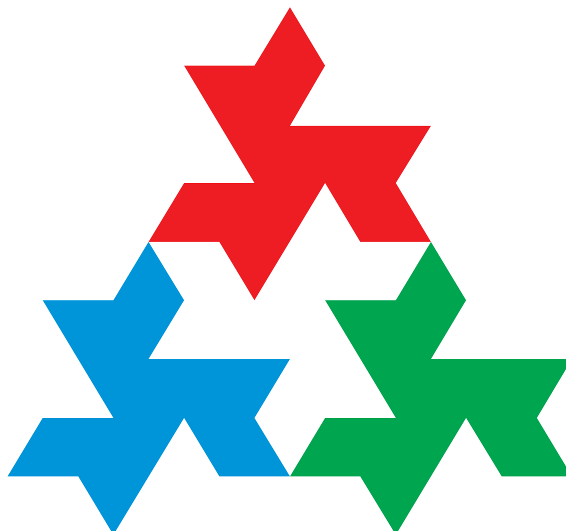
MINISTERSTWO EDUKACJI NARODOWEJ
FUNDACJA ROZWOJU INFORMATYKI
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ



XXI OLIMPIADA INFORMATYCZNA
2013/2014

WARSZAWA, 2015

Autorzy tekstów:

Krzysztof Diks
Jan Kanty Milczek
Wojciech Nadara
Jakub Pachocki
Paweł Parys
Marcin Pilipczuk
Adam Polak
Jakub Radoszewski
Wojciech Rytter
Marek Sommer
Bartosz Szreder
Bartosz Tarnawski
Jacek Tomasiewicz
Michał Włodarczyk

Autorzy programów:

Michał Adamczyk
Marcin Andrychowicz
Bartłomiej Dudek
Lech Duraj
Karol Farbiś
Adam Karczmarz
Krzysztof Kiljan
Bartosz Kostka
Aleksander Łukasiewicz
Jan Kanty Milczek
Paweł Parys
Karol Pokorski
Adam Polak
Jakub Radoszewski
Piotr Smulewicz
Szymon Stankiewicz
Tomasz Syposz

Opracowanie i redakcja:

Bartłomiej Gajewski
Tomasz Idziaszek
Jakub Radoszewski

Skład:

Bartłomiej Gajewski
Jakub Radoszewski

Tłumaczenie treści zadań:

Dawid Dąbrowski
Krzysztof Diks
Jakub Łącki
Jakub Pawlewicz
Jakub Radoszewski
Anna Zych

Pozycja dotowana przez Ministerstwo Edukacji Narodowej.

Sponsorzy Olimpiady:

© Copyright by Komitet Główny Olimpiady Informatycznej
Ośrodek Edukacji Informatycznej i Zastosowań Komputerów
ul. Nowogrodzka 73, 02-006 Warszawa

ISBN 978-83-64292-01-9

Spis treści

Sprawozdanie z przebiegu XXI Olimpiady Informatycznej 7

Regulamin Ogólnopolskiej Olimpiady Informatycznej 39

Zasady organizacji zawodów 51

Zasady organizacji zawodów II i III stopnia 59

Zawody I stopnia – opracowania zadań **63**

Bar sałatkowy 65

Hotele 73

Klocki 81

Kurierzy 85

Wąż 91

Zawody II stopnia – opracowania zadań **101**

Karty 103

Przestępcy 109

Superkomputer 115

Ptaszek 125

Rajd 129

Zawody III stopnia – opracowania zadań **137**

FarmerCraft 139

Dookoła świata 145

Mrowisko 149

Turystyka 155

Lampy słoneczne 165

Panele słoneczne 173

Załadunek 177

XXVI Międzynarodowa Olimpiada Informatyczna – treści zadań 185

Gra 187

Kolej 190

Ściana 193

Gondole 196

Przyjaciel 200

Wakacje 203

XX Bałtycka Olimpiada Informatyczna – treści zadań 207

Ciąg 209

Policjant i złodziej 211

Troje przyjaciół 215

Demarkacja 217

Portale 219

Sędziwi listonosze 221

XX Olimpiada Informatyczna Krajów Europy Środkowej — treści zadań 225

Paszor 227

Skarb 231

Tramwaj 234

Adriatyk 237

Plansza 240

Podlewanie 243

XXI Olimpiada Informatyczna Krajów Europy Środkowej — treści zadań 247

Karnawał 249

Las Fangorn 251

Pytanie 254

007 257

Ciasto 259
Mur 261
Literatura **265**

Wstęp

Po raz 21. oddajemy do rąk Czytelników „niebieską książeczkę” Olimpiady Informatycznej. Oprócz ciekawych danych statystycznych dotyczących przebiegu XXI Olimpiady Informatycznej, znajdziemy w niej wzorcowe rozwiązania wszystkich zadań, z którymi zmagali się uczestnicy Olimpiady. Jest to nieocenione źródło nie tylko w przygotowaniach do startów w kolejnych edycjach zawodów olimpijskich, lecz także znakomity materiał do studiowania „prawdziwej” informatyki, której sercem jest algorytmika. Rozwój technologii (pojemność dysków, sieci komputerowych, światowego Internetu) spowodował, że do przetwarzania i analizy olbrzymiej ilości informacji potrzebne są coraz wydajniejsze algorytmy. Bez postępów w algorytmice nie byłoby osiągnięć w medycynie, robotyce, sztucznej inteligencji, handlu elektronicznym, rozrywce elektronicznej, itp. Jeśli ktoś jest zainteresowany wpływaniem na rozwój świata w XXI wieku, to Olimpiada Informatyczna jest właściwym pierwszym krokiem w tym kierunku.

Organizacja Olimpiady jest złożonym przedsięwzięciem logistycznym, technologicznym i naukowym. Jej sukces, mierzony osiągnięciami laureatów Olimpiady w konkursach międzynarodowych, nie byłby możliwy bez zgodnej współpracy najlepszych uczelni informatycznych w Polsce (Uniwersytetu Jagiellońskiego, Uniwersytetu im. Mikołaja Kopernika w Toruniu, Uniwersytetu Warszawskiego, Uniwersytetu Wrocławskiego, Politechniki Białostockiej, Politechniki Gdańskiej, Politechniki Poznańskiej, Politechniki Śląskiej oraz Wyższej Szkoły Informatyki i Zarządzania w Rzeszowie), Ośrodka Edukacji i Zastosowań Komputerów w Warszawie oraz znakomitych firm komputerowych wspierających Olimpiadę (ASSECO Poland S.A., Jane Street i Codility), jak też Wydawnictwa Naukowego PWN. Oczywiście za instytucjami stoją ludzie. Gorące podziękowania za pracę przy XXI Olimpiadzie kieruję do członków Komitetu Głównego Olimpiady Informatycznej, członków komitetów okręgowych, jurorów, członków zespołu technicznego oraz administracji Olimpiady. Jestem przekonany, że wypracowane wzorce organizacyjne przyczynią się do sukcesów kolejnych edycji Olimpiady Informatycznej.

Krzysztof Diks
Przewodniczący Komitetu Głównego Olimpiady Informatycznej

Sprawozdanie z przebiegu XXI Olimpiady Informatycznej w roku szkolnym 2013/2014

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 XXI 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.

26 września 2013 roku rozesłano do 3477 szkół i zespołów szkół młodzieżowych ponadgimnazjalnych e-maile informujące o rozpoczęciu XXI Olimpiady. E-maile wysłano także do wszystkich zawodników startujących w zawodach XX Olimpiady Informatycznej, którzy w tym roku mogą startować w Olimpiadzie.

Zawody I stopnia rozpoczęły się 7 października 2013 roku. Ostatecznym terminem nadsyłania prac konkursowych był 4 listopada 2013 roku.

Zawody II i III stopnia były dwudniowymi sesjami stacjonarnymi, poprzedzonymi jednodniowymi sesjami próbnymi. Zawody II stopnia odbyły się w sześciu okręgach: Białymstoku, Gdańsku, Krakowie, Toruniu, Warszawie i Wrocławiu w dniach 11–13 lutego 2014 roku, natomiast zawody III stopnia odbyły się w Warszawie na Wydziale Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego, w dniach 1–3 kwietnia 2014 roku.

Uroczystość zakończenia XXI Olimpiady Informatycznej odbyła się 4 kwietnia 2014 roku w Auli Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego w Warszawie przy ul. Banacha 2.

SKŁAD OSOBOWY KOMITETÓW OLIMPIADY INFORMATYCZNEJ

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:
mgr Tomasz Idziaszek (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)
dr Marcin Kubica (Uniwersytet Warszawski)
dr 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 inż. Szymon Wąsik (Politechnika Poznańska)

sekretarz Komitetu Głównego:
mgr Monika Kozłowska-Zając (Ośrodek Edukacji Informatycznej i Zastosowań Komputerów 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 Białymstoku
przewodniczący:
dr hab. inż. Marek Krętowski, prof. PB (Politechnika Białostocka)

zastępca przewodniczącego:
dr inż. Krzysztof Jurczuk (Politechnika Białostocka)
sekretarz:
Jacek Tomasiewicz (student Uniwersytetu Warszawskiego)
członkowie:
Joanna Bujnowska (studentka Uniwersytetu Warszawskiego)
Adam Iwaniuk (student Uniwersytetu Warszawskiego)
Adrian Jaskółka (student Uniwersytetu Warszawskiego)
mgr inż. Konrad Kozłowski (Politechnika Białostocka)

Siedziba Komitetu Okręgowego: Wydział Informatyki, Politechnika Białostocka, 15-351 Białystok, ul. Wiejska 45a.

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)

Siedziba Komitetu Okręgowego: Politechnika Śląska w Gliwicach, 44-100 Gliwice, ul. Akademicka 16.

