB. Czy pomógłbyś mu w napisaniu programu, który wyznaczy odpowiednie cenniki?

Dodajmy dla jasności, że wszystkie połączenia kolejowe i lotnicze w Bajtocji są dwukierunkowe.

Wejście

Pierwszy wiersz standardowego wejścia zawiera pięć liczb całkowitych n, m, k, a oraz b ($2 \leqslant n \leqslant 100\,000$, $1 \leqslant m \leqslant 100\,000$, $1 \leqslant k \leqslant n$, $1 \leqslant a, b \leqslant 1000$) pooddzielanych pojedynczymi odstępami. Liczby n oraz m oznaczają odpowiednio liczbę miast oraz liczbę połączeń kolejowych w Bajtocji. Dla uproszczenia miasta w Bajtocji numerujemy od 1 do n. Kolejne liczby w wierszu oznaczają: k – numer miasta początkowego, dla którego należy wygenerować cennik opisujący najtańsze trasy; a – koszt biletu na jedno połączenie kolejowe; b – koszt biletu na jedno połączenie lotnicze.

Każdy z kolejnych m wierszy zawiera dwie liczby całkowite u_i oraz v_i ($1 \leqslant u_i, v_i \leqslant n$, $u_i \neq v_i$ dla $i = 1, 2, \ldots, m$) oddzielone pojedynczym odstępem, oznaczające numery miast połączonych bezpośrednim odcinkiem torów.

Możesz założyć, że z miasta numer k można dojechać koleją do wszystkich pozostałych miast kraju.

W testach wartych łącznie 30% punktów zachodzą dodatkowe warunki $n \leqslant 700$ oraz $m \leqslant 700$.

Wyjście

Twój program powinien wypisać na standardowe wyjście n wierszy. Wiersz o numerze i (dla $i = 1, 2, \ldots, n$) powinien zawierać jedną liczbę całkowitą: koszt najtańszej trasy z miasta numer k do miasta numer i. Spośród tych wierszy, wiersz o numerze k powinien zawierać liczbę 0.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`5 5 1 3 2`	`0`
`1 2`	`3`
`2 3`	`3`
`3 4`	`2`
`4 5`	`5`
`3 1`	

Wyjaśnienie do przykładu: *Najtańsza trasa z miasta numer 1 do miasta numer 5 wiedzie przez miasto numer 3 lub miasto numer 4. W obu przypadkach składa się ona z jednego połączenia kolejowego i jednego lotniczego.*

Rozwiązanie

Problem opisany w tym zadaniu ma naturalną interpretację grafową. Oznaczmy przez $G = (V, E)$ graf nieskierowany, w którym wierzchołki odpowiadają miastom ($|V| = n$), a krawędzie – połączeniom kolejowym ($|E| = m$). Niech dalej $G' = (V, E')$ oznacza graf reprezentujący połączenia lotnicze (oznaczmy $m' = |E'|$). Zbiór E' można całkiem zgrabnie opisać, korzystając z pojęcia *kwadratu* grafu. Kwadratem grafu $G = (V, E)$ nazywamy graf $G^2 = (V, E^2)$, w którym dwa wierzchołki są połączone krawędzią, jeśli w grafie G istnieje między nimi ścieżka o długości 2, tzn.:

$$E^2 = \{uw : uv, vw \in E \text{ dla pewnego } v \in V\}.$$

Krawędzie w grafie G' łączą wierzchołki położone w *odległości* dokładnie 2 w grafie G, więc $E' = E^2 \setminus E$. Celem zadania jest wyznaczenie długości najkrótszych ścieżek z ustalonego wierzchołka $k \in V$ w ważonym grafie $G'' = (V, E \cup E')$, w którym krawędzie ze zbioru E mają wagę a, a krawędzie ze zbioru E' – wagę b. Zakładamy dla uproszczenia, że same grafy G i G' nie są ważone.

Do rozwiązania zadania możemy zastosować algorytm Dijkstry zaimplementowany z pomocą kopca zupełnego, który pozwala wyznaczyć długości najkrótszych ścieżek ze źródła (tj. wierzchołka k) w czasie $O((n + m + m') \log n)$. Aby stwierdzić, na ile dobre jest to rozwiązanie, należy odpowiedzieć na pytanie, jak duże może być m'. Krawędziami ze zbioru E' łączymy tylko bliskie sobie wierzchołki w grafie G, a takich par wierzchołków w grafie intuicyjnie nie powinno być zbyt wiele... Niestety, istnieją grafy rzadkie (tj. spełniające warunek $m = O(n)$), w których wszystkie wierzchołki są położone stosunkowo blisko siebie. Przykładem może tu być *gwiazda*, czyli drzewo,

w którym $n-1$ wierzchołków jest podłączonych bezpośrednio do jednego, centralnego wierzchołka. Jeśli G jest taką gwiazdą, to graf G'' stanowi klikę (graf pełny) i wówczas $m' = \Theta(n^2)$. Nasz algorytm działa zatem pesymistycznie w czasie $\Theta(n^2 \log n)$. W takim przypadku już lepiej byłoby wykorzystać prostszą wersję algorytmu Dijkstry, w której do implementacji kolejki priorytetowej stosuje się tablicę przechowującą aktualne wagi wierzchołków. Wówczas złożoność czasowa spada do $O(n^2)$, jednak wciąż nie jest ona zadowalająca. Zgodnie z warunkiem z treści zadania, opisane tu rozwiązania mogły zdobyć 30% punktów. Program zawierający tego typu rozwiązanie można znaleźć w pliku `cens1.cpp`.

Najkrótsze ścieżki

Aby uzyskać bardziej efektywne rozwiązanie, zamiast od razu uruchamiać algorytm Dijkstry warto skorzystać ze specjalnej struktury grafu G''. Niech k będzie źródłem i niech $v \in V$ będzie dowolnym wierzchołkiem grafu. Zastanówmy się, jak może wyglądać najkrótsza ścieżka z k do v w grafie G''.

Na początek warto rozpatrzyć przypadek, w którym w ogóle nie opłaca się nam używać połączeń lotniczych. Jest tak, gdy koszty biletów spełniają nierówność $b \geqslant 2a$. Wówczas szukana najkrótsza ścieżka odpowiada po prostu najkrótszej ścieżce z k do v w grafie G. Długości takich ścieżek do wszystkich $v \in V$ możemy wyznaczyć w czasie $O(n + m)$ za pomocą algorytmu BFS (przeszukiwanie wszerz), więc ten przypadek możemy uznać za rozpatrzony.

Odtąd założymy, że $b < 2a$. Oznaczmy przez x długość najkrótszej ścieżki z k do v w grafie G, przez y – długość najkrótszej takiej ścieżki w grafie G', a przez z – szukaną długość najkrótszej ścieżki w ważonym grafie G''. Dalsza część rozwiązania zależy od parzystości liczby x. Rozważymy dwa przypadki, opisane w poniższych lematach.

Lemat 1. Jeśli $2 \mid x$, to $y = \frac{x}{2}$ i w konsekwencji $z = y \cdot b$.

Dowód: Uzasadnijmy najpierw, dlaczego w tym przypadku zachodzi $y = \frac{x}{2}$. Niech $P = (u_0, u_1, \ldots, u_x)$ będzie najkrótszą ścieżką łączącą wierzchołki $u_0 = k$ i $u_x = v$. Żadna para wierzchołków u_i, u_{i+2} nie jest połączona krawędzią w grafie G, gdyż wówczas w grafie G istniałaby ścieżka z k do v krótsza niż P. Tak więc ścieżka P wyznacza ścieżkę $P' = (u_0, u_2, \ldots, u_{x-2}, u_x)$ z k do v w grafie G' o długości $\frac{x}{2}$. Z drugiej strony, gdyby w grafie G' istniała ścieżka z k do v o długości mniejszej niż $\frac{x}{2}$, to wyznaczałaby ona ścieżkę w grafie G o długości mniejszej niż x, a założyliśmy, że takiej ścieżki nie ma.

Uzasadniliśmy już pierwszą część tezy lematu i tym samym skonstruowaliśmy ścieżkę z k do v w grafie G'' o długości $y \cdot b$. Wiemy też, że ani najkrótsza ścieżka w grafie G, ani najkrótsza ścieżka w grafie G' przeniesiona do grafu G'' nie ma kosztu mniejszego niż $y \cdot b$. W grafie G'' mogłaby teoretycznie istnieć ścieżka o mniejszej długości, złożona z krawędzi pochodzących zarówno z G, jak i z G'. Taka ścieżka, zawierająca p krawędzi z G i q krawędzi z G', odpowiadałaby ścieżce o długości $p+2q$ w grafie G. Jednak $p + 2q \geqslant x$, więc taka ścieżka miałaby w grafie G'' długość

$$p \cdot a + q \cdot b = \tfrac{1}{2}(p \cdot 2a + 2q \cdot b) > \tfrac{1}{2}b(p + 2q) \geqslant \tfrac{1}{2}b \cdot x = y \cdot b,$$

czyli nie byłaby krótsza niż ścieżka P'. ■

Lemat 2. Jeśli $2 \nmid x$, to $z = \min(\frac{x-1}{2} \cdot b + a,\ y \cdot b)$.

Dowód: Niech P będzie najkrótszą ścieżką z k do v w grafie G, a P' – najkrótszą taką ścieżką w grafie G'. Teza lematu orzeka, że najkrótsza ścieżka z k do v w grafie G'' to albo ścieżka P, na której pary kolejnych krawędzi zamieniono na krawędzie z G' (ostatnią krawędź ścieżki zostawiamy bez zmian), albo ścieżka P'. Wykażemy, że każda inna ścieżka w grafie G'' ma wagę nie mniejszą od tych dwóch wymienionych.

Niech $P'' = (u_0, \ldots, u_d)$ będzie najkrótszą ścieżką z $u_0 = k$ do $u_d = v$ w grafie G''. Jeśli jest więcej niż jedna taka ścieżka, wybieramy tę, która zawiera minimalną liczbę krawędzi pochodzących z G, a jeśli wciąż mamy wybór, wybieramy ścieżkę, w której pierwsza krawędź pochodząca z G występuje możliwie najpóźniej. Uzasadnimy, że P'' albo nie zawiera żadnych krawędzi z G, albo zawiera dokładnie jedną, położoną na samym końcu. To już nam wystarczy do wykazania tezy lematu: jeśli P'' nie zawiera żadnych krawędzi pochodzących z grafu G, to jest nie krótsza niż P', a w przeciwnym razie jej długość, wyrażona w krawędziach grafu G, jest nie mniejsza niż x, więc jej faktyczna długość jest nie mniejsza niż długość przekształconej w opisany powyżej sposób ścieżki P.

Dowód struktury ścieżki P'' przeprowadzimy w dwóch nietrudnych krokach. Przede wszystkim, P'' na pewno nie zawiera dwóch krawędzi z G występujących pod rząd. Faktycznie, gdyby pewne dwie krawędzie u_iu_{i+1}, $u_{i+1}u_{i+2}$ należały do E, wówczas albo $u_iu_{i+2} \in E$, albo $u_iu_{i+2} \notin E$, więc $u_iu_{i+2} \in E'$. W każdym z przypadków opłacałoby się nam zastąpić obie te krawędzie, o łącznej wadze $2a$, krawędzią u_iu_{i+2} o wadze a lub o wadze b.

Jeśli teraz P'' nie zawiera krawędzi pochodzących z G, to nie mamy już czego dowodzić. Załóżmy więc, że P'' zawiera krawędź z G i nie jest to ostatnia krawędź ścieżki. Niech $u_iu_{i+1} \in E$ będzie pierwszą taką krawędzią, mamy $u_{i+1}u_{i+2} \in E'$. Istnieje wówczas wierzchołek $w \in V$, taki że $u_{i+1}w, wu_{i+2} \in E$. Zastąpmy rozważaną parę krawędzi z P'' parą krawędzi u_iw, wu_{i+2}. Podobnie jak poprzednio, mamy $u_iw \in E'$ albo $u_iw \in E$. W pierwszym przypadku długość P'' nie zmieniła się, ale krawędź pochodząca z G znalazła się dalej na ścieżce. W drugim zaś przypadku również parę krawędzi u_iw, wu_{i+2} możemy zastąpić jedną krawędzią u_iu_{i+2} należącą do E albo do E', co powoduje skrócenie ścieżki P''. Widzimy zatem, że żadna z tych sytuacji nie była możliwa, i krawędź z E może wystąpić jedynie na końcu ścieżki P''. ■

Algorytm

Aby dokończyć rozwiązanie, wystarczy dla każdego wierzchołka $v \in V$ obliczyć długość najkrótszej ścieżki z k do v w grafie G oraz w grafie G'. Najkrótsze ścieżki w grafie G obliczamy wspomnianym już algorytmem BFS, jedyna kwestia to najkrótsze ścieżki w grafie G'.

Zauważmy, że gdyby w zadaniu należało znaleźć najkrótsze ścieżki w grafie $G^2 = (V, E^2)$, to rozwiązanie również byłoby proste. Mógłby to być algorytm BFS zastosowany dla grafu G, w którym dla każdego wierzchołka pamiętalibyśmy długości dwóch najkrótszych ścieżek z k: ścieżki o długości parzystej oraz ścieżki o długości nieparzystej (można by też wykonać przeszukiwanie BFS w nieznacznie przekształconym grafie G, co opisano w opracowaniu zadania *Morskie opowieści* w tej książeczce).

Naszym grafem jest jednak $G' = (V, E^2 \setminus E)$, co istotnie utrudnia sprawę. Algorytm w tym przypadku będzie podobny, lecz nieco bardziej subtelny.

W naszym przeszukiwaniu BFS będą występować dwa rodzaje stanów: (v) oraz (v, u). Pierwszy typ stanu reprezentuje wierzchołek v, do którego możemy dotrzeć z k za pomocą sekwencji krawędzi z grafu G'. Drugi typ stanu wskazuje na wierzchołek v, taki że istnieje wierzchołek u, do którego możemy dotrzeć z k za pomocą sekwencji krawędzi z grafu G', oraz w grafie G istnieje krawędź z u do v. Z pierwszego rodzaju stanów przechodzimy do drugiego rodzaju i na odwrót. Do przechodzenia grafu będziemy wykorzystywali tylko krawędzie ze zbioru E.

Stanem początkowym jest stan (k). Będąc w stanie postaci (v), rozważamy wszystkie krawędzie $vw \in E$ i odwiedzamy stany (w, v). Sumarycznie, dla wszystkich stanów tego rodzaju zajmie to czas $O(m)$.

Będąc w stanie postaci (v, u), rozważamy wszystkie krawędzie $vw \in E$ i dla każdej z nich:

(1) Jeśli istnieje krawędź uw, to ignorujemy krawędź vw (tj. nie możemy nią pójść).

(2) Jeśli nie istnieje krawędź uw, to idziemy krawędzią vw do stanu (w) i **usuwamy** skierowaną krawędź vw (ale tylko ze zbioru krawędzi rozpatrywanych przy stanach drugiego rodzaju!). Możemy to zrobić, bo wykonujemy przeszukiwanie wszerz, czyli w momencie przejścia do stanu (w) mamy już obliczoną długość najkrótszej ścieżki z (k) do (w), tak więc ponowne przechodzenie krawędzią vw tego wyniku nie poprawi (a moglibyśmy ją próbować ponownie odwiedzać dla innych stanów (v, u'), $u' \neq u$).

Ponieważ w przypadku (2) zawsze usuwamy jakąś krawędź z grafu w jednym z kierunków, więc znajdziemy się w nim łącznie co najwyżej $2m$ razy. Mogłoby się jednak wydawać, że z powodu przypadku (1) koszt algorytmu jest zbyt duży. Dokładniejsze oszacowanie pokazuje, że tak nie jest. Ustalmy wierzchołek v i niech $deg(v)$ oznacza jego stopień, czyli liczbę krawędzi z nim incydentnych. Wówczas liczba sytuacji, w których znajdziemy się w przypadku (1) dla ustalonego v, z jednej strony nie przekracza liczby wszystkich krawędzi $uw \in E$, czyli m, a z drugiej strony – liczby par krawędzi $uv, vw \in E$, czyli $deg(v)^2$. Sumaryczny koszt przypadku (1) szacuje się zatem przez:

$$\sum_{v \in V} \min(deg(v)^2, m) \leqslant \sum_{v \in V} \sqrt{deg(v)^2 \cdot m} = \sum_{v \in V} deg(v)\sqrt{m} = O(m\sqrt{m}).$$

W tym oszacowaniu skorzystaliśmy z nierówności między minimum z dwóch liczb a ich średnią geometryczną.

Podsumowując, złożoność czasowa całego rozwiązania to $O(m\sqrt{m})$, a złożoność pamięciowa jest liniowa. Stany drugiego rodzaju możemy przechowywać w tablicy z haszowaniem. Implementacje rozwiązania wzorcowego można znaleźć w plikach `cen.cpp`, `cen1.pas` i `cen2.cpp`. Natomiast w plikach `cens2.cpp` i `cens3.pas` znajdują się rozwiązania wolniejsze, w których nie usuwamy wykorzystanych krawędzi w kroku (2). Tego typu rozwiązania uzyskiwały na zawodach ok. 50% punktów (złożoność czasową takich rozwiązań można oszacować jako $O(\sum_{v \in V} deg(v)^2) = O(\sum_{v \in V} n \cdot deg(v)) = O(nm)$).

Testy

Testy są podzielone na pięć grup.

Testy z grupy a zawierają przypadek $b < a$. W testach tych najkrótsza ścieżka złożona z połączeń kolejowych ze źródła do pewnego wierzchołka ma długość nieparzystą, natomiast w optymalnym rozwiązaniu korzystamy jedynie z połączeń lotniczych. Testy te składają się z cyklu nieparzystej długości zawierającego źródło, do którego doczepiona jest duża gwiazda.

Testy z grupy b zawierają przypadek $a < b < 2a$. Wyglądają one podobnie jak testy z grupy a.

W testach z grupy c występuje dużo sytuacji, w których istnieją wszystkie krawędzie uv, vw, uw. Są szczególnie niewygodne dla rozwiązań rozważających graf E^2 zamiast $E^2 \setminus E$.

Testy z grupy d mają na celu odsiać rozwiązania, w których pesymistyczny czas działania algorytmu BFS lub Dijkstry, z powodu błędu w oznaczaniu wierzchołków, jest wykładniczy. Liczba możliwych ścieżek prowadzących ze źródła do poszczególnych wierzchołków jest wykładnicza.

Testy z grupy e są strukturalnie podobne do testów z grup a, b, ale parametry generatora zostały dobrane w celu uwypuklenia różnic czasowych programów (w szczególności, odsiewają programy nieusuwające krawędzi).

Jakub Wojtaszczyk
Treść zadania, Opracowanie

Adam Polak
Program

Dostępna pamięć: 128 MB. *OI, etap I, 15.10–12.11.2012*

Gobeliny

W Bajtockim Muzeum Sztuk Pięknych rozpoczyna się wystawa gobelinów. Główna sala wystawowa ma, patrząc z góry, kształt wielokąta (niekoniecznie wypukłego). Na każdej ze ścian sali powieszono jeden gobelin. Każdy gobelin zajmuje dokładnie całą powierzchnię ściany.

Do sali wstawiono lampę, która ma oświetlać wystawę. Lampa świeci równomiernie we wszystkich kierunkach. Wiadomo, że niektóre z gobelinów muszą być dobrze oświetlone, a niektórych – wręcz przeciwnie – nie można wystawiać na ostre światło.

Bajtazar, kustosz muzeum, zaczął przesuwać lampę po sali, ale nie udało mu się jej ustawić tak, żeby był zadowolony. Bajtazar jest przerażony – obawia się, że będzie trzeba przewieszać gobeliny (co wymaga dużo pracy), a do otwarcia wystawy zostało bardzo niewiele czasu. Może zdołasz mu pomóc i podpowiesz, czy jego wysiłki w ogóle mają sens?

Twoim zadaniem jest rozstrzygnięcie, czy istnieje położenie lampy spełniające poniższe warunki:

- *każda ściana musi być albo oświetlona w całości, albo zaciemniona w całości, w zależności od wymagań dotyczących danego gobelinu; natomiast nie może istnieć ściana, na której część pada światło, a na część nie;*
- *jeżeli lampa znajduje się dokładnie w linii ściany, to jej nie oświetla;*
- *lampy nie można wyłączyć ani zabrać z sali; musi być włączona i znajdować się wewnątrz sali (w szczególności nie może znajdować się na żadnej ze ścian ani w rogu sali).*

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita t ($1 \leqslant t \leqslant 20$), oznaczająca liczbę zestawów danych. W kolejnych wierszach znajdują się opisy zestawów danych.

W pierwszym wierszu opisu znajduje się jedna liczba całkowita n ($3 \leqslant n \leqslant 1000$), oznaczająca liczbę ścian sali wystawowej. W kolejnych n wierszach opisany jest kształt sali. W każdym z tych wierszy znajdują się dwie liczby całkowite x_i i y_i oddzielone pojedynczym odstępem ($-30\,000 \leqslant x_i, y_i \leqslant 30\,000$ dla $i = 1, 2, \ldots, n$) – są to współrzędne narożnika sali, czyli wierzchołka wielokąta opisującego kształt sali. Wierzchołki są podane w kolejności zgodnej z kierunkiem ruchu wskazówek zegara.

W kolejnych n wierszach opisane są wymagania dotyczące gobelinów wiszących na ścianach. W każdym z tych wierszy znajduje się jedna litera `S` *lub* `C`*, oznaczająca odpowiednio, że ściana ma być oświetlona lub zaciemniona. Litera znajdująca się w i-tym z tych wierszy (dla $1 \leqslant i \leqslant n-1$) dotyczy ściany łączącej wierzchołki i-ty oraz $(i+1)$-szy. Litera znajdująca się w ostatnim z tych wierszy dotyczy ściany łączącej ostatni wierzchołek z pierwszym.*

Wielokąt opisujący kształt sali nie ma samoprzecięć, tzn. poza sąsiednimi bokami, które siłą rzeczy mają wspólny koniec, żadne dwa boki wielokąta nie mają punktów wspólnych. Żadne trzy wierzchołki wielokąta nie są współliniowe.

W testach wartych łącznie 40% punktów zachodzi dodatkowy warunek $n \leqslant 20$. Dodatkowo, w testach wartych łącznie 10% punktów wszystkie ściany mają być oświetlone.

Wyjście

Dla każdego zestawu danych Twój program powinien wypisać na standardowe wyjście jeden wiersz zawierający jedno słowo: `TAK` *– jeżeli da się ustawić lampę zgodnie z podanymi warunkami, lub* `NIE` *– w przeciwnym przypadku.*

Przykład

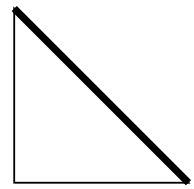
Na rysunkach pogrubione boki oznaczają ściany, które mają być zaciemnione, a pozostałe boki – ściany, które mają być oświetlone. Rysunek do drugiego przykładu przedstawia poprawne położenie lampy.

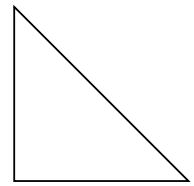
Dla danych wejściowych:

```
2
3
0 0
0 1
1 0
S
C
S
3
0 0
0 1
1 0
S
S
S
```

poprawnym wynikiem jest:

```
NIE
TAK
```



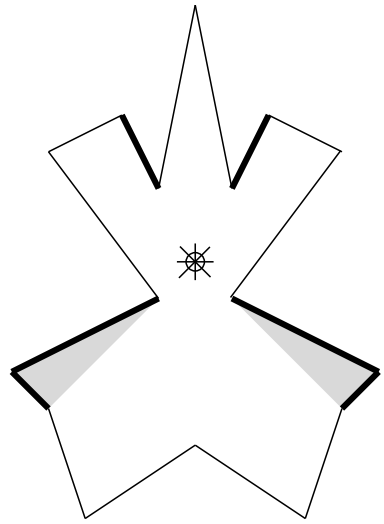


natomiast dla danych:

```
1
16
5 -3
4 -4
3 -7
0 -5
-3 -7
-4 -4
-5 -3
-1 -1
-4 3
-2 4
-1 2
0 7
1 2
2 4
4 3
1 -1
C
S
S
S
S
C
C
S
S
C
S
S
C
S
S
C
```

poprawnym wynikiem jest:

```
TAK
```



Rozwiązanie

Jasne ściany

Zadanie jest niełatwe, więc zacznijmy od rozpatrzenia jego istotnie łatwiejszej wersji – spróbujmy wpierw rozważyć przypadek, w którym kustosz chciałby, żeby wszystkie ściany były oświetlone.

Potrzebujemy zatem znaleźć punkt wewnątrz wielokąta, z którego będzie widać wszystkie ściany. Prostym warunkiem, który taki punkt musi spełniać, jest to, że musi znajdować się „po wewnętrznej stronie" każdej ze ścian. By wyrazić to dokładnej,

rozważmy dowolną ścianę wielokąta i przeprowadźmy przez nią linię prostą. W pobliżu ściany, po jednej stronie tej prostej znajduje się wnętrze muzeum, a po drugiej – zewnętrze; półpłaszczyznę po stronie wnętrza nazwiemy „wewnętrzną". Nasz warunek mówi, że dla każdej ściany lampa musi znajdować się po wewnętrznej stronie prostej zawierającej tę ścianę (nie może też znajdować się na samej prostej). Łatwo przekonać się, że jest to warunek konieczny.

Zaczniemy od pokazania, że ten warunek jest wystarczający, by lampa oświetliła wszystkie ściany, a potem zastanowimy się, jak ten warunek sprawdzić. Załóżmy, że lampa jest położona po wewnętrznej stronie każdej ściany, ale pewna ściana nie jest całkowicie oświetlona. Oznacza to, że promień światła idący od lampy do pewnego punktu tej ściany przecina zewnętrze muzeum. Jako że tuż przy ścianie promień znajduje się we wnętrzu muzeum, to gdzieś musi przechodzić z zewnętrza na wnętrze – a to oznacza, że przecina pewną ścianę „od zewnątrz", co nie może się zdarzyć (bo wtedy lampa leżałaby po zewnętrznej stronie tej ściany). Doszliśmy do sprzeczności, zatem każda ściana jest oświetlona. Ten sam argument pokazuje, że lampa znajduje się wewnątrz wielokąta.

Warto wiedzieć, że wielokąt, w którym da się z jednego punktu oświetlić wszystkie ściany, nazywa się *wielokątem gwiaździstym*, zaś zbiór punktów, z których da się wielokąt oświetlić (czyli przecięcie opisanych wyżej półpłaszczyzn), nazywa się jego *jądrem*.

Jasne ściany – algorytm

Jak jednak sprawdzić, czy wielokąt jest gwiaździsty? Najprościej będzie spróbować wyznaczyć jego jądro. Istnieje szereg algorytmów (opierających się na przecinaniu półpłaszczyzn) pozwalających wyznaczyć jądro (a zatem, w szczególności, stwierdzić, czy jest niepuste) w czasie $O(n \log n)$, mają one jednak tę wadę, że są nietrywialne w implementacji – a my przecież rozpatrujemy dopiero prosty przypadek naszego zadania. Na pomoc przyjdzie nam ograniczenie na liczbę boków naszego wielokąta (a zatem na liczbę półpłaszczyzn, które przecinamy) – jest ich co najwyżej 1000, a zatem stać nas na rozwiązanie w czasie $O(n^2)$.

Takie rozwiązanie może wyglądać następująco. Zauważmy, że jeśli jądro jest niepuste, to jest wielokątem wypukłym (bo jest przecięciem półpłaszczyzn, które same są wypukłe), a jego boki są zawarte w prostych wyznaczających półpłaszczyzny (czyli w prostych zawierających boki naszego wielokąta). Zatem możemy spróbować znaleźć bok jądra – bierzemy kolejno każdą z prostych i patrzymy na jej przecięcia z wszystkimi pozostałymi półpłaszczyznami. Te przecięcia są oczywiście półprostymi (ew. prostymi lub zbiorami pustymi, dla półpłaszczyzn wyznaczanych przez proste równoległe do naszej), a więc bardzo łatwo sprawdzić, czy ich przecięcie zawiera jakiś odcinek. Jeśli nie, to na tej prostej nie ma boku jądra, a jeśli tak – to znaleźliśmy bok (a zatem jądro jest niepuste). Nieprzyjemną sytuacją do rozważenia byłoby, gdyby dwie półpłaszczyzny były wyznaczone przez tę samą prostą, szczęśliwie w treści zadania mamy zagwarantowane, że żadne trzy wierzchołki wielokąta nie są współliniowe, co wyklucza ten przypadek.

Rozwiązania, które rozpatrywały tylko jasne ściany, uzyskiwały 10% punktów za to zadanie. Przykładowa implementacja znajduje się w pliku `gobb1.cpp`.

Ciemne ściany

Zacznijmy od zauważenia, że jeśli wszystkie ściany mają być ciemne, to lampy ustawić się nie da – światło gdzieś musi się podziać.

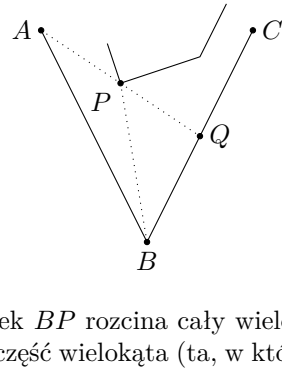
Byłoby pięknie, gdyby zbiór możliwych ustawień lampy wyznaczany przez ciemną ścianę też był półpłaszczyzną. Niestety, nie jest – lampa może ściany nie oświetlać dlatego, że znajduje się „za nią" (czyli w „zewnętrznej" półpłaszczyźnie), ale może też być zasłonięta przez jakąś inną ścianę; przykłady obydwu tych sytuacji widzimy w teście przykładowym.

Przyjrzenie się testowi przykładowemu może nam nasunąć jednak pewne przemyślenie. Otóż jeśli mamy obok siebie dwie sąsiadujące ciemne ściany, które tworzą kąt wypukły (czyli w punkcie ich styku jest skręt do środka wielokąta), to – jak widać na przykładowym obrazku – zacieniony jest cały trójkąt, którego dwoma bokami są te dwie ściany. Jest to prawdą przynajmniej wtedy, gdy w tym trójkącie nie ma żadnej innej ściany – faktycznie, gdyby docierał tam jakikolwiek promień światła, to musiałby skądś przyjść i gdzieś wyjść – a prosta nie może dwa razy przecinać tego samego boku trójkąta. Zatem, w tej szczególnej sytuacji, dwie zacienione ściany możemy zastąpić jedną, będącą trzecim bokiem rozważanego trójkąta, i ta ściana również musi być zacieniona.

To spostrzeżenie może budzić naszą nadzieję, że coś da się z zadaniem zrobić, albowiem pozwoliło nam zredukować zadanie do mniejszego – takiego, w którym mamy o jedną ciemną ścianę mniej. Spróbujmy sobie zatem poradzić z podobnymi przypadkami – gdy w trójkącie wyznaczonym przez dwie ciemne ściany znajduje się jakaś inna ściana oraz gdy kąt między dwiema ciemnymi ścianami jest wklęsły.

Dwie ciemne ściany i inne ściany w środku

Załóżmy, że w środku trójkąta wyznaczonego przez dwie ciemne ściany AB i BC znajduje się jakaś inna ściana (a zatem też jakiś wierzchołek naszego wielokąta; patrz rysunek). Wybierzmy taki wierzchołek P spośród znajdujących się we wnętrzu ABC, żeby kąt BAP był jak najmniejszy, i poprowadźmy prostą AP do przecięcia z BC w punkcie Q. Teraz odcinki AB oraz BQ muszą być nieoświetlone oraz we wnętrzu trójkąta ABQ nie leży już żaden punkt wielokąta – a zatem, podobnie jak poprzednio, do trójkąta ABQ nie dociera światło, a w szczególności odcinek BP musi być nieoświetlony.



Zauważmy teraz, że odcinek BP rozcina cały wielokąt na dwie części. Skoro jest nieoświetlony, to i cała jedna część wielokąta (ta, w której nie będzie lampy) musi być

nieoświetlona, bo każda droga od lampy do drugiej części poprowadzona wewnątrz wielokąta musiałaby przeciąć BP. Zatem albo wszystkie ściany od B do P po stronie zawierającej A, albo też po stronie zawierającej C, muszą być nieoświetlone; a stąd całą jedną część wielokąta będziemy mogli zastąpić ścianą BP.

Zauważmy, że powyższa operacja zastąpienia zmniejsza liczbę ścian w wielokącie co najmniej o jedną, a zatem operacji takich wykonamy co najwyżej n. Wobec tego możemy każdą z nich wykonywać w czasie $O(n)$ i nadal mieścić się w czasie. Zatem sprawdzenie, czy któryś z wierzchołków wielokąta leży we wnętrzu trójkąta ABC, wybranie punktu P, i wreszcie sprawdzenie, którą połowę wielokąta możemy usunąć, możemy zrealizować, po prostu przeglądając wszystkie możliwości.

Dwie ciemne ściany i kąt wklęsły

By osiągnąć sytuację, w której każda ciemna ściana sąsiaduje wyłącznie ze ścianami jasnymi, musimy jeszcze rozważyć przypadek, w którym dwie sąsiadujące ciemne ściany tworzą kąt wklęsły. Zakładamy przy tym, że pomiędzy żadnymi dwiema ciemnymi ścianami nie ma już kąta wypukłego, a zatem mamy pewien ciąg ścian ciemnych z kątami wklęsłymi pomiędzy każdą parą sąsiadów, a następnie po obydwu stronach tego ciągu ściany jasne. Pokażemy, że takiej sytuacji nie da się osiągnąć żadną pozycją lampy.

Niech A i B będą punktami styku ciągu ciemnych ścian ze ścianami jasnymi. Punkty tuż przy A i B są oświetlone, a zatem promienie świetlne poprowadzone z lampy do A i B idą we wnętrzu lub wzdłuż brzegów wielokąta. Powiedzmy, że lampa stoi w jakimś punkcie L. Punkt L nie może leżeć w jednej linii z A oraz B, bo wtedy – jako że pomiędzy A i B ściany ciemne tworzą kąt wklęsły – promień światła wychodziłby na zewnątrz wielokąta. Wiemy też, że skoro AL i BL nie przecinają boków wielokąta (mogą iść wzdłuż nich), to promień światła wypuszczony z L w jakimś kierunku pomiędzy AL i BL musi trafić w jakąś ścianę pomiędzy A i B. Ale ta ściana miała być nieoświetlona!

Czytelnikowi, który nie jest całkiem pewien, czy rozumie wszystkie detale, polecamy jako ćwiczenie zrozumieć, czemu powyższy argument nie działa, jeśli kąt pomiędzy ciemnymi ścianami jest wypukły.

Pojedyncze ciemne ściany

Doszliśmy zatem do sytuacji, w której każda ciemna ściana sąsiaduje po obu stronach ze ścianami jasnymi. Spróbujmy teraz powtórzyć powyższe rozumowanie – niech AB będzie ciemną ścianą, stawiamy lampę w jakimś punkcie L, światło z L dochodzi zarówno dowolnie blisko A, jak i dowolnie blisko B, czyli AL i BL nie opuszczają wielokąta (mogą iść wzdłuż jego brzegów). Gdyby A, B i L nie były współliniowe, to światło wypuszczone z L w jakimś kierunku pomiędzy AL i BL musiałoby zatrzymać się na ścianie AB (nie ma po drodze żadnej innej ściany wielokąta, bo żadna ściana nie przecina żadnego z trzech boków trójkąta ABL) – czyli przynajmniej kawałek AB byłby oświetlony. Zatem każda ciemna ściana wyznacza nam prostą, na której musi leżeć lampa.

To oznacza, że osiągnęliśmy sukces – po naszych redukcjach zarówno ściany jasne, jak i ściany ciemne wyznaczają łatwo wyrażalny warunek na położenie lampy (w przypadku ściany jasnej, lampa musi leżeć w półpłaszczyźnie wewnętrznej dla tej ściany, w przypadku ściany ciemnej – na prostej zawierającej tę ścianę). W szczególności, zauważmy, że oznacza to, że wielokąt otrzymany po redukcji musi już być gwiaździsty – a zatem spełnienie powyższych warunków będzie też wystarczające, by wszystkie jasne ściany były oświetlone.

Jeśli ścian ciemnych jest po redukcji więcej niż jedna, to przecinamy proste wyznaczone przez dowolne dwie z nich (są to różne proste, bo żadne trzy wierzchołki nie są współliniowe) i otrzymujemy dokładnie jednego kandydata na pozycję lampy, którego poprawność łatwo sprawdzić. W przypadku, gdy jest tylko jedna ściana ciemna, możemy łatwo sprawdzić, czy na prostej zawierającej tę ścianę możemy postawić lampę podobnie, jak robiliśmy to w przypadku samych ścian jasnych – patrzymy na półproste, które są przecięciami naszej prostej z półpłaszczyznami wyznaczonymi przez wszystkie pozostałe ściany (trzeba też pamiętać, że lampa nie może leżeć *na* ścianie, a zatem na koniec musimy sprawdzić, czy otrzymany odcinek możliwych pozycji lampy nie jest zawarty w ciemnej ścianie). Natomiast przypadek, gdy ścian ciemnych w ogóle nie ma, rozpatrzyliśmy już wcześniej.

Ostatecznie otrzymaliśmy rozwiązanie wzorcowe o złożoności czasowej $O(n^2)$. Można je znaleźć w plikach `gob.cpp`, `gob1.cpp` i `gob2.pas`.

Szybsze rozwiązanie

Czytelnikowi, który ma ochotę na odrobinę większe wyzwanie, można zaproponować próbę rozwiązania tego zadania w czasie $O(n \log n)$. Jak już wspomnieliśmy, sprawdzenie, czy wielokąt jest gwiaździsty, można przeprowadzić algorytmem przecięcia półpłaszczyzn. Pozostaje zatem problem redukcji ciemnych ścian, a konkretnie dwa problemy – sprawdzenie, czy we wnętrzu trójkąta znajduje się jakiś wierzchołek wielokąta, oraz sprawdzenie, który fragment wielokąta można usunąć. Istnieje kilka sposobów rozwiązania obydwu tych problemów, przedstawimy tu szkic jednego z nich.

Zauważmy, że ostatecznie naszym celem jest zawsze zredukowanie ciągu ścian ciemnych do pojedynczej ściany. Niech zatem A i B będą krańcami takiego ciągu. Jeśli ciąg ścian ciemnych „wystaje" poza odcinek AB (czego szczególnym przypadkiem jest ciąg ścian o kątach wklęsłych), to podobne rozumowanie, co w przypadku ciągu kątów wklęsłych, pokaże, że lampy ustawić się nie da. W przeciwnym razie, rozumowanie podobne do przypadku pojedynczej ciemnej ściany doprowadzi nas do wniosku, że lampa musi leżeć gdzieś na prostej AB, poza odcinkiem AB. Jeśli istnieje więcej niż jeden ciąg ścian ciemnych, to otrzymujemy dwie takie proste, i tylko jedno możliwe położenie lampy – a zatem będziemy mogli ograniczyć się do sprawdzenia poprawności takiego położenia (szczegóły takiego sprawdzenia w przypadku wielokąta niekoniecznie gwiaździstego pozostawimy niedopowiedziane). W przypadku, w którym ciąg do uproszczenia jest tylko jeden, możemy liniowo wyszukać punkt P jak powyżej (stać nas na to, gdyż zrobimy to tylko raz). Skoro punkt P jest krańcem ściany spoza naszego ciągu, to jest on oświetlony, tak więc zastępując jedną z części wielokąta przez ścianę AP lub BP, znajdziemy się w znanym już przypadku pojedynczej ciemnej ściany.

Testy

Przygotowano 10 plików testowych, każdy zawierający od kilku do kilkunastu testów. Małe testy (od *1* do *4*, w których $n \leqslant 20$) zostały wygenerowane ręcznie. Duże testy (od *5* do *10*) zostały wygenerowane dwuetapowo. Najpierw automatycznie wygenerowano wielokąty gwiaździste, a potem „popsuto" je ręcznie przy pomocy edytora graficznego. Ostatni test (*10*) zawiera wyłącznie ściany oświetlone.

Jakub Radoszewski, Wojciech Rytter
Treść zadania, Opracowanie

Przemysław Horban
Program

Dostępna pamięć: 256 MB.

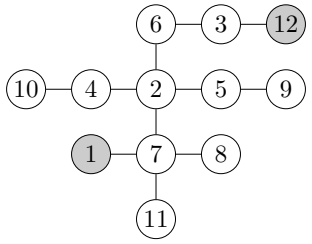
OI, etap I, 15.10–12.11.2012

Multidrink

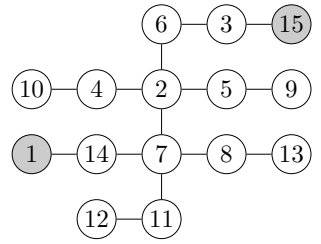
Bajtazar mieszka w Bajtowie, które słynie z tego, że przy każdym skrzyżowaniu ulic znajduje się bar mleczny. Pewnego dnia Bajtazar postanowił odwiedzić w celu tzw. „mlecznego multidrinka" wszystkie bary, każdy dokładnie raz. Bajtazar chciałby tak zaplanować trasę, żeby każdy kolejny bar był nie dalej niż dwa skrzyżowania od poprzedniego.

Skrzyżowania w Bajtowie są ponumerowane od 1 do n. Wszystkie ulice są dwukierunkowe. Między każdymi dwoma różnymi skrzyżowaniami jest tylko jedna trasa, na której żadne skrzyżowanie nie powtarza się. Bajtazar startuje przy skrzyżowaniu numer 1 i kończy przy skrzyżowaniu numer n.

Twoim zadaniem jest wyznaczenie jakiejkolwiek trasy spełniającej wymagania Bajtazara lub wypisanie informacji o braku takiej trasy.



Trasą spełniającą warunki zadania jest na przykład: $1, 11, 8, 7, 5, 9, 2, 10, 4, 6, 3, 12$.



Nie ma żadnej trasy spełniającej warunki zadania.

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita n ($2 \leqslant n \leqslant 500\,000$) oznaczająca liczbę skrzyżowań w Bajtowie. Każdy z kolejnych $n-1$ wierszy zawiera dwie różne liczby całkowite a_i oraz b_i ($1 \leqslant a_i, b_i \leqslant n$), oddzielone pojedynczym odstępem, reprezentujące ulicę łączącą skrzyżowania o numerach a_i i b_i.

W testach wartych łącznie 65% punktów zachodzi dodatkowy warunek $n \leqslant 5000$.

Wyjście

Jeśli nie istnieje żadna trasa spełniająca wymagania Bajtazara, w pierwszym i jedynym wierszu standardowego wyjścia Twój program powinien wypisać jedno słowo `BRAK`*. W przeciwnym przypadku Twój program powinien wypisać n wierszy, z których i-ty powinien zawierać numer i-tego skrzyżowania na przykładowej trasie spełniającej warunki Bajtazara. W pierwszym wierszu powinna znaleźć się liczba 1, a w n-tym wierszu – liczba n.*

Przykład

Dla danych wejściowych:

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

jednym z poprawnych wyników jest:

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

natomiast dla danych:

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

poprawnym wynikiem jest:

```
BRAK
```

Rozwiązanie

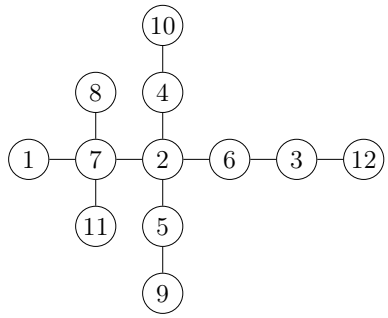
Sieć ulic z treści zadania można opisać jako drzewo, czyli graf o n wierzchołkach oraz $n-1$ krawędziach, spójny i bez cykli. *2-ścieżką* w drzewie nazwiemy ciąg wierzchołków, w którym każde dwa kolejne wierzchołki znajdują się w odległości co najwyżej 2. *2-cyklem* w drzewie nazwiemy 2-ścieżkę zaczynającą się i kończącą w tym samym wierzchołku. Możemy dalej zdefiniować *2H-ścieżkę* (2-ścieżkę Hamiltona) jako 2-ścieżkę przechodzącą przez każdy wierzchołek drzewa dokładnie raz i *2H-cykl* (2-cykl Hamiltona) jako 2-cykl przechodzący przez każdy wierzchołek drzewa dokładnie raz (poza wierzchołkiem początkowym, a zarazem końcowym, cyklu). W naszym zadaniu mamy znaleźć 2H-ścieżkę łączącą dwa zadane wierzchołki drzewa lub stwierdzić, że taka 2H-ścieżka nie istnieje.