Komitet Okręgowy w Krakowie
przewodniczący:
prof. dr hab. Paweł Idziak (Katedra Algorytmiki, Uniwersytet Jagielloński)
zastępca przewodniczącego:
dr Maciej Ślusarek (Katedra Algorytmiki, Uniwersytet Jagielloński)
sekretarz:
mgr Monika Gillert (Katedra Algorytmiki, Uniwersytet Jagielloński)
członkowie:
dr Iwona Cieślik (Katedra Algorytmiki, Uniwersytet Jagielloński)
dr Grzegorz Gutowski (Katedra Algorytmiki, Uniwersytet Jagielloński)
mgr Robert Obryk (Katedra Algorytmiki, Uniwersytet Jagielloński)
mgr Andrzej Pezarski (Katedra Algorytmiki, Uniwersytet Jagielloński)

Siedziba Komitetu Okręgowego: Katedra Algorytmiki Uniwersytetu Jagiellońskiego, 30-387 Kraków, ul. Łojasiewicza 6. Strona internetowa Komitetu Okręgowego: `http://www.tcs.uj.edu.pl/OI/`.

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)

Siedziba Komitetu Okręgowego: Politechnika Gdańska, Wydział Elektroniki, Telekomunikacji i Informatyki, 80-952 Gdańsk Wrzeszcz, ul. Gabriela Narutowicza 11/12.

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

Siedziba Komitetu Okręgowego: Instytut Informatyki Politechniki Poznańskiej, 60-965 Poznań, ul. Piotrowo 2.

Komitet Okręgowy w Toruniu
przewodniczący:
prof. dr hab. Grzegorz Jarzembski (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)

Siedziba Komitetu Okręgowego: Wydział Matematyki i Informatyki Uniwersytetu Mikołaja Kopernika w Toruniu, 87-100 Toruń, ul. Chopina 12/18.

Komitet Okręgowy w Warszawie
przewodniczący:
dr Jakub Pawlewicz (Uniwersytet Warszawski)
zastępca przewodniczącego:
mgr Tomasz Idziaszek (Uniwersytet Warszawski)
sekretarz:
mgr Monika Kozłowska-Zając (Ośrodek Edukacji Informatycznej i Zastosowań Komputerów w Warszawie)

członkowie:
mgr Szymon Acedański (Uniwersytet Warszawski)
prof. dr hab. Krzysztof Diks (Uniwersytet Warszawski)
dr Jakub Radoszewski (Uniwersytet Warszawski)

Siedziba Komitetu Okręgowego: Ośrodek Edukacji Informatycznej i Zastosowań Komputerów w Warszawie, 02-006 Warszawa, ul. Nowogrodzka 73.

Komitet Okręgowy we Wrocławiu
przewodniczący:
prof. dr hab. Krzysztof Loryś (Uniwersytet Wrocławski)
zastępca przewodniczącego:
dr Przemysława Kanarek (Uniwersytet Wrocławski)
sekretarz:
inż. Maria Woźniak (Uniwersytet Wrocławski)
członkowie:
dr Paweł Gawrychowski (Uniwersytet Wrocławski)
dr hab. Tomasz Jurdziński (Uniwersytet Wrocławski)
dr Rafał Nowak (Uniwersytet Wrocławski)

Siedziba Komitetu Okręgowego: Instytut Informatyki Uniwersytetu Wrocławskiego, 50-383 Wrocław, ul. Joliot-Curie 15.

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
Maciej Borsz
Maciej Dębski
Bartłomiej Dudek
dr Lech Duraj
Wojciech Dyżewski
Karol Farbiś
Adam Karczmarz
Krzysztof Kiljan
Bartosz Kostka
Aleksander Łukasiewicz
Maciej Matraszek
Jan Kanty Milczek
Wojciech Nadara
dr Paweł Parys
Karol Pokorski
Adam Polak
Piotr Smulewicz
Marek Sommer
Szymon Stankiewicz
Tomasz Syposz
Bartosz Tarnawski
Zbigniew Wojna

ZAWODY I STOPNIA

Zawody I stopnia XXI Olimpiady Informatycznej odbyły się w dniach 7 października – 4 listopada 2013 roku. Wzięło w nich udział 905 zawodników. Decyzją Komitetu Głównego zdyskwalifikowano 28 zawodników. Powodem dyskwalifikacji była niesamodzielność rozwiązań zadań konkursowych. Sklasyfikowano 877 zawodników.

Decyzją Komitetu Głównego do zawodów zostało dopuszczonych 51 uczniów gimnazjów. Pochodzili oni z następujących szkół:

nazwa gimnazjum	miejscowość	liczba uczniów
Gimnazjum nr 24	Gdynia	15
Gimnazjum nr 16 w Zespole Szkół Ogólnokształcących nr 7	Szczecin	5
Gimnazjum nr 58 z Oddziałami Dwujęzycznymi im. Króla Władysława IV	Warszawa	3
XIII Gimnazjum im. Stanisława Staszica w Zespole Szkół nr 82	Warszawa	3
Gimnazjum nr 42 z Oddziałami Dwujęzycznymi	Warszawa	3
Gimnazjum nr 50 w Zespole Szkół Ogólnokształcących nr 6	Bydgoszcz	2
Gimnazjum nr 24 w Zespole Szkół nr 7	Lublin	2
Publiczne Gimnazjum nr 23 w Zespole Szkół Ogólnokształcących nr 6 im. Jana Kochanowskiego	Radom	2
Gimnazjum Dwujęzyczne im. Świętej Kingi w Zespole Szkół Ogólnokształcących nr 2	Tarnów	2
Gimnazjum nr 13 im. Unii Europejskiej	Wrocław	2
Społeczne Gimnazjum nr 2 STO	Białystok	1
Gimnazjum w Zespole Szkół	Czarna	1
Gimnazjum im. św. Stanisława Kostki	Lublin	1
Szkoła Podstawowa i Gimnazjum EUREKA	Poznań	1
Publiczne Gimnazjum nr 8 z Oddziałami Integracyjnymi i Dwujęzycznymi im. Królowej Jadwigi	Radom	1
Gimnazjum nr 32 z Oddziałami Dwujęzycznymi w Zespole Szkół Ogólnokształcących nr 2	Szczecin	1
Gimnazjum z Oddziałami Sportowymi w Centrum Kształcenia Sportowego	Szczecin	1
Społeczne Gimnazjum nr 14 Społecznego Towarzystwa Oświatowego	Warszawa	1
Gimnazjum Dwujęzyczne nr 53 im. Stefanii Sempołowskiej w Zespole Szkół nr 53	Warszawa	1
Gimnazjum nr 121 im. Wojciecha Zawadzkiego	Warszawa	1
Gimnazjum nr 49 z Oddziałami Dwujęzycznymi w Zespole Szkół nr 14	Wrocław	1
Gimnazjum nr 23 im. Wandy Rutkiewicz	Wrocław	1

Kolejność województw pod względem liczby zawodników była następująca:

mazowieckie	183 zawodników	podkarpackie	35
małopolskie	110	zachodniopomorskie	31
pomorskie	98	lubelskie	27
dolnośląskie	78	warmińsko-mazurskie	19
kujawsko-pomorskie	68	łódzkie	16
śląskie	63	świętokrzyskie	15
podlaskie	62	opolskie	8
wielkopolskie	57	lubuskie	7

W zawodach I stopnia najliczniej reprezentowane były szkoły:

nazwa szkoły	miejscowość	liczba uczniów
Zespół Szkół nr 82 (XIV Liceum Ogólnokształcące im. Stanisława Staszica oraz Gimnazjum nr 13 im. Stanisława Staszica)	Warszawa	65
III Liceum Ogólnokształcące im. Marynarki Wojennej RP i Gimnazjum nr 24	Gdynia	59
V Liceum Ogólnokształcące im. Augusta Witkowskiego	Kraków	51
I Liceum Ogólnokształcące im. Adama Mickiewicza	Białystok	48
Zespół Szkół nr 14 (Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej oraz Gimnazjum nr 49 z Oddziałami Dwujęzycznymi)	Wrocław	27
Zespół Szkół Uniwersytetu Mikołaja Kopernika (Gimnazjum i Liceum Akademickie)	Toruń	25
Zespół Szkół Ogólnokształcących nr 6 (VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich oraz Gimnazjum nr 50)	Bydgoszcz	24
VIII Liceum Ogólnokształcące im. Adama Mickiewicza	Poznań	20
Zespół Szkół nr 15 (VIII Liceum Ogólnokształcące im. Króla Władysława IV i Gimnazjum nr 58 z Oddziałami Dwujęzycznymi im. Króla Władysława IV)	Warszawa	20
Zespół Szkół Ogólnokształcących nr 6 (VI Liceum Ogólnokształcące im. Jana Kochanowskiego oraz Publiczne Gimnazjum nr 23)	Radom	19
Zespół Szkół Ogólnokształcących nr 7 (XIII Liceum Ogólnokształcące oraz Gimnazjum nr 16)	Szczecin	17
Zespół Szkół i Placówek Oświatowych – V Liceum Ogólnokształcące	Bielsko-Biała	16
V Liceum Ogólnokształcące im. Stefana Żeromskiego	Gdańsk	15
I Liceum Ogólnokształcące im. Tadeusza Kościuszki	Legnica	15