Bardzo dawno temu, na II Olimpiadzie Informatycznej [2], pojawiło się zadanie pt. *Obchodzenie drzewa skokami*. W zadaniu tym należało w drzewie znaleźć 3H-cykl,

czyli cykl przechodzący przez każdy wierzchołek drzewa dokładnie raz, w którym każde dwa kolejne wierzchołki są oddalone co najwyżej o 3. Okazywało się wówczas, że każde drzewo zawiera 3H-cykl i istnieje liniowy algorytm, który pozwala wyznaczyć taki 3H-cykl. Nasz przypadek jest jednak bardziej skomplikowany, gdyż nie każde drzewo zawiera 2H-cykl lub nawet 2H-ścieżkę.

Algorytm warstwowy zachłanny

Oznaczmy zbiór wierzchołków drzewa przez $V = \{1, 2, \ldots, n\}$. Niech $1 = u_1$, u_2, …, $u_k = n$ będzie (zwykłą) ścieżką w naszym drzewie łączącą wierzchołki 1 oraz n. Ścieżkę tę będziemy nazywali *ścieżką główną*. Podzielimy wszystkie wierzchołki drzewa na *warstwy* odpowiadające wierzchołkom ścieżki głównej. W i-tej warstwie, oznaczanej jako V_i, znajdą się wszystkie wierzchołki „podłączone do wierzchołka u_i", a więc te wierzchołki, z których ścieżka prowadząca do wierzchołka 1 po raz pierwszy przecina ścieżkę główną właśnie w wierzchołku u_i (patrz rys. 1).



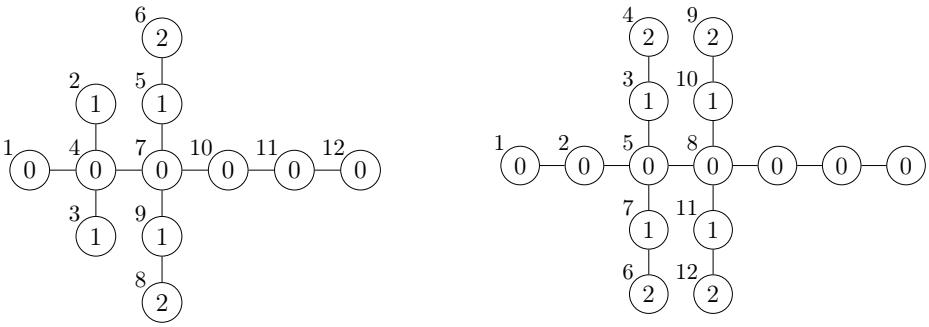
Rys. 1: Inna ilustracja pierwszego przykładu z treści zadania. Ścieżką główną w tym drzewie jest 1, 7, 2, 6, 3, 12, a kolejne warstwy to $\{1\}$, $\{7, 8, 11\}$, $\{2, 4, 5, 9, 10\}$, $\{6\}$, $\{3\}$, $\{12\}$.

Naszą 2H-ścieżkę będziemy konstruowali tak, że najpierw przejdziemy w pewnej kolejności przez wszystkie wierzchołki warstwy V_1, potem przez wszystkie wierzchołki warstwy V_2 itd. Taką 2H-ścieżkę nazwiemy 2H-ścieżką *warstwową*. Poprawność tej decyzji uzasadnia następujący, dość intuicyjny lemat, którego dowód odkładamy na później.

Lemat 1. Jeśli drzewo zawiera 2H-ścieżkę łączącą wierzchołki 1 oraz n, to zawiera także 2H-ścieżkę warstwową łączącą te dwa wierzchołki.

W oparciu o Lemat 1 możemy podać pierwsze rozwiązanie zadania, bazujące na podejściu zachłannym. W rozwiązaniu tym wierzchołki w ramach warstw odwiedzamy zgodnie z priorytetem określonym jako odległość od ścieżki głównej (rys. 2).

```
1:  Algorytm warstwowy zachłanny;
2:  begin
3:    foreach v in V do d(v) := odległość v od ścieżki głównej;
4:    start := 1;
5:    odwiedzaj wierzchołki warstwami, wybierając jako następny wierzchołek
6:      nieodwiedzony wierzchołek o maksymalnym d(v) i stopniu 1, a jeśli takiego
7:      nie ma, dowolny nieodwiedzony wierzchołek o maksymalnym d(v);
8:      { każdy kolejny wierzchołek musi leżeć w odległości 1 lub 2 od bieżącego }
9:    if odwiedziliśmy wszystkie wierzchołki then return true
10:   else return false;
11: end
```



Rys. 2: Priorytety poszczególnych wierzchołków drzew z przykładów w treści zadania (narysowanych warstwowo) oraz 2-ścieżki konstruowane przez algorytm warstwowy zachłanny. W pierwszym przypadku jest to 2H-ścieżka w drzewie, natomiast w drugim przypadku w kroku 12. trafiamy na „pułapkę bez wyjścia" – odwiedzanie kończy się niepowodzeniem, zatem nie ma szukanej 2H-ścieżki.

W implementacji trzeba jeszcze zwrócić uwagę na to, że liczba wierzchołków osiągalnych z jednego wierzchołka w jednym przeskoku może być duża. Aby sobie z tym poradzić, należy scalić każdą grupę liści podłączonych do tego samego wierzchołka w jeden liść. Jeśli po wykonaniu tej operacji jakiś wierzchołek ma stopień większy niż 5, z góry odpowiadamy, że w drzewie nie ma żądanej 2H-ścieżki. Później należy pamiętać, aby w momencie odwiedzania „scalonego" liścia na 2-ścieżce wypisać za jednym zamachem wszystkie liście, z których on powstał.

Okazuje się, że to już całe rozwiązanie. Na dodatek działa ono w czasie liniowym! Jednak nie bardzo widać, dlaczego to rozwiązanie zawsze znajduje 2H-ścieżkę w drzewie, jeśli takowa istnieje. Co więcej, wprowadzone ograniczenie na stopień wierzchołka wygląda co najmniej tajemniczo. Aby wyjaśnić, na jakiej podstawie to rozwiązanie działa, podamy teraz drugie rozwiązanie, oparte na zupełnie innej technice.

Gąsienice i programowanie dynamiczne

Oznaczmy przez $first_i$ pierwszy wierzchołek i-tej warstwy, który odwiedzimy na 2H-ścieżce, a przez $last_i$ – ostatni wierzchołek odwiedzony w tej warstwie. Oczywiście

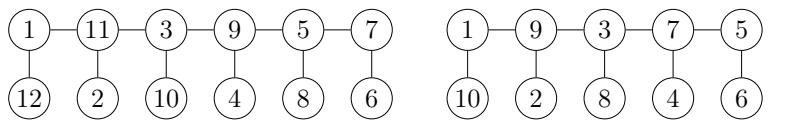
wymagamy, aby $first_1 = 1$ i $last_k = n$. Oznaczmy dalej przez S_i wierzchołek u_i wraz z jego sąsiadami nieleżącymi na ścieżce głównej. Aby dało się przeskoczyć z jednej warstwy bezpośrednio do następnej warstwy, każdy z wierzchołków $last_i$, $first_{i+1}$ musi być albo wierzchołkiem na ścieżce głównej, albo bezpośrednim sąsiadem wierzchołka ze ścieżki głównej, czyli należeć do zbioru S_i lub, odpowiednio, S_{i+1}. W tym rozwiązaniu dla każdego wierzchołka $w \in S_i$ stwierdzimy, czy istnieje 2-ścieżka startująca w wierzchołku 1, przechodząca przez wszystkie wierzchołki warstw $V_1, \ldots, V_i$ i kończąca w wierzchołku $last_i = w$.

Zauważmy, że każde dwa wierzchołki ze zbioru S_i są oddalone co najwyżej o 2. Ponieważ $first_i, last_i \in S_i$, prowadzi nas to do kluczowej obserwacji:

Obserwacja 1. 2H-ścieżka warstwowa ograniczona do jednej warstwy V_i jest 2H-cyklem w tej warstwie.

Powyższa obserwacja wymusza bardzo konkretną postać każdej warstwy. *Gąsienicą* nazwiemy drzewo składające się z jednej ścieżki (tzw. *osi*) oraz z pewnego zbioru wierzchołków (tzw. *odnóg*) „podczepionych" bezpośrednio do wewnętrznych wierzchołków osi. Z danego wierzchołka może wychodzić wiele odnóg, może też nie być do niego podczepiona żadna odnoga. Aby oś gąsienicy była określona jednoznacznie, przyjmujemy, że do końcowych wierzchołków gąsienicy muszą być podłączone jakieś odnogi (poza szczególnym przypadkiem, gdy gąsienica ma tylko jeden wierzchołek).

Jedynymi drzewami posiadającymi 2H-cykl są właśnie gąsienice. Metodę znajdowania 2H-cyklu w gąsienicy przedstawiono na rys. 3. W tym algorytmie, jeżeli jakiejś odnogi nie ma, to omijamy ją bezpośrednim przeskokiem do kolejnego wierzchołka na osi, a jeśli z jednego wierzchołka odchodzi wiele odnóg, odwiedzamy je wszystkie za jednym zamachem. Okazuje się, że jest to jedyny sposób znalezienia 2H-cyklu w gąsienicy. Dowód poniższego lematu również odkładamy na później.



Rys. 3: 2H-cykl w gąsienicy w zależności od parzystości długości osi.

Lemat 2. Drzewo posiada 2H-cykl wtedy i tylko wtedy, gdy jest gąsienicą. Co więcej, każdy 2H-cykl w gąsienicy jest postaci takiej jak na rys. 3.

W ten sposób otrzymujemy następujące rozwiązanie bazujące na metodzie programowania dynamicznego:

1. Wyznacz podział drzewa na warstwy $V_1, \ldots, V_k$.
2. Jeśli któraś z warstw nie jest gąsienicą, wypisz `BRAK`.
3. Dla każdych wierzchołków $v, w \in S_i$ sprawdź, czy gąsienica V_i zawiera 2H-cykl, w którym v i w sąsiadują.

4. Dla każdego $i = 1, \ldots, k$ oraz $w \in S_i$ wyznacz wartość logiczną $Last[w]$:
   (a) Dla każdego $w \in S_1$, $Last[w]$ jest prawdą wtedy i tylko wtedy, gdy V_1 zawiera 2H-cykl łączący 1 oraz w.
   (b) Dla każdego $i > 1$ i $w \in S_i$, $Last[w]$ jest prawdą wtedy i tylko wtedy, gdy istnieją wierzchołki $v \in S_i$ oraz $w' \in S_{i-1}$, takie że $Last[w']$ jest prawdą, wierzchołek v jest osiągalny z w' oraz V_i zawiera 2H-cykl łączący v oraz w.
5. Jeśli $Last[n]$ jest prawdą, odtwórz 2H-ścieżkę, cofając się po wartościach $Last$. W przeciwnym wypadku wypisz `BRAK`.

W powyższym rozwiązaniu $Last[w]$ dla $w \in S_i$ jest prawdą wtedy i tylko wtedy, gdy istnieje 2-ścieżka warstwowa startująca w wierzchołku 1, przechodząca przez wszystkie wierzchołki warstw $V_1, \ldots, V_i$ i kończąca w wierzchołku $last_i = w$. Czyli jest to dokładnie to, o co nam chodziło!

Rozwiązanie miałoby złożoność liniową, gdyby nie fakt, że zbiory S_i mogą być duże. Jednak w zbiorze S_i mogą znajdować się co najwyżej trzy wierzchołki niebędące liśćmi drzewa – w przeciwnym razie i-ta warstwa nie byłaby gąsienicą. Zauważmy, że wszystkie liście w zbiorze S_i są dla nas takie same. Stąd w powyższym algorytmie wystarczy w zbiorze S_i pozostawić maksymalnie dwa liście (jako kandydatów na wierzchołki v i w). W ten sposób S_i nie będzie miał nigdy więcej niż pięć elementów. Dzięki temu w każdej warstwie będziemy poszukiwać 2H-cyklu tylko stałą liczbę razy. Ponadto obliczenie $Last[w]$ będzie wymagało sprawdzenia $Last[w']$ tylko dla stałej liczby wierzchołków w'.

Ostatecznie całe rozwiązanie działa w czasie liniowym. Implementację rozwiązania opartego na programowaniu dynamicznym można znaleźć w pliku `mul.cpp`, natomiast implementację algorytmu warstwowego zachłannego – w pliku `mul2.pas`. Poniżej uzupełniamy brakujące dowody lematów.

Dowód lematu 1

Jeśli P jest 2H-ścieżką z 1 do n w drzewie, to przez $ind(P)$ oznaczymy najmniejsze $i \in \{1, \ldots, k-1\}$, takie że istnieje wierzchołek z warstwy V_i leżący później na P niż jakiś wierzchołek z dalszej warstwy (tj. warstwy V_j dla $j > i$). Jeśli taki indeks i nie istnieje, przyjmujemy $ind(P) = k$. Na mocy założeń lematu wiemy, że istnieje 2H-ścieżka prowadząca z 1 do n. Niech P oznacza 2H-ścieżkę maksymalizującą wartość $ind(P)$. Wykażemy, że P jest 2H-ścieżką warstwową.

Oczywiście jeśli $ind(P) = k$ to P jest warstwowa. Przyjmijmy zatem, że $ind(P) = i < k$. Udowodnimy, że prowadzi to do sprzeczności. Podzielmy P na fragmenty $X_1, X_2, X_3, \ldots, X_l$ w taki sposób, żeby każdy wierzchołek fragmentu typu X_{2p-1} (dla $p = 1, 2, \ldots$) należał do zbioru $\mathcal{L} = \bigcup_{j \leqslant i} V_j$, a każdy wierzchołek fragmentu typu X_{2p} (dla $p = 1, 2, \ldots$) należał do zbioru $\mathcal{R} = \bigcup_{j > i} V_j$. Na mocy założenia mamy, że $l \geqslant 4$.

Oznaczmy przez $N(v)$ wierzchołek v wraz z jego sąsiadami. Zauważmy, że

$$first(X_{2p-1}),\ last(X_{2p-1}) \quad \in \quad \{u_1\} \cup N(u_i)$$

$$\mathit{first}(X_{2p}),\ \mathit{last}(X_{2p}) \quad \in \quad \{u_k\} \cup N(u_{i+1}).$$

Ponadto, dla każdego $j = 1, \ldots, l-1$ musi zachodzić $\mathit{last}(X_j) \in \{u_i, u_{i+1}\}$ lub $\mathit{first}(X_{j+1}) \in \{u_i, u_{i+1}\}$.

Po chwili namysłu można dojść do wniosku, że jedyną możliwością spełnienia powyższych warunków jest sytuacja, gdy $l = 4$ oraz:

$$\mathit{last}(X_1) \in N(u_i) \setminus \{u_i\}, \quad \mathit{first}(X_2) = u_{i+1}, \quad \mathit{last}(X_2) \in N(u_{i+1}),$$

$$\mathit{first}(X_3) \in N(u_i), \quad \mathit{last}(X_3) = u_i, \quad \mathit{first}(X_4) \in N(u_{i+1}) \setminus \{u_{i+1}\}.$$

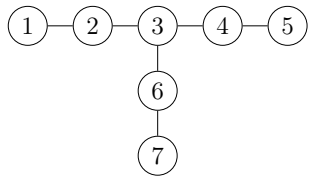
Sprawdźmy, że tak rzeczywiście jest. Ponieważ każda para ścieżek X_{2p-1}, X_{2p} zawiera jakiś wierzchołek spośród $\{u_i, u_{i+1}\}$, a l jest parzyste, więc l nie może być większe niż 4. Stąd $l = 4$.

Załóżmy teraz, że zachodziłoby $\mathit{last}(X_1) = u_i$. Wówczas $\mathit{last}(X_3) \neq u_i$, a zatem $\mathit{first}(X_4) = u_{i+1}$. Ponadto $\mathit{first}(X_3) \neq u_i$, więc także $\mathit{last}(X_2) = u_{i+1}$, co prowadzi do sprzeczności. W ten sposób wykazaliśmy, że $\mathit{last}(X_1) \neq u_i$, skąd otrzymujemy, że $\mathit{last}(X_1) \in N(u_i) \setminus \{u_i\}$ oraz $\mathit{first}(X_2) = u_{i+1}$. Musi dalej zachodzić $\mathit{last}(X_3) = u_i$, gdyż w przeciwnym razie mielibyśmy $\mathit{first}(X_4) = u_{i+1}$, co nie jest możliwe. W ten sposób otrzymujemy, że rzeczywiście P ma taką strukturę, jak opisana powyżej. (Można zauważyć, że struktura ta na styku X_2 i X_3 wymusza, że $X_2 = \{u_{i+1}\}$ lub $X_3 = \{u_i\}$, jednak to spostrzeżenie nie będzie nam potrzebne).

W tej sytuacji $P' = X_1 X_3 X_2 X_4$ także jest 2H-ścieżką w drzewie prowadzącą z 1 do n. Zauważmy, że kolejność występowania wierzchołków ze zbioru $\mathcal{L}$ na ścieżkach P i P' jest taka sama. Co więcej, na ścieżce P' wszystkie wierzchołki ze zbioru $\mathcal{R}$ występują później niż wszystkie wierzchołki ze zbioru $\mathcal{L}$. Stąd $\mathit{ind}(P') > i = \mathit{ind}(P)$, co daje żądaną sprzeczność wobec metody wyboru ścieżki P.

Dowód lematu 2

Zacznijmy od uzasadnienia stwierdzenia, jeśli drzewo ma 2H-cykl, to musi ono być gąsienicą. Przyjrzyjmy się prostemu drzewu z rys. 4 (trzy przedłużone odnogi). Łatwo zauważyć, że drzewo jest gąsienicą wtedy i tylko wtedy, gdy nie zawiera drzewa z rys. 4 jako poddrzewa. Na mocy poniższego lematu o przycinaniu, gdyby jakieś drzewo niebędące gąsienicą miało 2H-cykl, to również drzewo z rys. 4 miałoby 2H-cykl.



Rys. 4: Najprostsze drzewo niezawierające 2H-cyklu.

Lemat 3 (O przycinaniu). *Niech T będzie drzewem o co najmniej dwóch wierzchołkach, w – liściem drzewa T, a S – dowolną 2-ścieżką w T. Wówczas $S \setminus \{w\}$ (tj. S z usuniętymi wystąpieniami wierzchołka w) jest 2-ścieżką w drzewie $T \setminus \{w\}$.*

Dowód: Na 2-ścieżce S wierzchołek w może sąsiadować jedynie ze swoim bezpośrednim sąsiadem v w T oraz z sąsiadami tego sąsiada. Zauważmy, że każdy wierzchołek ze zbioru $N(v)$ (v oraz jego sąsiedzi) jest odległy od każdego innego wierzchołka tego zbioru co najwyżej o 2. To oznacza, że po usunięciu wszystkich wystąpień w z S dalej mamy 2-ścieżkę. ■

Łatwo sprawdzić, że drzewo z rys. 4 nie zawiera 2H-cyklu. Stąd rzeczywiście jedynymi drzewami mogącymi mieć 2H-cykl są gąsienice.

Kolejnym krokiem jest formalne uzasadnienie faktu, że każda gąsienica zawiera 2H-cykl. Najlepiej zacząć od znalezienia 2H-cyklu na osi gąsienicy. Łatwo podać ogólną postać takiego 2H-cyklu, zależnie od parzystości długości osi: startujemy z jednego końca osi, skaczemy co drugi wierzchołek do drugiego końca osi, po czym zawracamy i znów skaczemy co dwa (rys. 5). Co więcej, widzimy, że na danej osi jest dokładnie jeden 2H-cykl (z dokładnością do wyboru wierzchołka początkowego i kierunku obejścia).

Rys. 5: 2H-cykl na osi gąsienicy.

Jeśli gąsienica zawiera odnogi, wystarczy umieścić je w odpowiednim miejscu opisanego wyżej 2H-cyklu. Załóżmy, że osią gąsienicy jest ścieżka $v_1, \ldots, v_l$. Wówczas odnogi podłączone do wierzchołka v_i (gdzie $1 < i < l$) należy umieścić na 2H-cyklu między wystąpieniami wierzchołków v_{i-1} i v_{i+1}, natomiast odnogi wychodzące z wierzchołka v_1 (odpowiednio v_l) umieścić należy między wystąpieniami wierzchołków v_1 oraz v_2 (odpowiednio v_{l-1} oraz v_l). W ten sposób otrzymamy dokładnie taki 2H-cykl, jak na przedstawionym wcześniej rysunku 3.

Ostatnią rzeczą, jaka pozostała nam do uzasadnienia, jest fakt, że każdy 2H-cykl w gąsienicy jest dokładnie opisanej postaci. Dokładniej, poszczególne 2H-cykle w danej gąsienicy różnią się tylko kolejnością odwiedzania odnóg podłączonych do tego samego wierzchołka osi (a także, rzecz jasna, wyborem wierzchołka początkowego 2H-cyklu i kierunku obejścia).

W uzasadnieniu możemy bazować na poczynionym już spostrzeżeniu, iż 2H-cykl na osi gąsienicy jest określony jednoznacznie. Na mocy lematu o przycinaniu, 2H-cykl w dowolnej gąsienicy ograniczony do wierzchołków jej osi jest 2H-cyklem tej osi. Każda z odnóg wpisuje się zatem w którymś miejscu jedynego możliwego 2H-cyklu osi gąsienicy. Wystarczy uzasadnić, że miejsce to jest wyznaczone jednoznacznie. Rozważmy zatem odnogę podłączoną do wierzchołka v_i, dla $2 < i < l - 1$. Wówczas mogłaby ona zostać dołączona albo na fragmencie $\ldots, v_{i-1}, v_{i+1}, \ldots$, albo na fragmencie $\ldots, v_{i+2}, v_i, v_{i-2}, \ldots$ Widzimy, że nie zaburzy ona własności 2H-cyklu tylko wtedy, gdy umieścimy ją na 2H-cyklu między wierzchołkami v_{i-1} oraz v_{i+1}. Podobnie można sprawdzić, że umiejscowienie odnóg podłączonych do wierzchołków v_1, v_2, v_{l-1} oraz v_l w ramach 2H-cyklu osi gąsienicy jest wyznaczone jednoznacznie. Ten szczegół dowodu pozostawimy Czytelnikowi jako ćwiczenie.

Krzysztof Diks, Wojciech Rytter — Treść zadania, Opracowanie
Igor Adamski — Program

Dostępna pamięć: 64 MB. *OI, etap I, 15.10–12.11.2012*

Taksówki

Bajtazar chce przejechać taksówką z miejscowości Bajtodziura do miejscowości Bajtodół, odległej od Bajtodziury o m km. W odległości d km od Bajtodziury na trasie między tymi miastami znajduje się baza taksówek dysponująca n taksówkami, ponumerowanymi od 1 do n. Taksówka numer i ma zapas benzyny wystarczający na przejechanie x_i km.

Bajtazar może się przesiadać, zmieniając taksówki. Taksówki wyruszają z bazy, ale nie muszą do niej wracać. Twoim zadaniem jest sprawdzenie, czy można przewieźć Bajtazara z Bajtodziury do Bajtodołu, a jeżeli tak, to jaka jest minimalna liczba taksówek, jakie należy wykorzystać.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się trzy liczby całkowite m, d oraz n ($1 \leqslant d \leqslant m \leqslant 10^{18}$, $1 \leqslant n \leqslant 500\,000$), pooddzielane pojedynczymi odstępami. Oznaczają one odpowiednio: odległość z Bajtodziury do Bajtodołu, odległość z Bajtodziury do bazy taksówek oraz liczbę taksówek znajdujących się w bazie. W drugim wierszu wejścia znajduje się n liczb całkowitych $x_1, x_2, \ldots, x_n$ ($1 \leqslant x_i \leqslant 10^{18}$), pooddzielanych pojedynczymi odstępami. Liczba x_i oznacza dystans (w km), jaki maksymalnie może przejechać taksówka numer i.

W testach wartych łącznie 40% punktów zachodzi dodatkowy warunek $n \leqslant 5000$.

Wyjście

Twój program powinien wypisać na standardowe wyjście jedną liczbę całkowitą: minimalną liczbę taksówek, którymi musi jechać Bajtazar, aby dostać się z Bajtodziury do Bajtodołu. Jeżeli nie jest to możliwe, Twój program powinien wypisać liczbę 0.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
42 23 6 20 25 14 27 30 7	4

Wyjaśnienie do przykładu: *Bajtazar może jechać kolejno taksówkami numer: 4, 5, 1 i 2.*

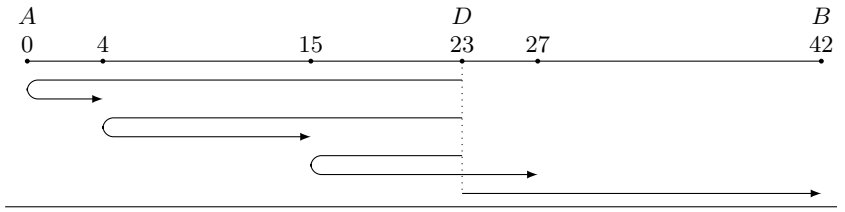
Rozwiązanie

Celem zadania jest przewieźć Bajtazara z miejscowości A do miejscowości B. Miejscowości te są oddalone o m kilometrów. Wyobraźmy sobie oś liczbową, na której A leży w punkcie 0, a B w punkcie m.

Do naszej dyspozycji jest n taksówek, z których i-ta może przejechać x_i kilometrów; będziemy mówili, że i-ta taksówka ma *zasięg* x_i. Dla zwięzłości, będziemy też

utożsamiali taksówkę z jej zasięgiem, mówiąc po prostu o taksówce x_i. Wszystkie taksówki są początkowo umiejscowione w punkcie osi o współrzędnej d, oznaczmy ten punkt przez D. Jeśli Bajtazara da się przewieźć z punktu A do punktu B, chcemy to zrobić, wykorzystując możliwie najmniejszą liczbę taksówek.

Poniższy rysunek przedstawia rozwiązanie testu przykładowego. W teście tym mamy $m = 42$, $d = 23$, a ciąg x zasięgów taksówek ma postać $(20, 25, 14, 27, 30, 7)$. W rozwiązaniu używamy kolejno taksówek $x_4 = 27$, $x_5 = 30$, $x_1 = 20$ i $x_2 = 25$.



Rys. 1: Jedno z rozwiązań testu przykładowego.

Już na podstawie analizy tego przykładu możemy poczynić pewne obserwacje. Pierwsza z wykorzystanych przez Bajtazara taksówek musi mieć zasięg co najmniej d (gdyż musi dojechać z punktu D do punktu A), a ostatnia z wykorzystanych taksówek musi mieć zasięg co najmniej $m - d$ (gdyż musi dojechać z punktu D do punktu B).

Oczywiście, taksówce niewiozącej Bajtazara na pewno nie opłaca się zawracać. Natomiast w momencie, gdy Bajtazar wsiada do jakiejś taksówki, powinna ona zacząć jechać w kierunku punktu B. Rzeczywiście, jeśli Bajtazar znajduje się wciąż przed punktem D, to w ten sposób przybliża się do każdego z punktów B i D, więc podróż w tę stronę jest jak najbardziej korzystna. Kiedy natomiast Bajtazar minie punkt D, to do dotarcia do celu wystarczy mu dokładnie jedna taksówka. Albo będzie to taksówka, w której minął punkt D, albo następna taksówka, o zasięgu nie mniejszym niż $m - d$.

Rozwiązanie wzorcowe

Odłóżmy na bok jedną taksówkę t o zasięgu co najmniej $m - d$. Jeśli żadnej takiej taksówki nie ma, od razu wiemy, że podróż Bajtazara jest niemożliwa. Natomiast jeśli jest więcej niż jedna taka taksówka, wybierzmy taką o najmniejszym zasięgu. Taksówkę t wykorzystamy jako ostatnią, będzie więc musiała dojechać do punktu B. Zatem taksówka ta może przejechać w kierunku punktu A odległość co najwyżej $\frac{1}{2}\big(t - (m - d)\big)$. Za pomocą pozostałych taksówek będziemy się więc starali dojechać do punktu o współrzędnej co najmniej $d_t = d - \frac{1}{2}\big(t - (m - d)\big) = \frac{1}{2}(m + d - t)$.

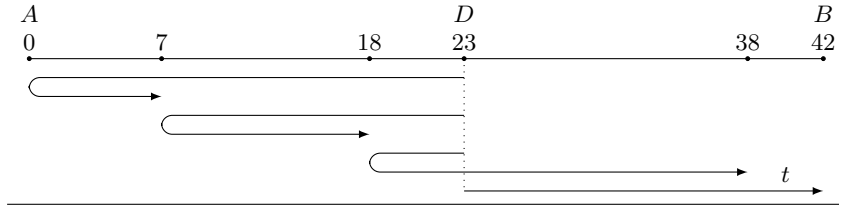
Posortujmy pozostałe taksówki *nierosnąco* względem ich zasięgów. Niech $y_1, \ldots, y_{n-1}$ oznacza tak uporządkowany ciąg zasięgów taksówek. W rozwiązaniu będziemy używać kolejnych taksówek z tego ciągu.

Niech b oznacza położenie Bajtazara przed rozważeniem i-tej taksówki. Załóżmy, że $b < m$. Jeśli $b \geqslant d_t$, to wystarczy użyć taksówki t i w ten sposób podróż Bajtazara się kończy.

W przeciwnym razie mamy $b < d_t \leqslant d$. Jeśli taksówka y_i nie jest w stanie dojechać do Bajtazara (tj. $y_i < d - b$), to stwierdzamy, że szukana trasa Bajtazara nie istnieje. Jeśli natomiast żaden z tych przypadków nie zachodzi, to używamy i-tej taksówki i przewozimy Bajtazara z punktu b do punktu $b + \big(y_i - (d - b)\big) = 2b - d + y_i$.

Zauważmy, że w ten sposób możemy też osiągnąć punkt B, nie używszy po drodze taksówki t.

Poniższy rysunek pokazuje, jak działa ta metoda dla testu przykładowego. Mamy tu $t = 20$, a ciąg y ma postać $(30, 27, 25, 14, 7)$.



Rys. 2: Schemat działania rozwiązania wzorcowego dla testu przykładowego.

Rozwiązanie wzorcowe działa w czasie $O(n \log n)$, jeśli do uporządkowania taksówek w ciąg y użyjemy efektywnego algorytmu sortowania, np. przez scalanie. Implementację można znaleźć w plikach `tak.cpp`, `tak1.pas` i `tak2.c`.

Uzasadnienie poprawności

W tej sekcji wykażemy, że jeśli rozwiązanie istnieje, nasz algorytm znajdzie rozwiązanie wykorzystujące minimalną liczbę taksówek.

Niech $z_1, \ldots, z_k$ oznacza zasięgi kolejnych taksówek w rozwiązaniu optymalnym. Jeśli jest więcej niż jedno rozwiązanie o tej samej wartości k, wybieramy ciąg taksówek największy leksykograficznie (powiemy, że ciąg $z_1, \ldots, z_k$ jest większy leksykograficznie niż ciąg $z'_1, \ldots, z'_k$, jeśli istnieje $1 \leqslant i \leqslant k$, takie że $z_1, \ldots, z_{i-1} = z'_1, \ldots, z'_{i-1}$ oraz $z_i > z'_i$). Spróbujmy rozpoznać strukturę rozwiązania $z_1, \ldots, z_k$.

Oczywiście zachodzi $z_k \geqslant t$, gdyż $z_k \geqslant m - d$, a t była taksówką o najmniejszym zasięgu nie mniejszym niż $m - d$. Ponadto możemy założyć, że $\{z_1, \ldots, z_k\}$ stanowią łącznie k taksówek o największych zasięgach. Gdyby tak nie było, istniałaby jakaś niewykorzystana taksówka x, $x > z_i$. Wówczas używając taksówki x zamiast taksówki z_i, otrzymalibyśmy poprawne rozwiązanie większe leksykograficznie niż $z_1, \ldots, z_k$.

Warto się następnie zastanowić nad kolejnością, w jakiej są ustawione taksówki $z_1, \ldots, z_k$. Poniższy lemat dostarcza kluczową własność tej kolejności.

Lemat 1. Jeśli $i < k$ i po wykorzystaniu taksówek $z_1, \ldots, z_i$ Bajtazar wciąż znajduje się przed punktem D, to $z_i \geqslant z_{i+1}$.

Dowód: Załóżmy przez sprzeczność, że przy założeniach lematu zachodzi $z_i < z_{i+1}$. Wykażemy, że wówczas ciąg taksówek $z_1, \ldots, z_{i-1}, z_{i+1}, z_i, z_{i+2}, \ldots, z_k$ również byłby poprawnym rozwiązaniem. To będzie stanowiło żądaną sprzeczność, gdyż jest to ciąg większy leksykograficznie niż $z_1, \ldots, z_k$.

Oznaczmy przez b pozycję Bajtazara po wykorzystaniu taksówek $z_1, \ldots, z_{i-1}$ i niech $\Delta = d - b$. Taksówka z_i przenosi Bajtazara do przodu o $z_i - \Delta$. Po tym ruchu Bajtazar znajduje się $\Delta - (z_i - \Delta) = 2\Delta - z_i$ jednostek od punktu D i z założenia wiemy, że nie osiągnął jeszcze punktu D. Stąd taksówka z_{i+1} przeniesie Bajtazara o $z_{i+1} - (2\Delta - z_i) = z_{i+1} + z_i - 2\Delta$ jednostek wprzód. Łącznie te dwie taksówki przeniosły Bajtazara o

$$z_{i+1} + 2z_i - 3\Delta \tag{1}$$

jednostek.

Gdyby jednak zamienić miejscami taksówki z_i i z_{i+1}, analogiczne rozumowanie pokazuje, że z ich użyciem Bajtazar albo przeniósłby się łącznie o

$$z_i + 2z_{i+1} - 3\Delta \tag{2}$$

jednostek do przodu, co jest wartością większą niż (1), albo już po wykorzystaniu lepszej taksówki z_{i+1} osiągnąłby punkt D i wtedy sama taksówka z_k wystarczyłaby mu do osiągnięcia celu. W obu przypadkach zamiana taksówek skutkuje poprawnym rozwiązaniem, co, na mocy założenia, nie może mieć miejsca. ■

Lemat 1 możemy na pewno zastosować dla dowolnego $i < k - 1$. Stąd wiemy, że $z_1 \geqslant z_2 \geqslant \ldots \geqslant z_{k-1}$. Jeśli teraz po wykorzystaniu taksówek $z_1, \ldots, z_{k-1}$ Bajtazar wciąż nie przekroczył punktu D, to na mocy lematu mamy $z_1 \geqslant z_2 \geqslant \ldots \geqslant z_{k-1} \geqslant z_k \geqslant t$. Wówczas $z_1, \ldots, z_k$ to po prostu k największych taksówek uporządkowanych nierosnąco, więc nasz algorytm znajdzie to rozwiązanie lub równoliczne rozwiązanie $z_1, \ldots, z_{k-1}, t$.

Drugi przypadek jest taki, że taksówki $z_1, \ldots, z_{k-1}$ umożliwiają Bajtazarowi przekroczenie punktu D. Wówczas wystarczy, aby $z_k \geqslant t$. Jeśli $z_k = t$, to $z_1, \ldots, z_{k-1}$ stanowią $k - 1$ taksówek o największych zasięgach poza t uporządkowanych nierosnąco, więc nasz algorytm również takie rozwiązanie znajdzie. Jeśli zaś $z_k > t$, to mamy dwa przypadki. Jeśli taksówka t występuje wśród $z_1, \ldots, z_{k-1}$, możemy zamienić tę taksówkę miejscami z z_k i w ten sposób otrzymamy poprawne rozwiązanie leksykograficznie większe (co nie jest możliwe). Jeśli zaś t nie występuje wśród $z_1, \ldots, z_{k-1}$, to musi zachodzić $z_1 \geqslant \ldots \geqslant z_{k-1} \geqslant z_k > t$, a ten przypadek rozpatrzyliśmy już wcześniej.

Rozwiązania błędne

W rozwiązaniu wzorcowym wybieraliśmy jedną taksówkę, którą oznaczaliśmy jako t, i odkładaliśmy ją na ostatnią przesiadkę. Inny pomysł na rozwiązanie mógłby zakładać, że po prostu sortujemy wszystkie taksówki nierosnąco i używamy ich w tej właśnie kolejności. Niestety jest to rozwiązanie błędne, o czym łatwo się przekonać, rozpatrując następujący przykład: $m = 25$, $d = 10$, mamy jedną taksówkę o zasięgu 15 i dowolnie wiele taksówek o zasięgu 11. Takie rozwiązanie nie uzyskiwało na zawodach żadnych punktów. Można je znaleźć w pliku `takb1.cpp`.

Innym błędem było korzystanie w rozwiązaniu tylko z liczb całkowitych 32-bitowych, które nie starczały do reprezentacji danych wejściowych. Rozwiązania wzorcowe z tym błędem uzyskiwały na zawodach 40% punktów. Takie rozwiązanie znajduje się w pliku `takb2.cpp`.

Jakub Pachocki, Wojciech Rytter — Treść zadania, Opracowanie
Igor Adamski — Program

Dostępna pamięć: 64 MB. *OI, etap I, 15.10–12.11.2012*

Usuwanka

Mała Bajtosia dostała w prezencie grę o nazwie **Usuwanka**. *W tej grze dany jest ciąg n przylegających do siebie klocków, ponumerowanych kolejno od 1 do n. Każdy klocek jest albo biały, albo czarny, przy czym białych klocków jest k razy więcej niż czarnych. Gra jest jednoosobowa. Celem gry jest usunięcie, za pomocą określonych ruchów, wszystkich klocków z ciągu.*

Pojedynczy ruch polega na usunięciu dokładnie k białych i jednego czarnego klocka bez zmiany pozycji pozostałych klocków. Ruch jest dozwolony, jeśli między żadnymi dwoma usuwanymi w tym ruchu klockami nie ma „dziury", czyli pustego miejsca po uprzednio usuniętym klocku.

Pomóż Bajtosi, ona tak bardzo Cię prosi... Znajdź dowolny ciąg dozwolonych ruchów prowadzący do usunięcia wszystkich klocków.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie liczby całkowite n oraz k ($2 \leqslant n \leqslant 1\,000\,000$, $1 \leqslant k \leqslant n-1$), oddzielone pojedynczym odstępem, oznaczające liczbę klocków w grze oraz liczbę białych klocków, które należy usunąć w każdym ruchu. We wszystkich testach zachodzi warunek $k+1 \mid n$.

W drugim wierszu znajduje się napis złożony z n liter `b` *oraz* `c`. *Kolejne litery napisu określają kolor kolejnych klocków:* `b` *– biały,* `c` *– czarny. Możesz założyć, że dla danych testowych istnieje szukany ciąg ruchów prowadzący do usunięcia wszystkich klocków.*

W testach wartych łącznie 40% punktów zachodzi dodatkowy warunek $n \leqslant 10\,000$.

Wyjście

Twój program powinien wypisać na standardowe wyjście $\frac{n}{k+1}$ wierszy. Kolejne wiersze powinny opisywać kolejne ruchy. Każdy z tych wierszy powinien zawierać $k+1$ liczb, podanych w kolejności rosnącej oraz rozdzielonych pojedynczymi odstępami, oznaczających numery klocków, które należy usunąć w danym ruchu.

Przykład

Dla danych wejściowych:

```
12 2
ccbcbbbbbbcb
```

poprawnym wynikiem jest:

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

Wyjaśnienie do przykładu: *Niech* ␣ *oznacza puste miejsce po usuniętym klocku. Wykonując podane powyżej ruchy, uzyskujemy kolejno następujące układy klocków:*

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
c	c	b	c	b	b	b	b	b	b	c	b
␣	c	b	c	b	b	b	␣	b	b	c	␣
␣	␣	b	c	b	␣	␣	␣	b	b	c	␣
␣	␣	␣	␣	␣	␣	␣	␣	b	b	c	␣
␣	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣

Rozwiązanie

Zastąpmy w naszym ciągu początkowym klocki białe przez wartości $+1$, a klocki czarne przez wartości $-k$. Otrzymamy w ten sposób ciąg liczbowy o sumie 0. Tego typu ciągi (o sumie zero, składające się z wartości $+1$ i $-k$) będziemy w tym zadaniu nazywać ciągami *usuwankowymi*.

Z treści zadania wiemy, że dla początkowego ciągu istnieje sekwencja dozwolonych ruchów usuwająca wszystkie klocki; powiemy, że ciąg ten jest *rozwiązywalny*. Ostatni ruch musi usuwać *spójny* ciąg $k+1$ klocków, tzn. ciąg klocków parami sąsiednich. Wynika z tego, że w naszym ciągu istnieje spójny usuwankowy podciąg o długości $k+1$. Okazuje się, że tę własność spełnia *dowolny* ciąg usuwankowy:

Lemat 1. W dowolnym ciągu usuwankowym istnieje spójny usuwankowy podciąg o długości $k+1$.

Dowód: Niech n oznacza długość ciągu. Podzielmy ciąg na $\frac{n}{k+1}$ bloków o długości $k+1$. Wyrazów równych $-k$ w ciągu również jest $\frac{n}{k+1}$, więc w każdym bloku jest dokładnie jedno $-k$ albo istnieje blok, w którym nie ma żadnego $-k$. W pierwszym przypadku każdy z bloków jest szukanym podciągiem. W drugim przypadku wybieramy ten blok, który zawiera same $+1$, i przesuwamy go w prawo bądź lewo do najbliższego $-k$, uzyskując w ten sposób podciąg o sumie 0. ■

Lemat 2. Każdy ciąg usuwankowy jest rozwiązywalny.

Dowód: Dowód przeprowadzimy przez indukcję ze względu na liczbę klocków. Dla $n = k+1$ teza jest oczywista. Jeśli $n > k+1$, to na mocy lematu 1 istnieje spójny podciąg długości $k+1$, który sumuje się do 0. Możemy go usunąć i scalić resztę (jeśli usunięcie rozspójniło ciąg). Na mocy indukcji wiemy, że istnieje sekwencja ruchów usuwająca nowo powstały ciąg, ponieważ ciąg ten jest krótszy (a nadal jest usuwankowy). Wstawiając z powrotem usunięty wcześniej podciąg, uzyskujemy poprawną sekwencję ruchów, która usuwa wszystko prócz tego fragmentu. Na końcu wystarczy zatem usunąć ten fragment. ■

Rozwiązanie wzorcowe

Rozwiązanie wzorcowe to efektywna implementacja dowodu lematu 2. W rozwiązaniu wykorzystamy stos S z następującymi operacjami:

- *push*(i, S) – połóż element i na szczyt stosu S,
- *pop*(S) – usuń element leżący na szczycie stosu S i zwróć jego wartość.

Dodatkowo będziemy chcieli mieć możliwość odpytywania o sumę elementów ciągu wejściowego na pozycjach odpowiadających „górnym" $k+1$ elementom stosu, do czego wykorzystamy operację *topSum*(S). Jeśli rozmiar stosu jest mniejszy niż $k+1$, to operacja ta będzie zwracać -1.

```
 1: function Usuwanka()
 2: begin
 3:   S := S' := pusty stos;
 4:   for i := 1 to n do begin
 5:     push(i, S);
 6:     if topSum(S) = 0 then
 7:       { przełóż k+1 górnych elementów z S do S' }
 8:       for j := 1 to k+1 do
 9:         push(pop(S), S');
10:   end
11:   for i := 1 to n do begin
12:     write(pop(S'));
13:     if i | k+1 then writeln();
14:   end
15: end
```

Łatwo zrealizować operację *topSum*(S) w czasie $O(k)$, co da nam algorytm działający w czasie $O(nk)$. Szybszą implementację uzyskamy, trzymając tablicę *sum*$[0..n]$, w której będziemy pamiętać sumy prefiksowe elementów ciągu wejściowego odpowiadające kolejnym elementom stosu S, tzn. *sum*$[i]$ jest sumą tych elementów ciągu wejściowego, które odpowiadają i „dolnym" elementom stosu S.

Podczas operacji *push*(i, S) musimy uaktualnić tablicę *sum*. Jeśli m jest wielkością stosu S po wykonaniu tej operacji, a a_i jest i-tym elementem ciągu wejściowego, to dajemy $sum[m] := sum[m-1] + a_i$. Operacja *topSum*(S) zwraca po prostu $sum[m] - sum[m-k-1]$ lub -1 jeśli $m < k+1$.

Tak zapisane rozwiązanie wzorcowe działa w czasie $O(n)$. Można je znaleźć w plikach `usu.cpp` i `usu1.pas`. Alternatywne rozwiązanie liniowe zapisano w plikach `usu2.cpp` i `usu3.pas`, natomiast w pliku `usu4.cpp` zaimplementowano rozwiązanie o złożoności $O(n \log k)$, które również otrzymywało na zawodach maksymalną liczbę punktów.

Testy

Testy *1-4* są w większości małymi testami losowymi. Testy *5a-8a* sprawiają kłopoty rozwiązaniom o złożoności $O(\frac{n^2}{k})$, a testy *5b-8b* mają tę samą strukturę, co odpowiadający test *a*, ale z dużym k. Testy z grupy *9* są testami maksymalnymi. Przy każdym teście w opisie podano, jak został on wygenerowany, przy czym *losowy*(b, c) oznacza losowy ciąg, który zawiera b białych klocków i c czarnych.

Nazwa	n	k	Opis
usu1a.in	2	1	bc – test minimalny
usu1b.in	15	2	$b^5c^5b^5$
usu1c.in	45	2	$(bcb)^{15}$
usu1d.in	9	2	*cbcbbbbbc*
usu2a.in	110	10	$b^{100}c^{10}$
usu2b.in	319	10	test losowy
usu3a.in	1200	5	test losowy
usu3b.in	1633	70	test losowy
usu4a.in	7994	3996	$(b^{\frac{2k}{3}}cb^{\frac{k}{3}})^2$
usu4b.in	9898	100	test losowy
usu5a.in	120 000	5	$c^{10\,000}b^{100\,000}c^{10\,000}$
usu5b.in	100 002	50 000	$b^{50\,000}ccb^{50\,000}$
usu6a.in	220 088	21	$cc(b^{19}cc)^{5000}b^{115\,084}cc$
usu6b.in	271 612	12 345	$cc(b^{12\,343}cc)^{10}b^{148\,160}$
usu7a.in	505 000	100	$b^{247\,500}losowy(5000, 5000)b^{247\,500}$
usu7b.in	300 010	30 000	$b^{150\,000}losowy(10, 10)b^{149\,990}$
usu8a.in	728 000	90	$losowy(300, 10)^{400}b^{480\,000}losowy(300, 10)^{400}$
usu8b.in	800 008	100 000	$losowy(150\,000, 2)^2b^{200\,000}losowy(150\,000, 2)^2$
usu9a.in	1 000 000	1	$b^{500\,000}c^{500\,000}$ – test maksymalny
usu9b.in	1 000 000	999 999	$cb^{999\,999}$ – test maksymalny
usu9c.in	1 000 000	9 999	$b^{499\,950}c^{100}b^{499\,950}$ – test maksymalny
usu9d.in	1 000 000	99	$(ccb^{396}cb^{250\,000}losowy(244\,604, 4996)c)^2$ – maksymalny test losowy

Zawody II stopnia

opracowania zadań

Wojciech Rytter — Treść zadania, Opracowanie
Wiktor Kuropatwa — Program

Dostępna pamięć: 256 MB. *OI, etap II, dzień próbny, 12.02.2013*

Spacer

Nazwy miast w Bajtocji mają postać ciągów n bitów. Oczywiście różne miasta mają różne nazwy. Wszystkich miast w Bajtocji jest $2^n - k$. Tak więc tylko k różnych ciągów n bitów nie jest nazwami miast.

Niektóre pary miast w Bajtocji są połączone drogami. Drogi te nie krzyżują się ze sobą poza miastami. Dwa miasta są połączone bezpośrednio drogą wtedy i tylko wtedy, gdy ich nazwy różnią się tylko jednym bitem.

Bajtazar wybiera się na spacer – chce przejść z miasta x do miasta y, korzystając jedynie z istniejących dróg. Twoim zadaniem jest napisanie programu, który sprawdzi, czy taki spacer jest możliwy.

Wejście

W pierwszym wierszu standardowego wejścia zapisane są dwie liczby całkowite n i k oddzielone pojedynczym odstępem ($1 \leqslant n \leqslant 60$, $0 \leqslant k \leqslant 1\,000\,000$, $k < 2^n - 1$, $n \cdot k \leqslant 5\,000\,000$). Oznaczają one odpowiednio liczbę bitów w nazwach miast oraz liczbę różnych ciągów n bitów, które nie są nazwami żadnych miast. W drugim wierszu znajdują się dwa napisy oddzielone pojedynczym odstępem, każdy złożony z n znaków `0` *i/lub* `1`*. Są to nazwy miast x i y. W kolejnych k wierszach są wymienione wszystkie ciągi n bitów, które nie są nazwami miast, po jednym w wierszu. Każdy taki ciąg bitów ma postać napisu złożonego z n znaków* `0` *i/lub* `1`*. Możesz założyć, że nazwy miast x i y nie pojawiają się wśród tych k ciągów bitów.*

W testach wartych łącznie 25% punktów zachodzi dodatkowy warunek $n \leqslant 22$.

Wyjście

Twój program powinien wypisać na standardowe wyjście słowo `TAK`*, jeżeli możliwe jest przejście z miasta x do miasta y, a w przeciwnym przypadku powinien wypisać słowo* `NIE`*.*

Przykład

Dla danych wejściowych:

```
4 6
0000 1011
0110
0111
0011
1101
1010
1001
```

poprawnym wynikiem jest:

```
TAK
```

a dla danych wejściowych:	*poprawnym wynikiem jest:*

`2 2`
`00 11`
`01`
`10`
	`NIE`

Wyjaśnienie do pierwszego przykładu: *Oto przykładowe trasy prowadzące z miasta* `0000` *do miasta* `1011`*:*

- `0000` → `1000` → `1100` → `1110` → `1111` → `1011`,
- `0000` → `0100` → `1100` → `1110` → `1111` → `1011`.

Testy „ocen”:

`1ocen`**:** *$n = 20$, $k = 215\,766$, zabronione są wszystkie miasta o liczbie zer podzielnej przez 5, chcemy wykonać spacer z $x =$* `100...0` *do $y =$* `011...1`*; odpowiedź* `NIE`*;*

`2ocen`**:** *$n = 50$, $k = 100\,000$, miasta x i y są połączone bezpośrednio drogą; odpowiedź* `TAK`*;*

`3ocen`**:** *ładny test z $n = 60$; odpowiedź* `TAK`*.*

Rozwiązanie

Zadanie to było nietypowe i opierało się na ciekawej własności kombinatorycznej pewnego grafu. Własność ta nie jest łatwa do udowodnienia (choć tego od uczestników Olimpiady w czasie zawodów się nie żąda), ale na intuicję można taką (lub podobną) własność zauważyć w oparciu o silny stopień wzajemnych powiązań wewnątrz grafu. Z podanych względów zadanie pojawiło się na sesji próbnej i nie liczyło się do punktacji.

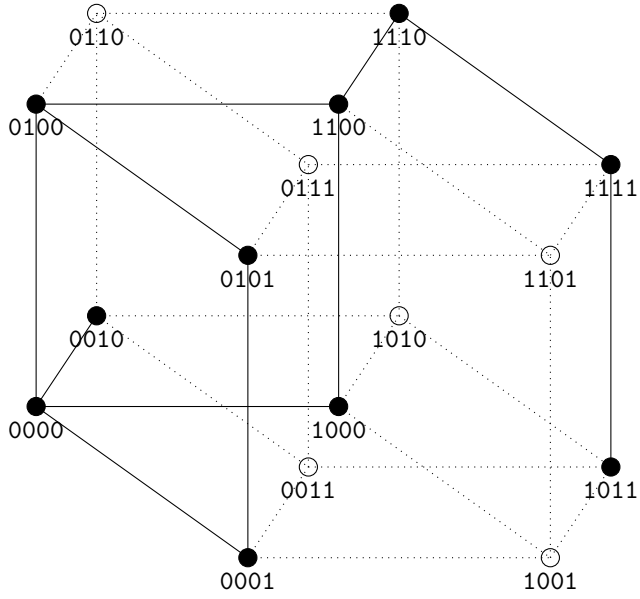
W przypadku $k = 0$ grafem rozważanym w zadaniu jest n-wymiarowa hiperkostka. Ma ona 2^n wierzchołków, a z każdego z nich wychodzi dokładnie n krawędzi (patrz rys. 1). To właśnie ta duża liczba krawędzi wychodzących z każdego wierzchołka powoduje, że zadanie *Spacer* ma rozwiązanie wielomianowe. Własność tę można trochę wzmocnić:

Twierdzenie 1. *Dla każdego podziału wierzchołków hiperkostki na dwa zbiory S i $V \setminus S$ liczba krawędzi prowadzących z jednego zbioru do drugiego jest co najmniej tak duża jak liczność mniejszego ze zbiorów:*

$$|\{(u, v) \colon u \in S, v \in V \setminus S\}| \geqslant \min(|S|, |V \setminus S|).$$

Dowód powyższego twierdzenia znajduje się na końcu opisu. Spójrzmy, co to twierdzenie nam daje.

W naszym zadaniu usunęliśmy ze zbioru wierzchołków hiperkostki pewien podzbiór V_K, o liczności k. To mogło spowodować podział pozostałych wierzchołków na więcej niż jedną spójną składową. Intuicyjnie, jeśli n jest duże, tylko jedna z tych spójnych składowych może być „duża”. Musieliśmy bowiem usunąć wszystkie krawędzie



Rys. 1: Ilustracja do pierwszego przykładu: hiperkostka dla $n = 4$, z której usunięto 6 wierzchołków.

prowadzące pomiędzy składowymi. Gdybyśmy zatem mieli dwie „duże" składowe, to wynikałoby z tego, że musieliśmy usunąć pomiędzy nimi „dużo" krawędzi. A usunęliśmy ich co najwyżej nk.

Uzasadnijmy to formalnie. Załóżmy, że mamy dwie spójne składowe o rozmiarze co najmniej $nk+1$ i niech S będzie jedną z nich. Wtedy z Twierdzenia 1 w hiperkostce pomiędzy S i $V \setminus S$ znajduje się co najmniej $nk+1$ krawędzi (rozmiar $V \setminus S$ jest równy co najmniej $nk+1$, bo znajduje się tam druga z rozważanych składowych). Ponieważ tylko nk z tych krawędzi może wchodzić do zbioru V_K, zatem między S a zbiorem $V \setminus (S \cup V_K)$ znajduje się co najmniej jedna krawędź. Daje to sprzeczność z tym, że S jest spójną składową. Zatem w naszym grafie jest co najwyżej jedna spójna składowa rozmiaru co najmniej $nk+1$.

Dzięki temu jesteśmy już w stanie napisać prosty algorytm działający w czasie $O(nk)$. Jeśli wierzchołki x i y znajdują się w tej samej spójnej składowej, to mamy dwie możliwości:

(1) Składowa ta ma rozmiar mniejszy niż $nk+1$, zatem przeszukując graf z wierzchołka x, znajdziemy ścieżkę do wierzchołka y po co najwyżej $nk+1$ krokach.

(2) Składowa ta jest jedyną składową o rozmiarze co najmniej $nk+1$. Do przetestowania tej możliwości wystarczy przeszukać graf z wierzchołka x, a następnie z wierzchołka y i sprawdzić, czy w obu przypadkach możemy odwiedzić co najmniej $nk+1$ wierzchołków. Po odwiedzeniu takiej liczby wierzchołków, kończymy przeszukiwanie.

Wystarczy zatem napisać funkcję *RestrictedSearch*(x, y, l), która przeszukuje graf, startując z wierzchołka x, i jeśli znajdzie wierzchołek y lub odwiedzi l wierzchołków, to zwraca **true**. Gdy te przypadki nie wystąpią, a funkcja przejrzy wszystkie osiągalne z x wierzchołki – zwraca **false**. Ścieżka pomiędzy x a y istnieje dokładnie wtedy, gdy zarówno *RestrictedSearch*$(x, y, nk+1)$, jak i *RestrictedSearch*$(y, x, nk+1)$ zwrócą **true**.

Rozwiązanie wzorcowe przegląda zatem $O(nk)$ wierzchołków i $O(n^2k)$ krawędzi. Wierzchołki zabronione utrzymywane są w tablicy z haszowaniem, co gwarantuje złożoność czasową rozwiązania $O(n^2k)$. Do przeglądania grafu wykorzystano przeszukiwanie wszerz, ze względu na łatwość implementacji i proste do przewidzenia zużycie pamięci, w przeciwieństwie do rozwiązań rekurencyjnych. Implementacje można znaleźć w plikach `spa.cpp`, `spa1.pas` i `spa2.cpp`.

Dowód własności podziałowej

Zamieńmy każdą krawędź hiperkostki na dwie krawędzie skierowane. *Ścieżką standardową* prowadzącą z wierzchołka u do wierzchołka v nazwiemy taką skierowaną ścieżkę najmniejszej długości, która odpowiada zmianie kolejnych niezgodnych bitów (od lewej do prawej) w ciągach bitów odpowiadających wierzchołkom u i v. Na przykład dla $u = \texttt{0101}$, $v = \texttt{0010}$ ścieżka standardowa odpowiada zmianie kolejno drugiego, trzeciego i czwartego bitu:

$$\texttt{0101} \rightarrow \texttt{0001} \rightarrow \texttt{0011} \rightarrow \texttt{0010}.$$

Ustalmy teraz pewną krawędź skierowaną e i zastanówmy się, ile standardowych ścieżek przechodzi przez e. Rozważmy ścieżkę standardową z u do v, która zawiera e. Jeśli krawędź e dotyczy zmiany i-tego bitu, to ostatnie $n - i$ bitów u oraz pierwsze $i - 1$ bitów v są zdeterminowane przez tę krawędź. Poza tym krawędź determinuje i-ty bit u i v. Zatem pozostaje $n - 1$ możliwości na wybór niezdeterminowanych bitów w ciągach odpowiadających wierzchołkom u i v. Zatem przez każdą krawędź skierowaną przechodzi dokładnie 2^{n-1} ścieżek standardowych.

Jesteśmy już gotowi do dowodu Twierdzenia 1. Oznaczmy $m = |S|$ i przyjmijmy, że $m \leqslant 2^n - m = |V \setminus S|$. Mamy $m \cdot (2^n - m)$ ścieżek standardowych prowadzących z wierzchołków S do wierzchołków $V \setminus S$. Każda z nich przechodzi przez co najmniej jedną krawędź skierowaną między S i $V \setminus S$. Ponieważ przez jedną taką krawędź przechodzi co najwyżej 2^{n-1} ścieżek standardowych, a $m \leqslant 2^{n-1}$, więc liczba krawędzi pomiędzy S i $V \setminus S$ wynosi co najmniej

$$\frac{m \cdot (2^n - m)}{2^{n-1}} \geqslant \frac{m \cdot 2^{n-1}}{2^{n-1}} = m = |S|.$$

Jakub Wojtaszczyk — Treść zadania, Opracowanie
Marcin Andrychowicz — Program

Dostępna pamięć: 128 MB. *OI, etap II, dzień pierwszy, 13.02.2013*

Inspektor

Inspektor Bajtazar bada zbrodnię, która wydarzyła się w biurze informatyków. Próbuje teraz ustalić chronologię zdarzeń. Niestety, informatycy są jednostkami stosunkowo roztrzepanymi. Żaden z nich nie udziela rozsądnych informacji, mówią raczej: „No, kiedy zajrzałem na serwer o 14:42, to zobaczyłem, że zalogowanych w pracy jest pięciu innych informatyków".

Wiadomo, że każdy z informatyków tego dnia przyszedł do biura, spędził w nim trochę czasu i wyszedł. Żaden informatyk nie opuszczał biura pomiędzy swoim przyjściem a wyjściem i nie pojawiał się w biurze poza tym odcinkiem czasu.

Bajtazar nie jest pewien, czy może polegać na zeznaniach informatyków. Chce się dowiedzieć, czy w ogóle możliwe jest, żeby wszyscy mówili prawdę. Poprosił Cię o pomoc.

Wejście

W pierwszym wierszu standardowego wejścia znajduje się liczba całkowita z ($1 \leqslant z \leqslant 50$), oznaczająca liczbę zestawów danych. W kolejnych wierszach znajdują się opisy z zestawów danych.

W pierwszym wierszu opisu znajdują się dwie liczby całkowite n i m oddzielone pojedynczym odstępem ($1 \leqslant n, m \leqslant 100\,000$). Oznaczają one odpowiednio liczbę informatyków w biurze i liczbę zeznań zebranych przez Bajtazara. Informatycy są ponumerowani od 1 do n.

W każdym z kolejnych m wierszy opisane jest jedno zeznanie. Każdy z tych wierszy zawiera trzy liczby całkowite t, j oraz i pooddzielane pojedynczymi odstępami ($1 \leqslant t \leqslant m$, $1 \leqslant j \leqslant n$, $0 \leqslant i < n$). Oznaczają one, że informatyk numer j zeznał, iż w chwili t był w biurze i oprócz niego było tam jeszcze i innych informatyków. Większa liczba oznacza późniejszą chwilę. Przyjmujemy, że informatycy przychodzili do pracy i wychodzili z pracy w czasie przed, po lub pomiędzy chwilami, których dotyczą zeznania.

W testach wartych łącznie 7% punktów zachodzi dodatkowy warunek $n, m \leqslant 5$. Ponadto w testach wartych łącznie 28% punktów zachodzi dodatkowy warunek $n, m \leqslant 101$.

Wyjście

Dla każdego zestawu danych Twój program powinien wypisać na standardowe wyjście jedną dodatnią liczbę całkowitą. Wypisanie liczby k ($1 \leqslant k \leqslant m$) oznacza, że pierwsze k zeznań podanych na wejściu dla danego zestawu może być prawdziwych, natomiast pierwsze $k + 1$ zeznań nie może być prawdziwych. W szczególności, jeśli $k = m$, to wszystkie zeznania podane na wejściu mogą być prawdziwe.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
2 3 5 1 1 1 1 2 1 2 3 1 4 1 1 4 2 1 3 3 3 3 0 2 2 0 1 1 0	4 3

Wyjaśnienie do przykładu: *W pierwszym zestawie danych nie wszystkie zeznania mogą być prawdziwe. Informatycy 1 i 2 zeznali, że byli w biurze przynajmniej od chwili 1 do chwili 4, natomiast informatyk 3 zeznał, że w chwili 2 w biurze była tylko jedna osoba oprócz niego. Jeśli odrzucimy ostatnie zeznanie, to pozostałe mogą już być prawdziwe. Wystarczy, żeby informatyk 2 wyszedł z biura między chwilą 1 i 2.*

W drugim zestawie danych wszystkie złożone zeznania mogą być prawdziwe.

Testy „ocen":

`1ocen`: *$z = 1$, $n = 5$, $m = 5$, wszystkie zeznania są ze sobą zgodne, żaden informatyk nie widział się z żadnym innym.*

`2ocen`: *$z = 50$, w każdym zestawie danych $n = 101$, $m = 101$. W zestawach o nieparzystych indeksach wszystkie zeznania są ze sobą zgodne i każdy informatyk widział się z każdym innym. Zestawy o parzystych indeksach różnią się tym, że zeznania dotyczące jednego momentu są składane przez większą liczbę informatyków, niż wynika to z owych zeznań.*

`3ocen`: *$z = 1$, $n = 100\,000$, $m = 50\,000$, wszystkie zeznania poza jednym wyglądają tak samo, zaś to jedno pozostałe jest z nimi sprzeczne.*

Rozwiązanie

Zacznijmy od zrozumienia, jakie informacje są zawarte w pojedynczym zeznaniu. Po pierwsze, jeśli informatyk j zeznaje, że w chwili t było w biurze i osób oprócz niego, to wiemy, że w chwili t łącznie było $i + 1$ osób. Po drugie, wiemy, że informatyk j był w biurze w chwili t. Z treści zadania wynika, że każdy informatyk był w pracy w pewnym spójnym przedziale czasu.

W zadaniu pytani jesteśmy o to, jak wiele początkowych zeznań może być niesprzeczne. Zauważmy, że jeśli początkowe k zeznań jest niesprzeczne, to i dowolna mniejsza początkowa liczba zeznań jest niesprzeczna – a zatem do znalezienia największego k takiego, że pierwsze k zeznań jest niesprzeczne, będziemy mogli zastosować

wyszukiwanie binarne. Do tego wystarczy nam umieć odpowiedzieć na pytanie „czy dany zbiór zeznań jest niesprzeczny".

Nierozróżnialni informatycy

Zaczniemy od prostszego zadania, w którym będziemy ignorować informacje drugiego typu (mówiące, że dany informatyk był w biurze w chwili, której dotyczy jego zeznanie). Innymi słowy, spróbujemy rozwiązać zadanie, w którym będziemy mieli zbiór zdarzeń postaci „w pracy jest i informatyków". Każde zdarzenie będzie reprezentowane przez parę (t_l, i_l), która oznacza, że w chwili t_l w biurze było i_l informatyków.

Po pierwsze, jeśli dla pewnej chwili t mamy więcej niż jedno zeznanie, to wszystkie te zeznania muszą się zgadzać. Ten warunek łatwo można sprawdzić na początku, sortując zeznania po czasie i sprawdzając dla sąsiednich zeznań, czy nie mają równych czasów a różnych liczb informatyków.

Po drugie, mogą nam się „skończyć" informatycy. Przykładowo, jeśli informatyków jest czterech i wiemy, że w chwilach $t = 1$ i $t = 3$ było ich w biurze po trzech, zaś w $t = 2$ był tylko jeden, to zeznania są sprzeczne. Wiemy z jednej strony, że dwóch informatyków wyszło z biura między $t = 1$ a $t = 2$, a z drugiej strony w $t = 3$ powinno być ich w biurze trzech – a przecież informatyk po opuszczeniu biura do niego nie wraca.

Taki problem też łatwo wykryć. Możemy przeglądać zeznania chronologicznie i pamiętać, ilu informatyków ostatnio było widzianych w biurze i ilu wiemy na pewno, że już wyszło. Suma tych dwóch liczb nie może przekroczyć łącznej liczby informatyków. Gdy rozważamy dane zeznanie, jeśli wymaga ono zwiększenia liczby informatyków w biurze, to zwiększamy ją, a jeśli wymaga zmniejszenia tej liczby, to odejmujemy odpowiednią wartość od liczby informatyków przebywających obecnie w biurze i dodajemy tę samą wartość do liczby informatyków, którzy już biuro opuścili. Uzasadnimy teraz, że jeśli przejrzymy wszystkie zdarzenia i nigdy nie przekroczymy liczby informatyków, to ich zeznania faktycznie są niesprzeczne.

Rozwiązaniem nazwiemy przypisanie każdemu z informatyków godziny przyjścia i wyjścia z biura. Powiemy, że rozwiązanie jest *poprawne*, jeśli zgadza się ze wszystkimi zeznaniami, które mamy do dyspozycji (w tej części oznacza to tyle, że w chwili dotyczącej dowolnego zeznania liczba informatyków się zgadza; dalej będziemy również sprawdzać, że każdy informatyk jest obecny w każdej chwili, dla której składa zeznanie).

Zauważmy teraz, że algorytm podany powyżej odpowiada zachłannej konstrukcji rozwiązania – zakładamy, że pomiędzy dwoma kolejnymi zeznaniami t_1 i t_2, jeśli $i_1 > i_2$, to wychodzi $i_1 - i_2$ informatyków, a jeśli $i_1 < i_2$, to przychodzi $i_2 - i_1$ informatyków. Zauważmy, że jest to jedyne możliwe poprawne rozwiązanie, jeśli tylko założymy, że w żadnej przerwie pomiędzy zeznaniami nie zachodzi sytuacja, w której pewien informatyk wychodzi z biura, a inny informatyk przychodzi do biura.

Załóżmy zatem na chwilę, że mamy poprawne rozwiązanie, w którym pomiędzy pewnymi kolejnymi zeznaniami t_1 i t_2 wychodzi pewien informatyk A, zaś przychodzi informatyk B. Nazwiemy taką sytuację *zmianą warty*. Rozważmy dowolną zmianę warty i zmodyfikujmy nasze rozwiązanie, przedłużając obecność w pracy informatyka A aż do momentu, w którym pierwotnie B wychodził z pracy, i przyjmując, że B

do pracy nie przyszedł w ogóle. W ten sposób otrzymamy rozwiązanie, w którym w pracy jest o jeden mniej informatyk i w chwili dotyczącej każdego zeznania liczba informatyków jest taka sama jak w oryginalnym rozwiązaniu (a zatem nowe rozwiązanie jest również poprawne). Powtarzajmy tę procedurę dopóty, dopóki w rozwiązaniu będą jakieś zmiany warty. Ponieważ każda modyfikacja powoduje, że liczba informatyków, którzy przyszli do pracy, maleje o jeden, tak więc w końcu musimy osiągnąć poprawne rozwiązanie, w którym nie ma żadnej zmiany warty. Zatem, o ile istnieje jakiekolwiek poprawne rozwiązanie, istnieje też rozwiązanie bez zmian warty – czyli takie, które zostanie znalezione przez nasz algorytm.

Rozróżnialni informatycy

Spójrzmy, jak rozróżnianie informatyków, spowodowane przez informacje drugiego typu, komplikuje nam życie. Przykładowo, jeśli informatyków jest trzech i złożyli oni pięć zeznań, z których wynika, że w chwilach $t = 1, 2, 4, 5$ w biurze jest po jednym informatyku, zaś w chwili $t = 3$ są wszyscy trzej, to ten układ zeznań jest niesprzeczny. Ale jeśli wiemy, że zeznanie dla chwili $t = 1$ składał informatyk A, zaś dla $t = 2$ – informatyk B, to zeznania już są sprzeczne. Istotnie, informatyk A musiał wyjść z biura przed chwilą $t = 2$, a zatem nie mógł być obecny w chwili $t = 3$.

Gdyby informatyków było niewielu, moglibyśmy pokusić się o rozwiązanie oparte o programowanie dynamiczne, w którym pamiętalibyśmy, jaki zbiór informatyków obecnie jest w biurze. Takie rozwiązanie może jednak zadziałać tylko dla bardzo małych n ($n = 30$ będzie już zbyt duże). Niestety, w naszym zadaniu przyszło nam rozwiązywać zagadkę zbrodni w dużej korporacji informatycznej, a nie w świeżo założonej firmie, a więc musimy wymyślić coś sprytniejszego. Spróbujmy zatem znaleźć rozwiązanie oparte na podobnej argumentacji co poprzednio, które umożliwi nam stwierdzenie, kto z biura wychodzi, a kto do niego przychodzi.

Zmiany warty

W tej wersji zadania poprawne rozwiązanie może wymagać zmiany warty. Przykładowo, jeśli informatyk A zeznał, że w chwili t_1 był w pracy sam, a informatyk B zeznał, że w chwili t_2 był w pracy sam, to pomiędzy chwilami t_1 a t_2 będzie musiała nastąpić zmiana warty. Zauważmy, że ta zmiana warty jest wymuszona tym, że B *musiał* przyjść do pracy, żeby zeznawać. Dodajmy zatem do zbioru rozważanych zdarzeń zdarzenia postaci „informatyk X zeznaje po raz pierwszy". Te zdarzenia będziemy rozpatrywali przed zdarzeniami dotyczącymi liczby informatyków w pracy. W momencie takiego zdarzenia, jeśli X jeszcze nie był w pracy, to dodajemy go do zbioru informatyków, którzy już przyszli.

Analogią faktu stosowanego w poprzednim rozwiązaniu jest następujące stwierdzenie. Jeśli istnieje jakiekolwiek poprawne rozwiązanie, to istnieje takie, w którym:

- Bezpośrednio przed zdarzeniem „informatyk X zeznaje po raz pierwszy" być może do pracy nie przyszedł nikt, jeśli X w pracy już był, lub tylko X, jeśli X w pracy nie było. Nikt bezpośrednio przed tym zdarzeniem nie wychodzi z pracy.

- Bezpośrednio przed zdarzeniem „w pracy jest i informatyków" do pracy może albo przyjść pewna liczba informatyków, albo może z niej wyjść pewna liczba informatyków, ale nie może być tak, że jednocześnie pewien informatyk przychodzi, a inny wychodzi.

Uzasadnienie jest podobne jak poprzednio – każde zdarzenie nieuwzględnione powyżej możemy przesunąć do przodu w czasie. Przykładowo, jeśli jakiś informatyk inny niż X przychodzi do pracy lub wychodzi z niej bezpośrednio przed zdarzeniem „informatyk X zeznaje po raz pierwszy", to poprawne rozwiązanie otrzymamy również, jeśli jego przyjście bądź wyjście przesuniemy bezpośrednio za to zdarzenie. Podobnie możemy przesunąć na później każdą zmianę warty następującą bezpośrednio przed liczeniem informatyków.

Kto wychodzi, kto przychodzi?

Wiemy zatem, kiedy informatycy przychodzą do pracy (albo bezpośrednio przed swoim pierwszym zeznaniem, albo przed liczeniem) i kiedy wychodzą (bezpośrednio przed liczeniem). W przypadku liczenia nie wiemy jednak, którzy informatycy przychodzą i wychodzą.

Wychodzenie jest prostsze. W momencie, w którym ktoś musi wyjść z pracy, żeby zmniejszyć liczbę obecnych informatyków, nie może to być nikt, kto jeszcze musi złożyć zeznanie. Wszyscy pozostali informatycy są z naszego punktu widzenia nierozróżnialni. Dokładniej: jeśli mamy poprawne rozwiązanie i przed pewnym liczeniem mamy w pracy x informatyków, którzy już nie będą składać zeznań, to otrzymamy również poprawne rozwiązanie, jeśli dowolnie przemieszamy godziny wyjścia tych x informatyków – poprawność gwarantuje to, że liczebności w żadnym momencie się nie zmienią, a informacji o zeznaniach składanych przez tych informatyków w przyszłości już nie ma.

Kluczem do rozwiązania zadania będzie spostrzeżenie, że możemy założyć, że informatycy przychodzą do pracy w tej samej kolejności, w której składają pierwsze zeznanie. Załóżmy, że mamy rozwiązanie R_1, w którym informatyk A składa pierwsze zeznanie przed pierwszym zeznaniem informatyka B, ale przychodzi do pracy po B. Wykażemy, że zamieniając godziny przyjścia do pracy informatyków A i B, dostaniemy rozwiązanie R_2, które również jest poprawne.

Po pierwsze, zauważmy, że godzina przybycia A do pracy w R_1 jest nie późniejsza niż pierwsze zeznanie A, które jest przed pierwszym zeznaniem B, zaś godzina przybycia B do pracy w R_1 jest jeszcze wcześniejsza niż godzina przyjścia do pracy A. Zatem w R_2 wszyscy informatycy są obecni w pracy w momentach, w których zeznają. W szczególności, każdy z informatyków przychodzi w R_2 do pracy wcześniej, niż z niej wychodzi (tutaj wykorzystujemy fakt, że B również złożył zeznanie; gdyby B nie złożył żadnego zeznania, to mogłoby się zdarzyć, że nasza zamiana byłaby niepoprawna). Po drugie, w każdej chwili liczba informatyków w biurze jest taka sama w obu rozwiązaniach, a zatem skoro R_1 było poprawne, to i R_2 jest poprawne.

Zatem, pomijając informatyków, którzy nie złożyli żadnych zeznań, możemy założyć, że informatycy przychodzą do biura w kolejności zgodnej z kolejnością składania pierwszego zeznania.

Pozostała kwestia informatyków, którzy w ogóle nie składali zeznań. Wiemy, że mogą oni wychodzić z pracy w dowolnym momencie. Pytanie, na które jeszcze nie znamy odpowiedzi, to – gdy wiemy, że jakiś informatyk musi przyjść do pracy – czy wybrać tego, który zeznanie złożył najwcześniej, czy jednego z tych, którzy w ogóle nie złożyli zeznania. Jak się jednak wkrótce okaże – nie będziemy musieli na to pytanie odpowiadać.

Rozwiązanie wzorcowe

W rozwiązaniu wzorcowym będziemy rozpatrywali chronologicznie trzy typy zdarzeń: „informatyk X zeznaje po raz pierwszy", „informatyk X zeznaje po raz ostatni" oraz „w pracy jest i informatyków". Podobnie jak pierwsze zeznanie danego informatyka rozpatrywaliśmy przed zdarzeniem „w pracy jest i informatyków" związanym z tym zeznaniem, to jego ostatnie zeznanie będziemy rozpatrywali po tym zdarzeniu.

Podzielimy naszych informatyków na pięć grup:

- *Śpiący* – informatycy, którzy na pewno jeszcze nie pojawili się w pracy.
- *Zeznający* – informatycy, którzy na pewno są w pracy, gdyż jesteśmy pomiędzy ich pierwszym a ostatnim zeznaniem.
- *Przed* – informatycy, którzy są w pracy, ale nie zeznawali (dlatego, że ich pierwsze zeznanie jeszcze nie nastąpiło albo że w ogóle nie zeznają).
- *Po* – informatycy, którzy jeszcze są w pracy, ale już złożyli ostatnie zeznanie (ta grupa nie zawiera informatyków, którzy w ogóle nie składają zeznań).
- *Wyszli* – informatycy, którzy już opuścili pracę.

W każdym momencie każdy z informatyków należy do dokładnie jednej powyższych pięciu grup. W rozwiązaniu wzorcowym nie będziemy śledzić, który z informatyków należy do której grupy, ale w każdej chwili będziemy znali *liczność* każdej z grup.

Spójrzmy, co powyższe rozważania mówią nam o przejściach informatyków między grupami w chwilach poszczególnych zdarzeń:

- W momencie zliczania informatyków, jeśli obecnie jest ich w pracy za mało, to ktoś będzie musiał przyjść do pracy – a zatem pewna liczba informatyków przejdzie z grupy *Śpiący* do grupy *Przed* (tak więc zmniejszamy liczność grupy *Śpiący* i zwiększamy liczność grupy *Przed*).
- Jeśli informatyków jest w momencie zliczania za dużo, to ktoś musi z pracy wyjść. Kandydatami są informatycy z grupy *Po* oraz pewna (nieznana nam) liczba informatyków, którzy znaleźli się w grupie *Przed*, bo nie składają żadnych zeznań. Wiemy, że poprawność rozwiązania *nie* zależy od tego, których z nich wybierzemy, więc zacznijmy od informatyków z grupy *Po*, bo wiemy, że każdy w tej grupie jest kandydatem do wyjścia z pracy. Jeśli ta grupa jest już pusta, to (o ile w ogóle istnieje poprawne rozwiązanie) w grupie *Przed* muszą znajdować się informatycy, którzy w ogóle nie składają zeznań. Możemy zatem usunąć ich z grupy *Przed*, zmniejszając liczność tej grupy.

- W momencie zdarzenia „informatyk X zeznaje po raz pierwszy” mamy dwie możliwości – albo informatyk, o którego chodzi, już był w pracy (wtedy przesuwamy go z grupy *Przed* do grupy *Zeznający*), albo nie (wtedy przesuwamy go z grupy *Śpiący* do grupy *Zeznający*). Za chwilę uzasadnimy, że jeśli tylko grupa *Przed* jest niepusta, to powinniśmy zmniejszyć liczność tej właśnie grupy.

- Najprostsze jest radzenie sobie ze zdarzeniem „informatyk X zeznaje po raz ostatni” – wystarczy przełożyć jednego informatyka z grupy *Zeznający* do grupy *Po*.

Przy założeniu, że udowodnimy ostatni brakujący fakt, algorytm rozwiązania zadania jest już jasny. Dla każdego informatyka zapamiętujemy jego najwcześniejsze i najpóźniejsze zeznanie, a następnie tworzymy zbiór zdarzeń i przechodzimy przez te zdarzenia chronologicznie. Początkowo grupa *Śpiący* zawiera n informatyków, a pozostałe grupy są puste. W każdym momencie pamiętamy liczność każdej z pięciu grup i poprawiamy ją według powyższych zasad.

Mogą wystąpić trzy przypadki, które oznaczają, że rozwiązanie nie istnieje:

- Jeśli w momencie zliczania informatyków chcemy dodać informatyka, a grupa *Śpiący* jest już pusta, lub

- jeśli w momencie pierwszego zeznania chcemy dodać informatyka, a zarówno grupa *Przed* jak i grupa *Śpiący* są puste, lub

- jeśli w momencie zliczania chcemy usunąć informatyka, a zarówno grupa *Przed* jak i grupa *Po* są puste.

Jeśli nasz algorytm zakończy się sukcesem, to łatwo będzie nam wskazać faktycznie poprawne rozwiązanie. Trudniejsze jest uzasadnienie, że jeśli istnieje poprawne rozwiązanie, to nasz algorytm je znajdzie. Kluczowe tu jest sprawdzenie, że możemy założyć, że jeśli przekładamy jakiegoś informatyka do grupy *Zeznający*, a grupa *Przed* jest niepusta, to możemy go przełożyć z grupy *Przed*. Zauważmy jednak, że jeśli mamy rozwiązanie, a w nim informatyka B, który nie składa żadnych zeznań, przychodzi do pracy przed informatykiem A, który składa zeznania, i B jest wciąż obecny w momencie, gdy A przychodzi do pracy, to możemy zamienić godziny przyjścia do pracy informatyków A i B i wciąż mieć poprawne rozwiązanie (a następnie przesunąć moment przyjścia B do pracy jeszcze później, po zdarzeniu „informatyk A zeznaje po raz pierwszy”). Możemy zatem faktycznie założyć, że jeśli ktokolwiek jest w grupie *Przed*, to jest tam między innymi informatyk, który zaczął teraz zeznawać. Zatem faktycznie, jeśli tylko istnieje poprawne rozwiązanie, to nasz algorytm je znajdzie.

Łączny czas działania wzorcowego algorytmu to $O((n + m) \log m)$ – w każdym z $O(\log m)$ kroków wyszukiwania binarnego będziemy musieli obliczyć w czasie $O(n + m)$ godziny pierwszego i ostatniego zeznania każdego informatyka, następnie w czasie $O(m)$ posortować kubełkowo wszystkie $O(m)$ zdarzeń w porządku chronologicznym i w czasie $O(m)$ przejść przez nie, aktualizując liczności grup informatyków.

Rozwiązanie wzorcowe zaimplementowano w plikach `ins.cpp` oraz `ins1.pas`.

Testy

Przygotowanie testów okazało się zadaniem nietrywialnym. W przypadku losowych testów przeróbki rozwiązania wzorcowego zawierające pojedynczy błąd przechodziły zdecydowaną większość testów.

Z tego powodu konieczne było umieszczenie wielu przypadków testowych w jednym pliku oraz bardziej przemyślany sposób generowania testów. Dużej liczby przypadków testowych nie dało się uniknąć, gdyż każdy test sprawdza co najwyżej jedną ścieżkę w rozwiązaniu – tę odpowiedzialną za pierwszą napotkaną sprzeczność. Stworzono 13 testów składających się z 50 przypadków testowych każdy.

Testy generowane były w sposób losowy z zapewnieniem następujących warunków: żądana liczba informatyków, żądana liczba zeznań, żądana liczba chwil, w których zostało złożone przynajmniej jedno zeznanie, oraz ograniczenie dolne na wynik.

Dodatkowo zapewniono, że w niektórych testach bardzo wiele zeznań złożonych jest w jednej chwili oraz że istnieje informatyk, który złożył bardzo wiele zeznań. Ponadto w większości testów wszystkie zeznania zgadzają się co do liczby osób, które były w biurze w poszczególnych chwilach. W części testów dodano informatyków, którzy nie składają żadnego zeznania.

Jacek Tomasiewicz — Treść zadania, Opracowanie
Karol Pokorski — Program

Dostępna pamięć: 128 MB. *OI, etap II, dzień pierwszy, 13.02.2013*

Łuk triumfalny

Król Bajtocji, Bajtazar, wraca po zwycięskiej bitwie do swojego kraju. W Bajtocji jest n miast połączonych zaledwie $n-1$ drogami. Wiadomo, że z każdego miasta da się dojechać do każdego innego dokładnie jedną trasą, złożoną z jednej lub większej liczby dróg. (Inaczej mówiąc, sieć dróg tworzy **drzewo***).*

Król właśnie wkroczył do stolicy Bajtocji. W mieście tym postawiono łuk triumfalny, czyli bramę, pod którą przejeżdża król po odniesieniu zwycięstwa. Bajtazar, zachwycony przyjęciem przez poddanych, zaplanował pochód triumfalny, w którym odwiedzi wszystkie miasta Bajtocji, zaczynając z miasta, w którym aktualnie przebywa.

Pozostałe miasta nie są przygotowane na przyjazd króla – nie zostały w nich jeszcze postawione łuki triumfalne. Nad budową łuków czuwa doradca Bajtazara. Chce on zatrudnić pewną liczbę ekip budowlanych. Każda ekipa codziennie może wybudować jeden łuk, w dowolnie wybranym mieście. Niestety nikt nie wie, w jakiej kolejności król będzie odwiedzał miasta. Wiadomo jedynie, że każdego dnia król przemieści się z miasta, w którym się aktualnie znajduje, do sąsiedniego miasta. Król może odwiedzać poszczególne miasta dowolnie wiele razy (jednak wystarczy mu jeden łuk triumfalny w każdym mieście).

Doradca Bajtazara musi zapłacić każdej ekipie budowlanej tyle samo, bez względu na to, ile zbuduje łuków triumfalnych. Z jednej strony, doradca chce mieć pewność, że każde miasto aktualnie odwiedzane przez króla będzie miało łuk triumfalny. Z drugiej jednak strony chciałby zatrudnić jak najmniej ekip budowlanych. Pomóż mu i napisz program, który pomoże wyznaczyć minimalną konieczną liczbę ekip budowlanych.

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą n ($1 \leqslant n \leqslant 300\,000$) oznaczającą liczbę miast w Bajtocji. Miasta są ponumerowane od 1 do n, przy czym stolica Bajtocji ma numer 1.

Sieć dróg Bajtocji jest opisana w kolejnych $n-1$ wierszach. Każdy z tych wierszy zawiera dwie liczby całkowite a, b ($1 \leqslant a, b \leqslant n$) oddzielone pojedynczym odstępem, oznaczające, że miasta a i b są połączone bezpośrednio dwukierunkową drogą.

W testach wartych łącznie 50% punktów zachodzi dodatkowy warunek $n \leqslant 10\,000$.

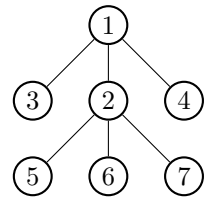
Wyjście

Pierwszy i zarazem jedyny wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą równą minimalnej liczbie ekip budowlanych, które powinien zatrudnić doradca Bajtazara.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*

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



Wyjaśnienie do przykładu: *W pierwszym dniu należy wybudować łuki triumfalne w miastach 2, 3, 4. Później wystarczy je zbudować w miastach 5, 6, 7.*

Testy „ocen":

`1ocen`: $n = 8\,191$, *miasta tworzą pełne drzewo binarne o wysokości 12;*

`2ocen`: $n = 300\,000$, *miasta* $1, \ldots, n$ *są ułożone wzdłuż jednej ścieżki;*

`3ocen`: $n = 5$, *prosty test poprawnościowy;*

`4ocen`: $n = 6$, *prosty test poprawnościowy;*

`5ocen`: $n = 10\,000$, *dziewięć długich ścieżek zwisających z korzenia.*

Rozwiązanie

Zadanie można rozpatrywać jako rodzaj gry rozgrywającej się na drzewie, którego wierzchołki oznaczać będą miasta Bajtocji, zaś krawędzie – drogi pomiędzy miastami. W grze tej pierwszym graczem jest król, a drugim – ekipy budujące łuki triumfalne.

Celem pierwszego gracza jest wejście do wierzchołka drzewa, w którym nie ma łuku triumfalnego, natomiast celem drugiego gracza jest niedopuszczenie do takiej sytuacji. Król może w ciągu jednego ruchu przemieścić się do sąsiedniego wierzchołka, natomiast każda ekipa budowlana może w jednym ruchu wybudować łuk triumfalny w jednym, dowolnym wierzchołku.

Zadanie polega na wyznaczeniu minimalnej liczby ekip budowlanych, które mają strategię wygrywającą w tej grze (czyli taką, że król nie jest w stanie wejść do wierzchołka bez łuku triumfalnego).

Strategie graczy

Ukorzeńmy drzewo w wierzchołku reprezentującym stolicę Bajtocji. Zauważmy, że jeśli król może wygrać – tzn. niezależnie od działań ekip budowlanych istnieje trasa króla kończąca się w pewnym wierzchołku v, taka że w chwili wejścia króla do v nie stoi tam łuk triumfalny – to może to równie dobrze zrobić, poruszając się jedynie w dół

drzewa – czyli wybierając najkrótszą ścieżkę z korzenia do wierzchołka v. Faktycznie, gdyby na ścieżce króla z drzewa do v jakiś wierzchołek w występował dwukrotnie, to po usunięciu fragmentu ścieżki między tymi dwoma wystąpieniami król byłby tylko w lepszej sytuacji (bo ekipy budowlane miałyby mniej czasu na budowanie łuków). Zatem jeśli stwierdzimy, że ekipy budowlane są w stanie wygrywać do momentu wejścia króla do liścia drzewa, to będziemy wiedzieli, że są już w stanie wygrywać do końca gry.