Zespół Szkół Ogólnokształcących nr 2 (II Liceum Ogólnokształcące im. Hetmana Jana Tarnowskiego oraz Gimnazjum Dwujęzyczne im. Świętej Kingi)	Tarnów	13
Liceum Ogólnokształcące nr III im. Adama Mickiewicza	Wrocław	13
IV Liceum Ogólnokształcące im. Marii Skłodowskiej-Curie	Olsztyn	11
VIII Liceum Ogólnokształcące z Oddziałami Dwujęzycznymi im. Marii Skłodowskiej-Curie	Katowice	10
I Liceum Ogólnokształcące im. Stanisława Staszica	Lublin	9
Zespół Szkół Łączności im. Obrońców Poczty Polskiej w Gdańsku	Kraków	8
Zespół Szkół Ogólnokształcących nr 4 (IV Liceum Ogólnokształcące im. Tadeusza Kościuszki)	Toruń	7
Zespół Szkół Zawodowych i Ogólnokształcących im. I Brygady Pancernej im. Bohaterów Westerplatte	Kartuzy	6
I Liceum Ogólnokształcące im. Stanisława Dubois	Koszalin	6
III Liceum Ogólnokształcące im. św. Jana Kantego	Poznań	6
VI Liceum Ogólnokształcące im. Juliusza Słowackiego	Kielce	5
VIII Liceum Ogólnokształcące im. Stanisława Wyspiańskiego	Kraków	5
Publiczne Liceum Ogólnokształcące nr II z Oddziałami Dwujęzycznymi im. Marii Konopnickiej	Opole	5
I Liceum Ogólnokształcące im. Bolesława Prusa	Siedlce	5
XXVII Liceum Ogólnokształcące im. Tadeusza Czackiego	Warszawa	5
XXVIII Liceum Ogólnokształcące im. Jana Kochanowskiego	Warszawa	5
Zespół Szkół Techniczno-Informatycznych	Gliwice	4
II Liceum Ogólnokształcące im. Króla Jana III Sobieskiego	Kraków	4
I Liceum Ogólnokształcące im. Stefana Żeromskiego	Lębork	4
Zespół Szkół Ogólnokształcących (I Liceum Ogólnokształcące im. Tadeusza Kościuszki)	Łomża	4
Zespół Szkół Technicznych im. Stanisława Staszica (Technikum nr 1 im. Stanisława Staszica)	Rybnik	4
Gimnazjum i Liceum im. Jana Pawła II Sióstr Prezentek	Rzeszów	4
Zespół Szkół Elektronicznych (Technikum nr 6)	Rzeszów	4
I Liceum Ogólnokształcące im. Bolesława Krzywoustego	Słupsk	4
III Liceum Ogólnokształcące im. Adama Mickiewicza	Tarnów	4
Zespół Szkół Elektronicznych im. Wojska Polskiego	Bydgoszcz	3

I Liceum Ogólnokształcące im. Cypriana Kamila Norwida	Bydgoszcz	3
IX Liceum Ogólnokształcące im. Cypriana Kamila Norwida	Częstochowa	3
Zespół Szkół Ponadgimnazjalnych nr 2 Centrum Kształcenia Praktycznego im. Tadeusza Kościuszki	Garwolin	3
Zespół Szkół Sióstr Nazaretanek im. św. Jadwigi Królowej	Kielce	3
I Liceum Ogólnokształcące im. Władysława Jagiełły	Krasnystaw	3
Zespół Szkół Ogólnokształcących (I Liceum Ogólnokształcące im. Mikołaja Kopernika i Gimnazjum Dwujęzyczne im. Mikołaja Kopernika)	Krosno	3
Zespół Szkół im. Świętego Stanisława Kostki (XXI Liceum Ogólnokształcące oraz Gimnazjum im. Świętego Stanisława Kostki)	Lublin	3
I Liceum Ogólnokształcące im. Mikołaja Kopernika	Łódź	3
Zespół Szkół Komunikacji im. Hipolita Cegielskiego	Poznań	3
I Liceum Ogólnokształcące im. Juliusza Słowackiego	Przemyśl	3
I Liceum Ogólnokształcące im. Księcia Adama Jerzego Czartoryskiego	Puławy	3
III Liceum Ogólnokształcące im. płk. Dionizego Czachowskiego	Radom	3
Liceum Ogólnokształcące WSIiZ	Rzeszów	3
Zespół Szkół Ponadgimnazjalnych nr 1 im. Stanisława Staszica	Siedlce	3
Zespół Szkół Politechnicznych „Energetyk”	Wałbrzych	3
Gimnazjum nr 42 z Oddziałami Dwujęzycznymi	Warszawa	3
Liceum Ogólnokształcące nr 10 im. Stefanii Sempołowskiej	Wrocław	3

Najliczniej reprezentowane były następujące miejscowości:

Warszawa	126 zawodników	Olsztyn	16
Kraków	74	Rzeszów	15
Gdynia	65	Gdańsk	12
Wrocław	54	Katowice	12
Białystok	52	Kielce	12
Poznań	35	Siedlce	9
Bydgoszcz	32	Gliwice	7
Toruń	32	Opole	7
Radom	27	Kartuzy	6
Szczecin	22	Koszalin	6
Tarnów	21	Rybnik	6
Bielsko-Biała	19	Przemyśl	5
Legnica	16	Słupsk	5
Lublin	16	Częstochowa	4

Kalisz	4	Krasnystaw	3
Lębork	4	Krosno	3
Łomża	4	Piła	3
Łódź	4	Puławy	3
Nowy Sącz	4	Suwałki	3
Garwolin	3	Wałbrzych	3
Gorzów Wielkopolski	3	Zawiercie	3

Zawodnicy uczęszczali do następujących klas:

do klasy I gimnazjum	1 uczeń
do klasy II gimnazjum	15 uczniów
do klasy III gimnazjum	35
do klasy I szkoły ponadgimnazjalnej	128
do klasy II szkoły ponadgimnazjalnej	310
do klasy III szkoły ponadgimnazjalnej	344
do klasy IV szkoły ponadgimnazjalnej	44

W zawodach I stopnia zawodnicy mieli do rozwiązania pięć zadań:

- „Bar sałatkowy" autorstwa Jacka Tomasiewicza
- „Hotele" autorstwa Jacka Tomasiewicza
- „Klocki" autorstwa Wojciecha Ryttera i Bartosza Szredera
- „Kurierzy" autorstwa Jacka Tomasiewicza
- „Wąż" autorstwa Jakuba Radoszewskiego

każde oceniane maksymalnie po 100 punktów.

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

- **BAR** – Bar sałatkowy

	BAR	
	liczba zawodników	czyli
100 pkt.	206	23,49%
75–99 pkt.	88	10,03%
50–74 pkt.	218	24,86%
1–49 pkt.	153	17,45%
0 pkt.	119	13,57%
brak rozwiązania	93	10,60%

- **HOT** – Hotele

	HOT	
	liczba zawodników	czyli
100 pkt.	209	23,83%
75–99 pkt.	45	5,13%
50–74 pkt.	98	11,17%
1–49 pkt.	139	15,85%
0 pkt.	35	3,99%
brak rozwiązania	351	40,02%

- **KLO** – Klocki

	KLO	
	liczba zawodników	czyli
100 pkt.	177	20,18%
75–99 pkt.	92	10,49%
50–74 pkt.	81	9,24%
1–49 pkt.	182	20,75%
0 pkt.	167	19,04%
brak rozwiązania	178	20,30%

- **KUR** – Kurierzy

	KUR	
	liczba zawodników	czyli
100 pkt.	125	14,25%
75–99 pkt.	31	3,53%
50–74 pkt.	13	1,48%
1–49 pkt.	531	60,55%
0 pkt.	53	6,04%
brak rozwiązania	124	14,14%

- **WAZ** – Wąż

	WAZ	
	liczba zawodników	czyli
100 pkt.	2	0,23%
75–99 pkt.	1	0,11%
50–74 pkt.	1	0,11%
1–49 pkt.	151	17,22%
0 pkt.	9	1,03%
brak rozwiązania	713	81,30%

W sumie za wszystkie 5 zadań konkursowych:

SUMA	liczba zawodników	czyli
500 pkt.	2	0,23%
375–499 pkt.	87	9,92%
250–374 pkt.	174	19,84%
125–249 pkt.	204	23,26%
1–124 pkt.	338	38,54%
0 pkt.	72	8,21%

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 11–13 lutego 2014 roku w sześciu okręgach, zakwalifikowano 322 zawodników, którzy osiągnęli w zawodach I stopnia wynik nie mniejszy niż 216 pkt.

Zawodnicy zostali przydzieleni do okręgów w następującej liczbie:

Białystok	39 zawodników	Toruń	44
Gdańsk	44	Warszawa	83
Kraków	82	Wrocław	30

Trzech zawodników nie stawiło się na zawody, w zawodach wzięło więc udział 319 zawodników.

Zawodnicy uczęszczali do szkół w następujących województwach:

mazowieckie	64 zawodników	lubelskie	12
małopolskie	52	wielkopolskie	12
podlaskie	38	podkarpackie	10
pomorskie	28	warmińsko-mazurskie	8
dolnośląskie	26	świętokrzyskie	5
kujawsko-pomorskie	25	łódzkie	4
śląskie	16	opolskie	3
zachodniopomorskie	16		

W zawodach II stopnia najliczniej reprezentowane były szkoły:

nazwa szkoły	miejscowość	liczba uczniów
V Liceum Ogólnokształcące im. Augusta Witkowskiego	Kraków	36
I Liceum Ogólnokształcące im. Adama Mickiewicza	Białystok	31
Zespół Szkół nr 82 (XIV Liceum Ogólnokształcące im. Stanisława Staszica oraz Gimnazjum nr 13 im. Stanisława Staszica)	Warszawa	31
III Liceum Ogólnokształcące im. Marynarki Wojennej RP i Gimnazjum nr 24	Gdynia	19
Zespół Szkół nr 14 (Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej oraz Gimnazjum nr 49 z Oddziałami Dwujęzycznymi)	Wrocław	19
Zespół Szkół Ogólnokształcących nr 6 (VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich oraz Gimnazjum nr 50)	Bydgoszcz	16
Zespół Szkół Ogólnokształcących nr 7 (XIII Liceum Ogólnokształcące oraz Gimnazjum nr 16)	Szczecin	13
Zespół Szkół nr 15 (VIII Liceum Ogólnokształcące im. Króla Władysława IV i Gimnazjum nr 58 z Oddziałami Dwujęzycznymi im. Króla Władysława IV)	Warszawa	11
Zespół Szkół i Placówek Oświatowych – V Liceum Ogólnokształcące	Bielsko-Biała	9
Zespół Szkół Ogólnokształcących nr 6 (VI Liceum Ogólnokształcące im. Jana Kochanowskiego)	Radom	7
Zespół Szkół Uniwersytetu Mikołaja Kopernika (Liceum Akademickie)	Toruń	7
V Liceum Ogólnokształcące im. Stefana Żeromskiego	Gdańsk	5
I Liceum Ogólnokształcące im Stanisława Staszica	Lublin	5
IV Liceum Ogólnokształcące im. Marii Skłodowskiej-Curie	Olsztyn	5

Zespół Szkół Ogólnokształcących nr 2 (II Liceum Ogólnokształcące im. Hetmana Jana Tarnowskiego)	Tarnów	5
I Liceum Ogólnokształcące im. Tadeusza Kościuszki	Legnica	3
Zespół Szkół Ogólnokształcących (I Liceum Ogólnokształcące im. Tadeusza Kościuszki)	Łomża	3
VIII Liceum Ogólnokształcące im. Adama Mickiewicza	Poznań	3
III Liceum Ogólnokształcące im. św. Jana Kantego	Poznań	3
Gimnazjum i Liceum im. Jana Pawła II Sióstr Prezentek	Rzeszów	3

Najliczniej reprezentowane były miejscowości:

Warszawa	50	Gdańsk	7
Kraków	39	Lublin	7
Białystok	33	Toruń	7
Wrocław	22	Olsztyn	5
Gdynia	19	Częstochowa	3
Bydgoszcz	17	Kielce	3
Szczecin	14	Legnica	3
Bielsko-Biała	9	Łomża	3
Poznań	9	Nowy Sącz	3
Radom	9	Opole	3
Tarnów	8	Rzeszów	3

11 lutego odbyła się sesja próbna, podczas której zawodnicy rozwiązywali nieliczące się do ogólnej klasyfikacji zadanie „Karty” autorstwa Adama Polaka. W dniach konkursowych zawodnicy rozwiązywali następujące zadania:

- w pierwszym dniu zawodów (12 lutego):
  - „Przestępcy” autorstwa Pawła Parysa
  - „Superkomputer” autorstwa Michała Włodarczyka
- w drugim dniu zawodów (13 lutego):
  - „Ptaszek” autorstwa Jacka Tomasiewicza
  - „Rajd” autorstwa Bartosza Tarnawskiego

każde oceniane maksymalnie po 100 punktów.

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

- **KAR – próbne** – Karty

	KAR – próbne	
	liczba zawodników	czyli
100 pkt.	45	14,11%
75–99 pkt.	0	0,00%
50–74 pkt.	49	15,36%
1–49 pkt.	161	50,47%
0 pkt.	43	13,48%
brak rozwiązania	21	6,58%

- **PRZ** – Przestępcy

	PRZ	
	liczba zawodników	czyli
100 pkt.	62	19,44%
75–99 pkt.	15	4,70%
50–74 pkt.	17	5,33%
1–49 pkt.	153	47,96%
0 pkt.	49	15,36%
brak rozwiązania	23	7,21%

- **SUP** – Superkomputer

	SUP	
	liczba zawodników	czyli
100 pkt.	9	2,82%
75–99 pkt.	9	2,82%
50–74 pkt.	9	2,82%
1–49 pkt.	143	44,83%
0 pkt.	64	20,06%
brak rozwiązania	85	26,65%

- **PTA** – Ptaszek

	PTA	
	liczba zawodników	czyli
100 pkt.	81	25,39%
75–99 pkt.	17	5,33%
50–74 pkt.	73	22,88%
1–49 pkt.	88	27,59%
0 pkt.	41	12,85%
brak rozwiązania	19	5,96%

- **RAJ** – Rajd

	RAJ	
	liczba zawodników	czyli
100 pkt.	14	4,39%
75–99 pkt.	1	0,31%
50–74 pkt.	7	2,19%
1–49 pkt.	143	44,83%
0 pkt.	83	26,02%
brak rozwiązania	71	22,26%

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

SUMA	liczba zawodników	czyli
400 pkt.	1	0,31%
300–399 pkt.	10	3,13%
200–299 pkt.	63	19,75%
100–199 pkt.	121	37,93%
1–99 pkt.	108	33,86%
0 pkt.	16	5,02%

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

ZAWODY III STOPNIA

Zawody III stopnia odbyły się na Wydziale Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego w dniach 1–3 kwietnia 2014 roku. Do zawodów III stopnia zakwalifikowano 100 najlepszych uczestników zawodów II stopnia, którzy uzyskali wynik nie mniejszy niż 168 pkt.

Zawodnicy uczęszczali do szkół w następujących województwach:

mazowieckie	28 zawodników	lubelskie	4
małopolskie	14	podkarpackie	4
dolnośląskie	13	łódzkie	2
podlaskie	11	opolskie	1 zawodnik
pomorskie	9	świętokrzyskie	1
śląskie	6	warmińsko-mazurskie	1
kujawsko-pomorskie	5	wielkopolskie	1

Niżej wymienione szkoły miały w finale więcej niż jednego zawodnika:

nazwa szkoły	miejscowość	liczba uczniów
Zespół Szkół nr 82 (XIV Liceum Ogólnokształcące im. Stanisława Staszica oraz Gimnazjum nr 13 im. Stanisława Staszica)	Warszawa	19
Zespół Szkół nr 14 (Liceum Ogólnokształcące nr 14 im. Polonii Belgijskiej)	Wrocław	12
V Liceum Ogólnokształcące im. Augusta Witkowskiego	Kraków	11
III Liceum Ogólnokształcące im. Marynarki Wojennej RP i Gimnazjum nr 24	Gdynia	8
I Liceum Ogólnokształcące im. Adama Mickiewicza	Białystok	7
Zespół Szkół i Placówek Oświatowych – V Liceum Ogólnokształcące	Bielsko-Biała	4
Zespół Szkół nr 15 (VIII Liceum Ogólnokształcące im. Króla Władysława IV)	Warszawa	4
Zespół Szkół Ogólnokształcących nr 6 (VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich)	Bydgoszcz	3
Zespół Szkół Uniwersytetu Mikołaja Kopernika (Liceum Akademickie)	Toruń	2

1 kwietnia odbyła się sesja próbna, podczas której zawodnicy rozwiązywali nieliczące się do ogólnej klasyfikacji zadanie „Farmercraft” autorstwa Jacka Tomasiewicza. W dniach konkursowych zawodnicy rozwiązywali następujące zadania:

- w pierwszym dniu zawodów (2 kwietnia):
  - „Dookoła świata” autorstwa Jana Kantego Milczka i Jakuba Pachockiego
  - „Mrowisko” autorstwa Jacka Tomasiewicza

  - „Turystyka” autorstwa Marcina Pilipczuka
- w drugim dniu zawodów (3 kwietnia):
  - „Lampy słoneczne” autorstwa Wojciecha Nadary
  - „Panele słoneczne” autorstwa Bartosza Tarnawskiego
  - „Załadunek” autorstwa Michała Włodarczyka

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:

- **FAR – próbne** – Farmercraft

	FAR – próbne	
	liczba zawodników	czyli
100 pkt.	47	47,00%
75–99 pkt.	1	1,00%
50–74 pkt.	1	1,00%
1–49 pkt.	13	13,00%
0 pkt.	32	32,00%
brak rozwiązania	6	6,00%

- **DOO** – Dookoła świata

	DOO	
	liczba zawodników	czyli
100 pkt.	13	13,00%
75–99 pkt.	0	0,00%
50–74 pkt.	4	4,00%
1–49 pkt.	57	57,00%
0 pkt.	20	20,00%
brak rozwiązania	6	6,00%

- **MRO** – Mrowisko

	MRO	
	liczba zawodników	czyli
100 pkt.	60	60,00%
75–99 pkt.	0	0,00%
50–74 pkt.	13	13,00%
1–49 pkt.	21	21,00%
0 pkt.	3	3,00%
brak rozwiązania	3	3,00%

- **TUR** – Turystyka

	TUR	
	liczba zawodników	czyli
100 pkt.	0	0,00%
75–99 pkt.	0	0,00%
50–74 pkt.	0	0,00%
1–49 pkt.	76	76,00%
0 pkt.	9	9,00%
brak rozwiązania	15	15,00%

- **LAM** – Lampy słoneczne

	LAM	
	liczba zawodników	czyli
100 pkt.	0	0,00%
75–99 pkt.	0	0,00%
50–74 pkt.	0	0,00%
1–49 pkt.	62	62,00%
0 pkt.	8	8,00%
brak rozwiązania	30	30,00%

- **PAN** – Panele słoneczne

	PAN	
	liczba zawodników	czyli
100 pkt.	10	10,00%
75–99 pkt.	1	1,00%
50–74 pkt.	2	2,00%
1–49 pkt.	59	59,00%
0 pkt.	21	21,00%
brak rozwiązania	7	7,00%

- **ZAL** – Załadunek

	ZAL	
	liczba zawodników	czyli
100 pkt.	10	10,00%
75–99 pkt.	2	2,00%
50–74 pkt.	6	6,00%
1–49 pkt.	33	33,00%
0 pkt.	10	10,00%
brak rozwiązania	39	39,00%

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

SUMA	liczba zawodników	czyli
600 pkt.	0	0%
450–599 pkt.	1	1,00%
300–449 pkt.	12	12,00%
150–299 pkt.	44	44,00%
1–149 pkt.	43	43,00%
0 pkt.	0	0%

4 kwietnia 2014 roku, na Wydziale Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego w Warszawie, ogłoszono wyniki finału XXI Olimpiady Informatycznej 2013/2014 i rozdano nagrody ufundowane przez: Ministerstwo Edukacji Narodowej, Asseco Poland SA, Jane Street Europe LTD, Ośrodek Edukacji Informatycznej i Zastosowań Komputerów w Warszawie, Software Mind SA, Gemius SA, Olimpiadę Informatyczną, Wydawnictwa Naukowe PWN i Wydawnictwo „Delta".