Zauważmy, że jeśli król znajduje się w pewnym wierzchołku v, to przed jego kolejnym ruchem we wszystkich synach wierzchołka v muszą zostać wybudowane łuki triumfalne. To sugeruje następującą strategię wygrywającą dla ekip budowlanych: po ruchu króla w dół drzewa do wierzchołka v, budowane są łuki triumfalne we wszystkich synach wierzchołka v. Jeśli K jest maksymalną liczbą synów, które posiada pewien wierzchołek drzewa, to ta strategia wymaga K ekip budowlanych.

Niestety, K nie musi być minimalną liczbą potrzebnych ekip budowlanych. Jeśli w teście przykładowym doczepilibyśmy dwa nowe wierzchołki do wierzchołka 2, to posiadałby on $K = 5$ synów. Do wygrania z królem wystarczą natomiast 4 ekipy (w pierwszym dniu budują one łuki triumfalne w wierzchołkach 2, 3, 4 i 5; w drugim dniu w pozostałych wierzchołkach).

Rozwiązanie stosujące tę strategię znajduje się w pliku `lukb1.cpp`. Nie otrzymuje ono żadnych punktów.

Rozwiązanie wzorcowe

Podstawowa obserwacja jest następująca: jeśli k ekip budowlanych może wygrać z królem, to tym bardziej uda się to $k+1$ ekipom. Dzięki tej obserwacji, wartość k możemy wyszukiwać binarnie. Jedyne, co pozostaje, to odpowiadać na pytania: czy mając do dyspozycji ustaloną liczbę k ekip budowlanych, można wygrać niezależnie od ruchów króla.

W rozwiązaniu wykorzystamy metodę programowania dynamicznego. Dla każdego wierzchołka v obliczymy następującą wartość:

$dp[v]$ – minimalna, niezbędna liczba wcześniej wybudowanych łuków triumfalnych w poddrzewie zaczepionym w wierzchołku v, która zapewnia strategię wygrywającą, jeśli król właśnie wchodzi do wierzchołka v, w którym jest już łuk triumfalny.

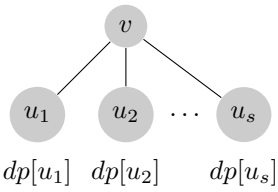
Nietrudno zauważyć, że po wyznaczeniu tych wartości, będzie można bezpośrednio odpowiedzieć na pytanie o strategię wygrywającą. Wystarczy bowiem odczytać wartość z wierzchołka o numerze 1, w którym znajduje się początkowo król:

- jeśli $dp[1] > 0$, to należałoby mieć wcześniej jakieś wybudowane łuki, a takich nie mamy, więc wtedy ekipy budowlane nie mają strategii wygrywającej,
- jeśli $dp[1] = 0$, to ekipy budowlane mają strategię wygrywającą.

Wyliczając wartości, poruszamy się od liści w kierunku korzenia. Jeśli wierzchołek v jest liściem, to nie ma żadnych innych wierzchołków w jego poddrzewie, zatem nie potrzeba tam budować innych łuków triumfalnych i $dp[v] = 0$.

Dla wierzchołka wewnętrznego v wyznaczamy $dp[v]$ na podstawie wartości w jego synach, których zbiór oznaczamy przez $synowie[v] = \{u_1, u_2, \ldots, u_s\}$ (patrz rys. 1). Dla każdego syna u_i musimy wybudować $dp[u_i]$ łuków triumfalnych w jego poddrzewie, a oprócz tego również łuk w samym wierzchołku u_i. Potrzebujemy zatem w sumie wybudować $\sum_{i=1}^{s}(dp[u_i]+1)$ łuków triumfalnych. W najbliższym ruchu k ekip budowlanych może wybudować co najwyżej k łuków triumfalnych, zatem liczba łuków, które muszą być zbudowane przed tym ruchem, jest równa:

$$dp[v] = \max\big(0,\ \sum_{i=1}^{s}(dp[u_i]+1) - k\big).$$



Rys. 1: Wyznaczanie wartości dp na podstawie wartości w synach $S = \{u_1, u_2, \ldots, u_s\}$.

Gdyby liczba wybudowanych łuków była mniejsza niż $dp[v]$, oznaczałoby to, że w którymś z synów wierzchołka v nie wybudowaliśmy łuku triumfalnego lub wchodząc do któregoś z synów, nie mamy wystarczająco dużo wybudowanych wcześniej łuków i tam król ma strategię wygrywającą. Król wybierze taki wierzchołek, jeśli tylko będzie mógł.

Implementacja

Rozwiązanie najprościej zaimplementować rekurencyjnie. Oto schematyczny zapis funkcji sprawdzającej:

```
1: function sprawdź(v, k)
2: begin
3:   suma := 0;
4:   foreach u ∈ synowie[v] do
5:     suma := suma + sprawdź(u, k) + 1;
6:   return max(0, suma − k);
7: end
```

W czasie implementacji zwykle nie przechowujemy oddzielnie wszystkich synów danego wierzchołka, a trzymamy zbiór wszystkich jego sąsiadów. Zawiera on także ojca tego wierzchołka w drzewie, powinniśmy więc umieć stwierdzać, czy analizowany sąsiad nie jest ojcem w drzewie. Najłatwiej to zrobić, przekazując do funkcji numer ojca jako argument:

```
1: function sprawdź(v, k, ojciec)
2: begin
3:   suma := 0;
4:   foreach u ∈ sasiedzi[v] do
5:     if u ≠ ojciec then
6:       suma := suma + sprawdź(u, k, v) + 1;
7:   return max(0, suma − k);
8: end
```

Złożoność czasowa takiej funkcji sprawdzającej jest liniowa względem liczby wierzchołków drzewa. Wobec tego cały algorytm, korzystający z wyszukiwania binarnego, działa w czasie $O(n \log n)$. Na tej zasadzie zostało zaimplementowane rozwiązanie wzorcowe, znajdujące się w pliku `luk.cpp` oraz `luk1.pas`.

Rozwiązanie wolne

Można było nie zauważyć możliwości wyszukiwania binarnego i uzyskać algorytm $O(n^2)$, sprawdzając kolejno każdą możliwą liczbę ekip budowlanych. Rozwiązanie takie otrzymywało około 50% punktów, a jego implementacja znajduje się w pliku `luks1.cpp` oraz `luks2.pas`.

Testy

W zestawie były cztery typy testów: małe testy poprawnościowe wygenerowane ręcznie; testy generowane losowo, w których wyniki są często bardzo małe; testy z długimi ścieżkami z małą częścią losową oraz testy przypominające drzewa d–arne (każdy wierzchołek albo jest liściem, albo ma kilku synów).

Bartłomiej Dudek
Treść zadania, Opracowanie

Michał Adamczyk
Program

Dostępna pamięć: 128 MB.

OI, etap II, dzień drugi, 14.02.2013

Konewka

Zostałeś ogrodnikiem u królowej Bajtoliny. Wspaniale, prawda? Skoro tak uważasz, to chyba jeszcze nie wiesz wszystkiego o tej pracy. Obok zamku królowej znajduje się wielki ogród z n drzewami ustawionymi po kolei jedno za drugim. To jeszcze nic strasznego, ale czy potrafisz o każdej porze dnia i nocy odpowiedzieć swojej władczyni, które z jej drzewek są teraz dojrzałe? Zakładamy, że drzewo jest dojrzałe, gdy ma przynajmniej k bajtymetrów wysokości.

Czasem królowa prosi Cię, abyś niektóre z jej drzewek podlał za pomocą magicznej konewki. Każda taka operacja powoduje, że wszystkie podlane drzewa rosną o dokładnie jeden bajtymetr.

Udowodnij, że nadajesz się do tej pracy i szybko odpowiedz na wszystkie pytania królowej!

Komunikacja

Napisz bibliotekę komunikującą się z programem oceniającym. Powinna ona zawierać przynajmniej następujące trzy funkcje, wywoływane przez program oceniający:

- `inicjuj(n, k, D)` – *ta funkcja zostanie wywołana dokładnie raz, na początku sprawdzania. Możesz ją wykorzystać do inicjalizacji swoich struktur danych. Przyjmuje jako parametry liczbę drzew n, dolne ograniczenie wysokości dojrzałego drzewa k oraz tablicę D o długości n zawierającą początkowe wysokości wszystkich drzew. Drzewa są ponumerowane kolejnymi liczbami całkowitymi od 0 do $n-1$.*
  - **C/C++:** `void inicjuj(int n, int k, int *D);`
  - **Pascal:** `procedure inicjuj(n, k : LongInt; var D : array of LongInt);`
- `podlej(a, b)` – *oznacza, że królowa poprosiła Cię o podlanie wszystkich drzew od a-tego do b-tego włącznie ($0 \leqslant a \leqslant b \leqslant n-1$). Wywołanie tej funkcji oznacza, że każde z tych drzew rośnie o 1 bajtymetr.*
  - **C/C++:** `void podlej(int a, int b);`
  - **Pascal:** `procedure podlej(a, b : LongInt);`
- `dojrzale(a, b)` – *królowa pyta Cię, ile spośród drzew o numerach od a do b włącznie ($0 \leqslant a \leqslant b \leqslant n-1$) jest już dojrzałych.*
  - **C/C++:** `int dojrzale(int a, int b);`
  - **Pascal:** `function dojrzale(a, b : LongInt) : LongInt;`

Twoja biblioteka **nie może** *czytać żadnych danych (ani ze standardowego wejścia, ani z plików).* **Nie może** *również nic wypisywać do plików ani na standardowe wyjście. Może pisać na standardowe wyjście diagnostyczne (*`stderr`*) – pamiętaj jednak, że zużywa to cenny czas.*

Jeśli piszesz w C/C++, Twoja biblioteka **nie może** *zawierać funkcji* `main`*. Jeśli piszesz w Pascalu, powinieneś dostarczyć moduł (patrz przykładowe programy na Twoim dysku).*

Twoje rozwiązanie uzyska punkty za dany test tylko wtedy, gdy spełni określone powyżej wymogi techniczne oraz odpowie poprawnie na wszystkie pytania królowej.

Ograniczenia

We wszystkich testach zachodzą następujące warunki:

- $1 \leqslant n \leqslant 300\,000$,
- $1 \leqslant k \leqslant 10^9$,
- *łączna liczba wywołań funkcji* `podlej` *i funkcji* `dojrzale` *nie przekracza 300 000*,
- *początkowe wysokości wszystkich drzew to dodatnie liczby całkowite nieprzekraczające* 10^9.

W testach wartych łącznie 20% punktów zachodzą dodatkowe warunki:

- $n \leqslant 2000$,
- *łączna liczba wywołań funkcji* `podlej` *i funkcji* `dojrzale` *nie przekracza 10 000*.

W testach wartych łącznie 50% punktów wszystkie wywołania funkcji `podlej` *następują przed wszystkimi wywołaniami funkcji* `dojrzale`.

Kompilacja

Twoja biblioteka – `kon.c`, `kon.cpp` *lub* `kon.pas` *– zostanie skompilowana z programem oceniającym przy użyciu następujących poleceń:*

- **C:** `gcc -O2 -static -lm kon.c kongrader.c -o kon`
- **C++:** `g++ -O2 -static -lm kon.cpp kongrader.cpp -o kon`
- **Pascal:**
```
ppc386 -O2 -XS -Xt kon.pas
ppc386 -O2 -XS -Xt kongrader.pas
mv kongrader kon
```

Przykładowe wykonanie

Poniższa tabela zawiera przykładowy ciąg wywołań funkcji oraz poprawne wyniki funkcji `dojrzale`.

Wywołanie	Wynik	Wyjaśnienie
`inicjuj(4, 6, D)` (`D[0]`=5, `D[1]`=4, `D[2]`=3, `D[3]`=7)		*Mamy $n = 4$ drzew o wysokościach 5, 4, 3 i 7, a $k = 6$.*
`dojrzale(2, 3)`	*1*	*Ile dojrzałych drzew jest na przedziale $[2, 3]$?*
`podlej(0, 2)`		*Podlej drzewa 0, 1 i 2.*
`dojrzale(1, 2)`	*0*	*Ile dojrzałych drzew jest na przedziale $[1, 2]$?*
`podlej(2, 3)`		*Podlej drzewa 2 i 3.*
`podlej(0, 1)`		*Podlej drzewa 0 i 1.*
`dojrzale(0, 3)`	*3*	*Ile dojrzałych drzew jest na przedziale $[0, 3]$?*

Testowanie

W katalogu `/home/zawodnik/kon/` *na dysku Twojego komputera możesz znaleźć przykładowy program oceniający (*`kongrader.c`*,* `kongrader.cpp` *i* `kongrader.pas`*). Żeby uruchomić program oceniający z Twoją biblioteką, powinieneś umieścić ją w pliku* `kon.c`*,* `kon.cpp` *lub* `kon.pas` *w odpowiednim katalogu (*`c`*,* `cpp` *lub* `pas`*). Na początku zawodów znajdziesz tam przykładowe, błędne rozwiązania tego zadania. Program oceniający wraz z Twoją biblioteką możesz skompilować za pomocą polecenia:*

```
make kon
```

które działa dokładnie tak, jak opisano w sekcji „Kompilacja". Kompilacja rozwiązania w C/C++ wymaga pliku `koninc.h`*, który również znajduje się w odpowiednich katalogach.*

Tak skompilowany program oceniający wczytuje ze standardowego wejścia specjalnie przygotowany opis testu, wywołuje odpowiednie funkcje Twojej biblioteki, a wyniki wypisuje na standardowe wyjście.

Opis testu powinien być w następującym formacie. Pierwszy wiersz testu składa się z dwóch liczb całkowitych n i k. Drugi wiersz zawiera n liczb oznaczających wysokości początkowe kolejnych drzew. Trzeci wiersz zawiera liczbę q. Dalej następuje q wierszy, z których każdy zawiera pojedynczą literę `p` *lub* `d` *oraz dwie nieujemne liczby całkowite. Litera określa, która funkcja powinna być wywołana:* `p` *dla* `podlej` *i* `d` *dla* `dojrzale`*, zaś liczby – z jakimi argumentami należy tę funkcję wywołać. Funkcja* `inicjuj` *zostanie wywołana od razu po wczytaniu pierwszych dwóch wierszy. Pamiętaj jednak, że przykładowy program oceniający nie sprawdza, czy dane są sformatowane poprawnie ani czy spełnione są ograniczenia podane w sekcji „Ograniczenia".*

W katalogu `/home/zawodnik/kon/` *znajdziesz plik* `kon0.in`*, który odpowiada przykładowemu wykonaniu programu opisanemu powyżej. Aby uruchomić program oceniający na podanym przykładzie, użyj następującego polecenia:*

```
./kon < kon0.in
```

Wyniki wywołań funkcji `dojrzale` *zostaną wypisane na standardowe wyjście. Poprawny wynik dla powyższego przykładowego wykonania znajdziesz na dysku w pliku* `kon0.out`*.*

Aby sprawdzić, czy wynik wypisany przez Twoje rozwiązanie jest prawidłowy, możesz również wysłać rozwiązanie (Twoją bibliotekę) do SIO.

Testy „ocen":

`1ocen`**:** *$n = 2000$, łączna liczba wywołań funkcji* `podlej` *i* `dojrzale` *wynosi 10 000, wszystkie te wywołania dotyczą pojedynczych drzew, początkowe wysokości drzew losowe z przedziału $[1, 100]$, $k = 100$;*

`2ocen`**:** *$n = 100\,000$, najpierw 100 000 wywołań funkcji* `podlej`*, dotyczące wszystkich drzew, potem 100 000 wywołań funkcji* `dojrzale` *o pojedyncze drzewa, początkowe wysokości drzew losowe z przedziału $[1, 1000]$, $k = 100\,500$.*

Rozwiązanie

Zanim przejdziemy do rozwiązania, opiszmy problem bardziej formalnym językiem. Dany jest ciąg liczb całkowitych długości n oraz q możliwych operacji, z których każda

jest typu `podlej` lub `dojrzale`. Dana jest także pewna stała całkowita k. Operacja `podlej`(a, b) powoduje zwiększenie o 1 wszystkich wyrazów ciągu na przedziale indeksów od a do b włącznie. Natomiast operacja `dojrzale`(a, b) jest pytaniem o liczbę wyrazów ciągu, o indeksach z przedziału od a do b włącznie, które są nie mniejsze niż k.

Zadaniem zawodnika jest przygotowanie biblioteki udostępniającej programowi oceniającemu trzy funkcje: `inicjuj`, która pozwala na przekazanie programowi zawodnika początkowych wartości, oraz `podlej` i `dojrzale` o działaniu opisanym powyżej. Zawodnik musi odpowiadać na zapytania `podlej` natychmiast, jeszcze przed poznaniem następnej operacji.

Rozwiązanie wolne

Od razu po zrozumieniu treści jesteśmy w stanie napisać rozwiązanie siłowe, które będzie wykonywało dokładnie to, co jest powiedziane w zadaniu. Dla każdego wywołania funkcji `podlej` rozwiązanie iteruje po wszystkich drzewach na przedziale $[a, b]$, zwiększając ich wysokość o jeden. Podobnie dla wywołań `dojrzale` iteruje po wszystkich drzewach na przedziale $[a, b]$, zliczając te o wysokości nie mniejszej niż k.

W pesymistycznym przypadku dla każdego z q zapytań nasze rozwiązanie będzie musiało przejrzeć wszystkie n drzew, zatem złożoność tego rozwiązania to $O(q \cdot n)$. Za takie rozwiązanie można było uzyskać 20 punktów. Implementacja tego rozwiązania znajduje się w plikach `kons1.cpp` i `kons2.pas`.

Rozwiązanie wzorcowe

Rozwiązanie wzorcowe opiera się na spostrzeżeniu, że każde drzewo tylko raz może stać się dojrzałe, a dojrzałe drzewo pozostanie dojrzałym na zawsze.

Rozważmy wszystkie drzewa, które są w danej chwili dojrzałe, i umieśćmy je w pewnej strukturze danych (nazwijmy ją A), udostępniającej dwie operacje:

- *WyznaczNaPrzedziale*(A, a, b) – wyznaczenie liczby drzew w strukturze znajdujących się na pozycjach z przedziału $[a, b]$; dzięki tej operacji wprost zaimplementujemy funkcję `dojrzale`.
- *UstawNaPozycji*(A, poz) – wstawienie jednego drzewa na pozycję poz.

W drugiej strukturze (nazwijmy ją B) przechowywać będziemy tylko drzewa niedojrzałe. Kiedy dzieje się cokolwiek z drzewami niedojrzałymi? Oczywiście tylko przy wywołaniu funkcji `podlej`, kiedy to wszystkie drzewa rosną o jeden bajtymetr i być może niektóre z nich stają się wtedy dojrzałe. Chcemy, aby po każdej takiej operacji nasze struktury były aktualne, tzn. musimy znaleźć w strukturze B wszystkie drzewa, które właśnie stały się dojrzałe, i przenieść je do struktury A.

Do efektywnego przechowywania i modyfikowania informacji o wysokościach niedojrzałych drzew będziemy potrzebowali struktury, która umożliwia szybkie wykonanie następujących operacji:

- *ZwiekszNaPrzedziale*(B, a, b) – zwiększenie o 1 wysokości wszystkich drzew na przedziale $[a, b]$;
- *PozycjaNajwyzszego*(B, a, b) oraz *WysokoscNaPozycji*(B, poz) – wyznaczenie pozycji najwyższego drzewa na przedziale $[a, b]$ i zwrócenie wysokości tego drzewa;
- *UstawNaPozycji*(B, poz, wys) – dowolna modyfikacja wysokości jednego drzewa na pozycji *poz*.

Funkcja `podlej` może teraz wyglądać następująco:

```
1:  function podlej(a, b)
2:  begin
3:    ZwiekszNaPrzedziale(B, a, b);
4:    while true do begin
5:      poz := PozycjaNajwyzszego(B, a, b);
6:      wys := WysokoscNaPozycji(B, poz);
7:      if wys < k then break;
8:      { Będziemy przenosić drzewo z pozycji poz }
9:      UstawNaPozycji(B, poz, −∞);
10:     UstawNaPozycji(A, poz);
11:   end
12: end
```

Wybór struktur danych

Zauważmy, że **drzewo przedziałowe** (opisane dokładnie w rozwiązaniach zadań *Tetris 3D* z XIII OI [13] oraz *Koleje* z IX OI [9], a także na *Wykładach z Algorytmiki Stosowanej* `http://was.zaa.mimuw.edu.pl`) będzie doskonale spisywać się w przypadku zarówno struktury A, jak i struktury B. Wszystkie żądane operacje na odpowiednio zaimplementowanych drzewach przedziałowych będziemy mogli zrealizować w złożoności czasowej $O(\log n)$.

Uściślając: strukturę danych A można zaimplementować jako drzewo przedziałowe typu punkt–przedział (czyli wstawiamy informację w pewnym punkcie, pytamy o liczbę punktów na przedziale), a strukturę B jako drzewo przedziałowe typu przedział–przedział z wyszczególnioną parą operacji $(+, \max)$ (zwiększamy każdą liczbę na przedziale o jeden, pytamy o maksimum na przedziale) oraz dodatkową operacją umożliwiającą ustawienie dowolnej wartości dla pewnej konkretnej pozycji.

Jako strukturę danych A możemy również wykorzystać **drzewo potęgowe**, opisane w czasopiśmie *Delta* w numerze 10/2008. Ta struktura również umożliwi nam wykonanie wszystkich potrzebnych operacji w czasie $O(\log n)$.

Analiza złożoności rozwiązania wzorcowego

Dzięki temu, że odpowiedź na każde z zapytań `dojrzale` sprowadza się do wywołania jednej operacji na drzewie przedziałowym, takie zapytanie możemy przetworzyć w czasie $O(\log n)$.

Jeśli chodzi o zapytania `podlej`, to mogłoby się wydawać, że skoro dla każdego z nich możemy przejrzeć nawet n drzew, to złożoność całego rozwiązania będzie rzędu $O(n \cdot q \cdot \log n)$. Na szczęście możemy lepiej oszacować pracę związaną z tymi zapytaniami.

Skoro każde drzewo co najwyżej raz może zmienić swój stan z niedojrzałego na dojrzałe, to co najwyżej raz będziemy je usuwać ze struktury B i dodawać do struktury A. Z tego powodu, podczas działania programu wykonamy $O(n)$ operacji przeniesienia pojedynczego drzewa. Każdą z tych operacji wykonujemy w czasie $O(\log n)$, więc koszt poprawienia stanu obu struktur po wywołaniu `podlej` zamortyzuje się do $O(\log n)$, a poprawień takich możemy wykonać co najwyżej n. Ostatecznie, złożoność czasowa całego rozwiązania wyniesie $O((q+n)\log n)$. Z kolei złożoność pamięciowa wyniesie $O(n)$.

Uwagi dotyczące poszczególnych implementacji

Zauważmy, że jeśli w naszym algorytmie będziemy przenosić dojrzałe drzewa pomiędzy strukturami A i B nie po wykonaniu funkcji `podlej`, ale przed wyznaczeniem wyniku przy zapytaniu `dojrzale`, to cała wcześniejsza analiza będzie dalej poprawna. Rozwiązania wzorcowe różnią się między sobą implementacją pierwszej struktury danych oraz momentem i sposobem przenoszenia drzew:

- Rozwiązania `kon.c`, `kon1.cpp` i `kon2.pas` przenoszą wszystkie dojrzałe drzewa z przedziału $[a, b]$ na początku wykonania funkcji `dojrzale`. Rozwiązanie `kon3.c` przenosi zaś wszystkie dojrzałe drzewa z przedziału $[a, b]$ na końcu wykonania funkcji `podlej`.
- Rozwiązania `kon.c` oraz `kon1.cpp` różnią się między sobą sposobem przenoszenia drzew dojrzałych ze struktury B do struktury A. Rozwiązanie `kon.cpp` wywołuje funkcję *ZnajdzDojrzale*, która dla przedziału, w którym maksimum jest nie mniejsze niż k, dzieli przedział na dwie równe części i wywołuje się rekurencyjnie (chyba, że przedział jest wielkości 1, wtedy drzewo o pozycji w tym przedziale przenoszone jest do pierwszej struktury danych). Rozwiązanie `kon1.cpp` realizuje zaś *stricte* zaprezentowany wyżej schemat rozwiązania, tj. dopóki najwyższe drzewo na przedziale jest ma wysokość nie mniejszą niż k, przenosi to drzewo, po czym sprawdza wysokość najwyższego drzewa pozostałego po tym przeniesieniu.
- Rozwiązanie `kon2.pas` implementuje strukturę A za pomocą drzewa potęgowego.

Wszystkie powyższe rozwiązania zdobywały na zawodach maksymalną punktację.

Rozwiązanie na połowę punktów

Rozwiązanie to zakłada, że wszystkie wywołania funkcji `podlej` nastąpią przed wywołaniami `dojrzale`. W takim wypadku można wczytać wszystkie przedziały, z jakimi wywoływana jest funkcja `podlej`, a przy pierwszym wywołaniu `dojrzale` obliczyć, dla każdego drzewa, o ile sumarycznie wcześniej urosło.

Aby to zrobić, przy pierwszym wywołaniu funkcji `dojrzale` przeglądamy wszystkie przedziały i w pomocniczej tablicy zaznaczamy, ile przedziałów w danym miejscu zaczyna się lub kończy. Teraz, przeglądając wszystkie drzewa po kolei od 0 do $n-1$, możemy jednocześnie łatwo wyznaczyć liczbę „otwartych" przedziałów, czyli takich, które obejmują swoim zakresem to drzewo. Znaleziona liczba będzie odpowiadała liczbie bajtymetrów, o jaką dane drzewo urosło, i na jej podstawie możemy powiedzieć, czy jest ono dojrzałe, czy nie.

Następnie w osobnej tablicy możemy zapamiętać dla każdego drzewa i, ile jest drzew dojrzałych spośród tych o numerach od 0 do i. Mając taką tablicę, można w czasie stałym odpowiadać na każde zapytanie o liczbę drzew dojrzałych na przedziale $[a, b]$.

Rozwiązanie to zostało zaimplementowane w pliku `konb1.cpp`. Zgodnie z treścią zadania można było za nie uzyskać 50 punktów.

Rozwiązania niepoprawne

W plikach `konb2.cpp`, `konb3.cpp`, `konb4.cpp`, `konb6.cpp` są przykłady rozwiązań, w których niepoprawnie zaimplementowano struktury danych przechowujące drzewa. Rozwiązania te dostawały 0 punktów.

Rozwiązanie `konb5.cpp` to rozwiązanie prawie wzorcowe – błąd polega na przyjęciu zbyt małej wartości nieskończoności, co dla niektórych testów powoduje wielokrotne zliczenie tego samego dojrzałego drzewa. Rozwiązanie to otrzymuje 60% maksymalnej punktacji. Należało pamiętać, że w trakcie całego programu wysokość drzewa może wzrosnąć o q, tak więc za $-\infty$ trzeba przyjąć liczbę mniejszą niż $k-q$.

Po co biblioteka?

Uważny Czytelnik na pewno zadaje sobie teraz pytanie, po co w tym zadaniu zostało wymuszone odpowiadanie „on-line" na zapytania, skoro nie widać łatwego rozwiązania wersji „off-line". Okazuje się, że „off-line" można rozwiązać nawet trudniejszy problem, w którym dodatkowo dla każdego drzewa jest określona wysokość h_i, od której dane drzewo jest uznawane za dojrzałe. Tę trudniejszą wersję da się rozwiązać w niewiele gorszej złożoności czasowej $O((q+n)\cdot \log n \cdot \log H)$, gdzie H to największa dopuszczalna wysokość drzew. Zachęcamy Czytelnika do rozwiązania tego problemu. Niestety, autorzy nie mogli dopuścić do wykorzystania takiego rozwiązania, gdyż z tą techniką uczestnicy Olimpiady zetknęli się już na finale XVIII OI w zadaniu pt. *Meteory* [18].

Testy

Każda grupa testów składa się z trzech testów, z czego pierwsze dwa są w całości losowe, a trzeci nie. W pierwszym z losowych testów w każdej grupie prawdopodobieństwo wystąpienia wywołania `podlej` było takie same, jak wywołania `dojrzale`. Testy w grupach o parzystych numerach zawierają wszystkie wywołania funkcji `podlej` przed pierwszym wywołaniem funkcji `dojrzale`.

Wiktor Kuropatwa
Treść zadania, Opracowanie

Mateusz Baranowski
Program

Dostępna pamięć: 128 MB.

OI, etap II, dzień drugi, 14.02.2013

Morskie opowieści

Młody Bajtinson uwielbia przesiadywać w portowej tawernie. Często wysłuchuje tam opowieści o przygodach wilków morskich. Początkowo wierzył we wszystkie, nawet najbardziej nieprawdopodobne zasłyszane historie. Z czasem stał się jednak podejrzliwy. Postanowił napisać program, który będzie sprawdzał, czy usłyszane przez niego opowieści są w ogóle możliwe. Niestety, kiepski z niego programista. Pomóż mu!

Na wodach, po których żeglują marynarze spotykani przez Bajtinsona, znajduje się n portów oraz m szlaków żeglownych między nimi. Istnienie szlaku żeglownego łączącego dwa porty oznacza, iż możliwe jest wykonanie rejsu, który zaczyna się w jednym z nich, zaś kończy w drugim. Taki rejs jest możliwy **w obie strony**.

Bajtinson poznał k historii morskich przygód. W każdej z nich opisywany marynarz rozpoczynał podróż w jednym z portów, wykonywał pewną liczbę rejsów szlakami żeglownymi i kończył w pewnym, być może tym samym porcie. Marynarz ten mógł odbyć wiele rejsów tym samym szlakiem żeglownym, w obu kierunkach.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się trzy liczby całkowite n, m oraz k ($2 \leqslant n \leqslant 5000$, $1 \leqslant m \leqslant 5000$, $1 \leqslant k \leqslant 1\,000\,000$). Oznaczają one kolejno: liczbę portów na wodach, po których żeglują marynarze spotkani przez Bajtinsona, liczbę szlaków żeglownych oraz liczbę poznanych opowieści.

Następne m wierszy zawiera opis istniejących szlaków żeglownych. Opis pojedynczego szlaku składa się z jednego wiersza zawierającego dwie liczby całkowite oddzielone pojedynczym odstępem, a oraz b ($1 \leqslant a, b \leqslant n$, $a \neq b$), oznaczające numery portów, które łączy dany szlak.

Kolejne k wierszy zawiera opis zasłyszanych przez Bajtinsona przygód. Opis pojedynczej przygody składa się z trzech liczb całkowitych pooddzielanych pojedynczymi odstępami: s, t oraz d ($1 \leqslant s, t \leqslant n$, $1 \leqslant d \leqslant 1\,000\,000\,000$). Opis taki oznacza, iż bohater danej przygody rozpoczął ją w porcie o numerze s, zakończył w porcie o numerze t oraz wykonał w jej trakcie **dokładnie** *d rejsów.*

W testach wartych łącznie 50% punktów zachodzi dodatkowy warunek $n \leqslant 800$.

Wyjście

Twój program powinien wypisać na standardowe wyjście k wierszy; i-ty wiersz powinien zawierać słowo `TAK`*, jeżeli i-ta zasłyszana przygoda (według kolejności z wejścia) była możliwa. W przeciwnym wypadku odpowiedni wiersz powinien zawierać słowo* `NIE`.