Poniżej zestawiono listę wszystkich laureatów i wyróżnionych finalistów:

(1) **Jarosław Kwiecień**, 1 klasa, Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej w Zespole Szkół nr 14, Wrocław, 450 pkt., laureat I miejsca

(2) **Stanisław Barzowski**, 3 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia, 420 pkt., laureat I miejsca

(3) **Michał Glapa**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków, 380 pkt., laureat II miejsca

(4) **Albert Citko**, 3 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica w Zespole Szkół nr 82, Warszawa, 378 pkt., laureat II miejsca

(5) **Maciej Hołubowicz**, 2 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza, Białystok, 364 pkt., laureat II miejsca

(6) **Wojciech Jabłoński**, 3 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica w Zespole Szkół nr 82, Warszawa, 358 pkt., laureat II miejsca

(7) **Jan Tabaszewski**, 1 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica w Zespole Szkół nr 82, Warszawa, 357 pkt., laureat II miejsca

(8) **Szymon Łukasz**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków, 350 pkt., laureat II miejsca

(9) **Tomasz Garbus**, 2 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia, 320 pkt., laureat II miejsca

(10) **Paweł Burzyński**, 3 klasa, Gimnazjum nr 24 przy III Liceum Ogólnokształcącym im. Marynarki Wojennej RP, Gdynia, 317 pkt., laureat II miejsca

(11) **Albert Gutowski**, 3 klasa, I Liceum Ogólnokształcące im. Stanisława Staszica, Lublin, 312 pkt., laureat II miejsca

(12) **Jakub Cisło**, 3 klasa, II Liceum Ogólnokształcące im. Mikołaja Kopernika, Mielec, 300 pkt., laureat II miejsca

(12) **Sebastian Jaszczur**, 3 klasa, VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich w Zespole Szkół Ogólnokształcących nr 6, Bydgoszcz, 300 pkt., laureat II miejsca

(14) **Konrad Paluszek**, 2 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica w Zespole Szkół nr 82, Warszawa, 296 pkt., laureat III miejsca

(15) **Marek Zbysiński**, 2 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica w Zespole Szkół nr 82, Warszawa, 292 pkt., laureat III miejsca

(16) **Jan Gwinner**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków, 278 pkt., laureat III miejsca

(16) **Marek Sokołowski**, 3 klasa, I Liceum Ogólnokształcące im. Tadeusza Kościuszki, Łomża, 278 pkt., laureat III miejsca

(18) **Karol Kaszuba**, 3 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica w Zespole Szkół nr 82, Warszawa, 270 pkt., laureat III miejsca

(19) **Michał Zawalski**, 2 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica w Zespole Szkół nr 82, Warszawa, 250 pkt., laureat III miejsca

(20) **Katarzyna Kowalska**, 2 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica w Zespole Szkół nr 82, Warszawa, 243 pkt., laureatka III miejsca

(21) **Michał Zieliński**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków, 242 pkt., laureat III miejsca

(22) **Michał Łuszczyk**, 3 klasa, IV Liceum Ogólnokształcące im. Jana Pawła II w Zespole Szkół Ogólnokształcących nr 1, Tarnów, 240 pkt., laureat III miejsca

(22) **Paweł Wegner**, 3 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia, 240 pkt., laureat III miejsca

(24) **Franciszek Stokowacki**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków, 239 pkt., laureat III miejsca

(25) **Bartosz Łukasiewicz**, 3 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia, 235 pkt., laureat III miejsca

(26) **Przemysław Jakub Kozłowski**, 3 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza, Białystok, 232 pkt., laureat III miejsca

(27) **Jakub Skorupski**, 3 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica w Zespole Szkół nr 82, Warszawa, 223 pkt., laureat III miejsca

(27) **Adam Trzaskowski**, 3 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica w Zespole Szkół nr 82, Warszawa, 223 pkt., laureat III miejsca

(29) **Tomasz Kościuszko**, 1 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica w Zespole Szkół nr 82, Warszawa, 220 pkt., laureat III miejsca

(30) **Jakub Boguta**, 3 klasa, Gimnazjum im. św. Stanisława Kostki, Lublin, 215 pkt., laureat III miejsca

(31) **Filip Czaplicki**, 3 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia, 211 pkt., laureat III miejsca

(32) **Jakub Staroń**, 3 klasa, V Liceum Ogólnokształcące, Bielsko-Biała, 208 pkt., laureat III miejsca

(33) **Weronika Grzybowska**, 2 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków, 204 pkt., laureatka III miejsca

(34) **Jarosław Dzikowski**, 3 klasa, Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej w Zespole Szkół nr 14, Wrocław, 198 pkt., wyróżniony finalista

(35) **Adam Kuczaj**, 2 klasa, Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej w Zespole Szkół nr 14, Wrocław, 194 pkt., wyróżniony finalista

(36) **Marcin Karpiński**, 3 klasa, I Liceum Ogólnokształcące im. Króla Stanisława Leszczyńskiego, Jasło, 190 pkt., wyróżniony finalista

(36) **Jakub Łabaj**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków, 190 pkt., wyróżniony finalista

(36) **Krzysztof Piesiewicz**, 1 klasa, VIII Liceum Ogólnokształcące im. Króla Władysława IV w Zespole Szkół nr 15, Warszawa, 190 pkt., wyróżniony finalista

(39) **Mateusz Wytrwał**, 2 klasa, II Liceum Ogólnokształcące w Zespole Szkół Ogólnokształcących nr 2, Tarnów, 186 pkt., wyróżniony finalista

(40) **Paweł Solecki**, 2 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica w Zespole Szkół nr 82, Warszawa, 185 pkt., wyróżniony finalista

(41) **Maciej Kucharski**, 2 klasa, Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej w Zespole Szkół nr 14, Wrocław, 184 pkt., wyróżniony finalista

(41) **Kasper Radek**, 3 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica w Zespole Szkół nr 82, Warszawa, 184 pkt., wyróżniony finalista

(41) **Maciej Sypetkowski**, 1 klasa, I Liceum Ogólnokształcące im. Władysława Jagiełły, Krasnystaw, 184 pkt., wyróżniony finalista

(44) **Michał Tepper**, 1 klasa, VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich w Zespole Szkół Ogólnokształcących nr 6, Bydgoszcz, 179 pkt., wyróżniony finalista

(45) **Zuzanna Pilat**, 3 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica w Zespole Szkół nr 82, Warszawa, 176 pkt., wyróżniona finalistka

(45) **Magdalena Szarkowska**, 3 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza, Białystok, 176 pkt., wyróżniona finalistka

(47) **Kamil Rychlewicz**, 3 klasa, I Liceum Ogólnokształcące im. Mikołaja Kopernika, Łódź, 175 pkt., wyróżniony finalista

(48) **Adrian Naruszko**, 2 klasa, VI Liceum Ogólnokształcące im. Jana Kochanowskiego w Zespole Szkół Ogólnokształcących nr 6, Radom, 170 pkt., wyróżniony finalista

(48) **Martyna Siejba**, 2 klasa, Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej w Zespole Szkół nr 14, Wrocław, 170 pkt., wyróżniona finalistka

Lista pozostałych finalistów w kolejności alfabetycznej:

- **Bartosz Barwikowski**, 3 klasa, I Liceum Ogólnokształcące im. Mikołaja Kopernika, Krosno
- **Anna Białokozowicz**, 1 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza, Białystok
- **Franciszek Budrowski**, 3 klasa, Społeczne Gimnazjum nr 2 Społecznego Towarzystwa Oświatowego, Białystok
- **Jakub Bujak**, 2 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica w Zespole Szkół nr 82, Warszawa
- **Mateusz Chołołowicz**, 3 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza, Białystok
- **Adam Chudaś**, 2 klasa, Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej w Zespole Szkół nr 14, Wrocław
- **Albert Cieślak**, 2 klasa, II Liceum Ogólnokształcące im. Marii Skłodowskiej-Curie, Końskie
- **Jan Derbisz**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków
- **Damian Dmochowski**, 4 klasa, Zespół Szkół nr 2, Ostrów Mazowiecka
- **Agnieszka Dudek**, 1 klasa, Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej w Zespole Szkół nr 14, Wrocław
- **Michał Figlus**, 3 klasa, I Liceum Ogólnokształcące im. Bolesława Chrobrego, Piotrków Trybunalski
- **Szymon Gajda**, 1 klasa, II Liceum Ogólnokształcące im. Heleny Malczewskiej, Zawiercie
- **Jan Góra**, 3 klasa, I Liceum Ogólnokształcące im. Księcia Adama Jerzego Czartoryskiego, Puławy
- **Marcin Gregorczyk**, 3 klasa, IV Liceum Ogólnokształcące im. Marii Skłodowskiej-Curie, Olsztyn
- **Tomasz Grześkiewicz**, 3 klasa, Gimnazjum nr 13 im. Stanisława Staszica w Zespole Szkół nr 82, Warszawa
- **Emil Hotkowski**, 3 klasa, Liceum Ogólnokształcące im. Stefana Żeromskiego, Żyrardów
- **Łukasz Kempys**, 3 klasa, V Liceum Ogólnokształcące, Bielsko-Biała
- **Eryk Kijewski**, 1 klasa, I Liceum Ogólnokształcące im. Marii Konopnickiej, Suwałki
- **Dominik Klemba**, 2 klasa, XVIII Liceum Ogólnokształcące im. Jana Zamoyskiego, Warszawa
- **Wiktoria Kośny**, 1 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica w Zespole Szkół nr 82, Warszawa

- **Krzysztof Krawczyk**, 3 klasa, VIII Liceum Ogólnokształcące im. Marii Skłodowskiej-Curie, Katowice
- **Mateusz Lewko**, 2 klasa, Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej w Zespole Szkół nr 14, Wrocław
- **Jan Ludziejewski**, 2 klasa, VIII Liceum Ogólnokształcące im. Króla Władysława IV w Zespole Szkół nr 15, Warszawa
- **Michał Łazowik**, 3 klasa, Liceum Akademickie w Zespole Szkół Uniwersytetu Mikołaja Kopernika, Toruń
- **Wojciech Mańke**, 3 klasa, VIII Liceum Ogólnokształcące im. Adama Mickiewicza, Poznań
- **Jakub Marciniszyn**, 3 klasa, V Liceum Ogólnokształcące, Bielsko-Biała
- **Martyna Mikoda**, 3 klasa, Publiczne Liceum Ogólnokształcące nr II z Oddziałami Dwujęzycznymi im. Marii Konopnickiej, Opole
- **Miłosz Mokwa**, 2 klasa, I Liceum Ogólnokształcące im. Marii Skłodowskiej-Curie, Starogard Gdański
- **Ngoc Khanh Nguyen**, 3 klasa, Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej w Zespole Szkół nr 14, Wrocław
- **Jakub Obuchowski**, 1 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza, Białystok
- **Jan Olkowski**, 3 klasa, Gimnazjum nr 42 z Oddziałami Dwujęzycznymi, Warszawa
- **Michał Orawiec**, 2 klasa, VIII Liceum Ogólnokształcące im. Króla Władysława IV w Zespole Szkół nr 15, Warszawa
- **Franciszek Piszcz**, 3 klasa, VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich w Zespole Szkół Ogólnokształcących nr 6, Bydgoszcz
- **Julian Pszczołowski**, 2 klasa, Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej w Zespole Szkół nr 14, Wrocław
- **Mateusz Puczel**, 3 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza, Białystok
- **Piotr Pusz**, 2 klasa, Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej w Zespole Szkół nr 14, Wrocław
- **Kamil Rajtar**, 2 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków
- **Łukasz Raszkiewicz**, 2 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia
- **Jakub Rówiński**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków
- **Jakub Schreiber**, 3 klasa, Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej w Zespole Szkół nr 14, Wrocław
- **Karol Smolak**, 1 klasa, Liceum Ogólnokształcące im. Tadeusza Kościuszki, Ropczyce
- **Jakub Sroka**, 3 klasa, Liceum Ogólnokształcące nr 1 im. Eugeniusza Romera, Rabka-Zdrój

- **Jakub Szymon Stanecki**, 3 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica w Zespole Szkół nr 82, Warszawa
- **Michał Stanisz**, 2 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków
- **Michał Swidziński**, 2 klasa, II Liceum Ogólnokształcące im. księżnej Anny z Sapiehów Jabłonowskiej w Zespole Szkół Ogólnokształcących nr 2, Białystok
- **Marcin Szwarc**, 3 klasa, VIII Liceum Ogólnokształcące im. Króla Władysława IV w Zespole Szkół nr 15, Warszawa
- **Michał Wiatrowski**, 3 klasa, V Liceum Ogólnokształcące, Bielsko-Biała
- **Krzysztof Wojculewicz**, 3 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia
- **Bartłomiej Wróblewski**, 2 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica w Zespole Szkół nr 82, Warszawa
- **Michał Zabłocki**, 2 klasa, Liceum Akademickie w Zespole Szkół Uniwersytetu Mikołaja Kopernika, Toruń
- **Jakub Zadrożny**, 1 klasa, Liceum Ogólnokształcące nr III, Wrocław

Komitet Główny Olimpiady Informatycznej przyznał następujące nagrody rzeczowe:

- puchar wręczono zwycięzcy XXI Olimpiady Jarosławowi Kwietniowi,
- złote, srebrne i brązowe medale przyznano odpowiednio laureatom I, II i III miejsca,
- laptopy (2 szt.) przyznano laureatom I miejsca,
- tablety (11 szt.) przyznano laureatom II miejsca,
- dyski twarde (20 szt.) przyznano laureatom III miejsca,
- pamięci USB (16 szt.) przyznano wyróżnionym finalistom,
- książki ufundowane przez PWN przyznano wszystkim finalistom,
- roczną prenumeratę miesięcznika „Delta” przyznano wszystkim laureatom i wyróżnionym finalistom.

Komitet Główny powołał reprezentacje na:

(1) **Międzynarodową Olimpiadę Informatyczną IOI'2014**, która odbędzie się na Tajwanie, w miejscowości Tajpej, w terminie 13–20 lipca 2014 roku

(2) oraz **Olimpiadę Informatyczną Krajów Europy Środkowej CEOI'2014**, która odbędzie się w Niemczech, w miejscowości Jena, w terminie 18–24 czerwca 2014 roku, w składzie:

   - Jarosław Kwiecień
   - Stanisław Barzowski
   - Michał Glapa
   - Albert Citko

rezerwowi:

- Maciej Hołubowicz
- Wojciech Jabłoński

(3) Na **Bałtycką Olimpiadę Informatyczną BOI'2014**, która odbyła się w terminie 26–30 kwietnia 2014 na Litwie w miejscowości Palanga, pojechali zawodnicy, którzy nie uczęszczają do klas maturalnych, w kolejności rankingowej:

- Jarosław Kwiecień
- Maciej Hołubowicz
- Jan Tabaszewski
- Tomasz Garbus
- Paweł Burzyński
- Konrad Paluszek

W tym roku wyniki osiągnięte przez naszą reprezentację były naprawdę wysokie: zwycięzcą całych zawodów został Jarosław Kwiecień, a wszyscy reprezentanci zdobyli w zawodach co najmniej srebrny medal. Wyniki szczegółowe reprezentacji Polski przedstawiają się następująco:

- złote medale zdobyli: Jarosław Kwiecień, Jan Tabaszewski i Konrad Paluszek,
- a srebrne medale uzyskali: Paweł Burzyński, Maciej Hołubowicz i Tomasz Garbus.

Komitet Główny podjął następujące uchwały o udziale młodzieży w obozach:

- w Obozie Naukowo-Treningowym im. A. Kreczmara w Kielnarowej k. Rzeszowa weźmie udział piętnastu pierwszych z rankingu laureatów i finalistów Olimpiady, którzy nie uczęszczają w tym roku szkolnym do programowo najwyższej klasy szkoły ponadgimnazjalnej,
- w Obozie Wyszehradzkim, który odbędzie się na Słowacji, wezmą udział reprezentanci na Międzynarodową Olimpiadę Informatyczną, wraz z zawodnikami rezerwowymi.

Ponieważ udało się pozyskać dodatkowe fundusze na organizację Obozu Naukowo-Treningowego im. A. Kreczmara, do uczestnictwa zaproszono wszystkich laureatów i finalistów XXI Olimpiady Informatycznej spełniających kryterium uczęszczanej klasy.

Sekretariat wystawił łącznie 33 zaświadczenia o uzyskaniu tytułu laureata, 16 zaświadczeń o uzyskaniu tytułu wyróżnionego finalisty oraz 51 zaświadczeń o uzyskaniu tytułu finalisty XXI Olimpiady Informatycznej.

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

- Szymon Acedański (doktorant Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego)

  - Paweł Burzyński – laureat II miejsca
- Michał Adamczyk (student Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego)
  - Jan Tabaszewski – laureat II miejsca
  - Kasper Radek – finalista z wyróżnieniem
- Marian Bąk (Zespół Szkół nr 14 we Wrocławiu)
  - Mateusz Lewko – finalista
- Iwona Bujnowska (I Liceum Ogólnokształcące im. Adama Mickiewicza w Białymstoku)
  - Magdalena Szarkowska – finalistka z wyróżnieniem
  - Anna Białokozowicz – finalistka
  - Mateusz Chołołowicz – finalista
  - Jakub Obuchowski – finalista
  - Mateusz Puczel – finalista
- Ireneusz Bujnowski (I Liceum Ogólnokształcące im. Adama Mickiewicza w Białymstoku)
  - Maciej Hołubowicz – laureat II miejsca
  - Przemysław Jakub Kozłowski – laureat III miejsca
  - Marek Sokołowski – laureat III miejsca
  - Magdalena Szarkowska – finalistka z wyróżnieniem
  - Anna Białokozowicz – finalistka
  - Mateusz Chołołowicz – finalista
  - Jakub Obuchowski – finalista
  - Mateusz Puczel – finalista
  - Michał Swidziński – finalista
- Magda Burakowska (Zespół Szkół Ogólnokształcących nr 2 w Olsztynie)
  - Marcin Gregorczyk – finalista
- Patryk Czajka (student Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego)
  - Tomasz Grześkiewicz – finalista
- Bartłomiej Dudek (student Instytutu Informatyki Uniwersytetu Wrocławskiego)
  - Jarosław Kwiecień – laureat I miejsca
  - Jarosław Dzikowski – finalista z wyróżnieniem
  - Maciej Kucharski – finalista z wyróżnieniem
  - Martyna Siejba – finalistka z wyróżnieniem
  - Agnieszka Dudek – finalistka
  - Mateusz Lewko – finalista
  - Julian Pszczołowski – finalista
  - Jakub Schreiber – finalista
  - Jakub Zadrożny – finalista

- Lech Duraj (Uniwersytet Jagielloński)
  - Michał Glapa – laureat II miejsca
  - Szymon Łukasz – laureat II miejsca
  - Jan Gwinner – laureat III miejsca
  - Michał Zieliński – laureat III miejsca
  - Jakub Rówiński – finalista
- Piotr Dybicz (student Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego)
  - Adam Trzaskowski – laureat III miejsca
- Andrzej Dyrek (V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie)
  - Michał Glapa – laureat II miejsca
  - Szymon Łukasz – laureat II miejsca
  - Jan Gwinner – laureat III miejsca
  - Michał Zieliński – laureat III miejsca
  - Franciszek Stokowacki – laureat III miejsca
  - Jakub Łabaj – finalista z wyróżnieniem
  - Jan Derbisz – finalista
  - Jakub Rówiński – finalista
- Karol Farbiś (student Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego)
  - Kasper Radek – finalista z wyróżnieniem
  - Albert Cieślak – finalista
- Elżbieta Gajda (Gimnazjum w Ogrodzieńcu)
  - Szymon Gajda – finalista
- Marek Gajda (Zespół Szkół im. Stanisława Staszica w Zawierciu)
  - Szymon Gajda – finalista
- Alina Gościniak (VIII Liceum Ogólnokształcące im. Adama Mickiewicza w Poznaniu)
  - Wojciech Mańke – finalista
- Józef Graboś (Liceum Ogólnokształcące im. Tadeusza Kościuszki w Ropczycach)
  - Karol Smolak – finalista
- Grzegorz Herman (Uniwersytet Jagielloński)
  - Michał Glapa – laureat II miejsca
  - Szymon Łukasz – laureat II miejsca
  - Jan Gwinner – laureat III miejsca
  - Michał Zieliński – laureat III miejsca
  - Franciszek Stokowacki – laureat III miejsca
  - Jakub Łabaj – finalista z wyróżnieniem
  - Jakub Rówiński – finalista