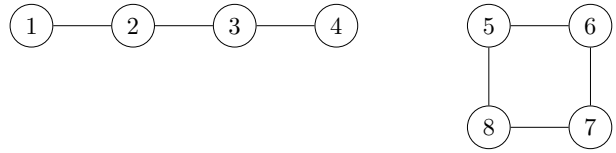
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
TAK
NIE
TAK
NIE
```



Testy „ocen”:

`1ocen`**:** *$n = 100$, $m = 4950$, każdy port połączony z każdym, milion losowych przygód, wszystkie z odpowiedzią* `TAK`*.*

`2ocen`**:** *$n = 100$, $m = 2450$, rejsy możliwe między portami o numerach o tej samej parzystości, milion losowych przygód, połowa z odpowiedzią* `TAK`*, połowa z* `NIE`*.*

Rozwiązanie

Postawiony w zadaniu problem łatwo wyrazić w języku teorii grafów. Mamy dany graf nieskierowany, którego wierzchołki reprezentują porty, zaś krawędzie – szlaki żeglowne. Ponadto danych jest k zapytań postaci: *„czy z wierzchołka s do wierzchołka t istnieje ścieżka długości dokładnie d?”*. Naszym zadaniem jest odpowiedzenie na każde z tych zapytań.

Zauważmy, że na każde zapytanie, które dotyczy wierzchołka izolowanego (tzn. takiego, z którego nie wychodzą żadne krawędzie), odpowiadamy negatywnie. W dalszych rozważaniach zakładamy zatem, że w grafie nie ma wierzchołków izolowanych.

Kluczowe obserwacje

Patrząc na limity na długość ścieżek przedstawione w zadaniu, łatwo zauważyć, iż jakiekolwiek rozwiązanie przeglądające poszukiwane ścieżki w całości nie będzie miało szans zmieścić się w limitach czasowych. Musimy zatem umieć odpowiadać na zapytania o istnienie ścieżek bez wyznaczania ich *explicite*. Z pomocą przychodzi tu poniższa obserwacja:

Lemat 1. Jeśli z wierzchołka s do wierzchołka t istnieje ścieżka długości l, to istnieją również ścieżki długości $l + 2$, $l + 4$, $l + 6$ itd.

Dowód: Rozważmy dowolną krawędź wychodzącą z wierzchołka t. Istniejącą ścieżkę z s do t długości l możemy przedłużyć o 2, przechodząc tą krawędzią raz w jedną, raz w drugą stronę. Tym samym, wykonując ten zabieg wielokrotnie, możemy przedłużyć długość danej ścieżki o dowolną parzystą wielkość. ■

Kluczową konsekwencją powyższego lematu jest to, że zbiór ścieżek między wierzchołkami s i t możemy bardzo łatwo opisać, znając dwie wartości:

- l_p – długość najkrótszej ścieżki długości parzystej między s i t oraz
- l_n – długość najkrótszej ścieżki długości nieparzystej między s i t.

Wtedy między wierzchołkami s i t istnieją ścieżki o długościach l_p, $l_p + 2$, $l_p + 4, \ldots$ oraz l_n, $l_n + 2$, $l_n + 4, \ldots$

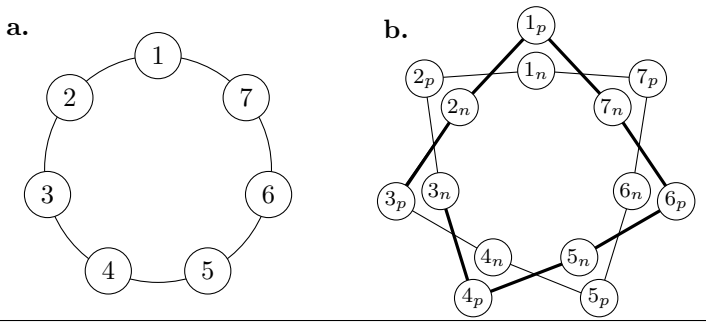
Przykładowo, na rys. 1a znajduje się graf G, będący cyklem o długości 7. Najkrótsza ścieżka o długości parzystej pomiędzy wierzchołkami 1 i 3 to $1 \to 2 \to 3$, a najkrótsza ścieżka o długości nieparzystej między nimi to $1 \to 7 \to 6 \to 5 \to 4 \to 3$. Między wierzchołkami 1 i 3 istnieją zatem ścieżki o długościach $2, 4, 6, \ldots$ oraz o długościach $5, 7, 9, \ldots$

Powyższe obserwacje są podstawą do sformułowania poniższego twierdzenia:

Twierdzenie 1. *Niech s i t będą wierzchołkami grafu, a l długością najkrótszej ścieżki o* **tej samej parzystości** *co liczba d. Odpowiedź na zapytanie „czy z wierzchołka s do wierzchołka t istnieje ścieżka długości dokładnie d?" jest twierdząca wtedy i tylko wtedy, gdy $d \geqslant l$.*

Dowód: Jeśli $d < l$, to oczywiste jest, że odpowiedź jest negatywna, gdyż l jest długością najkrótszej ścieżki o tej samej parzystości co d, więc krótsza ścieżka nie może istnieć. Jeśli zaś $d \geqslant l$, to możemy zapisać $d = l + 2k$ dla pewnej liczby naturalnej k. Między wierzchołkami s i t istnieje ścieżka długości l, zatem na mocy Lematu 1 istnieje również ścieżka długości d. ■

Twierdzenie to zachodzi też w przypadku, gdy ścieżka z s do t o szukanej parzystości długości nie istnieje. Wówczas przyjmujemy, że $l = \infty$.



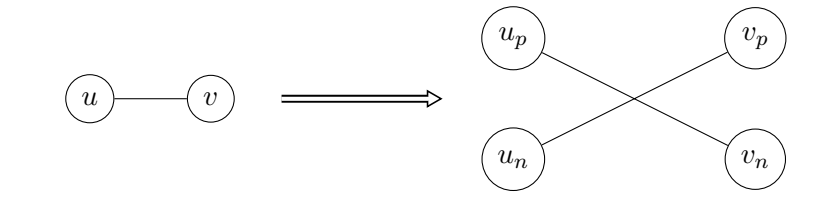
Rys. 1: **a.** Graf G będący cyklem długości 7. **b.** Graf G' uzyskany po transformacji grafu G z zaznaczonymi najkrótszymi ścieżkami długości parzystej i nieparzystej między wierzchołkami 1 i 3.

Tak więc, aby odpowiadać na zapytania z zadania, wystarczy obliczyć długość najkrótszej ścieżki długości parzystej bądź nieparzystej między wierzchołkami z zapytania i porównać ją z wartością d.

Najkrótsze ścieżki długości parzystej/nieparzystej

Potrzebujemy zatem rozwiązać następujący problem: mając dany graf nieskierowany $G = (V, E)$ i wyróżniony wierzchołek $s \in V$, należy wyznaczyć najkrótsze ścieżki długości parzystej/nieparzystej z wierzchołka s do wszystkich wierzchołków grafu.

W tym celu skonstruujmy graf nieskierowany $G' = (V', E')$. Dla każdego wierzchołka $u \in V$ dodajemy do V' dwa wierzchołki: u_p oraz u_n. Z kolei dla każdej krawędzi $uv \in E$ dodajemy do E' dwie krawędzie: $u_p v_n$ oraz $u_n v_p$; patrz też rys. 2.



Rys. 2: Transformacja krawędzi uv w grafie G w dwie krawędzie w grafie G'.

Zauważmy, iż każdej ścieżce długości parzystej z wierzchołka s do wierzchołka t w grafie G odpowiada ścieżka tej samej długości z wierzchołka s_p do wierzchołka t_p w grafie G'. Analogicznie, każdej ścieżce z wierzchołka s do wierzchołka t długości nieparzystej odpowiada ścieżka tej samej długości z wierzchołka s_p do wierzchołka t_n (patrz rys. 1b).

Ostatecznie, aby znaleźć najkrótsze ścieżki długości parzystej/nieparzystej z wierzchołka s do wszystkich pozostałych wierzchołków grafu G wystarczy skonstruować graf G', a następnie uruchomić na nim algorytm przeszukiwania wszerz (BFS) z wierzchołka s_p. Długość najkrótszej ścieżki o długości parzystej do wierzchołka t odczytujemy z odległości t_p od s_p, zaś nieparzystej z odległości t_n. Graf G' ma $2n$ wierzchołków i $2m$ krawędzi, zatem algorytm ten działa w czasie $O(n + m)$.

Rozwiązanie wzorcowe

Na początek wczytujemy wszystkie zapytania i dla każdego wierzchołka zapamiętujemy te, które go dotyczą. Teraz dla każdego wierzchołka:

(1) obliczamy wszystkie najkrótsze ścieżki długości parzystej/nieparzystej wychodzące z niego – wykorzystujemy w tym celu opisany powyżej algorytm,

(2) dysponując wyliczonymi w ten sposób wartościami, wyznaczamy odpowiedzi na zapytania dotyczące tego wierzchołka (wykorzystujemy tu Twierdzenie 1) i zapisujemy je w tablicy.

Na końcu wypisujemy odpowiedzi na zapytania w tej kolejności, w jakiej podane były na wejściu.

Zauważmy, że odpowiadając na zapytania dotyczące danego wierzchołka, potrzebujemy jedynie długości odpowiednich ścieżek wychodzących z tego wierzchołka. Jeśli spamiętamy wyniki przeszukiwań dotyczących tylko aktualnie rozważanego wierzchołka, możemy zaimplementować algorytm w złożoności pamięciowej $O(n+m+k)$.

Sumarycznie wykonamy $O(n)$ liniowych przeszukiwań grafu G', zaś odpowiedź na każde zapytanie zajmie czas stały. Ostatecznie, złożoność czasowa algorytmu wynosi więc $O(n(n+m)+k)$. Złożoność taka jest satysfakcjonująca przy limitach czasowych z zadania.

Rozwiązanie to znajduje się w plikach `mor.cpp` oraz `mor1.pas`, a także `mor2.cpp`.

Rozwiązania wolniejsze

Wykonując osobne przeszukiwanie dla każdego zapytania, otrzymujemy algorytm działający w złożoności $O(k(n+m))$. Implementacja tego rozwiązania znajduje się w plikach `mors1.cpp` oraz `mors2.pas`.

Rozwiązanie wykorzystujące potęgowanie macierzy do znalezienia ścieżki odpowiedniej długości nie wykorzystuje Twierdzenia 1 i działa w złożoności $O(kn^3 \log d)$, gdzie d to maksymalna długość ścieżki z zapytania. Zostało zaimplementowane w plikach `mors3.cpp` oraz `mors4.pas`.

Do wyznaczania długości najkrótszych ścieżek można też było użyć algorytmu Floyda-Warshalla. Uzyskane w ten sposób rozwiązanie działało w złożoności $O(n^3+k)$.

Rozwiązanie *online*

Wykonując wszystkie przeszukiwania przed wczytaniem zapytań i spamiętując ich wyniki, można było osiągnąć rozwiązanie o złożoności pamięciowej $O(n^2)$ zdolne do odpowiadania na zapytania *online*, czyli jedno zapytanie po drugim. Uważna implementacja pozwalała na zmieszczenie się w limitach pamięciowych i przejście wszystkich testów.

Testy

Rozwiązania sprawdzano na dziesięciu grupach testów. Testy *a* zawierały pojedynczą losowo wygenerowaną spójną składową. Testy *b* zawierały wiele losowych składowych, zaś *c* były zupełnie losowe. Ponadto, do części testów dodano wierzchołki izolowane.

Zawody III stopnia

opracowania zadań

Marcin Pilipczuk
Treść zadania, Opracowanie

Marcin Andrychowicz
Program

Dostępna pamięć: 128 MB.

OI, etap III, dzień próbny, 12.03.2013

Gra Tower Defense

Bajtuś gra w grę komputerową **Tower Defense**. *Jego zadaniem jest tak pobudować wieże strażnicze, by strzegły całego jego państwa. W państwie Bajtusia znajduje się wiele miast, a niektóre z nich połączone są dwukierunkowymi drogami. Jeśli Bajtuś postawi w pewnym mieście wieżę strażniczą, wieża ta strzeże tego miasta oraz wszystkich innych miast połączonych z nim bezpośrednią drogą.*

Gdy Bajtuś zastanawiał się nad rozmieszczeniem wież strażniczych w swoim państwie, do pokoju weszła jego starsza siostra Bajtunia. Bajtunia spojrzała na ekran komputera przedstawiający mapę państwa i po chwili stwierdziła: „Eee, nad czym się tu zastanawiać, przecież k wież wystarczy!".

Bajtuś, zły, że Bajtunia popsuła mu zabawę, przegonił siostrę z pokoju i zaczął się zastanawiać, co począć. Honor nie pozwoli mu teraz zbudować więcej niż k wież. Ma jednak w zanadrzu tajną broń: może wynaleźć technologię, dzięki której będzie mógł budować **ulepszone** *wieże strażnicze. Ulepszona wieża strażnicza pilnuje nie tylko miasta, w którym została wybudowana, i wszystkich bezpośrednio sąsiadujących miast, lecz także miast położonych trochę dalej. Formalnie, ulepszona wieża wybudowana w mieście u pilnuje miasta v, jeśli zachodzi jeden z przypadków:*

- $u = v$;
- *istnieje bezpośrednia droga z u do v;*
- *lub istnieje miasto w takie, że istnieją bezpośrednie drogi z u do w oraz z w do v.*

Oczywiście, Bajtuś dalej musi wybudować co najwyżej k wież, będą to jednak ulepszone wieże. Pomóż mu.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się trzy liczby całkowite n, m oraz k ($2 \leqslant n \leqslant 500\,000$, $0 \leqslant m \leqslant 1\,000\,000$, $1 \leqslant k \leqslant n$), rozdzielone pojedynczymi odstępami, oznaczające, odpowiednio, liczbę miast i dróg w państwie Bajtusia oraz liczbę k wypowiedzianą przez Bajtunię. Miasta w państwie Bajtusia są ponumerowane liczbami od 1 do n. Dalej następuje m wierszy opisujących drogi. W każdym wierszu znajdują się dwie liczby całkowite a_i, b_i ($1 \leqslant a_i, b_i \leqslant n$, $a_i \neq b_i$) oznaczające, że miasta o numerach a_i i b_i łączy bezpośrednia dwukierunkowa droga. Każda para miast jest połączona co najwyżej jedną drogą.

Wyjście

Twój program powinien wypisać na standardowe wyjście dwa wiersze opisujące rozmieszczenie ulepszonych wież w państwie Bajtusia. Pierwszy wiersz powinien zawierać liczbę całkowitą r ($1 \leqslant r \leqslant k$), oznaczającą liczbę ulepszonych wież, które powinien zbudować Bajtuś. Drugi wiersz powinien zawierać opis rozmieszczenia tych wież: r parami różnych liczb całkowitych

oznaczających numery miast, w których należy zbudować ulepszone wieże strażnicze. Numery miast można podać w dowolnej kolejności.

W przypadku, gdy istnieje więcej niż jedno rozwiązanie, wystarczy wypisać dowolne z nich. Zwracamy uwagę, że należy wypisać dowolne rozmieszczenie nie więcej niż k ulepszonych wież – nie trzeba używać minimalnej możliwej liczby ulepszonych wież. Możesz założyć, że Bajtunia nie pomyliła się, tzn. że całe państwo Bajtusia można strzec przy pomocy k zwykłych (nieulepszonych) wież. W szczególności oznacza to, że zawsze istnieje rozwiązanie.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*

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

Testy „ocen”:

`1ocen`: *$n = m = 10$, $k = 5$, sieć dróg tworzy cykl;*

`2ocen`: *$n = 1414$, $m = 998\,991$, $k = 100$, każda para miast jest połączona drogą; zauważ, że w tym przypadku wystarczyłoby wybudować tylko jedną wieżę;*

`3ocen`: *$n = 500\,000$, $m = 499\,999$, $k = 250\,000$, sieć dróg tworzy ścieżkę.*

Rozwiązanie

Naturalnym jest, by państwo Bajtusia reprezentować jako nieskierowany graf, w którym wierzchołki odpowiadają miastom, a krawędzie – bezpośrednim drogom między nimi. Przełóżmy więc pozostałe definicje z treści zadania na język teorii grafów.

Niech $G = (V, E)$ będzie grafem o zbiorze wierzchołków V i zbiorze nieskierowanych krawędzi E. Powiemy, że dwa wierzchołki $u, v \in V$ są *w odległości nie większej niż r*, jeśli w grafie G istnieje ścieżka łącząca u i v, zawierająca co najwyżej r krawędzi. Dla liczby całkowitej dodatniej r, zbiór wierzchołków X w grafie $G = (V, E)$ nazwiemy *r-dominującym*, jeśli dla każdego wierzchołka ze zbioru V istnieje wierzchołek ze zbioru X w odległości nie większej niż r. Zbiór 1-dominujący będziemy nazywać zazwyczaj po prostu *dominującym*.

Zauważmy, że $X = V$ jest zbiorem dominującym w grafie G. Ciekawym problemem jest jednak znajdowanie *jak najmniejszego* zbioru dominującego lub r-dominującego. W szczególności, problem postawienia k (nieulepszonych) wież strażniczych w państwie Bajtusia to, używając właśnie wprowadzonej definicji, problem znalezienia w danym grafie zbioru dominującego o co najwyżej k wierzchołkach. Stwierdzenie Bajtuni

możemy więc przetłumaczyć na język teorii grafów jako „W tym grafie istnieje zbiór dominujący o k wierzchołkach!".

Co zaś zmieniają ulepszone wieże? Zauważmy, że problem postawienia k ulepszonych wież to tak naprawdę problem znalezienia zbioru 2-dominującego o co najwyżej k wierzchołkach. Zadanie, które ma przed sobą Bajtuś, brzmi więc:

> Wiedząc, że w grafie G istnieje zbiór dominujący wielkości k, znajdź w G zbiór 2-dominujący wielkości k.

Rozwiązanie wzorcowe

Opiszmy najpierw rozwiązanie wzorcowe, na które wpadła znaczna część uczestników. Spróbujmy postąpić zachłannie: dopóki istnieje choć jedno niepilnowane miasto, postawmy w nim ulepszoną wieżę. A w języku teorii grafów: konstruujemy zbiór wierzchołków X, zaczynając od $X = \emptyset$, i, dopóki X nie jest zbiorem 2-dominującym, bierzemy dowolny wierzchołek v w odległości większej niż 2 od wszystkich wierzchołków z X i dodajemy go do zbioru X. Oczywiście, w ten sposób otrzymamy zbiór 2-dominujący. Nie jest jednak jasne, że nie postawimy w ten sposób więcej niż k wież.

By to pokazać, rozważmy zbiór dominujący Z wielkości co najwyżej k, który miała na myśli Bajtunia. Przeanalizujemy jeden krok naszego algorytmu, w którym do zbioru X dodajemy wierzchołek v (czyli stawiamy ulepszoną wieżę w v). Zgodnie z definicją zbioru dominującego, istnieje wierzchołek $z \in Z$ w odległości co najwyżej 1 od v. Poczyńmy kluczową obserwację: każdy wierzchołek, pilnowany przez (nieulepszoną) wieżę postawioną w wierzchołku z, jest również pilnowany przez ulepszoną wieżę postawioną w v. Innymi słowy, w tym kroku zbiór wierzchołków pilnowanych przez postawione dotychczas ulepszone wieże „połyka" wszystkie wierzchołki pilnowane przez (nieulepszoną) wieżę w z. W związku z tym w każdym kroku algorytmu mamy do czynienia z innym wierzchołkiem z. A ponieważ $|Z| \leqslant k$, więc wykonamy nie więcej niż k kroków, czyli postawimy co najwyżej k ulepszonych wież.

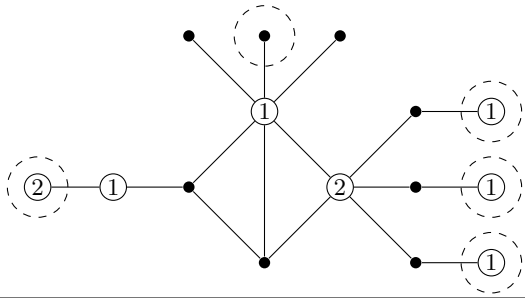
Rozwiązanie wzorcowe w innym języku

Powyższy opis dla niektórych czytelników może wydać się za mało formalny. Spróbujmy więc uczynić go przejrzystszym. Posłuży nam do tego dodatkowa definicja. Zbiór wierzchołków Y nazwiemy *r-rozrzuconym*, jeśli dowolne dwa różne wierzchołki ze zbioru Y są w odległości większej niż r. Zbiór r-rozrzucony Y jest *maksymalny ze względu na zawieranie*, jeśli nie można do niego dodać już żadnego innego wierzchołka, tj. zbiór $Y \cup \{v\}$ nie jest r-rozrzucony dla każdego wierzchołka $v \in V \setminus Y$. Na rysunku 1 pokazany jest przykład zbioru dominującego, 2-dominującego i 3-rozrzuconego.

Poczyńmy następujące dwie proste obserwacje, które powiążą tę definicję ze zbiorami dominującymi.

Lemat 1. Jeśli zbiór Y jest maksymalnym ze względu na zawieranie zbiorem r-rozrzuconym, to jest też zbiorem r-dominującym.

Dowód: Załóżmy, że Y jest zbiorem r-rozrzuconym, ale nie jest zbiorem r-dominującym. Wówczas istnieje wierzchołek $v \in V$ taki, że każdy wierzchołek



Rys. 1: Przykładowy graf z zaznaczonym zbiorem dominującym wielkości 5 (wierzchołki z jedynkami), zbiorem 2-dominującym wielkości 2 (wierzchołki z dwójkami) i zbiorem 3-rozrzuconym wielkości 5 (wierzchołki otoczone przerywaną linią). Jako że przedstawiony zbiór 3-rozrzucony jest oczywiście też zbiorem 2-rozrzuconym, zgodnie z lematem 2, w tym grafie nie istnieje zbiór dominujący o mniej niż pięciu wierzchołkach.

$y \in Y$ jest w odległości większej niż r od v; w szczególności $v \notin Y$. Ale wtedy $Y \cup \{v\}$ jest również zbiorem r-rozrzuconym, czyli Y nie jest maksymalny ze względu na zawieranie. ■

Lemat 2. Jeśli graf G zawiera zbiór $2r$-rozrzucony wielkości k, to każdy zbiór r-dominujący w grafie G ma co najmniej k wierzchołków.

Dowód: Niech Y będzie zbiorem $2r$-rozrzuconym w grafie G, a Z zbiorem r-dominującym. Weźmy dowolny wierzchołek $y \in Y$. Zgodnie z definicją zbioru r-dominującego, istnieje wierzchołek $z \in Z$ w odległości nie większej niż r od wierzchołka y. Zauważmy, że nie może się zdarzyć tak, że dwa wierzchołki $y_1, y_2 \in Y$ są w odległości nie większej niż r od jednego wierzchołka $z \in Z$, gdyż wówczas y_1 i y_2 byłyby w odległości nie większej niż $2r$, co przeczyłoby definicji zbioru $2r$-rozrzuconego. Zatem $|Y| \leqslant |Z|$, co kończy dowód lematu. ■

Zauważmy, że wcześniej omawiany algorytm tak naprawdę w zachłanny sposób konstruuje maksymalny w sensie zawierania zbiór 2-rozrzucony. Lemat 1 pokazuje, że na końcu otrzymamy zbiór 2-dominujący. Lemat 2 pokazuje zaś, że nie będzie miał on więcej niż k wierzchołków. Zwróćmy uwagę na podobieństwo argumentów użytych w dowodzie lematu 2 i poprzedniego opisu poprawności algorytmu: jest to tak naprawdę to samo rozumowanie.

Implementacja

Zadanie wymagało zaimplementowania omówionego rozwiązania zachłannego tak, by działało w czasie liniowym od wielkości grafu wejściowego. Rozważmy następującą naturalną implementację: gdy dodajemy wierzchołek v do konstruowanego zbioru X (stawiamy ulepszoną wieżę w v), przeglądamy wszystkie wierzchołki w odległości co najwyżej 2 od wierzchołka v i odznaczamy je jako „strzeżone”. W kolejnym kroku algorytmu szukamy dowolnego niestrzeżonego jeszcze wierzchołka i stawiamy tam kolejną wieżę.

Jak szybko jesteśmy w stanie przejrzeć wierzchołki strzeżone przez nowo postawioną wieżę? Musimy do tego przejrzeć listę sąsiadów v oraz listę sąsiadów każdego wierzchołka połączonego bezpośrednią drogą z wierzchołkiem v. Zauważmy, że jeśli przejrzymy listę sąsiadów wierzchołka w zarówno przy okazji stawiania wieży w wierzchołku v_1, jak i w wierzchołku v_2, to odległość między v_1 i v_2 wynosi co najwyżej 2, co jest w sprzeczności z zasadą działania naszego algorytmu. Tak więc przejrzymy listę sąsiadów każdego wierzchołka co najwyżej raz, zatem algorytm działa w czasie liniowym od wielkości grafu.

Rozwiązanie wzorcowe można znaleźć w plikach `gra.cpp`, `gra1.pas` i `gra2.cpp`.

Testy

Przygotowano 11 zestawów testowych, z których każdy zawiera co najmniej 6 testów. Testy zostały wygenerowane w sposób losowy, zapewniając istnienie pokrycia k nieulepszonymi wieżami.

Kilka słów o podobnych, ale trudniejszych problemach

Chcielibyśmy wykorzystać zadanie *Gra Tower Defense* jako pretekst, by opowiedzieć o podobnych problemach, którymi zajmują się badacze i które stanowią przedmiot niejednej publikacji naukowej w dziedzinie algorytmiki.

Jak sprawdzić czy Bajtunia miała rację?

Bardzo ciekawym pytaniem jest, skąd Bajtunia wiedziała, że w państwie Bajtusia wystarczy k wież. I jak Bajtuś mógłby sprawdzić, czy Bajtunia ma rację? W języku teorii grafów nasze pytanie brzmi:

> Mając dany graf G i liczbę k, sprawdź, czy w grafie G istnieje zbiór dominujący o co najwyżej k wierzchołkach.

Okazuje się, że jest to problem NP-trudny, co oznacza, że nie spodziewamy się algorytmu rozwiązującego go, który działałby w czasie wielomianowym od wielkości grafu G. Co więcej, jest to też problem co najmniej tak trudny jak problemy w być może dość egzotycznej klasie $W[2]$, co z kolei oznacza (pomijając tutaj skomplikowaną definicję tej klasy), że raczej nie spodziewamy się algorytmu działającego istotnie szybciej niż algorytm kompletnie naiwny, sprawdzający wszystkie możliwe rozwiązania i działający w czasie mniej więcej $O(n^k)$.

Jedną z metod, jakimi staramy się radzić sobie z problemami bardzo trudnymi, jest *aproksymacja*: zamiast próbować rozwiązać problem dokładnie, postarajmy się dać może niekoniecznie optymalne, ale dość dobre rozwiązanie. W naszym problemie przekłada się to na następujące sformułowanie:

> Mając dany graf G i liczbę k, stwierdź, że w G nie istnieje zbiór dominujący o co najwyżej k wierzchołkach, lub podaj zbiór dominujący o co najwyżej αk wierzchołkach.

Parametr α nazywamy *współczynnikiem aproksymacji*; im jest on mniejszy, tym algorytm jest lepszy – lepiej przybliża nam prawdziwe rozwiązanie.

Nie jest bardzo trudno pokazać (zachęcamy Czytelnika do próby samodzielnego dowodu), że dość naturalny algorytm zachłanny (który buduje wieżę w takim wierzchołku, by strzegła jak najwięcej dotychczas niestrzeżonych wierzchołków) jest algorytmem aproksymacyjnym o współczynniku aproksymacji $\alpha = 1 + \ln n$, gdzie ln oznacza logarytm naturalny. Innymi słowy, w grafie, w którym istnieje zbiór dominujący wielkości k, algorytm ten wyznaczy zbiór dominujący wielkości co najwyżej $k(1 + \ln n)$. Okazuje się, że lepiej się nie da, co (przy pewnych dość standardowych założeniach teoriozłożonościowych) wykazał Uriel Feige w 1996 roku [41].

Podsumowując, nie mamy pojęcia, jak Bajtunia odkryła, że można strzec całe państwo Bajtusia przy pomocy tylko k nieulepszonych wież.

Problem remiz strażackich

W naszym zadaniu zastosowaliśmy inny kierunek aproksymacji: zamiast stawiać więcej wież strażniczych, Bajtuś budował ulepszone wieże. To przybliża nas do następującego problemu, który najczęściej formułowany jest w języku budowy remiz strażackich.

Mamy dany graf $G = (V, E)$, w którym wierzchołki grafu odpowiadają różnym ważnym częściom miasta lub państwa, a krawędzie – połączeniom między nimi. Chcemy postawić w niektórych wierzchołkach remizy strażackie. Naszym celem jest, by do każdego wierzchołka naszego grafu straż pożarna dojeżdżała możliwie najszybciej. Formalnie, jeśli X będzie zbiorem wierzchołków, w których zbudujemy remizy, to miarą jakości naszego rozwiązania jest

$$\max_{v \in V} \min_{x \in X} \text{odległość}(v, x).$$

Innymi słowy, dla każdego wierzchołka v patrzymy, gdzie jest najbliższa remiza do wierzchołka v, i znajdujemy ten wierzchołek, który ma najdalej do najbliższej remizy.

Oczywiście, idealnym rozwiązaniem byłoby zbudować remizę w każdym wierzchołku naszego grafu. Niestety, mamy ograniczony budżet: możemy zbudować tylko k remiz. Jak więc je rozmieścić?

Zauważmy, że problem Bajtusia to tak naprawdę problem rozmieszczenia k remiz tak, by straż pożarna była w odległości 1 od każdego wierzchołka, jeśli Bajtuś używa nieulepszonych wież, a w odległości 2 w przypadku wież ulepszonych. Spróbujmy więc uogólnić nasze rozwiązanie. Załóżmy, że ktoś nam powiedział, że można rozmieścić te k remiz tak, by straż pożarna była w odległości co najwyżej r od każdego wierzchołka grafu. Rozważmy następujący algorytm, analogiczny do rozwiązania wzorcowego: tak długo, jak istnieje wierzchołek, od którego najbliższa remiza jest w odległości większej niż $2r$, stawiamy w jednym takim wierzchołku remizę. Omówione wcześniej rozumowanie przechodzi właściwie bez zmian: postawimy co najwyżej k remiz.

Dostajemy więc algorytm o współczynniku aproksymacji 2 dla problemu remiz strażackich. Zauważmy, że nie możemy liczyć na algorytm o lepszym współczynniku aproksymacji, gdyż wówczas taki algorytm mógłby (dokładnie) rozwiązywać problem zbioru dominującego. A ten problem, jak już mówiliśmy, jest bardzo trudny.

Omówiony algorytm został zauważony już w latach 80. ubiegłego wieku [40]. Od tego czasu naukowcy badali różne warianty problemu remiz strażackich, analizując,

które z nich mają równie dobre algorytmy aproksymacyjne. Jednym z najciekawszych kierunków badań jest rozważanie wariantu *z pojemnościami*.

Załóżmy, że w naszym grafie istnieje jeden wierzchołek v, z którego prowadzi bezpośrednia droga do każdego innego wierzchołka grafu. Wówczas optymalnym rozwiązaniem jest postawić jedną remizę strażacką w v. Czy to jednak jest dobre rozwiązanie? Jeśli miasto, reprezentowane przez nasz graf, jest naprawdę duże, to może nie wystarczyć, mimo, że każdy wierzchołek jest blisko remizy. W dużym mieście może wybuchnąć kilka pożarów w różnych częściach miasta, i strażaków z jednej remizy będzie za mało, by je wszystkie ugasić.

Dodajmy więc do naszego zadania założenie o *pojemnościach*: jedna remiza strażacka jest w stanie pilnować co najwyżej L wierzchołków. Formalnie, nasze zadanie zdefiniowane jest następująco.

> Mając dany graf G i liczby k oraz L, znajdź zbiór wierzchołków X wielkości k oraz funkcję $f\colon V \to X$ tak, by dla każdego $x \in X$ zachodziło $|f^{-1}(x)| \leqslant L$ oraz by zminimalizować wielkość
>
> $$\max_{v \in V} \text{odległość}(v, f(v)).$$

Funkcja f przyporządkowuje każdemu wierzchołkowi v remizę $f(v)$, która strzeże v. Warunek $|f^{-1}(x)| \leqslant L$ oznacza, że remiza w wierzchołku x strzeże co najwyżej L wierzchołków.

Problem z pojemnościami okazuje się istotnie trudniejszy. Khuller i Sussmann w 2000 roku pokazali algorytm 6-aproksymacyjny dla tego wariantu [42]. Jeśli dodatkowo utrudnimy sobie zadanie i założymy, że pojemności mogą się różnić między wierzchołkami (tzn. każdy wierzchołek $x \in V$ ma daną pojemność $L(x)$, i remiza postawiona w wierzchołku x może strzec co najwyżej $L(x)$ wierzchołków), wówczas otrzymujemy problem, do którego pierwszy znany algorytm o stałym współczynniku aproksymacji został odkryty dopiero w 2012 roku [43], a obecnie najlepszy znany algorytm ma współczynnik aproksymacji 9 [44].

Jacek Tomasiewicz
Treść zadania, Opracowanie

Maciej Matraszek
Program

Dostępna pamięć: 128 MB.

OI, etap III, dzień pierwszy, 13.03.2013

Bajtokomputer

Dany jest ciąg n liczb całkowitych $x_1, x_2, \ldots, x_n$ o wartościach ze zbioru $\{-1, 0, 1\}$. **Bajtokomputer** *to urządzenie, które umożliwia wykonywanie tylko jednego rodzaju operacji na tym ciągu: zwiększenia wartości x_{i+1} o wartość x_i, dla dowolnego $1 \leqslant i < n$. Liczby całkowite, jakie może pamiętać bajtokomputer, nie są ograniczone. W szczególności elementy przetwarzanego ciągu mogą przybierać dowolnie duże wartości.*

Zaprogramuj bajtokomputer, aby za pomocą minimalnej liczby operacji przekształcił dany ciąg w ciąg niemalejący, czyli taki, że $x_1 \leqslant x_2 \leqslant \ldots \leqslant x_n$.

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą n ($1 \leqslant n \leqslant 1\,000\,000$), oznaczającą liczbę elementów w danym ciągu. Drugi wiersz zawiera n liczb całkowitych $x_1, x_2, \ldots, x_n$ ($x_i \in \{-1, 0, 1\}$) stanowiących kolejne elementy danego ciągu, pooddzielane pojedynczymi odstępami.

W testach wartych łącznie 24% punktów zachodzi dodatkowy warunek $n \leqslant 500$, a w testach wartych łącznie 48% punktów zachodzi $n \leqslant 10\,000$.

Wyjście

Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą, równą minimalnej liczbie operacji, które musi wykonać bajtokomputer, aby przekształcić dany ciąg w ciąg niemalejący, lub jedno słowo `BRAK`*, gdy otrzymanie takiego ciągu nie jest możliwe.*

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`6` `-1 1 0 -1 0 1`	`3`

Wyjaśnienie do przykładu: *Za pomocą trzech operacji bajtokomputer może uzyskać ciąg $-1, -1, -1, -1,\ 0,\ 1$.*

Testy „ocen”:

`0ocen`**:** *$n = 6$, mały test z odpowiedzią* `BRAK`*;*

`1ocen`**:** *$n = 6$, mały test z odpowiedzią 4;*

`2ocen`**:** *$n = 500$, wszystkie elementy ciągu równe 1;*

`3ocen:` $n = 10\,000$, $x_1 = x_2 = \ldots = x_{9\,000} = -1$, $x_{9\,001} = \ldots = x_{9\,900} = 1$, $x_{9\,901} = \ldots = x_{10\,000} = 0$;

`4ocen:` $n = 1\,000\,000$, $x_1 = x_2 = \ldots = x_{999\,997} = -1$, $x_{999\,998} = 1$, $x_{999\,999} = 1$ i $x_{1\,000\,000} = -1$.

Rozwiązanie

Treść zadania można wyrazić całkiem zwięźle: mając dany ciąg $x_1, x_2, \ldots, x_n$ o wartościach w zbiorze $\{-1, 0, 1\}$, chcemy przekształcić go w ciąg niemalejący za pomocą minimalnej liczby operacji $x_{i+1} := x_{i+1} + x_i$. Dla jasności, wartości w początkowym ciągu będziemy oznaczać przez $\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_n$.

Za chwilę udowodnimy, że w trakcie przekształcania ciągu opłaca się tworzyć tylko wartości $x_i \in \{-1, 0, 1\}$. Ponadto wszystkie operacje mogą być wykonywane od lewej do prawej.

Ścisły dowód nie jest prosty. Czytelnik niepotrzebujący dowodu może pominąć poniższy rozdział i od razu przejść do opisu implementacji.

Dowód

Oznaczmy przez o_i operację $x_i := x_i + x_{i-1}$. Niech O będzie pewną optymalną (tj. najkrótszą) sekwencją operacji, która przekształca początkowy ciąg wartości w ciąg niemalejący. Po wykonaniu operacji z O, końcowy ciąg można podzielić na trzy (być może puste) bloki:

- blok ujemny – pierwszy blok z lewej, zawierający liczby ujemne,
- blok zerowy – środkowy blok, zawierający same zera,
- blok dodatni – ostatni blok, zawierający liczby dodatnie.

Blok dodatni

Na początek przeanalizujemy, jak może wyglądać sekwencja operacji, która doprowadziła do powstania bloku dodatniego (o ile jest on niepusty). Załóżmy, że w sekwencji O operacja o_n jest wykonywana k razy. Możemy przekształcić sekwencję O tak, żeby te k operacji było wykonywanych w momencie, gdy x_{n-1} jest największe. Wtedy na końcu ostatni wyraz ciągu będzie nie mniejszy niż poprzednio.

Załóżmy teraz, że blok dodatni ma co najmniej dwa wyrazy i że któraś operacja o_{n-1} w sekwencji O zmniejsza wartość x_{n-1}. Zobaczmy, co się stanie, jeśli zamiast tej operacji dołożymy jedną operację typu o_n (tak jak poprzednio, gdy x_{n-1} jest największe).

Oznaczmy przez m_{n-1} maksymalną wartość x_{n-1} w tym zmodyfikowanym ciągu. Mamy więc teraz $k+1$ operacji typu o_n.

- Jeśli $k \geqslant 1$, to $m_{n-1} \geqslant 1$ i na końcu dostajemy:

$$x_n = (k+1) \cdot m_{n-1} + \hat{x}_n \geqslant 2 \cdot m_{n-1} - 1 \geqslant m_{n-1} \geqslant x_{n-1}.$$

- Jeśli $k = 0$, to wiemy, że $\hat{x}_n = 1$. Skoro operacja zmniejszająca wartość x_{n-1} była w którymś momencie potrzebna, to znaczy, że bez niej byłoby na końcu $x_{n-1} > 1$ (inaczej sekwencja O nie byłaby najkrótsza), czyli także $m_{n-1} > 1$. Zamieniając ją na operację o_n w najdogodniejszym momencie, dostajemy

$$x_n = 1 + m_{n-1} > x_{n-1}.$$

Wobec tego możemy przekształcić sekwencję O także tak, żeby nigdy nie zmniejszała x_{n-1} i żeby wykonywała operacje o_n tylko jak x_{n-1} jest maksymalne. Ale to z kolei oznacza, że można wykonać wszystkie operacje o_n po wszystkich operacjach o_{n-1}. Rozumując indukcyjnie, można dzięki temu pokazać, że na całym bloku dodatnim możemy wykonywać operacje od lewej do prawej, nie tracąc na optymalności.

Załóżmy, że na pierwszej pozycji j w bloku dodatnim mamy $\hat{x}_j < 1$. Wobec tego w którymś momencie x_{j-1} musiało być dodatnie (żeby x_j stało się dodatnie), a potem musiało stać się niedodatnie. Możemy przeprowadzić podobne rozumowanie jak wyżej i zamienić operacje tak, żeby po tym, jak x_{j-1} było dodatnie, już go nie zmniejszać, a w zamian za to zwiększać x_j i w konsekwencji rozszerzyć nasz blok. Stąd wynika, że jeśli blok dodatni zaczyna się wartością mniejszą niż 1, to można go zawsze rozszerzyć w lewo, nie powiększając długości ciągu operacji.

Zastanówmy się teraz, ile co najmniej operacji trzeba wykonać, żeby blok dodatni stał się niemalejący, przy założeniu, że operacje wykonujemy od lewej do prawej i tylko zwiększamy wartości wyrazów ciągu. Każdy wyraz $\hat{x}_i = 0$ musi być zwiększony co najmniej raz, bo zwiększamy go tylko o ostateczną wartość wyrazu x_{i-1}. Podobnie, każdy wyraz $\hat{x}_i = -1$ musi być zwiększony co najmniej dwa razy. Zauważmy, że możemy, idąc od lewej do prawej, każdy wyraz ciągu początkowo równy 0 zwiększyć raz, a każdy wyraz początkowo równy -1 dokładnie dwa razy. Dzięki temu zamienimy wszystkie wyrazy bloku na jedynki.

Ostatecznie oznacza to tyle, że jako dodatnie bloki możemy rozważać sufiksy ciągu początkowego rozpoczynające się jedynką i optymalnie jest zamieniać kolejno wszystko w takim bloku na 1 (od lewej do prawej).

Blok ujemny

Ponieważ żadna operacja nie zmienia wartości pierwszego wyrazu ciągu, zatem blok ujemny (jeśli jest niepusty) musi początkowo zaczynać się od $\hat{x}_1 = -1$, a na końcu mieć wszystkie wyrazy równe -1.

Rozważmy pewną optymalną sekwencję operacji, która do tego prowadzi. Zauważmy, że aby ostatecznie wyraz $\hat{x}_i = 0$ zawierał -1, trzeba co najmniej raz wykonać operację o_i. Jeśli natomiast $\hat{x}_i = 1$, to albo potrzebujemy co najmniej dwóch operacji o_i (gdy $x_{i-1} = -1$ w momencie wykonania tej operacji), albo co najmniej jednej operacji zmniejszającej x_i (o co najmniej 2) i co najmniej jednej operacji zwiększającej x_{i-1}. W obu przypadkach potrzebujemy zatem co najmniej dwóch operacji na jedynkę.

Zupełnie podobnie jak wcześniej w przypadku bloku dodatniego, możemy osiągnąć to dolne oszacowanie na liczbę operacji, idąc od lewej do prawej i zamieniając wszystko na -1, wykonując po jednej operacji dla każdego 0 i po dwie dla każdej 1.

Blok zerowy

Przyjmijmy, że po optymalnym ciągu operacji dostajemy blok $x_p, x_{p+1}, \ldots, x_{p+l}$ z zerami, który początkowo nie był tej postaci. Zauważmy, że jeśli $\hat{x}_p = -1$, to równie dobrze moglibyśmy (z zerowym kosztem) rozszerzyć w prawo blok ujemny. Załóżmy zatem, że $\hat{x}_p \geqslant 0$, po którym następowało dokładnie k zer. Jeśli $k \neq l$, to mamy $\hat{x}_{p+1} = \ldots = \hat{x}_{p+k} = 0$ i $\hat{x}_{p+k+1} \neq 0$. Aby wyzerować wyraz $\hat{x}_{p+k+1}$, w którymś momencie każdy z wyrazów $\hat{x}_{p+k-i}$ musiał stać się tego samego znaku co $\hat{x}_{p+k+1}$ (dla i nieparzystego) lub przeciwnego znaku do $\hat{x}_{p+k+1}$ (dla i parzystego), po czym z powrotem stać się zerem. Zatem potrzebujemy co najmniej $2k + 1 + \hat{x}_p$ operacji na wyzerowanie wyrazów $x_p, \ldots, x_{p+k+1}$. Co więcej, blok ujemny nie może być w takim wypadku pusty, zatem zakładamy, że $x_{p-1} = -1$. Zauważmy jednak, że wystarczy nam $k + 1 + \hat{x}_p$ operacji na rozszerzenie bloku ujemnego w prawo o $k + 1$ wyrazów oraz dodatkowa operacja na wyzerowanie x_{p+k+1}, jeśli $\hat{x}_{p+k+1} = 1$. Jeśli więc $k \geqslant 1$, to opłaca nam się rozszerzyć blok ujemny (dla $k = 0$ łatwo sprawdzić, że to również jest prawdą).

Ostatecznie zatem, istnieje optymalne rozwiązanie, w którym blok zerowy zajmuje przestrzeń, na której pierwotnie były same zera, ewentualnie z jedną jedynką na początku (przy czym ten drugi przypadek może mieć miejsce jedynie, gdy blok ujemny jest niepusty).

Implementacja

W poprzednim rozdziale udowodniliśmy, że wystarczy rozważać końcowe ciągi, które składają się z trzech bloków zawierających wartości -1, 0 i 1. Oznaczmy te bloki literami A, B i C. Każdy z tych bloków może być pusty. Udowodniliśmy ponadto, że blok B musi początkowo składać się z samych zer, ewentualnie poprzedzonych jedną jedynką (ale w tym drugim przypadku blok A musi być niepusty). Natomiast blok C (o ile jest niepusty) musi początkowo rozpoczynać się jedynką, a blok A (o ile jest niepusty) musi rozpoczynać się wartością -1 (rys. 1).

Rozwiązanie wzorcowe

Rozwiązanie wzorcowe będzie przebiegać następująco. Rozważamy wszystkie możliwe długości pierwszego bloku. Dla tak ustalonego bloku A znajdujemy najdłuższy blok B, rozpatrując dwa przypadki (gdy B zaczyna się od 0 i od 1). Dzięki temu znamy też

-1	?	?	?	$\frac{0}{1}$	0	0	0	1	?	?	?
-1	-1	-1	-1	0	0	0	0	1	1	1	1
	A				B				C		

Rys. 1: Początkowy i końcowy wygląd ciągu w podziale na bloki.

długość bloku C. Jeśli bloki A, B i C są poprawne, wyznaczamy liczbę potrzebnych operacji do utworzenia tych bloków i porównujemy z dotychczas znalezionym minimum.

Znalezienie najdłuższego bloku B możemy wykonać w czasie stałym. Niech $mZero[i]$ będzie maksymalną liczbą kolejnych zer w ciągu, poczynając od pozycji i. W tym celu wystarczy przeglądać ciąg od końca, pamiętając maksymalną liczbę zer:

```
1: ileZer := 0;
2: for k := n downto 1 do begin
3:    if x̂[k] = 0 then
4:       ileZer := ileZer + 1
5:    else
6:       ileZer := 0;
7:    mZero[k] := ileZer;
8: end
```

Koszt utworzenia bloku B jest łatwy do wyznaczenia. W pierwszym przypadku (gdy blok ten składa się z samych zer) koszt jest zerowy. W drugim przypadku (gdy blok ten zaczyna się jedynką i blok A jest niepusty) wystarczy jedna operacja, która wyzeruje pierwszy element.

Dla bloków A i C możemy równie szybko wyznaczyć koszt ich utworzenia. Skupmy się na bloku A (blok C jest analogiczny). Jeśli $\hat{x}_1 = -1$, to blok A o długości a można zamienić na same -1, używając po dwie operacje na każdą jedynkę i jedną operację na każde zero, czyli:

$$zamiany = 2 \cdot jed(1, a) + zer(1, a).$$

Funkcja $jed(l, p)$ zwraca liczbę jedynek w ciągu w przedziale od pozycji l do pozycji p. Analogicznie funkcja $zer(l, p)$ zwraca liczbę zer w tym przedziale. Aby je efektywnie zapisać, możemy uprzednio przygotować tablice, będące sumami prefiksowymi. Przykładowo do wyliczania liczby jedynek:

```
1: jedynki[0] := 0;
2: for k := 1 to n do begin
3:    jedynki[k] := jedynki[k − 1];
4:    if x̂[k] = 1 then
5:       jedynki[k] := jedynki[k] + 1;
6: end
```

Dzięki temu wyznaczenie liczby jedynek w dowolnym przedziale jest proste:

$$jed(l, p) = jedynki[p] - jedynki[l - 1].$$

W ten sposób możemy wyznaczyć liczbę $1, 0$ i -1 w dowolnym przedziale w czasie $O(1)$. Wobec tego każdy blok A analizujemy w czasie stałym, więc złożoność całego rozwiązania wynosi $O(n)$. Zostało ono zaimplementowane w plikach `baj.cpp`, `baj1.cpp` i `baj2.pas`.

Rozwiązanie siłowe $O(n^3)$

Można wprost rozpatrywać wszystkie możliwe podziały ciągu na trzy bloki i dla każdego podziału symulować wykonywanie operacji. Wszystkich podziałów możemy mieć $O(n^2)$, a zliczenie potrzebnych operacji trwa $O(n)$. Stąd dostajemy algorytm w złożoności $O(n^3)$. Takie rozwiązanie zostało zaimplementowane w plikach `bajs1.cpp` i `bajs2.pas` i otrzymywało 24% punktów.

Rozwiązanie wolne $O(n^2)$

Usprawnieniem poprzedniego rozwiązania jest wyliczenie sum prefiksowych, przez co zliczanie potrzebnych operacji możemy wykonać w czasie $O(1)$. Stąd całkowita złożoność to $O(n^2)$. Takie rozwiązanie zostało zaimplementowane w plikach `bajs3.cpp` i `bajs4.pas`. Otrzymywało ono 48% punktów.

Rozwiązanie alternatywne

Wiedząc, że istnieje optymalne rozwiązanie, takie że wszystkie operacje wykonujemy od lewej do prawej i mamy tylko wartości $x_i \in \{-1, 0, 1\}$, możemy napisać rozwiązanie oparte o programowanie dynamiczne.

Wyznaczamy minimalną liczbę operacji do uzyskania ciągu o prefiksie długości i, kończącego się wartościami $0, 1$ i -1. Poruszamy się od najmniejszych i – wyliczamy wynik na podstawie uprzednio wyliczonych wartości. Takie rozwiązanie samo troszczy się o wszystkie przypadki i jest prostsze w implementacji.

Rozwiązanie znajduje się w pliku `baj3.cpp`. Za takie rozwiązanie otrzymywało się maksymalną liczbę punktów.

Testy

Przygotowano 8 grup testów:

- grupa *1* – małe ręczne testy poprawnościowe,
- grupa *2* i *3* – większe testy poprawnościowe,
- testy *4a–8a* – losowe testy z krótkimi przedziałami takich samych liczb,
- testy *4b–8b* – losowe testy z długimi przedziałami (około $\sqrt{n}$) takich samych liczb.

Tomasz Idziaszek
Treść zadania, Opracowanie

Maciej Borsz
Program

Dostępna pamięć: 128 MB. *OI, etap III, dzień pierwszy, 13.03.2013*

Labirynt

Bajtazar przeczytał niedawno ciekawą historię. Jej bohaterem był jakiś grecki królewicz, który pokonał straszliwego potwora za pomocą kłębka wełny, lub coś w tym rodzaju. Ale to nie to tak zafascynowało Bajtazara. Najbardziej spodobało mu się, że kluczowe wydarzenia działy się w labiryncie. Od tej pory Bajtazar ma bzika na punkcie labiryntów.

Bajtazar rysuje plany labiryntów na kratkowanej kartce papieru. Każdy taki plan jest wielokątem, którego boki (reprezentujące ściany labiryntu) są równoległe do brzegów kartki (tj. osi prostokątnego układu współrzędnych) i każde dwa kolejne boki są do siebie prostopadłe. Bajtazar zauważył, że jeśli na jednej ze ścian takiego labiryntu umieścimy wejście, a następnie wejdziemy do niego i, idąc, cały czas będziemy trzymać się prawą ręką ściany, to na pewno obejdziemy cały labirynt, wracając na końcu z powrotem do wejścia.

Co więcej, podczas takiego obejścia możemy notować wykonywane przez nas zakręty. Zapisujemy literę `L`*, jeśli podczas przechodzenia na kolejną ścianę obracamy się w lewo, zaś* `P`*, jeśli obracamy się w prawo. Bajtazar zastanawia się, dla jakich słów złożonych z liter* `L` *i* `P` *istnieje labirynt, który spowoduje, że zanotujemy takie słowo podczas obchodzenia tego labiryntu.*

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedno n-literowe słowo ($1 \leqslant n \leqslant 100\,000$) złożone z liter `L` *i* `P`*, które opisuje ciąg kolejnych zakrętów napotykanych podczas obchodzenia labiryntu.*

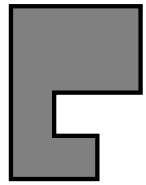
W testach wartych 50% punktów zachodzi dodatkowy warunek $n \leqslant 2500$.

Wyjście

Jeśli nie da się skonstruować labiryntu według opisu z wejścia, na standardowym wyjściu należy wypisać słowo `NIE`*. W przeciwnym wypadku na wyjście należy wypisać dokładnie n wierszy zawierających opis przykładowego labiryntu. W i-tym z nich powinny znaleźć się dwie liczby całkowite x_i i y_i ($-10^9 \leqslant x_i, y_i \leqslant 10^9$) oddzielone pojedynczym odstępem, oznaczające współrzędne i-tego wierzchołka na planie labiryntu. Wierzchołki powinny zostać wypisane zgodnie z kolejnością ich występowania na obwodzie wielokąta, przeciwnie do ruchu wskazówek zegara; można zacząć od dowolnego wierzchołka i nie trzeba zaznaczać umiejscowienia wejścia.*

Przykład

Dla danych wejściowych:
`LLLLPPLL`
poprawnym wynikiem jest:
```
0 0
2 0
2 2
-1 2
-1 -2
1 -2
1 -1
0 -1
```



Testy „ocen":

`1ocen`**:** *$n = 12$, słowo* `LLLLLLLLLLLL`*, odpowiedź* `NIE`*;*

`2ocen`**:** *$n = 100$, spirala zakręcająca w lewo;*

`3ocen`**:** *$n = 100\,000$, schodki.*

Wizualizator wyjść

Do tego zadania dołączony jest wizualizator wyjść, który dla pliku zgodnego z formatem wyjścia rysuje odpowiadający mu labirynt. Aby go uruchomić, wejdź do folderu `/home/zawodnik/labwiz` *i wykonaj polecenie:*