- Krzysztof Hyżyk (VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich w Bydgoszczy)
  - Sebastian Jaszczur – laureat II miejsca
  - Franciszek Piszcz – finalista
- Andrzej Jackowski (Społeczne Gimnazjum nr 2 Społecznego Towarzystwa Oświatowego w Białymstoku)
  - Franciszek Budrowski – finalista
- Wiktor Janas (student Instytutu Informatyki Uniwersytetu Wrocławskiego)
  - Jarosław Kwiecień – laureat I miejsca
  - Jarosław Dzikowski – finalista z wyróżnieniem
  - Maciej Kucharski – finalista z wyróżnieniem
  - Martyna Siejba – finalistka z wyróżnieniem
  - Agnieszka Dudek – finalistka
  - Mateusz Lewko – finalista
  - Ngoc Khanh Nguyen – finalista
  - Julian Pszczołowski – finalista
  - Jakub Zadrożny – finalista
- Henryk Kawka (Zespół Szkół nr 7 w Lublinie)
  - Albert Gutowski – laureat II miejsca
  - Jakub Boguta – laureat III miejsca
  - Jan Góra – finalista
- Krzysztof Kiewicz (student Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego)
  - Krzysztof Piesiewicz – finalista z wyróżnieniem
- Konrad Kijewski (student Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego)
  - Eryk Kijewski – finalista
- Karol Konaszyński (student Instytutu Informatyki Uniwersytetu Wrocławskiego)
  - Jarosław Kwiecień – laureat I miejsca
  - Maciej Kucharski – finalista z wyróżnieniem
  - Martyna Siejba – finalistka z wyróżnieniem
  - Agnieszka Dudek – finalistka
  - Mateusz Lewko – finalista
  - Julian Pszczołowski – finalista
  - Jakub Schreiber – finalista
  - Jakub Zadrożny – finalista
- Bartosz Kostka (student Instytutu Informatyki Uniwersytetu Wrocławskiego)
  - Jarosław Kwiecień – laureat I miejsca
  - Maciej Kucharski – finalista z wyróżnieniem
  - Martyna Siejba – finalistka z wyróżnieniem
  - Agnieszka Dudek – finalistka
  - Mateusz Lewko – finalista

  - Julian Pszczołowski – finalista
  - Jakub Zadrożny – finalista
- Konrad Kras (Fabre Polska sp. z o.o. w Krakowie)
  - Michał Łuszczyk – laureat III miejsca
- Wiktor Kuropatwa (student Uniwersytetu Jagiellońskiego)
  - Michał Glapa – laureat II miejsca
  - Szymon Łukasz – laureat II miejsca
  - Jan Gwinner – laureat III miejsca
  - Michał Zieliński – laureat III miejsca
  - Albert Cieślak – finalista
  - Jakub Rówiński – finalista
- Jan Kwaśniak (student Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego)
  - Albert Cieślak – finalista
- Anna Beata Kwiatkowska (Liceum Akademickie Uniwersytetu Mikołaja Kopernika w Toruniu)
  - Michał Łazowik – finalista
  - Michał Zabłocki – finalista
- Ryszard Lisoń (Publiczne Liceum Ogólnokształcące nr II w Opolu)
  - Martyna Mikoda – finalistka
- Piotr Łowicki (I Liceum Ogólnokształcące w Łomży)
  - Marek Sokołowski – laureat III miejsca
- Tom Macieszczak (student Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego)
  - Kasper Radek – finalista z wyróżnieniem
  - Tomasz Grześkiewicz – finalista
- Paweł Mateja (I Liceum Ogólnokształcące im. Mikołaja Kopernika w Łodzi)
  - Kamil Rychlewicz – finalista z wyróżnieniem
- Dawid Matla (Zespół Szkół nr 14 we Wrocławiu)
  - Jarosław Kwiecień – laureat I miejsca
  - Maciej Kucharski – finalista z wyróżnieniem
  - Martyna Siejba – finalistka z wyróżnieniem
  - Agnieszka Dudek – finalistka
  - Mateusz Lewko – finalista
  - Julian Pszczołowski – finalista
  - Jakub Schreiber – finalista
  - Jakub Zadrożny – finalista
- Ewa Matuszewska (Liceum Ogólnokształcące im. Stefana Żeromskiego w Żyrardowie)
  - Emil Hotkowski – finalista

- Jan Kanty Milczek (student Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego)
  - Filip Czaplicki – laureat III miejsca
  - Tomasz Kościuszko – laureat III miejsca
- Mirosław Mortka (Zespół Szkół Ogólnokształcących nr 6 w Radomiu)
  - Adrian Naruszko – finalista z wyróżnieniem
- Wojciech Nadara (student Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego)
  - Kasper Radek – finalista z wyróżnieniem
- Rafał Nowak (Instytut Informatyki Uniwersytetu Wrocławskiego)
  - Jarosław Kwiecień – laureat I miejsca
  - Jarosław Dzikowski – finalista z wyróżnieniem
  - Maciej Kucharski – finalista z wyróżnieniem
  - Martyna Siejba – finalistka z wyróżnieniem
  - Agnieszka Dudek – finalistka
  - Mateusz Lewko – finalista
  - Ngoc Khanh Nguyen – finalista
  - Julian Pszczołowski – finalista
  - Jakub Schreiber – finalista
  - Jakub Zadrożny – finalista
- Zofia Olędzka (VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich w Bydgoszczy)
  - Michał Tepper – finalista z wyróżnieniem
- Małgorzata Piekarska (VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich w Bydgoszczy)
  - Franciszek Piszcz – finalista
  - Michał Tepper – finalista z wyróżnieniem
  - Sebastian Jaszczur – laureat II miejsca
- Mirosław Pietrzycki (I Liceum Ogólnokształcące im. Stanisława Staszica w Lublinie)
  - Albert Gutowski – laureat II miejsca
- Andrzej Piotrowski (I Liceum Ogólnokształcące im. Mikołaja Kopernika w Krośnie)
  - Bartosz Barwikowski – finalista
- Karol Pokorski (student Instytutu Informatyki Uniwersytetu Wrocławskiego)
  - Jarosław Kwiecień – laureat I miejsca
  - Maciej Kucharski – finalista z wyróżnieniem
  - Adam Kuczaj – finalista z wyróżnieniem
  - Martyna Siejba – finalistka z wyróżnieniem
  - Adam Chudaś – finalista
  - Agnieszka Dudek – finalistka
  - Mateusz Lewko – finalista
  - Julian Pszczołowski – finalista

  - Piotr Pusz – finalista
  - Jakub Zadrożny – finalista
- Adam Polak (doktorant Uniwersytetu Jagiellońskiego)
  - Michał Glapa – laureat II miejsca
  - Szymon Łukasz – laureat II miejsca
  - Jan Gwinner – laureat III miejsca
  - Weronika Grzybowska – laureatka III miejsca
  - Jakub Łabaj – finalista z wyróżnieniem
  - Albert Cieślak – finalista
  - Kamil Rajtar – finalista
  - Jakub Rówiński – finalista
  - Michał Stanisz – finalista
- Krzysztof Pszeniczny (student Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego)
  - Jakub Cisło – laureat II miejsca
- Włodzimierz Raczek (V Liceum Ogólnokształcące w Bielsku-Białej)
  - Jakub Staroń – laureat III miejsca
  - Michał Wiatrowski – finalista
- Damian Rusak (student Instytutu Informatyki Uniwersytetu Wrocławskiego)
  - Jarosław Kwiecień – laureat I miejsca
  - Jarosław Dzikowski – finalista z wyróżnieniem
  - Maciej Kucharski – finalista z wyróżnieniem
  - Martyna Siejba – finalistka z wyróżnieniem
  - Agnieszka Dudek – finalistka
  - Mateusz Lewko – finalista
  - Julian Pszczołowski – finalista
  - Jakub Schreiber – finalista
  - Jakub Zadrożny – finalista
- Marek Rusinowski (student Uniwersytetu Jagiellońskiego)
  - Jakub Cisło – laureat II miejsca
- Agnieszka Samulska (VIII Liceum Ogólnokształcące im. Króla Władysława IV w Warszawie)
  - Krzysztof Piesiewicz – finalista z wyróżnieniem
  - Michał Orawiec – finalista
  - Marcin Szwarc – finalista
- Piotr Smulewicz (student Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego)
  - Jan Ludziejewski – finalista
- Marek Sommer (student Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego)
  - Jan Tabaszewski – laureat II miejsca
  - Tomasz Kościuszko – laureat III miejsca

  - Katarzyna Kowalska – laureatka III miejsca
  - Jakub Skorupski – laureat III miejsca
  - Michał Zawalski – laureat III miejsca
  - Marek Zbysiński – laureat III miejsca
  - Kasper Radek – finalista z wyróżnieniem
  - Albert Cieślak – finalista
  - Michał Orawiec – finalista
  - Jakub Stanecki – finalista
- Hanna Stachera (XIV Liceum Ogólnokształcące im. Stanisława Staszica w Warszawie)
  - Wojciech Jabłoński – laureat II miejsca
- Szymon Stankiewicz (student Uniwersytetu Jagiellońskiego)
  - Adrian Naruszko – finalista z wyróżnieniem
- Jacek Stańczak (I Liceum Ogólnokształcące im. Bolesława Chrobrego w Piotrkowie Trybunalskim)
  - Michał Figlus – finalista
- Rafał Stefański (student Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego)
  - Jan Tabaszewski – laureat II miejsca
  - Katarzyna Kowalska – laureatka III miejsca
  - Michał Zawalski – laureat III miejsca
  - Marek Zbysiński – laureat III miejsca
  - Kasper Radek – finalista z wyróżnieniem
  - Michał Orawiec – finalista
- Damian Straszak (student Instytutu Informatyki Uniwersytetu Wrocławskiego)
  - Jarosław Kwiecień – laureat I miejsca
  - Martyna Siejba – finalistka z wyróżnieniem
  - Maciej Kucharski – finalista z wyróżnieniem
  - Agnieszka Dudek – finalistka
  - Mateusz Lewko – finalista
  - Julian Pszczołowski – finalista
  - Jakub Zadrożny – finalista
- Tomasz Syposz (student Instytutu Informatyki Uniwersytetu Wrocławskiego)
  - Jarosław Kwiecień – laureat I miejsca
  - Martyna Siejba – finalistka z wyróżnieniem
  - Maciej Kucharski – finalista z wyróżnieniem
  - Agnieszka Dudek – finalistka
  - Mateusz Lewko – finalista
  - Julian Pszczołowski – finalista
  - Jakub Zadrożny – finalista