```
./labwiz
```

Dla plików, które nie zawierają opisu labiryntu zgodnego z formatem wyjścia, zachowanie wizualizatora jest nieokreślone.

Rozwiązanie

Dane jest n-literowe słowo składające się tylko z liter `L` i `P`. Zadanie polega na wygenerowaniu takiego labiryntu – wielokąta o bokach prostopadłych do osi układu współrzędnych – że, obchodząc jego obwód (zaczynając od wybranej ściany), wykonamy określoną sekwencję skrętów w lewo (litera `L` w słowie) i w prawo (`P`).

Ponieważ dopuszczalną odpowiedzią jest też stwierdzenie, że żądany labirynt nie istnieje, zastanówmy się najpierw, kiedy taka sytuacja będzie miała miejsce. Niech l oznacza liczbę skrętów w lewo w słowie, a $p = n - l$ liczbę skrętów w prawo. Dość łatwo się przekonać, że warunkiem koniecznym na istnienie labiryntu jest to, żeby skrętów w lewo było dokładnie o cztery więcej niż skrętów w prawo, czyli

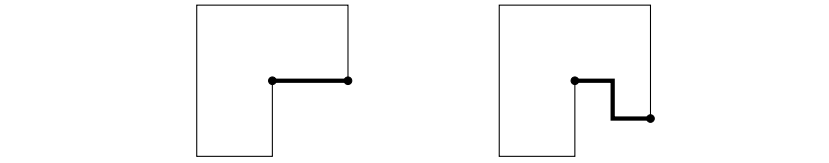
$$l = p + 4. \qquad (\star)$$

Istotnie, aby mieć szansę skończyć wędrówkę w tym samym miejscu, w którym ją zaczęliśmy, musimy (poruszając się przeciwnie do ruchu wskazówek zegara) sumarycznie wykonać obrót o dokładnie $4 \cdot 90^\circ = 360^\circ$ w lewo. (W szczególności oznacza to, że długość słowa opisującego labirynt musi być parzysta.)

Mniej oczywiste jest to, że jest to również warunek wystarczający na istnienie labiryntu. Dowód tego faktu przeprowadzimy przez indukcję po długości słowa opisującego labirynt. Najmniejszy wielokąt ma co najmniej 4 boki. Dla $n = 4$ daje to $l = 4$ i $p = 0$, a rozwiązaniem jest dowolny prostokąt o współrzędnych całkowitoliczbowych.

Załóżmy zatem, że labirynt istnieje dla wszystkich słów długości $n-2$ spełniających warunek $(\star)$, i rozważmy dowolne słowo w długości n spełniające ten warunek. Skoro $n \geqslant 6$, to w słowie w znajduje się co najmniej jedna litera `P` i co najmniej jedna litera `L`. Co więcej, możemy bez straty ogólności założyć, że pierwsze dwie litery słowa w to `P` i `L`, czyli $w = \texttt{PL}w'$ dla pewnego słowa w' o długości $n-2$. Jeżeli tak nie jest, to możemy zastąpić w jego obrotem cyklicznym, który zaczyna się od `PL` (zauważmy, że labirynt będzie taki sam dla każdego obrotu cyklicznego słowa w).

Zauważmy, że słowo w' również spełnia warunek $(\star)$, zatem z założenia indukcyjnego istnieje labirynt opisywany tym słowem. Możemy teraz zmodyfikować ten labirynt, aby dostać labirynt opisywany słowem $\texttt{PL}w'$, poprzez dodanie dwóch zakrętów na ścianie labiryntu zawierającej wejście. Przykładowo, jeśli jest to ściana łącząca wierzchołki (x, y) i $(x + 1, y)$, tak jak na rys. 1, to zastępujemy ją trzema ścianami łączącymi kolejno wierzchołki (x, y), $(x + \frac{1}{2}, y)$, $(x + \frac{1}{2}, y - \frac{1}{2})$, $(x + 1, y - \frac{1}{2})$ i na pierwszej z tych ścian umieszczamy wejście. Następnie mnożymy wszystkie współrzędne wierzchołków przez 2, aby z powrotem stały się całkowite.



Rys. 1: Labirynt dla słowa $w' = \texttt{LLLLLP}$ oraz dla słowa $w = \texttt{PL}w'$.

Rozwiązanie kwadratowe

Powyższy dowód istnienia labiryntu jest konstruktywny i może stać się podstawą do zapisania rekurencyjnego algorytmu wyznaczającego labirynt. Każda z $O(n)$ faz rekurencji będzie znajdować odpowiedni obrót cykliczny słowa w, wyznaczać rekurencyjnie labirynt dla słowa w' i poprawiać go zgodnie z powyższym opisem. Pojedyncza faza rekurencji trwa $O(n)$, zatem cały algorytm będzie działał w czasie $O(n^2)$.

Pozostał jeszcze jeden szczegół techniczny do dopracowania. Mnożąc w każdej fazie współrzędne wierzchołków przez 2, dość szybko wyjdziemy poza dopuszczalny prostokąt $|x|, |y| \leqslant 10^9$, w którym ma się zmieścić labirynt. Aby temu zapobiec, wystarczy na koniec każdej fazy algorytmu *skompresować* labirynt, przenumerowując wszystkie współrzędne występujące w wyniku na pewne liczby z zakresu od 1 do $\frac{n}{2}$. Taki zakres współrzędnych nam wystarczy, gdyż każda z $\frac{n}{2}$ ścian pionowych łączy dwa wierzchołki

o równej współrzędnej x, podobnie dla ścian poziomych i współrzędnych y. Ponadto dowolne przekształcenie „ściskające" labirynt wzdłuż osi układu współrzędnych daje nadal poprawny labirynt. Jeśli chodzi o implementację, to używamy standardowej sztuczki polegającej na zmapowaniu każdej współrzędnej na liczbę współrzędnych mniejszych od niej przed przemapowaniem. Takie przemapowanie wymaga sortowania i zajmuje czas $O(n \log n)$. W przypadku naszego zadania, przemapowanie będziemy musieli robić po każdej fazie, będzie ono jednak kosztować $O(n)$, gdyż możemy zastosować sortowanie przez zliczanie.

Rozwiązanie wzorcowe

Słowo złożone z takiej samej liczby liter `L` i `P`, w którego każdym prefiksie jest co najmniej tyle samo liter `L`, co liter `P`, nazwiemy *zbalansowanym*. Rozwiązanie wzorcowe działa w oparciu o obserwację, że ze słowa opisującego labirynt możemy usunąć takie cztery skręty w lewo, aby pozostałe cztery fragmenty były słowami zbalansowanymi (przyjmujemy przy tym, że pierwsza litera słowa sąsiaduje z ostatnią). Na przykład ze słowa `PLLPLLLPLPPLLLLP` można usunąć podkreślone litery `L`, aby pozostałe fragmenty `LP`, `LLPLPP`, ϵ (słowo puste) i `LLPP` były zbalansowane. Każdy z tych fragmentów będzie rozpatrywany osobno, a wynikowy labirynt powstanie przez sklejenie labiryntów odpowiadających tym fragmentom. (Formalnie, zbalansowanemu słowu odpowiada pewna łamana niebędąca wielokątem, ale dla prostoty opisu będziemy ją również nazywać labiryntem.)

Labirynt dla zbalansowanego słowa będziemy tworzyć rekurencyjnie. Słowu pustemu (brak skrętów) odpowiada jedna ściana. Jeśli słowo w jest niepuste, to rozważymy dwa przypadki:

(1) Pewien właściwy prefiks w też jest zbalansowany, tzn. $w = w_1 w_2$ dla niepustych słów w_1 i w_2, takich że w_1 jest zbalansowane. Wówczas w_2 również jest zbalansowane i labirynt dla w powstaje z połączenia labiryntów dla w_1 i w_2.

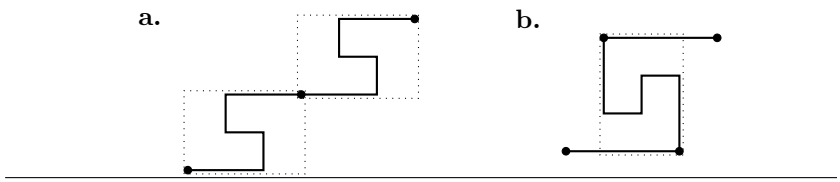
(2) W przeciwnym wypadku $w = \texttt{L}w'\texttt{P}$ i słowo w' jest zbalansowane. Labirynt dla w powstaje z obrotu labiryntu dla w' w lewo i dołożeniu dwóch ścian.

A konkretniej: zakładamy, że konstruowany przez nas labirynt dla zbalansowanego słowa długości n ma mieć $n+1$ ścian, ma rozpoczynać się w wierzchołku $(0, 0)$, kończyć w wierzchołku (x, y) i ma mieścić się w prostokącie o rozmiarze $x \times y$, tak że poza końcami żadne inne punkty labiryntu nie dotykają prawej i lewej krawędzi prostokąta. Wtedy opisane powyżej przypadki rozpatrujemy następująco (patrz rys. 2):

(0) Labirynt dla słowa pustego składa się ze ściany łączącej punkty $(0, 0)$ i $(1, 0)$.

(1) Jeśli labirynt dla zbalansowanego słowa w_i mieści się w prostokącie rozmiaru $x_i \times y_i$, to labirynt dla słowa $w = w_1 w_2$ powstaje przez przesunięcie labiryntu dla w_2 do punktu (x_1, y_1) i mieści się w prostokącie rozmiaru $(x_1 + x_2) \times (y_1 + y_2)$.

(2) Jeśli labirynt dla zbalansowanego słowa w' mieści się w prostokącie rozmiaru $x \times y$, to labirynt dla słowa $w = \texttt{L}w'\texttt{P}$ powstaje przez obrót tego labiryntu, by mieścił się w prostokącie wyznaczonym przez punkty $(y + 1, 0)$ i $(1, x)$, oraz



Rys. 2: Konstrukcja labiryntu **a.** dla słowa ww i **b.** słowa LwP, gdzie w = LLPP.

dodanie ścian łączących punkty $(0,0)$ i $(y+1,0)$ oraz punkty $(1,x)$ i $(y+2,x)$. Cały labirynt mieści się więc w prostokącie rozmiaru $(y+2) \times x$.

Poniższe dwie funkcje wzajemnie rekurencyjne pozwalają na wyznaczenie rozmiarów prostokąta otaczającego labirynt dla niepustego słowa zbalansowanego w. Zakładamy, że i jest globalną zmienną oznaczającą aktualnie podglądaną literę w słowie w, początkowo zainicjowaną na 1. Funkcja *prefiksy*, dopóki może, odcina zbalansowane prefiksy. Ponieważ są one najkrótsze możliwe, więc funkcja *rek* zakłada, że ma do czynienia z przypadkiem (2). Przeskakuje ona końcowe litery fragmentu, a środek rozdziela na zbalansowane fragmenty znów z wykorzystaniem funkcji *prefiksy*.

```
1:  function prefiksy()
2:  begin
3:    if w_i = P then return (1,0);
4:    x := 0; y := 0;
5:    while i < n and w_i = L do begin
6:      (x',y') := rek();
7:      x := x + x'; y := y + y';
8:    end
9:    return (x,y);
10: end
```

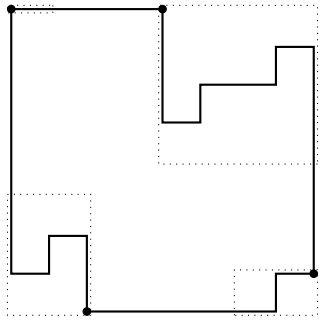
```
1: function rek()
2: begin
3:   { w_i = L }
4:   i := i + 1;
5:   (x,y) := prefiksy();
6:   { w_i = P }
7:   i := i + 1;
8:   return (y+2,x);
9: end
```

Jeśli spojrzymy na zbiór zbalansowanych fragmentów słowa w jak na drzewo (iterujemy po kolejnych literach i interpretujemy literę `L` jako początek opisu nowego poddrzewa, a literę `P` jako jego koniec – analogicznie do interpretacji nawiasowań jako drzew), to powyższe funkcje obliczają dla każdego poddrzewa prostokąt, jaki jest potrzebny do jego narysowania.

W następnym kroku możemy (korzystając z obliczonych rozmiarów prostokątów) rekurencyjnie wyznaczyć ciągi ścian odpowiadające zbalansowanym fragmentom słowa w.

Zauważmy, że nie potrzebujemy kompresować uzyskanych labiryntów; łatwo wykazać, że labirynt dla n-literowego słowa będzie miał współrzędne wierzchołków nie większe niż n. Na końcu łączymy cztery labirynty w jeden. Wystarczy umieścić prostokąty ograniczające te labirynty w wierzchołkach dużego kwadratu i odpowiednio przedłużyć ich początkowe ściany (patrz rys. 3).

Powyższe rozwiązanie działa w optymalnym czasie $O(n)$ i zostało zaimplementowane w plikach `lab.cpp` i `lab3.pas`.



Rys. 3: Łączenie czterech labiryntów dla słowa PLLPLLLPLPPLLLLP.

Aby uzasadnić poprawność rozwiązania, pozostaje tylko udowodnić fakt, że dla słowa spełniającego warunek ($\star$) zawsze istnieje podział zbalansowany. Pozostawimy to jako ćwiczenie dla Czytelnika, gdyż jego dowód jest bardzo podobny do dowodu znanego faktu, który mówi, że dla każdego słowa zawierającego taką samą liczbę liter `L` i `P` istnieje obrót cykliczny będący słowem zbalansowanym.

Tomasz Waleń — Treść zadania, Opracowanie
Krzysztof Kiljan — Program

Dostępna pamięć: 128 MB. *OI, etap III, dzień pierwszy, 13.03.2013*

Łańcuch kolorowy

Mały Bajtuś bardzo lubi bawić się kolorowymi łańcuchami. Zebrał już ich sporą kolekcję, jednak niektóre z nich lubi bardziej niż inne. Każdy z łańcuchów składa się z pewnej liczby kolorowych ogniw. Bajtazar zauważył, że Bajtuś ma bardzo precyzyjne preferencje estetyczne. Otóż Bajtuś uważa jakiś spójny fragment łańcucha za ładny, jeśli zawiera on dokładnie l_1 ogniw koloru c_1, l_2 koloru c_2, ..., l_m koloru c_m, a ponadto nie zawiera żadnych ogniw innych kolorów. Atrakcyjność łańcucha odpowiada liczbie (spójnych) fragmentów, które są ładne. Bajtazar metodą prób i błędów odkrył wartości $c_1, \ldots, c_m$ i $l_1, \ldots, l_m$. Teraz chciałby kupić nowy łańcuch i prosi Cię o napisanie programu, który pomoże mu w zakupach.

Wejście

Pierwszy wiersz standardowego wejścia zawiera dwie liczby całkowite n i m ($1 \leqslant m \leqslant n \leqslant 1\,000\,000$) oddzielone pojedynczym odstępem. Oznaczają one odpowiednio długość łańcucha i długość opisu ładnego fragmentu. Drugi wiersz zawiera m liczb całkowitych $l_1, \ldots, l_m$ ($1 \leqslant l_i \leqslant n$) pooddzielanych pojedynczymi odstępami. Trzeci wiersz zawiera m liczb całkowitych $c_1, \ldots, c_m$ ($1 \leqslant c_i \leqslant n$, $c_i \neq c_j$ dla $i \neq j$) pooddzielanych pojedynczymi odstępami. Ciągi $l_1, \ldots, l_m$ i $c_1, \ldots, c_m$ opisują definicję ładnego fragmentu – musi on zawierać dokładnie l_i ogniw koloru c_i. Czwarty wiersz zawiera n liczb całkowitych $a_1, \ldots, a_n$ ($1 \leqslant a_i \leqslant n$) pooddzielanych pojedynczymi odstępami, oznaczających kolory kolejnych ogniw łańcucha.

W testach wartych 50% punktów zachodzi dodatkowy warunek $1 \leqslant m \leqslant n \leqslant 5000$.

Wyjście

W pierwszym i jedynym wierszu standardowego wyjścia Twój program powinien wypisać jedną liczbę całkowitą – liczbę ładnych spójnych fragmentów w łańcuchu.

Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
2
```

Wyjaśnienie do przykładu: *Dwa ładne fragmenty tego łańcucha to 2, 1, 3, 1 oraz 1, 3, 1, 2.*

Testy „ocen”:

`1ocen`: *$n = 9$, $m = 3$, dwa ładne fragmenty następują po sobie, ale nie nakładają się;*

`2ocen`: *$n = 10$, $m = 5$, długość ładnego fragmentu przekracza długość całego łańcucha (wynik to 0);*

`3ocen`: *$n = 19$, $m = 7$, trzy ładne fragmenty nachodzące na siebie;*

`4ocen`: *$n = 1000$, $m = 500$, ładny fragment zawiera po jednym kolorze z $\{1, \ldots, 500\}$, a łańcuch to ciąg ogniw kolorów 1, 2, ..., 499, 500, 500, 499, ..., 2, 1 (wynik to 2);*

`5ocen`: *$n = 1\,000\,000$, $m = 2$, ładny fragment zawiera jeden kolor 1 oraz dwa kolory 2, łańcuch to 1, 2, 2, 1, 2, 2, ... (wynik to 999 998).*

Rozwiązanie

Prawidłowe rozwiązanie zadania *Łańcuch kolorowy* nie sprawiło zbyt wielu kłopotów uczestnikom Olimpiady. Aby osiągnąć efektywne rozwiązanie zadania, wystarczy zauważyć kilka prostych faktów.

Szukane ładne fragmenty łańcucha mają zawsze tę samą długość równą $d := \sum_{i=1}^{m} l_i$. Stąd nie musimy rozważać wszystkich możliwych spójnych fragmentów łańcucha (których jest $\Theta(n^2)$), a jedynie $\Theta(n)$ fragmentów o długości dokładnie d. Jeśli zauważymy, że wartość d przekracza długość łańcucha n, to możemy zakończyć obliczenia i zwrócić wartość 0, ponieważ łańcuch jest zbyt krótki, aby zawierał ładny fragment. W tym kroku ukryta była pewna pułapka techniczna, gdyż zgodnie z opisem danych wejściowych wartość d mogła być bardzo duża i konieczne było użycie dla niej typu danych o odpowiednim rozmiarze (na przykład 64-bitowej liczby całkowitej).

Kolejną prostą obserwacją jest fakt, że dwa kolejne fragmenty długości d rozpoczynające się odpowiednio na pozycjach a_i oraz a_{i+1} nie różnią się zbyt wiele, jeśli chodzi o krotności występujących w nich kolorów. Oba fragmenty różnią się jedynie elementami a_i oraz a_{i+d}. Możemy wykorzystać tę obserwację, aby znacznie przyspieszyć testowanie, czy aktualnie badany fragment jest ładny.

Aby dokładniej opisać rozwiązanie, zdefiniujmy funkcję $ile_w(c)$ zliczającą liczbę wystąpień ogniw koloru c w spójnym fragmencie łańcucha $w = a_i, \ldots, a_j$:

$$ile_w(c) = |\{a_p = c : i \leqslant p \leqslant j\}|.$$

Korzystając z poprzedniej obserwacji, zauważamy, że dla dwóch kolejnych fragmentów $w = a_i, \ldots, a_{i+d-1}$ oraz $u = a_{i+1}, \ldots, a_{i+d}$ funkcje $ile_w(c)$ oraz $ile_u(c)$ albo są identyczne (w przypadku gdy $a_i = a_{i+d}$), albo (w przypadku gdy $a_i \neq a_{i+d}$) różnią się na dokładnie dwóch pozycjach: $ile_u(a_i) = ile_w(a_i) - 1$ oraz $ile_u(a_{i+d}) = ile_w(a_{i+d}) + 1$.

Na podstawie definicji ładnego fragmentu definiujemy analogiczną funkcję $oczekiwane(c)$, która dla każdego koloru c zwraca nam liczbę oczekiwanych ogniw l w ładnym fragmencie: $oczekiwane(c_i) = l_i$ dla $1 \leqslant i \leqslant m$ oraz $oczekiwane(c) = 0$ dla pozostałych kolorów.

Oczywiście, fragment $w = a_i, \ldots, a_{i+d-1}$ jest ładny wtedy i tylko wtedy, gdy $ile_w(c) = oczekiwane(c)$ dla $1 \leqslant c \leqslant n$.

Aby przyspieszyć testowanie powyższego warunku, możemy utrzymywać wartość *zleKolory*, równą liczbie kolorów c dla których $ile_w(c) \neq oczekiwane(c)$. Dzięki tej wartości możemy bardzo łatwo testować, czy fragment jest ładny: wystarczy sprawdzić, czy $zleKolory = 0$.

Korzystając z powyższych obserwacji, otrzymujemy następujące rozwiązanie o złożoności czasowej i pamięciowej $O(n)$:

```
1:  function policzLadneFragmenty(A[1..n], L[1..m], C[1..m])
2:  begin
3:    d := ∑_{i=1}^{m} L[i];
4:    if d > n then return 0;
5:    wyznacz tablicę oczekiwane[1..n] na podstawie L[1..m], C[1..m];
6:    { wyznaczanie wartości ile i zleKolory dla a_1, ..., a_d }
7:    wyznacz tablicę Ile[1..n], gdzie Ile[c] = |{A[i] = c : 1 ⩽ i ⩽ d}|;
8:    zleKolory := |{Ile[c] ≠ oczekiwane[c] : 1 ⩽ c ⩽ n}|;
9:    ladneFragmenty := 0;
10:   if zleKolory = 0 then ladneFragmenty := ladneFragmenty + 1;
11:   for i := 1 to n − d do begin
12:     { wyznaczanie wartości ile i zleKolory dla a_{i+1}, ..., a_{i+d} }
13:     if A[i] ≠ A[i + d] then begin
14:       aktywneKolory := {A[i], A[i + d]};
15:       d_1 := |{Ile[c] ≠ oczekiwane[c] : c ∈ aktywneKolory}|;
16:       Ile[A[i]] := Ile[A[i]] − 1;
17:       Ile[A[i + d]] := Ile[A[i + d]] + 1;
18:       d_2 := |{Ile[c] ≠ oczekiwane[c] : c ∈ aktywneKolory}|;
19:       zleKolory := zleKolory − d_1 + d_2;
20:     end
21:     if zleKolory = 0 then ladneFragmenty := ladneFragmenty + 1;
22:   end
23:   return ladneFragmenty;
24: end
```

Testy

Do oceny zadania przygotowano 10 zestawów testowych, z których każdy zawierał od 4 do 5 testów. Testy zostały podzielone na trzy rodzaje:

- ciągi zupełnie losowe,
- ciągi, których ładne podciągi nachodzą na siebie,
- ciągi zawierające rozłączne ładne podciągi.

Ponadto w każdej grupie był test zawierający ładny podciąg zaczynający się na pierwszej pozycji ciągu.

Jakub Pachocki	Adam Polak, Jakub Radoszewski	Adam Polak
Treść zadania	Opracowanie	Program

Dostępna pamięć: 128 MB. *OI, etap III, dzień drugi, 14.03.2013*

Gdzie jest jedynka?

Bajtazar wymyślił sobie pewną permutację[1] *P liczb od 1 do n. Bajtazar chce, abyś zgadł(a), na którym miejscu tej permutacji znajduje się liczba 1. Abyś nie musiał(a) zgadywać w ciemno, Bajtazar będzie udzielać Ci odpowiedzi na pytania postaci:*

- *f(i, j, d): czy różnica między i-tym a j-tym elementem permutacji P jest podzielna przez d, tzn. czy d | P[i] − P[j]?*
- *g(i, j): czy i-ty element permutacji P jest większy niż j-ty element permutacji P?*

W powyższych pytaniach i, j są dowolnymi indeksami ze zbioru $\{1, 2, \ldots, n\}$, zaś d jest dowolną liczbą całkowitą dodatnią.

Podczas zgadywanki możesz dowolnie wiele razy użyć pytania typu f, natomiast liczba wykonanych pytań typu g musi być jak najmniejsza.

Napisz program komunikujący się z biblioteką dostarczoną przez Bajtazara, który rozwiąże tę zagadkę.

Ocenianie

Niech $M(n)$ będzie najmniejszą możliwą liczbą pytań typu g, za pomocą której da się znaleźć jedynkę w permutacji o ustalonej długości n, niezależnie od tego, jaka jest ta permutacja. Twój program otrzyma punkty za dany test, tylko jeśli liczba pytań typu g, jaką wykona, nie przekroczy $M(n)$. Poza tym program musi zmieścić się w limicie czasowym, a więc wykonać rozsądną liczbę pytań typu f (choć niekoniecznie minimalną).

We wszystkich testach zachodzi warunek $1 \leqslant n \leqslant 500\,000$. W testach wartych 28% punktów zachodzi dodatkowy warunek $n \leqslant 5000$.

Komunikacja

Aby użyć biblioteki, należy wpisać na początku programu:

- **C/C++:** `#include "cgdzlib.h"`
- **Pascal:** `uses pgdzlib;`

Biblioteka udostępnia następujące funkcje i procedury:

- `inicjuj` – *zwraca liczbę n. Powinna zostać użyta dokładnie raz, na samym początku działania programu.*
  - **C/C++:** `int inicjuj();`
  - **Pascal:** `function inicjuj : LongInt;`

[1] Permutacja liczb od 1 do n to taki ciąg o długości n, w którym każda liczba od 1 do n występuje dokładnie raz.

- `f(i, j, d)` – *zadaje bibliotece Bajtazara pytanie typu f. Jej wynikiem jest jedna liczba: 1, jeśli $d \mid P[i] - P[j]$, a 0 w przeciwnym przypadku.*
  - **C/C++:** `int f(int i, int j, int d);`
  - **Pascal:** `function f(i, j, d : LongInt) : LongInt;`
- `g(i, j)` – *zadaje bibliotece Bajtazara pytanie typu g. Jej wynikiem jest jedna liczba: 1, jeśli $P[i] > P[j]$, a 0 w przeciwnym przypadku.*
  - **C/C++:** `int g(int i, int j);`
  - **Pascal:** `function g(i, j : LongInt) : LongInt;`
- `odpowiedz(k)` – *odpowiada bibliotece Bajtazara, że jedynka znajduje się na k-tej pozycji w permutacji (tzn. że $P[k] = 1$). Uruchomienie tej procedury/funkcji* **kończy działanie Twojego programu**.
  - **C/C++:** `void odpowiedz(int k);`
  - **Pascal:** `procedure odpowiedz(k : LongInt);`

Twój program **nie może** *czytać żadnych danych (ani ze standardowego wejścia, ani z plików).* **Nie może** *również nic wypisywać do plików ani na standardowe wyjście. Może pisać na standardowe wyjście diagnostyczne (*`stderr`*) – pamiętaj jednak, że zużywa to cenny czas.*

Przykładowe wykonanie

Poniższa tabela zawiera przykładowy ciąg pytań do biblioteki Bajtazara prowadzący do odgadnięcia pozycji, na której znajduje się jedynka.

Numer pytania	Wywołanie	Wynik	Wyjaśnienie
	`inicjuj`	*5*	$n = 5$
1	`f(1, 2, 2)`	*0*	$2 \nmid P[1] - P[2]$
2	`g(1, 2)`	*0*	$P[1] < P[2]$
3	`f(3, 2, 3)`	*1*	$3 \mid P[3] - P[2]$
4	`g(2, 5)`	*1*	$P[2] > P[5]$
5	`f(1, 3, 2)`	*1*	$2 \mid P[1] - P[3]$
6	`f(1, 4, 3)`	*1*	$3 \mid P[1] - P[4]$
	`odpowiedz(4)`		*Odpowiadamy, że $k = 4$.*

Po drugim pytaniu wiemy, że $P[2] \neq 1$. Stąd po trzecim pytaniu zachodzi jedna z możliwości: ($P[2] = 2$, $P[3] = 5$) lub ($P[2] = 4$, $P[3] = 1$), lub ($P[2] = 5$, $P[3] = 2$). Czwarte pytanie eliminuje pierwszą z tych możliwości. Piąte pytanie pozwala teraz stwierdzić, że $P[1] \in \{3, 4\}$. Skoro więc w szóstym pytaniu mamy $3 \mid P[1] - P[4]$, to $P[1] = 4$, $P[4] = 1$. Szukaną pozycją w permutacji jest $k = 4$.

Podana sekwencja pytań poprawnie znajduje pozycję jedynki w permutacji, jednak nie uzyskałaby żadnych punktów za ten test, gdyż w dowolnej permutacji pięciu elementów jedynkę da się znaleźć za pomocą co najwyżej jednego pytania typu g (tj. $M(5) = 1$). Zauważ, że liczba wykonanych pytań typu f nie gra tu roli.

Eksperymenty

W katalogu **/home/zawodnik/rozw/gdz** *na stacji roboczej dostępna jest przykładowa biblioteka, która pozwoli Ci przetestować poprawność formalną rozwiązania. Biblioteka wczytuje opis permutacji ze standardowego wejścia w następującym formacie:*

- *w pierwszym wierszu liczba całkowita n – długość permutacji;*
- *w drugim wierszu n pooddzielanych pojedynczymi odstępami liczb od 1 do n – kolejne elementy permutacji.*

Przykładowe wejście dla biblioteki znajduje się w pliku `gdz0.in`*. Po zakończeniu działania programu biblioteka wypisuje na standardowe wyjście informację, czy odpowiedź udzielona przez Twoje rozwiązanie była poprawna, oraz liczbę zadanych pytań typu g.*

W tym samym katalogu znajdują się przykładowe rozwiązania `gdz.c`*,* `gdz.cpp` *i* `gdz.pas` *korzystające z biblioteki. Rozwiązania te nie minimalizują liczby pytań typu g.*

Do kompilacji rozwiązania wraz z biblioteką służą polecenia:

- **C:** `gcc -O2 -static cgdzlib.c gdz.c -lm -o gdz`
- **C++:** `g++ -O2 -static cgdzlib.c gdz.cpp -lm -o gdz`
- **Pascal:** `ppc386 -O2 -XS -Xt gdz.pas`

Plik z rozwiązaniem i biblioteka powinny znajdować się w tym samym katalogu.

Rozwiązanie

W zadaniu mamy do czynienia z permutacją o długości n. Możemy zadawać pytania dwóch typów: „czy i-ty i j-ty wyraz dają taką samą resztę z dzielenia przez d?" oraz „czy i-ty wyraz jest większy od j-tego?". Naszym celem jest stwierdzenie, na której pozycji w permutacji znajduje się jedynka. Powinniśmy przy tym zadać minimalną możliwą (dla danej długości permutacji) liczbę pytań drugiego typu oraz niezbyt dużą liczbę pytań pierwszego typu (żeby nie przekroczyć limitów czasowych).

Warto odnotować, że gdyby nie ograniczenie na pytania drugiego typu, zadanie byłoby bardzo proste. Wystarczyłoby zastosować najprostszy algorytm wyznaczania minimum w tablicy, który wykonuje dokładnie $n-1$ porównań elementów (czyli pytań drugiego typu).

Rozwiązanie wolne

Spróbujmy najpierw poszukać rozwiązania, które wykonuje możliwie mało pytań drugiego typu, natomiast nie przejmujmy się na razie liczbą pytań pierwszego typu.

Zauważmy, że jedynka jest jedynym elementem permutacji, dla którego istnieje element większy o $n-1$. Najprostsze rozwiązanie może więc polegać na przejrzeniu wszystkich par różnych indeksów $i < j$ w permutacji i zadaniu dla każdej z nich pytania pierwszego typu z parametrem $d = n-1$, tzn. sprawdzeniu, czy $P[i] - P[j]$ dzieli się przez $n-1$. To pozwoli zidentyfikować dwa indeksy x i y, dla których $n-1$

dzieli $P[x] - P[y]$. Niestety nie będziemy wciąż wiedzieli, czy $P[x] = 1$ i $P[y] = n$, czy jest odwrotnie. W tym miejscu zastosujemy więc pytanie drugiego typu i sprawa stanie się jasna.

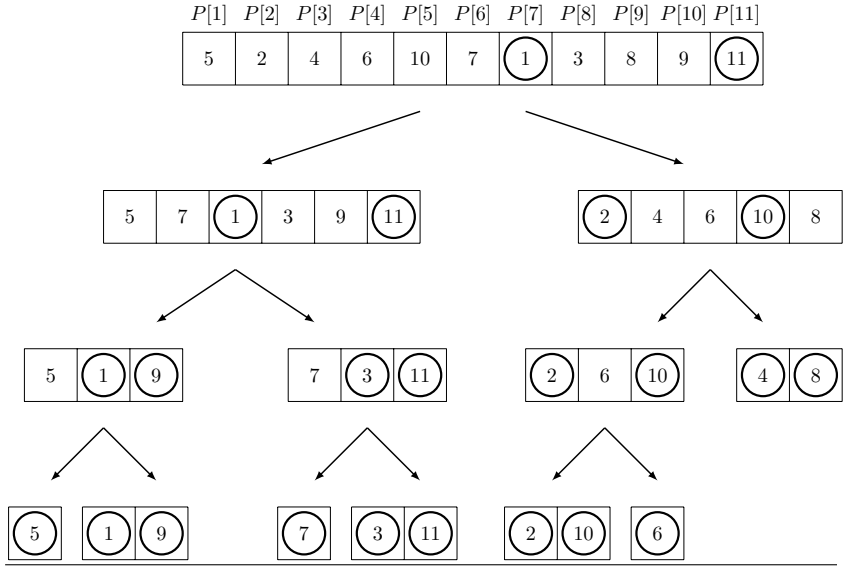
Czy to rozwiązanie rzeczywiście zadaje minimalną liczbę pytań drugiego typu? Okazuje się, że tak: nie licząc przypadku $n = 1$, konieczne jest zadanie przynajmniej jednego pytania drugiego typu. Dzieje się tak, gdyż pytania pierwszego typu nie są w stanie odróżnić permutacji $1, 2, \ldots, n$ od permutacji $n, n-1, \ldots, 1$ i, ogólnie, permutacji $P[i]$ od permutacji $P'[i] = n + 1 - P[i]$.

Opisane rozwiązanie zadaje $O(n^2)$ pytań pierwszego typu i optymalną liczbę pytań drugiego typu. Przykładowa implementacja znajduje się w plikach `gdzs0.c` oraz `gdzs1.pas`. Zgodnie z treścią zadania, takie rozwiązanie uzyskiwało na zawodach 28% punktów.

Rozwiązanie wzorcowe

Poprzednie rozwiązanie sprowadzało się do wyznaczenia pary indeksów x, y takiej, że $\{P[x], P[y]\} = \{1, n\}$, wyłącznie przy pomocy pytań pierwszego typu. Znając taką parę, mogliśmy łatwo znaleźć jedynkę, używając jednego pytania drugiego typu. W rozwiązaniu wzorcowym osiągniemy to samo, wykonując tylko $O(n \log n)$ pytań pierwszego typu. W tym celu skorzystamy z metody „dziel i zwyciężaj”.

W fazie „dziel” wybierzemy dowolny element permutacji (np. pierwszy) i podzielimy pozostałe elementy na te o takiej samej parzystości co wybrany i te o przeciwnej parzystości. Wystarczy do tego $n - 1$ pytań pierwszego typu z parametrem $d = 2$. Następnie wywołamy się rekurencyjnie na obu połówkach, szukając ich minimów



Rys. 1: Przykład działania rozwiązania wzorcowego dla permutacji o $n = 11$ elementach. Kółkami zaznaczono minima i maksima.

i maksimów. Wiemy, że minimum i maksimum całej permutacji znajdują się wśród minimów i maksimów tych dwóch połówek. Aby je wyznaczyć, w fazie „zwyciężaj" sprawdzimy cztery pary elementów, zadając pytanie, czy ich różnica dzieli się przez $n-1$. Na końcu nie będziemy wiedzieli, który z wyznaczonych elementów jest minimum, a który maksimum permutacji; stwierdzimy to, zadając jedyne pytanie drugiego typu.

Kolejne poziomy wywołania rekurencyjnego obsługujemy podobnie, z niewielkimi zmianami: nie sprawdzamy parzystości tylko podzielność przez kolejne potęgi dwójki, a szukając pary elementów stanowiącej minimum i maksimum, nie dzielimy przez $n-1$, tylko liczbę odpowiednio mniejszą. Dokładniej, elementy, z którymi mamy do czynienia na i-tym poziomie rekursji, stanowią ciąg arytmetyczny o różnicy 2^i, więc znając jego długość, możemy wywnioskować, jaka jest wartość bezwzględna różnicy minimum i maksimum ciągu. Wywołania rekurencyjne przerywamy, gdy pozostały nam do rozważenia co najwyżej dwa elementy. Przykład pełnego drzewa rekursji znajduje się na rys. 1.

Czas działania rozwiązania wzorcowego opisuje równanie

$$T(n) = 2T(n/2) + O(n),$$

którego rozwiązaniem jest $T(n) = O(n \log n)$. Implementacje można znaleźć w plikach `gdz.cpp` oraz `gdz1.pas`.

Możemy poczynić jeszcze jedną obserwację, która przyspiesza program dla niektórych wartości n. A mianowicie, jeśli na którymś poziomie rekursji dostajemy fragment nieparzystej długości, to po podziale na dwie części nie musimy się wywoływać na krótszej połówce. Wynika to z tego, że minimum i maksimum przy podziale na pewno wpadną do dłuższej połówki.

Biblioteka

Biblioteka używana do sprawdzania rozwiązań wczytuje z wejścia opis permutacji i odpowiada zgodnie z tym opisem. Jedyna jej „złośliwość" polega na wykorzystaniu faktu, że dla $n > 1$ nie da się obejść bez pytań drugiego typu – jeśli zawodnik próbuje odpowiedzieć przed zadaniem jakiegokolwiek tego typu pytania, biblioteka sygnalizuje błędną odpowiedź.

Testy

Przygotowano 13 losowych testów, z czego pierwsze dwa (jeden z nich zawiera przypadek $n = 1$) są zgrupowane.

Istnieje pewien heurystyczny argument, że w przypadku tego zadania testy losowe są równie dobre co każde inne: gdyby było inaczej i testy losowe miałyby być w jakimś sensie prostsze, rozwiązanie mogłoby zaczynać od wylosowania permutacji i złożenia jej z permutacją wejściową, co w wyniku dałoby również losową permutację, na której można by dalej operować.

W niektórych testach n jest potęgą dwójki. Dzięki temu implementacje rozwiązania wzorcowego muszą wykonać wszystkie wywołania rekurencyjne (za każdym razem fragment ma długość parzystą).

Karol Pokorski
Treść zadania, Opracowanie

Lech Duraj
Program

Dostępna pamięć: 128 MB.

OI, etap III, dzień drugi, 14.03.2013

Laser

Kapitan Bajtazar poluje na odcinkowce na planecie Tumulum VI. Jego statek stoi w miejscu lądowania (które będziemy uważać za początek układu współrzędnych) i wyposażony jest w laser ogłuszający. Laserem tym można obracać, tak aby promień lasera był skierowany pod dowolnym kątem. Zasilania statku wystarczy na co najwyżej k strzałów, z których każdy może być oddany pod dowolnie wybranym kątem. W momencie, gdy laser jest włączony, nie można nim obracać.

Na planecie znajduje się n odcinkowców – każdy z nich jest istotą jednowymiarową (odcinkiem) o końcach w punktach o współrzędnych dodatnich i całkowitych. Celem Bajtazara jest trafienie promieniem lasera jak największej liczby odcinkowców, przy czym zabronione jest trafienie jakiegoś odcinkowca więcej niż raz – kapitan chce je sprzedać z dobrym zyskiem, a do tego muszą być w nienagannym stanie fizycznym i psychicznym. Promień lasera rozchodzi się wzdłuż prostej, a gdy trafia odcinek, przenika przez niego i biegnie dalej. Jeśli promień lasera przejdzie przez sam koniec lub wzdłuż odcinkowca, to również jest on trafiony.

Napisz program, który wyznaczy maksymalną liczbę odcinkowców, które można trafić promieniem lasera, zgodnie z podanymi powyżej zasadami.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie liczby całkowite k i n ($1 \leqslant k \leqslant 100$, $1 \leqslant n \leqslant 500\,000$), oddzielone pojedynczym odstępem. W kolejnych n wierszach opisane są odcinkowce, po jednym w wierszu. W każdym z tych wierszy znajdują się po cztery dodatnie liczby całkowite x_1, y_1, x_2, y_2 ($1 \leqslant x_1, y_1, x_2, y_2 \leqslant 1\,000\,000$), rozdzielone pojedynczymi odstępami. Liczby takie reprezentują odcinkowca o końcach (x_1, y_1) i (x_2, y_2).

W testach wartych 36% punktów zachodzi dodatkowy warunek $k \leqslant 2$, w testach wartych 45% punktów zachodzi $n \leqslant 2000$ i $k \leqslant 30$, natomiast w testach wartych 81% punktów zachodzi $n \leqslant 200\,000$ i $k \leqslant 50$.

Wyjście

Twój program powinien wypisać w pierwszym (i jedynym) wierszu standardowego wyjścia dokładnie jedną liczbę całkowitą: maksymalną liczbę odcinkowców, które można trafić promieniem lasera (każdego dokładnie raz), wykonując co najwyżej k strzałów.

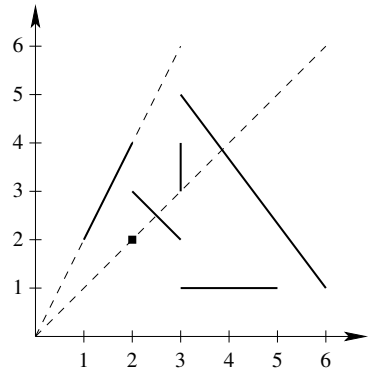
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
5
```



Wyjaśnienie do przykładu: *Wystarczy uruchomić laser dwukrotnie. Promień lasera zaznaczono na rysunku linią przerywaną.*

Testy „ocen”:

`1ocen`: $k = 4$, $n = 5$, *mały test poprawnościowy;*

`2ocen`: $k = 2$, $n = 5$, *tylko odcinkowce zerowej długości;*

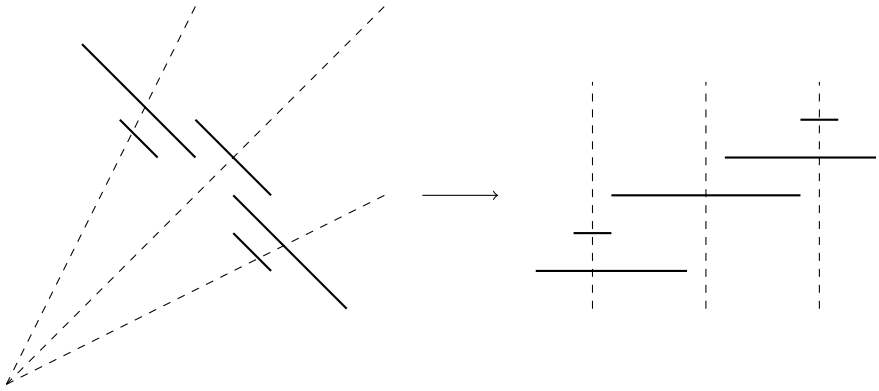
`3ocen`: $k = 2$, $n = 3$, *złośliwy test poprawnościowy;*

`4ocen`: $k = 3$, $n = 500\,000$, *test maksymalnego rozmiaru.*

Rozwiązanie

Niezbyt długą historyjkę z zadania *Laser* można jeszcze skrócić i powiedzieć, że sprowadza się ono do wyznaczenia k półprostych (zaczepionych w początku układu współrzędnych) w taki sposób, aby przecięły łącznie jak największą liczbę zadanych odcinków, przy czym żadnego odcinka nie wolno przeciąć dwa razy.

Geometryczna otoczka tego zadania może trochę odstraszać, szczęśliwie jednak okazuje się, że stanowi ona jedynie szczegół implementacyjny. Łatwo przekonać się, że nie potrzebujemy znać dokładnych współrzędnych końców odcinka, a jedynie kąty, pod jakimi należy ustawić laser, aby jego promień przechodził przez te końce. Wówczas każdy odcinek możemy utożsamić z przedziałem kąta lasera, który trafia w ów odcinek (rys. 1). Dzięki tej obserwacji do rozwiązania mamy nieco łatwiejsze zadanie: *Danych jest n odcinków na prostej, które należy przeciąć co najwyżej k pionowymi liniami, aby jak najwięcej odcinków zostało przeciętych, a żaden nie został przecięty dwukrotnie.* Zauważmy, że korzystamy tutaj z faktu, że wszystkie odcinki są zawarte w jednej ćwiartce układu współrzędnych. Od tego momentu możemy zapomnieć o oryginalnym zadaniu i skupić się na jego nowej wersji.



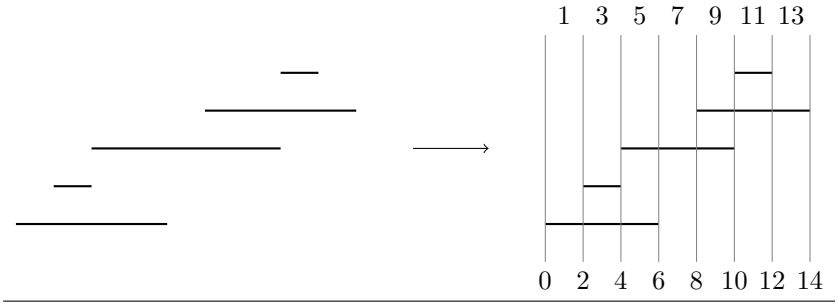
Rys. 1: Sprowadzenie problemu dwuwymiarowego do jednowymiarowego.

Rozwiązanie wzorcowe

Ponieważ kolejność wykonywania strzałów nie ma znaczenia, będziemy je rozpatrywać w kolejności rosnących współrzędnych x. To pozwoli nam wprowadzić do problemu przestrzeń stanów i myśleć w kategoriach programowania dynamicznego. Niech $r[p, x]$ oznacza maksymalną liczbę odcinków, które można przeciąć (nie przecinając żadnego dwukrotnie), wykorzystując dokładnie p strzałów, jeśli ostatni strzał nastąpił nie później niż na współrzędnej x.

Przy takiej definicji stanów jest nieskończenie wiele. Zauważmy jednak, że jeśli między współrzędnymi x_1 i x_2 nie zaczyna się ani nie kończy żaden odcinek – można strzelić w dowolną z nich, a wynik pozostanie niezmieniony. Takie współrzędne można więc bezpiecznie utożsamić (a nawet przeskalować tak, aby były, na przykład, niedużymi liczbami całkowitymi; patrz rys. 2).

Co ważne, nie można sobie pozwolić tylko na analizę punktów pomiędzy krańcami odcinków (na rysunku: punktów o współrzędnych nieparzystych), gdyż w jednej współrzędnej x mogą jednocześnie zaczynać i kończyć się różne odcinki (na rysunku



Rys. 2: Przeskalowanie współrzędnych. Współrzędne parzyste odpowiadają początkom i końcom odcinków, a współrzędne nieparzyste – maksymalnym przedziałom, które nie zawierają krańców odcinków.

dzieje się tak np. dla $x = 10$). Z tego samego powodu oraz ze względu na konieczność co najwyżej jednokrotnego trafienia każdego odcinka, nie wystarczy także analizowanie tylko początków i końców odcinków (na rysunku: punktów o współrzędnych parzystych). Można to także zilustrować na naszym prostym przykładzie: gdyby chcieć wykonać dokładnie trzy strzały, możliwe jest trafienie wszystkich odcinków z powyższego rysunku – jednak aby tego dokonać, niezbędne jest dokonanie strzału w $x = 7$.

Od tego momentu rozważana przestrzeń stanów jest skończona, a nawet dość mała – zawiera bowiem $\Theta(nk)$ stanów. Pozostaje nadal pytanie, jak obliczać wartości $r[p, x]$. Załóżmy, że znane są już wyniki dla wszystkich stanów dla $p-1$ strzałów oraz wyniki dla stanów $r[p, x']$, dla $x' < x$. Wówczas rozpatrujemy dwa przypadki, w zależności od tego, czy wykonujemy strzał we współrzędnej x.

Jeśli nie wykonujemy strzału w x, to wszystkie p strzałów musimy wykonać (zgodnie z zasadami zadania) przed współrzędną x, zatem będzie ich $r[p, x-1]$.

W drugim przypadku sytuacja jest bardziej skomplikowana. Musimy zadbać o to, aby żaden z odcinków przeciętych w strzałach wcześniejszych nie został przecięty w p-tym strzale. Biorąc pod uwagę kolejność obliczeń, wystarczy zapewnić, aby zbiory odcinków przeciętych w $(p-1)$-szym i p-tym strzale były rozłączne. Zatem współrzędna $(p-1)$-szego strzału musi być mniejsza od współrzędnych wszystkich początków odcinków zawierających współrzędną x.

Aby móc zaimplementować wydajne obliczanie wyników dla poszczególnych stanów, warto najpierw wyznaczyć kilka pomocniczych wartości:

- $w[x]$ – liczba odcinków, które trafia strzał wykonany w x,
- $l[x]$ – najmniejsza współrzędna początku odcinka trafionego strzałem w x.

W drugim przypadku zestrzelimy więc $r[p, l[x]-1] + w[x]$ odcinków. Ostatecznie rekurencja (pomijając warunki brzegowe) ma postać:

$$r[p, x] = \max(r[p, x-1], r[p, l[x]-1] + w[x]).$$

Wartości $w[x]$ można obliczyć, rozbijając każdy odcinek na dwa zdarzenia: początek (który zwiększa liczbę przeciętych w danym miejscu odcinków o jeden) oraz koniec odcinka (który tę wartość zmniejsza). Takie zdarzenia można posortować względem współrzędnej x, remisy rozstrzygając na korzyść punktów rozpoczynających odcinki. Następnie wystarczy już tylko rozpatrzyć te zdarzenia zgodnie z nowym posortowaniem i zapisać dla kolejnych x liczbę aktualnie przeciętych odcinków.

Wartości $l[x]$ można obliczyć analogicznie, najłatwiej będzie jednak analizować zdarzenia względem malejących wartości x. Utrzymujemy minimalną współrzędną dla początków odcinków przecinających x i, jeśli napotkamy koniec nowego odcinka, sprawdzamy, czy jego początek jest wcześniej niż aktualnie znane minimum.

W ten sposób dostajemy rozwiązanie o złożoności obliczeniowej $O(nk)$ i takiej samej złożoności pamięciowej. Tę drugą można łatwo zredukować, zauważając, że aby obliczyć wyniki dla p strzałów, wystarczy pamiętać tylko wyniki dla $p-1$ strzałów. Zastosowanie tej optymalizacji pozwala zredukować złożoność pamięciową do $O(n+k)$. Wyżej opisane rozwiązanie zostało zaimplementowane w plikach `las.cpp`, `las1.pas` i `las2.cpp` i jest wystarczające do uzyskania maksymalnej punktacji. Rozwiązania bez optymalizacji pamięci pozwalały uzyskać na zawodach około 80% maksymalnej punktacji (plik: `lasb1.cpp`).

Rozwiązanie alternatywne

Na problem można także spojrzeć z innej perspektywy i spróbować zastosować programowanie dynamiczne inaczej. Zamiast dla ustalonego x (oraz liczby strzałów) pamiętać maksymalną liczbę odcinków możliwych do przecięcia, można zapytać odwrotnie: jaka jest najmniejsza możliwa współrzędna końca odcinka, który został trafiony, przy założeniu, że użyto p strzałów oraz łącznie zestrzelono t odcinków. Można o tym myśleć jako o minimalizacji kąta ostatniego, p-tego strzału lasera dla wyniku t. Rozwiązanie to ma gorszą złożoność czasową: $O(nk \log(nk))$, z uwagi na konieczność zastosowania wyszukiwania binarnego, aby wyszukiwać najmniejszy możliwy kąt zapewniający zadany wynik. Przy prawidłowej implementacji rozwiązanie tego typu miało szansę uzyskać około 80% maksymalnej punktacji.

Rozwiązania naiwne oraz błędne

W przypadku braku pomysłu na rozwiązanie szybkie i poprawne, zawodnicy czasami podejmują próby uzyskania częściowej punktacji. Przykładowe rozwiązanie, które jest poprawne, ale niezbyt szybkie, opiera się na metodzie przeszukiwania z nawrotami: spośród możliwych pozycji potencjalnych strzałów wystarczy wybrać co najwyżej k – można spróbować albo przetestować je wszystkie, albo pewien ich podzbiór (i liczyć na to, że rozwiązanie optymalne zostanie znalezione). Takie rozwiązania miały szansę uzyskać około 20–30% punktów i zostały zaimplementowane w plikach `lass1.cpp` oraz `lass2.pas`.

Można było także zadanie próbować rozwiązywać zachłannie: na przykład pierwszy strzał wybrać tak, aby przeciął największą liczbę odcinków, kolejny, aby przeciął największą liczbę pozostałych itd. Takie rozwiązania (błędne algorytmicznie) warte były najczęściej 0 punktów. Rozwiązania tego typu zaimplementowane są w plikach `lasb2.cpp` oraz `lasb3.cpp`.

Testy

Rozwiązania były oceniane za pomocą 11 grup testów, z których większość zawierała po trzy testy:

- Testy z grupy a były testami losowymi; zadbano jednak, aby laser przecinał pewne zbiory końców odcinków pod tym samym kątem.
- Testy z grupy b zawierały odcinki krótkich długości; możliwe było przestrzelenie prawie wszystkich z nich i bezpieczne dokonanie k strzałów (aby żadnego odcinka nie przeciąć dwa razy).
- Testy z grupy c zostały przygotowane specjalnie w celu obalenia rozwiązań zachłannych – strzał w największą liczbę odcinków (lub jedno z kilku największych) uniemożliwiał osiągnięcie najlepszego wyniku.

Jan Kanty Milczek	Jan Kanty Milczek Bartosz Tarnawski	Bartosz Tarnawski
Treść zadania	Opracowanie	Program

Dostępna pamięć: 128 MB. *OI, etap III, dzień drugi, 14.03.2013*

Polaryzacja

Każdy wiedział, że to tylko kwestia czasu. Ale cóż z tego? Kiedy zagrożenie wisi nad ludźmi przez lata, staje się częścią codzienności. Traci swoją wartość...

Dzisiaj stała się jawna treść listu bitockiego chara Ziemobita do bajtockiego króla Bajtazara. Bitocja zażądała aneksji całej Bajtocji, grożąc Bitowym Magnesem Polaryzującym (BMP). W wyniku jego działania każda droga Bajtocji stałaby się jednokierunkowa. Wróg doskonale wie, że byłby to cios, którego sąsiad może nie wytrzymać, bo infrastruktura Bajtocji jest minimalna – między każdą parą miast da się przejechać na dokładnie jeden sposób.

Jak może wyglądać infrastruktura Bajtocji po działaniu BMP? Oblicz minimalną i maksymalną liczbę par miast, takich że z jednego będzie się dało przejechać do drugiego, nie naruszając skierowania dróg.

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita n ($1 \leqslant n \leqslant 250\,000$), oznaczająca liczbę miast w Bajtocji. W kolejnych $n-1$ wierszach znajdują się opisy dróg. Każdy z nich składa się z dwóch liczb całkowitych u i v ($1 \leqslant u < v \leqslant n$), oznaczających, że istnieje (na razie jeszcze dwukierunkowa) droga łącząca miasta o numerach u i v.

W testach wartych 60% punktów zachodzi dodatkowy warunek $n \leqslant 10\,000$, a w części tych testów, wartych 30% punktów, zachodzi warunek $n \leqslant 100$.

Wyjście

W pierwszym wierszu standardowego wyjścia powinny znaleźć się dwie liczby całkowite oddzielone pojedynczym odstępem. Pierwsza z nich to minimalna, a druga – maksymalna liczba par różnych miast, między którymi będzie można przejechać po spolaryzowaniu dróg.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*

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

a dla danych wejściowych:	*poprawnym wynikiem jest:*
8	7 28
1 2	
2 3	
3 4	
4 5	
5 6	
6 7	
7 8	

Testy „ocen":

`1ocen:` *$n = 10$, test typu „grzebień";*

`2ocen:` *$n = 15$, pełne drzewo binarne;*

`3ocen:` *$n = 100$, dwie „gwiazdy" połączone krawędzią;*

`4ocen:` *$n = 10\,000$, dwie „gwiazdy" połączone krawędzią;*

`5ocen:` *$n = 250\,000$, „gwiazda" o maksymalnym rozmiarze.*

Rozwiązanie

W zadaniu dane jest n-wierzchołkowe drzewo T. Niech s będzie funkcją przyporządkowującą każdej nieskierowanej krawędzi $\{x, y\}$ drzewa jedno z dwóch możliwych skierowań (x, y) lub (y, x). Funkcję tę nazwiemy *skierowaniem* drzewa, a przez T_s oznaczymy drzewo powstałe z T po skierowaniu jego krawędzi według s.

Powiemy, że para różnych wierzchołków (x, y) jest *dobra* w skierowaniu s, jeżeli w T_s istnieje ścieżka z x do y. Zadanie polega na znalezieniu minimalnej i maksymalnej liczby dobrych par w T, jakie możemy uzyskać przy różnych skierowaniach tego drzewa.

Zauważmy, że żadne skierowanie nie da nam wyniku mniejszego niż $n - 1$. Dzieje się tak dlatego, że każda krawędź, niezależnie od skierowania, tworzy co najmniej jedną dobrą parę. Łatwo zauważyć, że dokładnie taki wynik uzyskamy, jeśli skierujemy krawędzie tak, aby każdy wierzchołek był albo *źródłem* (wszystkie krawędzie z nim incydentne są skierowane od niego), albo *ujściem* (wszystkie krawędzie z nim incydentne są skierowane do niego). W tym celu możemy wykonać dwukolorowanie naszego drzewa, a następnie skierować wszystkie krawędzie tak, aby prowadziły z wierzchołka pierwszego koloru do wierzchołka koloru drugiego.

Widzimy zatem, że pierwszą z liczb, o jakich wypisanie jesteśmy proszeni na wyjściu, jest zawsze $n - 1$. Możemy podejrzewać, że najtrudniejsza część zadania jest dopiero przed nami...

W celu wyznaczenia maksymalnej liczby dobrych par pomocne nam będą następujące obserwacje.

Obserwacja 1. Jeśli odwrócimy skierowanie w całym drzewie, to liczba dobrych par nie zmieni się.

Jest to oczywiste, gdyż każdej dobrej parze (x, y) w starym skierowaniu będzie odpowiadać dobra para (y, x) w nowym skierowaniu.

Obserwacja 2. W optymalnym rozwiązaniu tylko liście mogą być źródłami i ujściami.

Intuicyjnym jest, że jeśli chcąc zminimalizować liczbę dobrych par, maksymalizowaliśmy liczbę wierzchołków będących źródłami i ujściami, to chcąc zmaksymalizować wynik, powinniśmy minimalizować liczbę źródeł i ujść. Jeśli jakikolwiek wierzchołek wewnętrzny (nie liść) byłby źródłem lub ujściem, to moglibyśmy wybrać jego dowolne poddrzewo i odwrócić skierowanie wszystkich jego krawędzi. Zgodnie z obserwacją 1 liczba dobrych par w tym poddrzewie nie zmieniłaby się, za to w całym drzewie doszłaby co najmniej jedna nowa dobra para. Od teraz we wszystkich rozważaniach zakładamy, że żaden wierzchołek wewnętrzny nie jest źródłem ani ujściem.

Jeżeli istnieje wierzchołek, który pojawia się na ścieżce z x do y dla każdej dobrej pary *liści* (x, y), nazwiemy go *superwierzchołkiem*. Skierowania zawierające superwierzchołek mają bardzo regularną strukturę – ukorzeniwszy drzewo w superwierzchołku, zobaczymy, że każde jego poddrzewo jest w całości skierowane albo do niego, albo od niego. Wykażemy teraz, że prawdziwa jest

Obserwacja 3. Optymalne rozwiązanie zawsze zawiera superwierzchołek.

Jeśli w skierowaniu nie istnieje superwierzchołek, to istnieje para wierzchołków (x, y) taka, że:

- istnieje ścieżka z x do y;
- każdy wierzchołek pomiędzy x a y (może ich być zero, jeśli istnieje krawędź $\{x, y\}$) ma stopień 2;
- od x wychodzi co najmniej jedna krawędź nieprowadząca do y;
- do y wchodzi co najmniej jedna krawędź nieprowadząca od x.

Udowodnienie tego faktu pozostawiamy jako ćwiczenie dla Czytelnika (dowód jest intuicyjny i prosty, ale mozolny). Oznaczmy teraz przez m_x liczbę wierzchołków, z których istnieje ścieżka do x, a przez m_y liczbę wierzchołków, do których istnieje ścieżka z y. Jeśli $m_x \leqslant m_y$, to odwracając skierowanie poddrzew wychodzących z wierzchołka x (oprócz tego zawierającego y), poprawimy wynik o co najmniej jeden. Jeśli zaś $m_y \leqslant m_x$, to odwracając skierowanie poddrzew wchodzących do y (oprócz tego zawierającego x), również poprawimy wynik o co najmniej jeden. Wykonajmy jedno z tych odwróceń i znowu poszukajmy superwierzchołka. Jeśli znów takiego nie będzie, znajdźmy nową parę (x, y) i powtórzmy rozumowanie. Jako że wynik nie może się zwiększać w nieskończoność, w końcu dotrzemy do sytuacji z superwierzchołkiem.

Rozwiązanie $O(n^2)$

Skonstruujemy teraz rozwiązanie bazujące na powyższych obserwacjach. Dla każdego wierzchołka drzewa chcemy wiedzieć, jaki byłby wynik, gdyby to on był superwierzchołkiem.

Ustalmy zatem wierzchołek drzewa v. Policzenie liczby dobrych par znajdujących się wewnątrz poddrzew wierzchołka v możemy wykonać np. algorytmem DFS w czasie $O(n)$. Liczba ta nie zależy od skierowania poddrzew (obserwacja 1). Do wyniku dochodzi jeszcze iloczyn liczby wierzchołków w poddrzewach skierowanych do v i liczby wierzchołków w poddrzewach skierowanych od v. Musimy zatem dla każdego poddrzewa tak wybrać skierowanie, aby ten iloczyn był jak największy. Iloczyn postaci $a(n-1-a)$ jest największy, gdy a jest możliwie najbliższe $\frac{n-1}{2}$. Używając algorytmu pakowania plecaka (wydawania reszty), ustalamy, jak bardzo możemy się zbliżyć do tej wartości. Złożoność czasowa tego algorytmu to $O(n \cdot \deg(v))$, gdzie $\deg(v)$ to stopień wierzchołka v.

Ostatecznie, łączna złożoność czasowa dla całego drzewa będzie więc wynosiła $O\big(\sum_v (n + n \cdot \deg(v))\big) = O(n^2)$. Rozwiązanie to zostało zaimplementowane w plikach `pols5.cpp` oraz `pols6.pas`, dostaje ok. 50% punktów.

Rozwiązanie $O(n^2)$, wersja 2

Zauważmy, że jeśli uruchomimy rozwiązanie problemu plecakowego dla wierzchołka v, w którym jedno z poddrzew (oznaczmy, że jest ukorzenione w wierzchołku u) ma rozmiar większy niż $\frac{n}{2}$, to wszystkie inne poddrzewa skierujemy w przeciwną stronę. W takim razie, całe poddrzewo ukorzenione w v zostanie skierowane w stronę u. Zatem równie dobre rozwiązanie moglibyśmy uzyskać, rozpatrując problem plecakowy skonstruowany w wierzchołku u. Kontynuując to rozumowanie, dotrzemy w końcu do wierzchołka, którego wszystkie poddrzewa mają rozmiar nie większy niż $\frac{n}{2}$. Taki wierzchołek nazywa się *centroidem* drzewa. Każde drzewo ma co najwyżej dwa centroidy, można zatem wykorzystać spostrzeżenie, że superwierzchołkiem musi być centroid, i tylko dla nich rozwiązywać problem plecakowy. Niestety, jeśli zrobimy to w czasie $O(n^2)$, to nie uzyskamy lepszej złożoności niż w poprzednim rozwiązaniu. Jednak w przypadku poprzedniego rozwiązania czas wykonania zdominowany jest przez n-krotne przeszukiwanie drzewa. Jeżeli wyznaczymy najpierw centroid, to większość czasu zajmą operacje na tablicy i rozwiązanie będzie działać istotnie szybciej na wszystkich wygenerowanych testach. Takie rozwiązanie uzyskiwało ok. 60% punktów. Zostało zaimplementowane w plikach `pols3.cpp` oraz `pols4.pas`.

Rozwiązanie wzorcowe $O(n\sqrt{n})$

Rozwiązanie można przyspieszyć, sprytniej rozwiązując problem plecakowy. Zauważmy, że występuje w nim co najwyżej n liczb o łącznej sumie $n-1$. Takie programowanie dynamiczne można zaimplementować w czasie $O(n\sqrt{n})$. Korzystamy ze spostrzeżenia, że przedmiotów większych niż $\sqrt{n}$ jest co najwyżej $\sqrt{n}$, natomiast przedmioty mniejsze niż $\sqrt{n}$ rozpatrujemy w pakietach złożonych z jednakowych przedmiotów. Rozwiązanie to zostało opisane szczegółowo w opracowaniu zadania *Banknoty* z XII OI [12]. Jeden ze sposobów na przyspieszenie rozwiązania został pokrótce opisany poniżej.

Załóżmy, że mamy pewną grupę przedmiotów tej samej wielkości s. Rozpatrując $(j+1)$-szy przedmiot z tej grupy, sprawdzamy jedynie, czy osiągalne są sumy postaci $x + s$, gdzie x jest sumą osiągalną dopiero po dodaniu j-tego przedmiotu wielkości s.

Wtedy łączny czas przetwarzania przedmiotów, które nie są rozpatrywane jako pierwsze w swoim rozmiarze, szacuje się przez sumaryczną liczbę wartości, które w pewnym momencie stały się dostępne. Tych zaś jest najwyżej n. Takie rozwiązanie zostało zaimplementowane w plikach `pol.cpp` i `pol2.pas`. Rozwiązanie autorskie, które jest podobne i trochę prostsze, zostało zaimplementowane w pliku `pol1.cpp`.

Rozwiązanie alternatywne $O(n\sqrt{n})$

Każda liczba ze zbioru $\{0, 1, \ldots, m\}$ jest sumą elementów pewnego podzbioru $\{2^0, 2^1, 2^2, \ldots, 2^{t-1}, m - (2^t - 1)\}$ dla $t = \lfloor \log_2 m \rfloor$ i odwrotnie, suma elementów każdego podzbioru tego (multi)zbioru należy do $\{0, 1, \ldots, m\}$.

Jeżeli m przedmiotów ma taką samą wielkość s, to zamiast m-krotnie uaktualniać tablicę dostępnych wartości, możemy „posklejać" przedmioty z tej grupy w bloki wielkości $s, 2s, \ldots, 2^{t-1}s, (m - (2^t - 1))s$.

Złożoność czasowa takiego rozwiązania łatwo szacuje się przez $O(n\sqrt{n}\log n)$. Tak naprawdę złożoność to $O(n\sqrt{n})$, ale dowód tego nie jest istotą tego opracowania. Takie rozwiązanie zostało zaimplementowane w plikach `pol5.cpp` oraz `pol6.pas`.

Rozwiązanie $O(n^2)$, wersja 3

Nie korzystając z obserwacji o dystrybucji wielkości przedmiotów, można przyspieszyć rozwiązanie problemu plecakowego, korzystając z masek bitowych. Zamiast trzymać tablicę zmiennych logicznych informującą, czy dana wartość jest osiągalna, reprezentujemy ją jako kolejne bity słów 32-bitowych. Dzięki temu zmniejsza się stała przy aktualizacji dostępnych stanów po dodaniu nowego elementu. Rozwiązanie działające w czasie $O(n^2)$ z tą optymalizacją zostało zaimplementowane w plikach `pols1.cpp` oraz `pols2.pas`. Dostaje ok. 70–80% punktów.

Rozwiązanie alternatywne $O(n\sqrt{n})$, wersja 2

Tak samo korzystając z masek bitowych, można zmniejszyć stałą w rozwiązaniu $O(n\sqrt{n})$ (pliki `pol3.cpp` oraz `pol4.pas`). Rozwiązanie to działa szybciej od wzorcowego, przy czym różnica w czasie wykonania staje się znacząca dopiero dla danych wejściowych przekraczających limity z treści zadania.

Rozwiązania wolne

Istnieje rozwiązanie wielomianowe nie korzystające z powyższych obserwacji, które także opiera się na programowaniu dynamicznym. W tym rozwiązaniu wypełniamy tablicę d, w której $d[v, x, y]$ to maksymalny wynik częściowy dla poddrzewa ukorzenionego w wierzchołku v przy założeniu, że

(a) istnieje x wierzchołków w poddrzewie v, z których prowadzi ścieżka do v,

(b) istnieje y wierzchołków w poddrzewie v, do których prowadzi ścieżka z v.

Wartości tablicy można obliczyć podczas przeszukiwania drzewa, a stan można wyznaczyć w czasie $O(n)$, jeżeli pamięta się dodatkowo najlepsze wyniki przy założeniu samego warunku (a) lub (b). Sumaryczny czas działania tego algorytmu to $O(n^4)$,

został on zaimplementowany w plikach `pols7.cpp` oraz `pols8.pas`. Dostaje ok. 30% punktów.

Najprostszym, choć niestety bardzo wolnym, podejściem było generowanie wszystkich (lub wielokrotne losowanie) skierowań krawędzi, a następnie liczenie liczby dobrych par algorytmem DFS (wywoływanym raz lub z każdego wierzchołka). Przykładowe rozwiązanie wykładnicze, które sprawdza wszystkie możliwe skierowania krawędzi w czasie $O(2^n \cdot n)$, zostało zaimplementowane w plikach `pols9.cpp` i `pols10.pas`. Takie rozwiązanie dostawało 10% punktów.

Rozwiązania niepoprawne

Ponieważ maksymalnym wynikiem jest $\binom{n}{2}$, zadanie wymaga użycia arytmetyki 64-bitowej. Rozwiązanie o złożoności $O(n\sqrt{n})$, ale używające do reprezentacji wyniku typu 32-bitowego, otrzymuje 80% punktów (`polb7.cpp`).

Zaimplementowane rozwiązania, które dostają 0-10% punktów, to:

1. Wielokrotne generowanie losowego skierowania krawędzi (`polb1.cpp`).
2. Szukanie centrum (środka średnicy drzewa) zamiast centroidu (`polb2.cpp`).
3. Szukanie wierzchołka o największym stopniu zamiast centroidu (`polb3.cpp`).
4. Ustawianie synów centroidu w ciąg według rosnących wielkości poddrzew, szukanie prefiksu tego ciągu o sumie wielkości poddrzew możliwie zbliżonej do $\frac{n-1}{2}$ (`polb4.cpp`).
5. Heurystyka polegająca na tym, że po znalezieniu centroidu nie rozwiązuje się problemu plecakowego, ale uznaje się, że suma $\frac{n-1}{2}$ jest osiągalna (`polb5.cpp`).

Heurystyka zaimplementowana w pliku `polb6.cpp` zdobywa maksymalnie 30% punktów. Po znalezieniu centroidu wielokrotnie losowo ustawia jego synów w ciąg, a następnie szuka prefiksu o sumie poddrzew najbardziej zbliżonej do $\frac{n-1}{2}$.

Testy

Funkcje generujące testy do zadania dzielą się na dwie grupy: ogólne (generujące drzewa według pewnych parametrów, ale bez wyróżnionych wierzchołków) oraz z ustalonym centroidem (wybierające jeden wierzchołek jako centroid, a następnie generujące poddrzewa jego synów o ustalonych własnościach).

1. Testy ogólne:
   (a) gwiazda,
   (b) ścieżka,
   (c) ścieżka, do której podczepione są mniejsze drzewa,
   (d) losowe drzewo,
   (e) drzewo binarne (zrównoważone i niezrównoważone),
   (f) połączenie ścieżki z jednym z drzew wymienionych powyżej.

2. Testy z ustalonym centroidem:
   (a) Synowie centroidu mają poddrzewa o losowych wielkościach; ustalone są następujące parametry: liczba synów centroidu, maksymalna i minimalna wielkość ich poddrzew.
   (b) Jak wyżej, ale synowie centroidu podzieleni są na dwie grupy, dla każdej osobno ustalamy parametry.
   (c) Wielkości poddrzew rzadko się powtarzają.
   (d) Wielkości poddrzew często się powtarzają.
   (e) Wielkości wszystkich poddrzew są podzielne przez ustaloną liczbę.
   (f) a synów centroidu ma poddrzewa wielkości b, b synów centroidu ma poddrzewa wielkości a oraz $\mathrm{NWD}(a, b) = 1$. Ten test skutecznie wykrywa heurystykę `polb6.cpp`.

Testy są grupowane po 6, jest 10 grup. W każdej grupie znajdują się 2–3 testy ogólne, wśród nich zawsze pojawia się gwiazda i ścieżka. W każdej grupie pojawia się też test typu *2e*.

XXV Międzynarodowa Olimpiada Informatyczna,

Brisbane, Australia 2013

Jakub Łącki, Jakub Radoszewski
Tłumaczenie

IOI 2013, dzień pierwszy, 08.07.2013

Czas snu

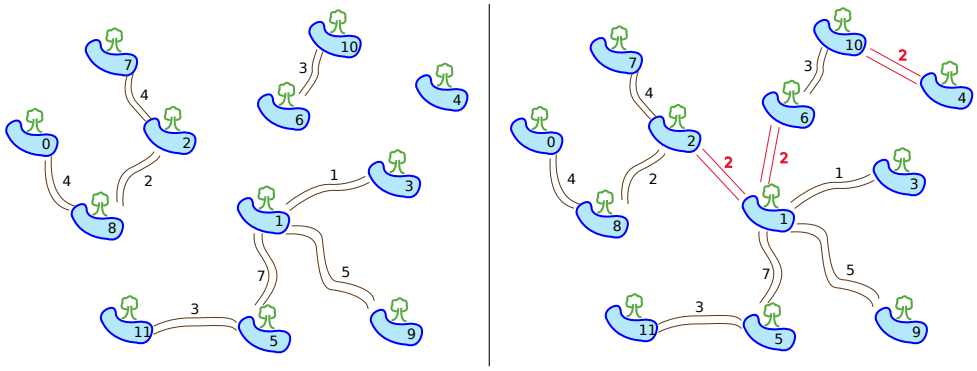
Rzecz dzieje się dawno temu, niedługo po powstaniu świata, w czasach, gdy o IOI jeszcze nikomu się nie śniło.

Pewien Wąż żyje w kraju, w którym znajduje się N kałuż ponumerowanych liczbami od 0 do $N-1$. Kałuże połączone są za pomocą M dwukierunkowych ścieżek, po których może poruszać się Wąż. Każda para kałuż jest połączona (pośrednio lub bezpośrednio) co najwyżej jednym ciągiem ścieżek. Pewne pary kałuż mogą być ze sobą w ogóle nie połączone (zatem $M \leqslant N-1$). Przebycie każdej ścieżki zajmuje Wężowi pewną liczbę dni; liczba ta może być różna dla różnych ścieżek.

Kangur, przyjaciel Węża, chciałby wytyczyć (wydeptać) $N-M-1$ nowych ścieżek tak, by Wąż mógł przemieszczać się między dowolnymi dwiema kałużami. Kangur może połączyć dowolne dwie kałuże ścieżką. Przebycie każdej z tych nowych ścieżek będzie zajmowało Wężowi dokładnie L dni.

Kangur chciałby, by Wąż przemieszczał się możliwie efektywnie, dlatego wytyczy nowe ścieżki tak, by najdłuższy czas podróży między dowolną parą kałuż był jak najkrótszy. Pomóż zwierzętom wyznaczyć maksymalny czas podróży pomiędzy dowolnymi dwiema kałużami, po tym jak Kangur wydepcze nowe ścieżki.

Przykład



Rysunek po lewej przedstawia $N = 12$ kałuż i $M = 8$ ścieżek. Przyjmijmy, że $L = 2$, tj. Wąż potrzebuje dwóch dni na przebycie każdej nowej ścieżki. W tej sytuacji Kangur może wydeptać np. trzy nowe ścieżki: pomiędzy kałużami 1 i 2, pomiędzy kałużami 1 i 6 oraz pomiędzy kałużami 4 i 10.

Rysunek po prawej przedstawia końcowy układ ścieżek. Najdłuższy możliwy czas podróży między dwiema kałużami to 18 dni – tyle trwa podróż między kałużami 0 i 11. Jest to optymalny wynik, gdyż niezależnie od tego, jak Kangur wydepcze ścieżki, będzie istniała taka para kałuż, że najkrótszy czas podróży Węża pomiędzy tymi kałużami to co najmniej 18 dni.

Implementacja

Powinieneś zgłosić plik z implementacją funkcji `travelTime`*:*

C/C++

```
int travelTime(int N, int M, int L, int A[], int B[], int T[]);
```

Pascal

```
function travelTime(N, M, L : LongInt; var A, B, T : array of LongInt)
            : LongInt;
```

Funkcja `travelTime` *powinna obliczać największy czas podróży (w dniach) pomiędzy dowolną parą kałuż, przy założeniu, że Kangur wydeptał $N - M - 1$ ścieżek w taki sposób, że wszystkie kałuże są połączone ścieżkami, a najdłuższy czas podróży pomiędzy dowolną parą kałuż jest jak najmniejszy.*

Argumenty tej funkcji to N – liczba kałuż ($1 \leqslant N \leqslant 100\,000$), M – liczba istniejących ścieżek ($0 \leqslant M \leqslant N - 1$), L – liczba dni, których potrzebuje Wąż na przebycie ścieżki wydeptanej przez Kangura ($1 \leqslant L \leqslant 10\,000$), oraz A, B i T – tablice rozmiaru M, które określają końce oraz czasy podróży dla każdej obecnie istniejącej ścieżki; i-ta ścieżka (dla $0 \leqslant i \leqslant M - 1$) łączy kałuże $A[i]$ oraz $B[i]$, a przebycie jej zajmuje $T[i]$ dni ($0 \leqslant A[i], B[i] \leqslant N - 1$, $1 \leqslant T[i] \leqslant 10\,000$).

Przykład

Wywołanie funkcji

```
travelTime(12, 8, 2,
           [0,8,2,5,5,1,1,10],
           [8,2,7,11,1,3,9,6],
           [4,2,4,3,7,1,5,3])
```

daje wynik 18.

Ograniczenia i podzadania

Maksymalny czas działania: 1 sekunda. Limit pamięci: 64 MiB.

Podzadanie	Punkty	Dodatkowe ograniczenia
1	14	*$M = N - 2$ i z każdej kałuży wychodzą jedna lub dwie ścieżki. Innymi słowy, kałuże podzielone są na dwa spójne zbiory, a w ramach każdego zbioru są one połączone w graf będący ścieżką prostą.*
2	10	*$M = N - 2$ i $N \leqslant 100$*
3	23	*$M = N - 2$*
4	18	*Z każdej kałuży wychodzi co najwyżej jedna ścieżka.*
5	12	*$N \leqslant 3000$*
6	23	*(brak)*

Jakub Łącki, Jakub Radoszewski
Tłumaczenie

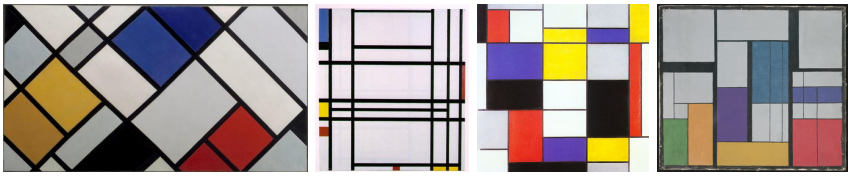
IOI 2013, dzień pierwszy, 08.07.2013

Historia sztuki

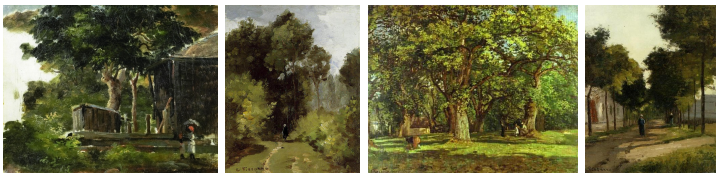
Zbliża się egzamin z historii sztuki. Niestety, zamiast chodzić na wykłady, zajmowałeś się informatyką. Musisz więc napisać program, który zda egzamin za Ciebie.

Na egzaminie otrzymasz pewną liczbę obrazów. Każdy obraz to przykład jednego z czterech nurtów malarstwa, które numerujemy liczbami 1, 2, 3 i 4.

Nurt 1 to neoplastycyzm. Na przykład:

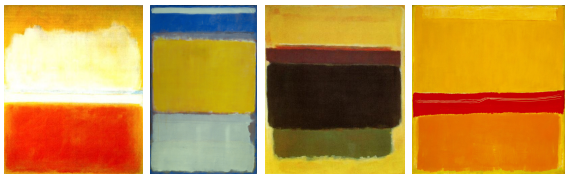


Nurt 2 to krajobrazy impresjonistyczne. Na przykład:



Nurt 3 to action painting (jeden z nurtów ekspresjonizmu). Na przykład:

Nurt 4 to malarstwo barwnych płaszczyzn. Na przykład:



Twoim zadaniem jest napisanie programu, który rozpozna, do którego nurtu należy podany obraz.

Jury IOI zebrało wiele obrazów namalowanych w każdym z powyższych stylów. Dziewięć losowo wybranych obrazów umieszczono w materiałach do zadania dostępnych na Twoim komputerze. Możesz je obejrzeć i użyć ich do testów. Pozostałe obrazy zostaną użyte w trakcie oceny Twojego programu.

Obraz podany jest w postaci macierzy $H \times W$ pikseli. Wiersze obrazu są ponumerowane liczbami od 0 do $H-1$, począwszy od najwyższego, zaś kolumny liczbami od 0 do $W-1$, począwszy od lewej. Piksele są opisane za pomocą dwuwymiarowych macierzy R, G oraz B, które podają poszczególne składowe kolorów (czerwonego, zielonego i niebieskiego) w odpowiednim pikselu obrazu. Wartości te są z zakresu od 0 (minimalne nasycenie koloru) do 255 (maksymalne nasycenie koloru).

Implementacja

Powinieneś zgłosić plik z implementacją funkcji `style`*:*

C/C++

```
int style(int H, int W, int R[500][500], int G[500][500], int B[500][500]);
```

Pascal

```
type artArrayType = array[0..499, 0..499] of LongInt;
function style(H, W : LongInt; var R, G, B : artArrayType) : LongInt;
```

Funkcja `style` *powinna określić nurt, z którego pochodzi obraz (zwracając liczbę 1, 2, 3 lub 4, zgodnie z wcześniejszym opisem).*

Jej argumenty to H, W – liczba wierszy i kolumn pikseli na obrazie ($100 \leqslant H, W \leqslant 500$) oraz R, G i B – dwuwymiarowe macierze rozmiaru $H \times W$, określające ilość koloru (odpowiednio czerwonego, zielonego i niebieskiego) w każdym pikselu obrazu. Piksel znajdujący się na przecięciu i-tego wiersza oraz j-tej kolumny (dla $0 \leqslant i \leqslant H-1$ oraz $0 \leqslant j \leqslant W-1$) jest opisany wartościami $R[i,j]$, $G[i,j]$ oraz $B[i,j]$, przy czym $0 \leqslant R[i,j], G[i,j], B[i,j] \leqslant 255$.

Ograniczenia

Maksymalny czas działania: 5 sekund. Limit pamięci: 64 MiB.

Ocenianie

W tym zadaniu nie ma podzadań. Twój wynik zależy od tego, jak wiele obrazów zostanie poprawnie rozpoznanych przez Twój program. Przyjmijmy, że Twój program poprawnie sklasyfikuje P procent obrazów ($0 \leqslant P \leqslant 100$). Wówczas:

- *Jeśli $P < 25$, otrzymasz 0 punktów.*
- *Jeśli $25 \leqslant P < 50$, otrzymasz między 0 a 10 punktów; liczba punktów zależy liniowo od P, tj. jest obliczana według wzoru $10 \cdot (P-25)/25$ i zaokrąglana w dół do liczby całkowitej.*
- *Jeśli $50 \leqslant P < 90$, otrzymasz między 10 a 100 punktów; liczba punktów zależy liniowo od P, tj. jest obliczana według wzoru $10 + 90 \cdot (P-50)/40$ i zaokrąglana w dół do liczby całkowitej.*
- *Jeśli $90 \leqslant P$, otrzymasz 100 punktów.*

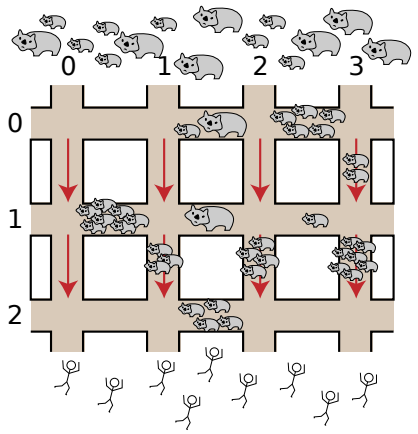
Jakub Łącki, Jakub Radoszewski
Tłumaczenie

IOI 2013, dzień pierwszy, 08.07.2013

Wombaty

W Brisbane grasują wielkie zmutowane wombaty. Pomóż ocalić mieszkańców miasta od tej plagi torbaczy.

Drogi w Brisbane tworzą regularną kratkę. Jest R poziomych dróg biegnących ze wschodu na zachód, ponumerowanych liczbami od 0 do $R-1$ w kolejności z północy na południe, i C pionowych dróg biegnących z północy na południe, ponumerowanych liczbami od 0 do $C-1$ w kolejności z zachodu na wschód (patrz rysunek poniżej).



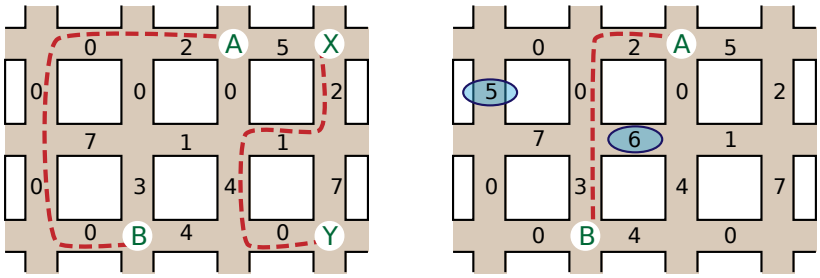
Wombaty przybywają do miasta z północy, dlatego mieszkańcy próbują uciekać na południe. Podczas ucieczki mogą oni biec ulicami poziomymi w dowolnym kierunku, jednak wzdłuż ulic pionowych biegną zawsze w kierunku południowym, bo tam czeka ocalenie.

Przecięcie poziomej drogi o numerze P z drogą pionową o numerze Q będziemy oznaczać przez (P, Q). Na każdym fragmencie drogi położonym między dwoma sąsiednimi skrzyżowaniami znajduje się pewna liczba wombatów. Liczba ta może dodatkowo zmieniać się w czasie. Twoim zadaniem jest pomóc zaprowadzić każdą osobę z określonego skrzyżowania położonego na północy (tj. na poziomej drodze o numerze 0) na określone przez nią skrzyżowanie położone na południu (tj. na poziomej drodze o numerze $R-1$), tak aby na swojej drodze spotkała jak najmniej wombatów.

Znasz rozmiary mapy oraz liczby wombatów znajdujących się początkowo na każdym fragmencie drogi. Ponadto dany jest opis zdarzeń dwóch typów:

- *zdarzenie zmiany, które odpowiada zmianie liczby wombatów na danym fragmencie drogi,*
- *zdarzenie ucieczki, w którym dana osoba znajduje się przy skrzyżowaniu położonym na drodze poziomej o numerze 0, a Twoim zadaniem jest znaleźć drogę do danego skrzyżowania położonego na drodze poziomej o numerze $R-1$, na której osoba ta spotka najmniejszą możliwą liczbę wombatów.*

Przykład



Rysunek po lewej przedstawia mapę złożoną z $R = 3$ dróg poziomych i $C = 4$ dróg pionowych. Na każdym fragmencie drogi znajduje się pewna liczba wombatów. Rozważmy następujący ciąg zdarzeń:

- *Osoba znajduje się przy skrzyżowaniu $A = (0, 2)$ i chce uciec do skrzyżowania $B = (2, 1)$. Najmniejsza liczba wombatów, jakie może napotkać w trakcie ucieczki, to 2. Odpowiednią trasę ucieczki zaznaczono na rysunku linią przerywaną.*
- *Inna osoba znajduje się przy skrzyżowaniu $X = (0, 3)$ i chce uciec do skrzyżowania $Y = (2, 3)$. Najmniejsza liczba wombatów, jakie może napotkać w trakcie ucieczki, to 7. Odpowiednia trasa ucieczki jest oznaczona na rysunku.*
- *Dalej mają miejsce dwa zdarzenia zmiany: liczba wombatów przy górnym fragmencie pionowej drogi o numerze 0 zmienia się na 5, a liczba wombatów na środkowym fragmencie poziomej drogi o numerze 1 zmienia się na 6. Pozycje te zostały zakreślone na rysunku po prawej.*
- *Trzecia osoba znajduje się przy skrzyżowaniu $A = (0, 2)$ i chce uciec do skrzyżowania $B = (2, 1)$. Teraz najmniejsza liczba napotkanych wombatów to 5.*

Implementacja

Powinieneś zgłosić plik z implementacją funkcji `init`, `changeH`, `changeV` *i* `escape`*:*

C/C++

```
void init(int R, int C, int H[5000][200], int V[5000][200]);
void changeH(int P, int Q, int W);
void changeV(int P, int Q, int W);
int escape(int V1, int V2);
```

Pascal

```
type wombatsArrayType = array[0..4999, 0..199] of LongInt;
procedure init(R, C : LongInt; var H, V : wombatsArrayType);
procedure changeH(P, Q, W : LongInt);
procedure changeV(P, Q, W : LongInt);
function escape(V1, V2 : LongInt) : LongInt;
```

Procedura `init` *przekazuje Ci początkowy wygląd mapy. W tej procedurze możesz zainicjować wszystkie swoje zmienne globalne i struktury danych. Zostanie ona wywołana tylko raz, przed wszystkimi wywołaniami funkcji* `changeH`, `changeV` *i* `escape`.

Jej argumenty to R *– liczba dróg poziomych (*$2 \leqslant R \leqslant 5000$*),* C *– liczba dróg pionowych (*$1 \leqslant C \leqslant 200$*),* H *– dwuwymiarowa tablica rozmiaru* $R \times (C-1)$*, gdzie* $H[P,Q]$ *oznacza liczbę wombatów na fragmencie drogi poziomej ograniczonym skrzyżowaniami* (P,Q) *i* $(P,Q+1)$*, oraz* V *– dwuwymiarowa tablica rozmiaru* $(R-1) \times C$*, gdzie* $V[P,Q]$ *oznacza liczbę wombatów na fragmencie drogi pionowej ograniczonym skrzyżowaniami* (P,Q) *i* $(P+1,Q)$*. Na każdym fragmencie drogi będzie co najwyżej 1000 wombatów*

Wywołanie procedury `changeH` *oznacza zmianę liczby wombatów znajdujących na fragmencie drogi poziomej ograniczonym skrzyżowaniami* (P,Q) *i* $(P,Q+1)$*, przy czym* $0 \leqslant P \leqslant R-1$ *oraz* $0 \leqslant Q \leqslant C-2$*. Argument* W *oznacza nową liczbę wombatów na tym fragmencie drogi (*$0 \leqslant W \leqslant 1000$*).*

Wywołanie procedury `changeV` *oznacza zmianę liczby wombatów znajdujących na fragmencie drogi pionowej ograniczonym skrzyżowaniami* (P,Q) *i* $(P+1,Q)$*, przy czym* $0 \leqslant P \leqslant R-2$ *oraz* $0 \leqslant Q \leqslant C-1$*. Argument* W *oznacza nową liczbę wombatów na tym fragmencie drogi (*$0 \leqslant W \leqslant 1000$*).*

Funkcja `escape` *ma wyznaczyć najmniejszą możliwą liczbę wombatów, jakie musi napotkać osoba w trakcie ucieczki ze skrzyżowania* $(0,V_1)$ *do skrzyżowania* $(R-1,V_2)$*, przy czym* $0 \leqslant V_1, V_2 \leqslant C-1$*.*

Przykład

Wywołanie funkcji	**Wynik**
`init(3, 4, [[0,2,5], [7,1,1], [0,4,0]], [[0,0,0,2], [0,3,4,7]])`	
`escape(2,1)`	2
`escape(3,3)`	7
`changeV(0,0,5)`	
`changeH(1,1,6)`	
`escape(2,1)`	5

Ograniczenia i podzadania

Maksymalny czas działania: 20 sekund. Limit pamięci: 256 MiB. Liczba zdarzeń zmiany (tj. wywołań funkcji `changeH` *i* `changeV`*) nie przekracza 500, a liczba wywołań funkcji* `escape` *nie przekracza 200 000.*

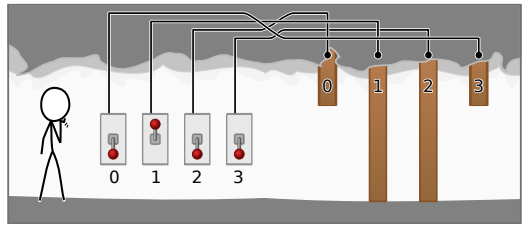
Podzadanie	**Punkty**	**Dodatkowe ograniczenia**
1	9	$C = 1$
2	12	$R, C \leqslant 20$ *i brak wywołań funkcji* `changeH` *i* `changeV`
3	16	$R, C \leqslant 100$ *i co najwyżej 100 wywołań funkcji* `escape`
4	18	$C = 2$
5	21	$C \leqslant 100$
6	24	*(brak)*

Jakub Łącki, Jakub Radoszewski
Tłumaczenie

IOI 2013, dzień drugi, 10.07.2013

Jaskinia

W trakcie drogi z akademika do biblioteki zauważyłeś wejście, które okazało się prowadzić do sekretnej jaskini ukrytej głęboko pod kampusem. Dostępu do jaskini broni korytarz, w którym znajduje się N drzwi, umiejscowionych jedne za drugimi. Zauważyłeś także układ N przełączników. Przełączniki te są połączone z drzwiami i służą do sterowania nimi. Każdy przełącznik jest podłączony do innych drzwi.



Drzwi są ponumerowane kolejno liczbami od 0 do $N-1$, przy czym drzwi numer 0 znajdują się najbliżej Ciebie. Przełączniki są również ponumerowane liczbami od 0 do $N-1$, jednak nie wiesz, którymi drzwiami steruje każdy z przełączników.

Przełączniki znajdują się przy wejściu do jaskini. Każdy przełącznik może być albo w pozycji „w górę", albo w pozycji „w dół". Tylko jedna z tych dwóch pozycji przełącznika powoduje, że podłączone do niego drzwi otwierają się. Jeśli przełącznik jest w drugiej pozycji, odpowiednie drzwi pozostają zamknięte. Pozycje otwierające drzwi mogą być różne dla różnych drzwi i, niestety, nie są Ci znane.

Chciałbyś rozkminić system bezpieczeństwa jaskini. W tym celu możesz dowolnie przestawiać przełączniki, a następnie wyruszyć korytarzem w kierunku jaskini, aby znaleźć pierwsze drzwi, które zostały zamknięte. Drzwi nie są przezroczyste, zatem dotarłszy do tych drzwi, nie widzisz żadnych z dalszych drzwi.

Twój wolny czas pozwala Ci na wypróbowanie co najwyżej 70 000 ustawień przełączników. Twoim zadaniem jest określenie, z którymi drzwiami jest połączony każdy z przełączników, jak również, w jakiej pozycji musi być on ustawiony, by otwierał te drzwi.

Implementacja

Powinieneś zgłosić plik z implementacją procedury `exploreCave`*:*

C/C++

```
int tryCombination(int S[]);
void answer(int S[], int D[]);
void exploreCave(int N);
```

Pascal

```
function tryCombination(var S : array of LongInt) : LongInt;
procedure answer(var S, D : array of LongInt);
procedure exploreCave(N : longint);
```

Funkcja modułu oceniającego `tryCombination` *zostanie zlinkowana z Twoim rozwiązaniem. Pozwala ona dowiedzieć się, dla podanego ustawienia przełączników, jaki jest numer pierwszych drzwi, które są zamknięte. Jeśli wszystkie drzwi są otwarte, funkcja zwraca* -1*. Funkcja działa w czasie* $O(N)$*. Tę funkcję można wywołać co najwyżej 70 000 razy.*

Jej argumentem jest S *– tablica długości* N *określająca pozycję każdego przełącznika. Wartość* $S[i]$ *opisuje pozycję przełącznika* i *(dla* $0 \leqslant i \leqslant N-1$*). Jeśli jest ona równa* 0*, to przełącznik* i *jest w pozycji „w górę”. Z kolei wartość 1 odpowiada pozycji „w dół”.*

Wywołaj procedurę modułu oceniającego `answer`*, gdy odkryjesz ustawienie przełączników, które otwiera wszystkie drzwi, oraz układ połączeń między przełącznikami a drzwiami.*

Jej argumenty to S *– tablica rozmiaru* N *określająca ustawienie przełączników, które otwiera wszystkie drzwi, oraz* D *– tablica rozmiaru* N *określająca, do których drzwi jest podłączony każdy z przełączników. Format tablicy* S *jest taki sam, jak (podany powyżej) format argumentu funkcji* `tryCombination`*. Wartość* $D[i]$ *powinna być równa numerowi drzwi połączonych z przełącznikiem* i *(dla* $0 \leqslant i \leqslant N-1$*).*

Procedura `exploreCave` *powinna używać (dostarczonej) funkcji* `tryCombination`*, aby wyznaczyć ustawienie przełączników otwierające wszystkie drzwi oraz układ połączeń między drzwiami a przełącznikami. Na końcu powinna wywołać funkcję* `answer`*. Jej argumentem jest* N *– liczba przełączników oraz liczba drzwi w jaskini (*$1 \leqslant N \leqslant 5000$*).*

Przykład

Załóżmy, że drzwi i przełączniki są połączone tak, jak na zamieszczonym wcześniej obrazku.

Wywołanie	**Wynik**	**Wyjaśnienie**
`tryCombination([1,0,1,1])`	`1`	*Ten układ odpowiada rysunkowi. Przełączniki 0, 2 i 3 są w pozycji „w dół”, zaś przełącznik 1 w pozycji „w górę”. Funkcja zwraca 1, co oznacza, że drzwi 1 to pierwsze (od lewej) zamknięte drzwi.*
`tryCombination([0,1,1,0])`	`3`	*Drzwi 0, 1 i 2 są otwarte, zaś drzwi 3 są zamknięte.*
`tryCombination([1,1,1,0])`	`-1`	*Przestawienie przełącznika 0 na pozycję „w dół” sprawia, że wszystkie drzwi są otwarte. Funkcja zwraca więc* -1*.*
`answer([1,1,1,0], [3,1,0,2])`	*Program kończy się*	*Orzekamy, że ustawienie* $[1,1,1,0]$ *otwiera wszystkie drzwi, a przełączniki 0, 1, 2 i 3 są połączone, odpowiednio, z drzwiami 3, 1, 0 i 2.*

Ograniczenia i podzadania

Maksymalny czas działania: 2 sekundy. Limit pamięci: 32 MiB.

Podzadanie	Punkty	Dodatkowe ograniczenia
1	12	*Dla każdego i, przełącznik i jest podłączony do drzwi i. Twoim zadaniem jest wyznaczenie ustawienia otwierającego drzwi.*
2	13	*Ustawienie $[0, 0, 0, \ldots, 0]$ powoduje otwarcie wszystkich drzwi. Twoim zadaniem jest wyznaczenie połączeń między przełącznikami a drzwiami.*
3	21	$N \leqslant 100$
4	30	$N \leqslant 2000$
5	24	*(brak)*

Jakub Łącki, Jakub Radoszewski
Tłumaczenie

IOI 2013, dzień drugi, 10.07.2013

Robociki

Braciszek Bajtyny porozrzucał w pokoju mnóstwo zabawek. Bajtyna chciałaby zaprowadzić porządek w pokoju i odstawić wszystkie zabawki z powrotem na półkę. Aby się zbytnio nie namęczyć, zaprzęgnie do tej czynności skręcone naprędce robociki. Bajtyna poprosiła Cię o pomoc w określeniu, które robociki powinny zgarnąć poszczególne zabawki.

Na podłodze leży łącznie T zabawek. Każda z nich charakteryzuje się dwoma parametrami wyrażającymi się liczbami całkowitymi: masą $W[i]$ i rozmiarem $S[i]$. Bajtyna ma dwa typy robocików: „słabe" oraz „małe".

- *Jest A słabych robocików. Każdy z nich ma pewną wytrzymałość $X[i]$, co oznacza, że może podnieść dowolny przedmiot o masie ściśle mniejszej niż $X[i]$. Rozmiar podnoszonego przedmiotu jest nieistotny.*
- *Jest B małych robocików. Każdy z nich ma pewną pojemność $Y[i]$, co oznacza, że może podnieść dowolny przedmiot rozmiaru ściśle mniejszego niż $Y[i]$. Masa podnoszonego przedmiotu jest nieistotna.*

Odstawienie zabawki na półkę zajmuje każdemu robocikowi jedną minutę. Różne robociki mogą odstawiać różne zabawki jednocześnie.

Zadaniem Twojego programu jest stwierdzić, czy robociki Bajtyny mogą odstawić wszystkie zabawki na półkę oraz, jeśli odpowiedź jest twierdząca, wyznaczyć najkrótszy czas, w którym mogą to zrobić.

Przykłady

Przykład 1. *Załóżmy, że Bajtyna ma $A = 3$ słabe robociki o wytrzymałościach $X = [6, 2, 9]$ i $B = 2$ małe robociki o pojemnościach $Y = [4, 7]$, a na podłodze leży $T = 10$ następujących zabawek:*

Numer zabawki	0	1	2	3	4	5	6	7	8	9
Masa	4	8	2	7	1	5	3	8	7	10
Rozmiar	6	5	3	9	8	1	3	7	6	5

W tej sytuacji uporządkowanie wszystkich zabawek zajmie robocikom Bajtyny dokładnie trzy minuty:

	Słaby robocik 0	**Słaby robocik 1**	**Słaby robocik 2**	**Mały robocik 0**	**Mały robocik 1**
Pierwsza minuta	*Zabawka 0*	*Zabawka 4*	*Zabawka 1*	*Zabawka 6*	*Zabawka 2*
Druga minuta	*Zabawka 5*		*Zabawka 3*		*Zabawka 8*
Trzecia minuta			*Zabawka 7*		*Zabawka 9*

Przykład 2. *Załóżmy, że Bajtyna ma $A = 2$ słabe robociki o wytrzymałościach $X = [2, 5]$ i $B = 1$ mały robocik o pojemności $Y = [2]$, a na podłodze leżą $T = 3$ następujące zabawki:*

Numer zabawki	0	1	2
Masa	3	5	2
Rozmiar	1	3	2

Żaden z robocików nie jest w stanie podnieść zabawki o masie 5 i rozmiarze 3, więc w tej sytuacji nie da się uporządkować wszystkich zabawek.

Implementacja

Powinieneś zgłosić plik z implementacją funkcji `putaway`*:*

C/C++

```
int putaway(int A, int B, int T, int X[], int Y[], int W[], int S[]);
```

Pascal

```
function putaway(A, B, T : LongInt; var X, Y, W, S : array of LongInt)
            : LongInt;
```

Funkcja `putaway` *powinna obliczyć, jaki jest najkrótszy czas (w minutach) potrzebny na uporządkowanie wszystkich zabawek. Jeśli uporządkowanie zabawek nie jest możliwe, funkcja powinna zwrócić* -1*.*

Jej argumenty to A *i* B *– liczba słabych i małych robocików (*$0 \leqslant A, B \leqslant 50\,000$ *oraz* $1 \leqslant A + B$*),* T *– liczba zabawek (*$1 \leqslant T \leqslant 1\,000\,000$*),* X *– tablica liczb całkowitych rozmiaru* A *określająca wytrzymałości poszczególnych słabych robocików,* Y *– tablica liczb całkowitych rozmiaru* B *określająca pojemności poszczególnych małych robocików,* W *– tablica liczb całkowitych rozmiaru* T *określająca masy poszczególnych zabawek oraz* S *– tablica liczb całkowitych rozmiaru* T *określająca rozmiary poszczególnych zabawek. Liczby w tablicach spełniają ograniczenie* $1 \leqslant X[i], Y[i], W[i], S[i] \leqslant 2\,000\,000\,000$*.*

Przykłady

Wywołanie funkcji `putaway(3, 2, 10, [6,2,9], [4,7], [4,8,2,7,1,5,3,8,7,10], [6,5,3,9,8,1,3,7,6,5])` *daje wynik 3. Wywołanie funkcji* `putaway(2, 1, 3, [2,5], [2], [3,5,2], [1,3,2])` *daje wynik* -1*.*

Ograniczenia i podzadania

Maksymalny czas działania: 3 sekundy. Limit pamięci: 64 MiB.

Podzadanie	**Punkty**	**Dodatkowe ograniczenia**
1	14	$T = 2$ *i* $A + B = 2$ *(dokładnie dwie zabawki i dwa robociki)*
2	14	$B = 0$ *(wszystkie robociki są słabe)*
3	25	$T \leqslant 50$ *i* $A + B \leqslant 50$
4	37	$T \leqslant 10\,000$ *i* $A + B \leqslant 1000$
5	10	*(brak)*

Jakub Łącki, Jakub Radoszewski
Tłumaczenie

IOI 2013, dzień drugi, 10.07.2013

Gra

Bazza i Shazza grają w grę. Plansza do gry jest prostokątem podzielonym na $R \times C$ kwadratowych pól ułożonych w R wierszy po C kolumn. Wiersze są ponumerowane liczbami od 0 do $R-1$, zaś kolumny liczbami od 0 do $C-1$. Przez (P,Q) oznaczamy pole w wierszu P i kolumnie Q. Każde pole zawiera nieujemną liczbę całkowitą. Na początku gry wszystkie te liczby są równe zeru.

Gra przebiega następująco. W każdym momencie Bazza może albo

- *zaktualizować wartość w komórce (P,Q), albo*
- *poprosić Shazzę o obliczenie największego wspólnego dzielnika (NWD) wszystkich liczb w prostokątnym fragmencie planszy, którego skrajne pola, położone w przeciwległych rogach, to (P,Q) i (U,V).*

Bazza wykona $N_U + N_Q$ czynności (dokona N_U aktualizacji i zada N_Q pytań), zanim znudzi się i pójdzie pograć w krykieta. Twoim zadaniem jest udzielenie poprawnych odpowiedzi na pytania Bazzy.

Przykład

Przyjmijmy $R = 2$ i $C = 3$. Bazza wykonuje najpierw aktualizacje zmieniające wartość pola $(0,0)$ na 20, wartość pola $(0,2)$ na 15 oraz wartość pola $(1,1)$ na 12.

20	0	15
0	12	0

Wynikowa plansza została pokazana na powyższym rysunku. Bazza zadaje następnie pytania o NWD liczb z prostokątnych fragmentów wyznaczonych przez pola:

- *$(0,0)$ i $(0,2)$: w tym prostokącie znajdują się trzy liczby: 20, 0 i 15; ich NWD wynosi 5.*
- *$(0,0)$ i $(1,1)$: w tym prostokącie znajdują się cztery liczby: 20, 0, 0 i 12; ich NWD wynosi 4.*

Przyjmijmy teraz, że Bazza wykonuje aktualizacje zmieniające wartość pola $(0,1)$ na 6 oraz wartość pola $(1,1)$ na 14.

20	6	15
0	14	0

Nowa plansza została pokazana na powyższym rysunku. Bazza zadaje następnie pytania o NWD liczb z prostokątnych fragmentów wyznaczonych przez pola:

- *$(0, 0)$ i $(0, 2)$: trzy liczby w tym prostokącie to teraz 20, 6 i 15, a ich NWD wynosi 1.*
- *$(0, 0)$ i $(1, 1)$: cztery liczby w tym prostokącie to teraz 20, 6, 0 i 14, a ich NWD wynosi 2.*

Bazza wykonał $N_U = 5$ aktualizacji i zadał $N_Q = 4$ pytania.

Implementacja

Powinieneś zgłosić plik z implementacją funkcji `init`, `update` *i* `calculate` *opisanych poniżej.*

Dla ułatwienia, szablony rozwiązań umieszczone na Twoim komputerze (`game.c`, `game.cpp` *i* `game.pas`*) zawierają funkcję* `gcd2(X,Y)`*, która oblicza największy wspólny dzielnik dwóch nieujemnych liczb całkowitych X i Y. Jeśli $X = Y = 0$,* `gcd2(X,Y)` *zwraca 0.*

Funkcja ta działa na tyle szybko, że za jej pomocą można napisać rozwiązanie uzyskujące maksymalną punktację. W szczególności, jej czas działania jest w najgorszym razie proporcjonalny do $\log(X + Y)$.

C/C++

```
void init(int R, int C);
void update(int P, int Q, long long K);
long long calculate(int P, int Q, int U, int V);
```

Pascal

```
procedure init(R, C : LongInt);
procedure update(P, Q : LongInt; K : Int64);
function calculate(P, Q, U, V : LongInt) : Int64;
```

Procedura `init` *otrzymuje wymiary planszy jako argumenty i pozwala Ci na zainicjowanie zmiennych globalnych oraz struktur danych. Zostanie wywołana dokładnie raz, przed wszystkimi wywołaniami funkcji* `update` *i* `calculate`*. Argumenty tej procedury to R i C – liczba wierszy i kolumn planszy ($1 \leqslant R, C \leqslant 10^9$).*

Procedura `update` *zostanie wywołana za każdym razem, gdy Bazza zmieni liczbę na planszy. Aktualizacja polega na zmianie wartości pola (P, Q) na K, przy czym $0 \leqslant P \leqslant R - 1$, $0 \leqslant Q \leqslant C - 1$ oraz $0 \leqslant K \leqslant 10^{18}$. Nowa wartość może być równa poprzedniej wartości na tym polu.*

Funkcja `calculate` *powinna obliczyć największy wspólny dzielnik wszystkich liczb zawartych w prostokącie, którego skrajne pola położone w przeciwległych rogach to (P, Q) oraz (U, V), przy czym $0 \leqslant P \leqslant U \leqslant R - 1$ oraz $0 \leqslant Q \leqslant V \leqslant C - 1$. Jeśli wszystkie liczby w prostokątnym fragmencie są zerami, wynikiem funkcji także powinno być zero.*

Przykład

Wywołanie funkcji	**Wynik**
`init(2, 3)`	
`update(0, 0, 20)`	
`update(0, 2, 15)`	
`update(1, 1, 12)`	
`calculate(0, 0, 0, 2)`	5
`calculate(0, 0, 1, 1)`	4
`update(0, 1, 6)`	
`update(1, 1, 14)`	
`calculate(0, 0, 0, 2)`	1
`calculate(0, 0, 1, 1)`	2

Ograniczenia i podzadania

Maksymalny czas działania: 13 sekund. Limit pamięci: 230 MiB.

Podzadanie	**Punkty**	**Dodatkowe ograniczenia**			
		R	C	N_U	N_Q
1	10	$\leqslant 100$	$\leqslant 100$	$\leqslant 100$	$\leqslant 100$
2	27	$\leqslant 10$	$\leqslant 100\,000$	$\leqslant 10\,000$	$\leqslant 250\,000$
3	26	$\leqslant 2000$	$\leqslant 2000$	$\leqslant 10\,000$	$\leqslant 250\,000$
4	17	$\leqslant 10^9$	$\leqslant 10^9$	$\leqslant 10\,000$	$\leqslant 250\,000$
5	20	$\leqslant 10^9$	$\leqslant 10^9$	$\leqslant 22\,000$	$\leqslant 250\,000$

XIX Bałtycka Olimpiada Informatyczna,

Rostock, Niemcy 2013

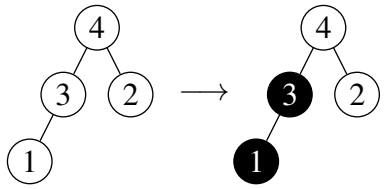
Dawid Dąbrowski, Jakub Radoszewski
Tłumaczenie

BOI 2013, dzień pierwszy, 9.04.2013

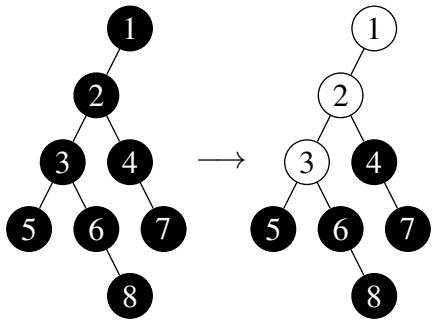
Kulki

Maszyna kulkowa *to ukorzenione drzewo o N węzłach, ponumerowanych liczbami od 1 do N. Każdy węzeł jest albo pusty, albo zawiera dokładnie jedną kulkę. Początkowo wszystkie węzły są puste. Maszyna potrafi wykonywać dwa rodzaje operacji:*

1. *Wrzucenie k kulek. Kulki są wrzucane do maszyny pojedynczo. Staczają się w dół drzewa, począwszy od korzenia, aż do osiągnięcia węzła, który nie ma żadnego pustego syna. Jeśli kulka znajduje się w węźle, który ma przynajmniej jednego pustego syna, to stacza się do tego spośród pustych synów, którego poddrzewo zawiera węzeł o minimalnym numerze. Przykładowo, jeśli do maszyny przedstawionej na poniższym rysunku wrzucimy dwie kulki, to znajdą się one ostatecznie w węzłach o numerach 1 i 3. Dokładniej, pierwsza kulka najpierw stoczy się z węzła 4 do węzła 3, ponieważ węzeł numer 3 jest w tym momencie pusty i zawiera w swoim poddrzewie węzeł 1. W następnym kroku kulka stoczy się z węzła 3 do węzła 1. Druga kulka stoczy się z węzła 4 do węzła 3 i tam już pozostanie.*



2. *Usunięcie kulki z wybranego węzła. W wyniku tej operacji węzeł staje się pusty, a jeśli w węzłach powyżej znajdują się kulki, to staczają się one w dół drzewa. Dokładniej, jeśli w ojcu jakiegoś pustego węzła znajduje się kulka, to kulka ta stoczy się w dół. Przykładowo, jeśli w maszynie z rysunku poniżej usuniemy kulki z węzłów o numerach 5, 7 i 8 (w tej kolejności), to węzły 1, 2 oraz 3 staną się puste.*



Wejście

W pierwszym wierszu wejścia znajdują się dwie liczby całkowite N i Q ($1 \leqslant N, Q \leqslant 100\,000$) oznaczające liczbę węzłów maszyny oraz liczbę operacji do wykonania. Kolejne N wierszy zawiera opis maszyny. W i-tym z tych wierszy znajduje się opis węzła numer i: jest to numer ojca tego węzła albo 0, jeśli węzeł numer i jest korzeniem.

Każdy z kolejnych Q wierszy zawiera dwie liczby całkowite i opisuje operację do wykonania. Operacja pierwszego rodzaju jest oznaczana przez napis „1 k", gdzie k jest liczbą kulek do wrzucenia. Operacja drugiego rodzaju jest oznaczana przez napis „2 x", gdzie x oznacza numer węzła, z którego usuwamy kulkę.

Możesz założyć, że wszystkie operacje są poprawne, tzn. że nigdy nie wrzucamy więcej kulek niż jest pustych węzłów oraz nigdy nie usuwamy kulki z pustego węzła.

Wyjście

Dla każdej operacji pierwszego rodzaju wypisz numer węzła, w którym zatrzymała się ostatnia wrzucona kulka. Dla operacji drugiego rodzaju wypisz liczbę kulek, które stoczyły się w wyniku jej wykonania.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
8 4	1
0	3
1	2
2	2
2	
3	
3	
4	
6	
1 8	
2 5	
2 7	
2 8	

Ocenianie

W testach wartych 25 punktów każdy węzeł ma zero lub dwóch synów. Co więcej, wszystkie węzły nieposiadające synów znajdują się w tej samej odległości od korzenia.

W testach wartych 30 punktów operacje będą wykonywane w taki sposób, że żadna kulka nie stoczy się w wyniku wykonania operacji drugiego rodzaju.

W testach wartych 40 punktów wykonywana jest tylko jedna operacja pierwszego rodzaju. Jest to zarazem pierwsza wykonywana operacja.

Te trzy grupy testów są parami rozłączne.

Dawid Dąbrowski, Jakub Radoszewski
Tłumaczenie

BOI 2013, dzień pierwszy, 9.04.2013

Liczby antypalindromiczne

Słowo nazywamy palindromem, jeśli czytane wspak brzmi tak samo jak czytane normalnie. Liczbę nazywamy **antypalindromiczną**, *jeśli jej zapis w systemie dziesiętnym nie zawiera spójnego fragmentu będącego palindromem o długości większej niż 1. Dla przykładu, liczba 16276 jest antypalindromiczna, natomiast liczba 17276 nie jest antypalindromiczna, ponieważ jej zapis dziesiętny zawiera palindrom 727.*

Twoim zadaniem jest wyznaczyć liczbę liczb antypalindromicznych w danym przedziale liczb.

Wejście

Na wejściu znajdują się dwie liczby całkowite a oraz b ($0 \leqslant a \leqslant b \leqslant 10^{18}$).

Wyjście

Twój program powinien wypisać jedną liczbę całkowitą – liczbę liczb antypalindromicznych w przedziale od a do b (przedział zawiera również liczby a i b).

Przykład

Dla danych wejściowych:
```
123 321
```
poprawnym wynikiem jest:
```
153
```

natomiast dla danych wejściowych:
```
123456789 987654321
```
poprawną odpowiedzią jest:
```
167386971
```

Ocenianie

W testach wartych 25 punktów zachodzi dodatkowy warunek $b - a \leqslant 100\,000$.

Dawid Dąbrowski, Jakub Radoszewski
Tłumaczenie

BOI 2013, dzień pierwszy, 9.04.2013

Rury

W Bajtogrodzie grasuje znany złoczyńca Bitold. Aby uprzykrzyć życie mieszkańcom, Bitold postanowił pomajstrować przy kanalizacji. Woda przeznaczona na użytek miasta jest przechowywana w N zbiornikach połączonych układem M rur. Z każdego zbiornika do każdego innego istnieje ścieżka złożona z jednej lub większej liczby rur. Każda z rur łączy dwa różne zbiorniki, a każda para zbiorników jest połączona co najwyżej jedną rurą.

Bitold podpiął się do niektórych rur i zaczął odciągać z nich wodę. Nie wiedzieć czemu, z każdej z nich odciąga zawsze parzystą liczbę metrów sześciennych wody na sekundę (m^3/s). Jeśli Bitold odciąga $2d$ m^3/s wody z rury łączącej zbiorniki u oraz v, to w efekcie każdy ze zbiorników u, v traci d m^3/s wody.

Celem Bitolda nie jest jednak wzbogacenie się w wodę, tylko mydlenie ludziom oczu. Dlatego do niektórych kontrolowanych przez siebie rur Bitold pompuje wodę, zamiast ją odciągać. Do każdej z tych rur Bitold pompuje parzystą liczbę m^3/s wody. Jeśli Bitold pompuje $2p$ m^3/s wody do rury łączącej zbiorniki u oraz v, to w efekcie każdy ze zbiorników u, v zyskuje p m^3/s wody. Łączna zmiana stanu wody w zbiorniku jest sumą zysków i strat pochodzących ze wszystkich rur dochodzących do tego zbiornika. Tak więc jeśli do zbiornika są podłączone rury, z których jest odpompowywane, odpowiednio, $2d_1$, $2d_2$, ..., $2d_a$ m^3/s wody oraz rury, do których jest pompowane, odpowiednio, $2p_1$, $2p_2$, ..., $2p_b$ m^3/s wody, to zmiana stanu wody w tym zbiorniku jest równa $p_1 + p_2 + \ldots + p_b - d_1 - d_2 - \ldots - d_a$.

Burmistrz Bajtogrodu, Bajtazar, zainstalował w kanalizacji system czujników. Niestety, czujniki znajdują się w zbiornikach, ale nie w rurach. Tak więc Bajtazar może wyznaczyć zmianę stanu wody w każdym zbiorniku, jednak nie wie, ile wody jest odciągane z bądź pompowane do poszczególnych rur.

Twoim zadaniem jest napisanie programu, który pomoże Bajtazarowi rozgryźć plan Bitolda. Znając układ sieci zbiorników oraz zmiany stanu wody w poszczególnych zbiornikach, Twój program powinien stwierdzić, czy na tej podstawie da się jednoznacznie odtworzyć plan Bitolda. Powiemy, że plan można odtworzyć jednoznacznie, jeśli można jednoznacznie ustalić, ile wody jest odciągane z i pompowane do poszczególnych rur. Zauważ, że objętość wody odciąganej lub pompowanej nie musi być taka sama dla wszystkich rur. Jeśli plan można odtworzyć jednoznacznie, Twój program powinien go wypisać.

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby N i M ($1 \leqslant N \leqslant 100\,000$, $1 \leqslant M \leqslant 500\,000$) oznaczające liczbę zbiorników i liczbę rur w sieci kanalizacyjnej.

Kolejne N wierszy opisuje wskazania czujników; i-ty z tych wierszy zawiera jedną liczbę całkowitą c_i ($-10^9 \leqslant c_i \leqslant 10^9$) oznaczającą zmianę stanu wody w zbiorniku numer i.

Kolejne M wierszy opisuje układ rur w sieci; i-ty z tych wierszy zawiera dwie liczby całkowite u_i oraz v_i ($1 \leqslant u_i, v_i \leqslant N$). Oznaczają one, że zbiorniki u_i oraz v_i są połączone rurą.

Możesz założyć, że podany na wejściu opis zawsze odpowiada jakiemuś planowi Bitolda.

Wyjście

Jeśli nie da się jednoznacznie odtworzyć planu Bitolda, Twój program powinien wypisać jeden wiersz zawierający liczbę 0. W przeciwnym razie Twój program powinien wypisać M wierszy; i-ty z nich powinien zawierać jedną liczbę całkowitą x_i. Jeśli Bitold odciąga d_i m^3/s wody z rury łączącej zbiorniki u_i oraz v_i, powinno zachodzić $x_i = -d_i$. Jeśli Bitold pompuje p_i m^3/s wody do rury łączącej zbiorniki u_i oraz v_i, powinno zachodzić $x_i = p_i$. Jeśli zaś Bitold nie zaburza przepływu w rurze łączącej zbiorniki u_i oraz v_i, powinno zachodzić $x_i = 0$.

Możesz założyć, że jeśli jednoznaczne odtworzenie planu Bitolda jest możliwe, wówczas dla poprawnej odpowiedzi zachodzi warunek $-10^9 \leqslant x_i \leqslant 10^9$.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*

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

Dla danych wejściowych:	*poprawnym wynikiem jest:*

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

Ocenianie

W przypadkach testowych wartych 30 punktów, sieć kanalizacyjna jest drzewem.

Dawid Dąbrowski, Jakub Radoszewski
Tłumaczenie

BOI 2013, dzień drugi, 10.04.2013

Urodziny Bajtyny

Pomijając jej zamiłowanie do informatyki teoretycznej, Bajtyna jest zupełnie normalną siedmiolatką. Zbliżają się jej urodziny, na które zamierza zaprosić swoje koleżanki i kolegów. Wymyśliła dla nich następującą grę. Na początku gry wszystkie dzieci biegają po pokoju, dopóki arbiter nie krzyknie pewnej liczby k. Wtedy dzieci starają się uformować grupy składające się z dokładnie k osób. Dopóki jest co najmniej k osób niedobranych jeszcze w grupy, tworzą się kolejne grupy. Na końcu dzieci niemające grupy (będzie ich mniej niż k) wypadają z gry. Potem następują kolejne rundy, z kolejnymi liczbami wybieranymi przez arbitra. Gra kończy się, jeśli wszystkie osoby wypadną z gry.

Bajtyna poprosiła, by w rolę arbitra gry wcielił się jej tata, Biton. Niestety Bitonowi ta gra w ogóle nie przypadła do gustu, a kiedy po raz pierwszy próbowali w nią zagrać, bez namysłu jako pierwszą liczbę wybrał ∞. Bajtyna uważa, że taka sytuacja podczas urodzin byłaby niedopuszczalna, więc dała mu listę m liczb pierwszych, z których może wybierać podczas każdej rundy. Zauważ, że Biton może użyć tej samej liczby pierwszej wielokrotnie.

Biton chciałby zakończyć grę tak szybko, jak to możliwe, ponieważ chce pójść dzisiaj na mecz swojego ulubionego klubu piłkarskiego Legia Bitawa. Niestety, Bajtyna nie podała mu dokładnej liczby gości na przyjęciu. Biton chce więc wiedzieć, dla pewnych Q liczb gości na przyjęciu, ile co najmniej liczb z listy musi krzyknąć, by zakończyć zabawę.

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby całkowite m i Q ($1 \leqslant m, Q \leqslant 100\,000$) oznaczające rozmiar puli liczb, których może użyć Biton, oraz liczbę sytuacji, które rozważa.

W drugim wierszu znajduje się m różnych liczb pierwszych $p_1, \ldots, p_m$ w kolejności rosnącej ($2 \leqslant p_i \leqslant 10\,000\,000$). Są to liczby, których może użyć Biton.

Kolejne Q wierszy zawiera sytuacje rozważane przez Bitona. W i-tym z nich znajduje się liczba całkowita n_i ($1 \leqslant n_i \leqslant 10\,000\,000$) oznaczającą liczbę gości na przyjęciu w i-tej sytuacji.

Wyjście

Na wyjściu powinno znaleźć się Q wierszy; i-ty z nich powinien zawierać odpowiedź dla i-tej sytuacji z wejścia. Jeśli Biton może zakończyć grę, powinna znaleźć się tam minimalna liczba rund potrzebna do zakończenia zabawy, a w przeciwnym przypadku należy wpisać oo *(tj. dwie małe litery* o*, oznaczające ∞).*

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
2 2 2 3 5 6	3 oo

Ocenianie

W testach wartych 20 punktów zachodzi warunek $m, n_i, Q \leqslant 10\,000$.
W testach wartych dodatkowe 20 punktów zachodzi warunek $Q = 1$.

Dawid Dąbrowski, Jakub Radoszewski
Tłumaczenie

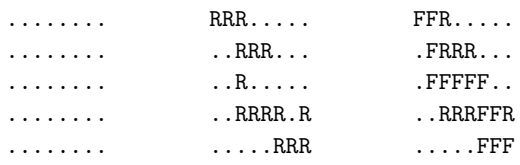
BOI 2013, dzień drugi, 10.04.2013

Ślady

Wczorajszej nocy padało, wskutek czego prostokątna polana w pobliskim lesie pokryta została warstwą świeżego śniegu (co artysta uwiecznił w lewej części poniższego rysunku).

Zające oraz lisy, które żyją w lesie, przechodzą przez polanę i zostawiają na śniegu ślady. Zawsze wchodzą na polanę w jej lewym górnym rogu oraz opuszczają ją w prawym dolnym rogu. Poruszając się w dowolny sposób, bawią się na śniegu, a nawet chodzą po własnych śladach. W każdej chwili na polanie znajduje się co najwyżej jedno zwierzę. Żadne zwierzę nie odwiedza polany więcej niż raz. Aby opisać ruchy zwierząt, podzielimy polanę na kwadraty jednostkowe. Zwierzęta przechodzą między kwadratami jednostkowymi, które mają wspólny bok (nigdy nie przechodzą między kwadratami jednostkowymi po przekątnej i nigdy nie przeskakują kwadratu jednostkowego). Gdy zwierzę wchodzi do kwadratu jednostkowego, zostawia tam swój ślad, zacierając wszystkie inne pozostawione w tym miejscu ślady.

W poniższym przykładzie na początku przez polanę przeszedł zając, zaczynając w lewym górnym, a kończąc w prawym dolnym rogu. Jego ślady są oznaczone na środkowej części rysunku literą `R`*. Następnie na polanie bawił się lis, a jego ślady (oznaczone literą* `F`*) częściowo przykryły ślady zająca (patrz prawa część rysunku).*



Po pewnym czasie nasz artysta znowu uwiecznił sytuację na polanie, nanosząc na mapę widoczne ślady i zaznaczając, czy należą one do zająca, czy do lisa. Chciałbyś oszacować populację zwierzyny w okolicy polany. Napisz program, który wyznaczy najmniejszą możliwą liczbę N zwierząt, które musiały przejść przez polanę, by zostawić na niej dany układ śladów.

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby całkowite H i W ($1 \leqslant H, W \leqslant 4\,000$) oznaczające wysokość oraz szerokość mapy polany. Dalej następuje opis mapy składający się z H wierszy po W znaków każdy. Znak `.` *(kropka) oznacza czysty śnieg, litera* `R` *oznacza miejsce, gdzie ślad zostawił królik, a litera* `F` *oznacza miejsce, gdzie ślad zostawił lis. Na polanie znajduje się co najmniej jeden ślad.*

Wyjście

Wyjście powinno składać się z jednej liczby całkowitej – minimalnej liczby N zwierząt, które mogły zostawić ślady na polanie, której mapa dana jest na wejściu.

Przykład

Dla danych wejściowych:

```
5 8
FFR.....
.FRRR...
.FFFFF..
..RRRFFR
.....FFF
```

poprawnym wynikiem jest:

```
2
```

Ocenianie

W testach wartych 30 punktów zachodzą dodatkowe warunki $N \leqslant 200$ *oraz* $H, W \leqslant 500$.

Dawid Dąbrowski, Jakub Radoszewski
Tłumaczenie

BOI 2013, dzień drugi, 10.04.2013

Vim

Ernest Vincent Wright to amerykański pisarz, znany z tego, że jego powieść „Gadsby" nie zawiera ani jednego wystąpienia litery `e`*. Wacek jest wielkim fanem Ernesta Wrighta i też chciałby napisać równie fajną powieść. Postanowił jednak podjąć się ambitniejszego wyzwania i napisać ją, używając jedynie pierwszych dziesięciu liter alfabetu angielskiego (tj. liter* `abcdefghij`*). Niestety, jak na ironię, w połowie pisania popsuł mu się klawisz z literą* `e`*. Aby zatem zachować spójność tekstu i zarazem nawiązać do dzieła Wrighta, postanowił on także i ze swojej powieści usunąć wszystkie wystąpienia litery* `e`*. Znajomy Wacka, programista, polecił mu użycie do tego celu edytora tekstu Vim. Niestety Wacek nie jest zbytnio obeznany z Vimem i zna tylko trzy komendy tego edytora:* `x`*,* `h` *oraz* `f`*.*

- *Komenda* `x` *usuwa znak wskazywany przez kursor. Pozycja kursora (patrząc od lewej strony) nie zmienia się. Wacek obiecuje nie używać tej komendy, kiedy kursor będzie znajdował się na końcu dokumentu.*
- *Komenda* `h` *przesuwa kursor o jedną pozycję w tył (tj. w lewo). Jeśli kursor znajduje się na początku dokumentu, komenda ta nie ma żadnego efektu.*
- *Komenda* `f` *czeka na to, aż użytkownik wprowadzi kolejny znak C, a następnie przenosi kursor do następnego wystąpienia znaku C w dokumencie (nawet jeśli znak, na którym obecnie znajduje się kursor, jest akurat równy C). Jeśli znak C nie występuje na prawo od kursora, komenda ta nie ma żadnego efektu.*

Przykładowo, jeśli bieżący tekst dokumentu miałby postać:

`jeffiehadabigidea`

przy czym znak pod kursorem jest oznaczony ramką `i`*, to*

- *komenda* `x` *prowadziłaby do sytuacji* `jeffehadabigidea`
- *komenda* `h` *prowadziłaby do sytuacji* `jeffiehadabigidea`
- *komenda* `fi` *prowadziłaby do sytuacji* `jeffiehadabigidea`

Napisz program, który wyznaczy minimalną liczbę naciśnięć klawiszy, jakie Wacek musi wykonać, aby usunąć z dokumentu wszystkie wystąpienia litery `e` *(innych liter nie wolno usuwać). Początkowo kursor znajduje się na pierwszej pozycji tekstu. Zauważ, że klawisz z literą* `e` *jest popsuty, tak więc Wacek nie może użyć komendy* `fe`*.*

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę całkowitą N ($1 \leqslant N \leqslant 70\,000$) oznaczającą długość dokumentu. Drugi wiersz zawiera tekst dokumentu, będący N-literowym napisem składającym się z małych liter alfabetu angielskiego z zakresu od `a` *do* `j`*. Możesz założyć, że pierwsza i ostatnia litera tekstu jest różna od* `e`*.*

Wyjście

Jedyny wiersz wyjścia powinien zawierać jedną liczbę całkowitą – minimalną liczbę naciśnięć klawiszy, jakie Wacek musi wykonać, aby usunąć z dokumentu wszystkie wystąpienia litery `e`.

Przykład

Dla danych wejściowych:
```
35
chefeddiefedjeffeachbigagedegghehad
```
poprawnym wynikiem jest:
```
36
```

Optymalne rozwiązanie dla testu przykładowego to:

```
fdhxhhxffhxfahxhhhxhhhxfdhxfghxfahhx
```

Aby zobaczyć, jak to działa w praktyce, włącz edytor Vim! Wpisz polecenie `vim file.txt`*, aby otworzyć plik o nazwie* `file.txt`*; aby wyjść z edytora, wpisz w nim komendę* `:q`⟨`ENTER`⟩.

Ocenianie

W testach wartych 50 punktów zachodzi warunek $N \leqslant 500$.
W testach wartych dodatkowe 10 punktów zachodzi warunek $N \leqslant 5\,000$.

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] *XV Olimpiada Informatyczna 2007/2008*. Warszawa, 2008.

[16] *XVI Olimpiada Informatyczna 2008/2009*. Warszawa, 2009.

[17] *XVII Olimpiada Informatyczna 2009/2010*. Warszawa, 2010.

[18] *XVIII Olimpiada Informatyczna 2010/2011*. Warszawa, 2011.

[19] *XIX Olimpiada Informatyczna 2011/2012*. Warszawa, 2012.

[20] A. V. Aho, J. E. Hopcroft, J. D. Ullman. *Projektowanie i analiza algorytmów komputerowych*. PWN, Warszawa, 1983.

[21] L. Banachowski, A. Kreczmar. *Elementy analizy algorytmów*. WNT, Warszawa, 1982.

[22] L. Banachowski, K. Diks, W. Rytter. *Algorytmy i struktury danych.* WNT, Warszawa, 1996.

[23] J. Bentley. *Perełki oprogramowania.* WNT, Warszawa, 1992.

[24] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Wprowadzenie do algorytmów.* WNT, Warszawa, 2004.

[25] M. de Berg, M. van Kreveld, M. Overmars. *Geometria obliczeniowa. Algorytmy i zastosowania.* WNT, Warszawa, 2007.

[26] R. L. Graham, D. E. Knuth, O. Patashnik. *Matematyka konkretna.* PWN, Warszawa, 1996.

[27] D. Harel. *Algorytmika. Rzecz o istocie informatyki.* WNT, Warszawa, 1992.

[28] D. E. Knuth. *Sztuka programowania.* WNT, Warszawa, 2002.

[29] W. Lipski. *Kombinatoryka dla programistów.* WNT, Warszawa, 2004.

[30] E. M. Reingold, J. Nievergelt, N. Deo. *Algorytmy kombinatoryczne.* WNT, Warszawa, 1985.

[31] K. A. Ross, C. R. B. Wright. *Matematyka dyskretna.* PWN, Warszawa, 1996.

[32] R. Sedgewick. *Algorytmy w C++. Grafy.* RM, 2003.

[33] S. S. Skiena, M. A. Revilla. *Wyzwania programistyczne.* WSiP, Warszawa, 2004.

[34] P. Stańczyk. *Algorytmika praktyczna. Nie tylko dla mistrzów.* PWN, Warszawa, 2009.

[35] M. M. Sysło. *Algorytmy.* WSiP, Warszawa, 1998.

[36] M. M. Sysło. *Piramidy, szyszki i inne konstrukcje algorytmiczne.* WSiP, Warszawa, 1998.

[37] R. J. Wilson. *Wprowadzenie do teorii grafów.* PWN, Warszawa, 1998.

[38] N. Wirth. *Algorytmy + struktury danych = programy.* WNT, Warszawa, 1999.

[39] *W poszukiwaniu wyzwań. Wybór zadań z konkursów programistycznych Uniwersytetu Warszawskiego.* Warszawa, 2012.

[40] Dorit S. Hochbaum, David B. Shmoys. A unified approach to approximation algorithms for bottleneck problems. *J. ACM*, 33(3):533–550, 1986.

[41] Uriel Feige. A threshold of ln n for approximating set cover. *J. ACM*, 45(4):634–652, 1998.

[42] Samir Khuller, Yoram J. Sussmann. The capacitated k-center problem. *SIAM J. Discrete Math.*, 13(3):403–418, 2000.

[43] Marek Cygan, MohammadTaghi Hajiaghayi, Samir Khuller. LP rounding for k-centers with non-uniform hard capacities. In *53rd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2012*, pages 273–282, 2012.

[44] Hyung-Chan An, Aditya Bhaskara, Ola Svensson. Centrality of trees for capacitated k-center. *CoRR*, abs/1304.2983, 2013.

Niniejsza publikacja stanowi wyczerpujące źródło informacji o zawodach XX Olimpiady Informatycznej, przeprowadzonych w roku szkolnym 2012/2013. 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 XXV Międzynarodowej Olimpiady Informatycznej oraz XIX Bałtyckiej Olimpiady Informatycznej.

XX 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

ARRETO
POLAND

ISBN 978-83-64292-00-2