- Ryszard Szubartowski (III Liceum Ogólnokształcące im. Marynarki Wojennej RP w Gdyni)
  - Stanisław Barzowski – laureat I miejsca
  - Paweł Burzyński – laureat II miejsca
  - Tomasz Garbus – laureat II miejsca
  - Filip Czaplicki – laureat III miejsca
  - Bartosz Łukasiewicz – laureat III miejsca
  - Paweł Wegner – laureat III miejsca
  - Miłosz Mokwa – finalista
  - Łukasz Raszkiewicz – finalista
  - Krzysztof Wojculewicz – finalista
- Joanna Śmigielska (XIV Liceum Ogólnokształcące im. Stanisława Staszica w Warszawie)
  - Albert Citko – laureat II miejsca
  - Wojciech Jabłoński – laureat II miejsca
  - Jan Tabaszewski – laureat II miejsca
  - Tomasz Kościuszko – laureat III miejsca
  - Jakub Skorupski – laureat III miejsca
  - Michał Zawalski – laureat III miejsca
  - Marek Zbysiński – laureat III miejsca
  - Katarzyna Kowalska – laureatka III miejsca
  - Zuzanna Pilat – finalistka z wyróżnieniem
  - Paweł Solecki – finalista z wyróżnieniem
  - Jakub Bujak – finalista
  - Tomasz Grześkiewicz – finalista
  - Bartłomiej Wróblewski – finalista
  - Wiktoria Kośny – finalistka
- Cezary Urban (Zespół Szkół nr 14 we Wrocławiu)
  - Mateusz Lewko – finalista
- Grzegorz Witek (II Liceum Ogólnokształcące w Mielcu)
  - Jakub Cisło – laureat II miejsca
- Arkadiusz Wróbel (student Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego)
  - Kasper Radek – finalista z wyróżnieniem
  - Tomasz Grześkiewicz – finalista
- Robert Zieliński (Zespół Szkół Ogólnokształcących nr 2 w Tarnowie)
  - Michał Łuszczyk – laureat III miejsca
  - Mateusz Wytrwał – finalista z wyróżnieniem
- Piotr Żurkowski (Politechnika Poznańska)
  - Wojciech Mańke – finalista

Podobnie jak w ubiegłych latach w przygotowaniu jest publikacja zawierająca pełną informację o XXI Olimpiadzie Informatycznej, zadania konkursowe oraz wzorcowe rozwiązania. W publikacji tej znajdą się także zadania z międzynarodowych zawodów informatycznych.

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

Warszawa, 13 czerwca 2014 roku

Regulamin Ogólnopolskiej Olimpiady Informatycznej

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 z późniejszymi zmianami (Dz. U. 2002, nr 13, poz. 125).

Cele Olimpiady:

- stworzenie motywacji dla zainteresowania nauczycieli i uczniów informatyką,
- rozszerzanie współdziałania nauczycieli akademickich z nauczycielami szkół w kształceniu młodzieży uzdolnionej,
- stymulowanie aktywności poznawczej młodzieży informatycznie uzdolnionej,
- kształtowanie umiejętności samodzielnego zdobywania i rozszerzania wiedzy informatycznej,
- stwarzanie młodzieży możliwości szlachetnego współzawodnictwa w rozwijaniu swoich uzdolnień, a nauczycielom – warunków twórczej pracy z młodzieżą,
- 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 najlepszych uczestników olimpiady,
- przygotowywanie i publikowanie materiałów edukacyjnych dla uczniów zainteresowanych udziałem w olimpiadach i ich nauczycieli.

Rozdział I – Olimpiada i jej organizator

§1 PRAWA I OBOWIĄZKI ORGANIZATORA

(1) Organizatorem Olimpiady jest Fundacja Rozwoju Informatyki z siedzibą w Warszawie przy ul. Banacha 2. Organizator prowadzi działania związane z Olimpiadą poprzez Komitet Główny Olimpiady Informatycznej, mieszczący się w Warszawie przy ul. Nowogrodzkiej 73, tel. 22 626 83 90, fax 22 626 92 50, e-mail: `olimpiada@oi.edu.pl`, strona internetowa: `http://oi.edu.pl`.

(2) W organizacji Olimpiady Organizator 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, Ośrodkiem Edukacji Informatycznej i Zastosowań Komputerów, a także z innymi środowiskami akademickimi, zawodowymi i oświatowymi działającymi w sprawach edukacji informatycznej.

(3) Zadaniem Organizatora jest:

- przygotowanie zadań konkursowych na poszczególne etapy Olimpiady,
- realizacja Olimpiady zgodnie z postanowieniami jej regulaminu i zasad organizacji zawodów,
- organizacja komitetów okręgowych,
- zapewnienie logistyki przedsięwzięcia (dystrybucja materiałów informacyjnych oraz zadań, organizacja procesu zgłoszeń, zapewnienie odpowiednich środków do realizacji zawodów, komunikacja z uczestnikami, organizacja dystrybucji wyników poszczególnych etapów, rezerwacja sal, rezerwacja noclegów, organizacja wyżywienia finalistów, organizacja finału i uroczystego zakończenia, prowadzenie rozliczeń finansowych),
- kontakt z uczestnikami – rozwiązywanie problemów i sporów,
- działania promocyjne upowszechniające Olimpiadę.

(4) Komitet Główny Olimpiady w imieniu Organizatora ma prawo:

- do anulowania wyników poszczególnych etapów lub nakazywania powtórzenia zawodów w razie ujawnienia istotnych (naruszających regulamin Olimpiady) nieprawidłowości,
- do wykluczenia z udziału w Olimpiadzie uczestników łamiących regulamin lub zasady organizacji zawodów Olimpiady,
- prawo do reprezentowania Olimpiady na zewnątrz,
- prawo do rozstrzygania sporów i prowadzenia arbitrażu w sprawach dotyczących Olimpiady i jej uczestników,

- prawo do nawiązywania współpracy z partnerami zewnętrznymi (np. sponsorami).

Organizator ma prawo bieżącej kontroli zgodności działań Komitetu Głównego z przepisami prawa.

§2 STRUKTURA ORGANIZACYJNA OLIMPIADY

(1) Struktura organizacyjna

1. Olimpiadę przeprowadza Komitet Główny. Za organizację zawodów II stopnia w okręgach odpowiadają komitety okręgowe lub komisje powołane w tym celu przez Komitet Główny.

(2) Komitet Główny

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: przewodniczący, dwóch wiceprzewodniczących, sekretarz naukowy, kierownik Jury, kierownik techniczny i koordynator Olimpiady.
3. Komitet dokonuje zmian w swoim składzie za zgodą Organizatora.
4. Komitet powołuje i rozwiązuje komitety okręgowe Olimpiady.
5. Komitet:
   - opracowuje szczegółowe zasady organizacji zawodów, które są ogłaszane razem z treścią zadań zawodów I stopnia Olimpiady,
   - wybiera zadania na wszystkie zawody Olimpiady,
   - udziela wyjaśnień w sprawach dotyczących Olimpiady,
   - zatwierdza listy rankingowe oraz listy laureatów i wyróżnionych uczestników,
   - przyznaje uprawnienia i nagrody rzeczowe wyróżniającym się uczestnikom Olimpiady,
   - 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. W jawnych posiedzeniach Komitetu mogą brać udział przedstawiciele organizacji wspierających, jako obserwatorzy z głosem doradczym.

9. Do organizacji zawodów II stopnia w miejscowościach, których nie obejmuje żaden komitet okręgowy, Komitet powołuje komisję zawodów co najmniej miesiąc przed terminem rozpoczęcia zawodów.
10. Decyzje Komitetu we wszystkich sprawach dotyczących przebiegu i wyników zawodów są ostateczne.
11. Komitet dysponuje funduszem Olimpiady za zgodą Organizatora i za pośrednictwem koordynatora Olimpiady.
12. Komitet przyjmuje plan finansowy Olimpiady na przyszły rok na ostatnim posiedzeniu w roku poprzedzającym.
13. Komitet przyjmuje sprawozdanie finansowe z przebiegu Olimpiady na ostatnim posiedzeniu w roku, na dzień 30 listopada.
14. 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 roku.
15. Pracami Komitetu kieruje przewodniczący, a w jego zastępstwie lub z jego upoważnienia jeden z wiceprzewodniczących.
16. Przewodniczący:
    - czuwa nad całokształtem prac Komitetu,
    - zwołuje posiedzenia Komitetu,
    - przewodniczy tym posiedzeniom,
    - reprezentuje Komitet na zewnątrz,
    - rozpatruje odwołania uczestników Olimpiady osobiście lub za pośrednictwem kierownika Jury,
    - czuwa nad prawidłowością wydatków związanych z organizacją i przeprowadzeniem Olimpiady oraz zgodnością działalności Komitetu z przepisami.
17. Komitet Główny przyjął następujący tryb opracowywania zadań olimpijskich:
    - Autor zgłasza propozycję zadania, które powinno być oryginalne i nieznane, do sekretarza naukowego Olimpiady.
    - 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.
    - Wyboru zestawu zadań na zawody dokonuje Komitet Główny, spośród zadań, które zostały opracowane i uzyskały pozytywną opinię.
    - Wszyscy uczestniczący w procesie przygotowania zadania są zobowiązani do zachowania tajemnicy do czasu jego wykorzystania w zawodach lub ostatecznego odrzucenia.
18. Kierownik Jury w porozumieniu z przewodniczącym powołuje i odwołuje członków Jury Olimpiady, które jest odpowiedzialne za opracowanie i sprawdzanie zadań.

19. Kierownik techniczny odpowiada za stronę techniczną przeprowadzenia zawodów.

(3) **Komitety okręgowe**

1. Komitet okręgowy składa się z przewodniczącego, jego zastępcy, sekretarza i co najmniej dwóch członków.
2. Komitety okręgowe są powoływane przez Komitet Główny. Członków komitetów okręgowych rekomendują lokalne środowiska akademickie, zawodowe i oświatowe działające w sprawach edukacji informatycznej.
3. Zadaniem komitetów okręgowych jest organizacja zawodów II stopnia oraz popularyzacja Olimpiady.

Rozdział II – Organizacja Olimpiady

§3 UCZESTNICY OLIMPIADY

(1) Adresatami Olimpiady są uczniowie wszystkich typów szkół ponadgimnazjalnych i szkół średnich dla młodzieży, dających możliwość uzyskania matury, zainteresowani tematyką związaną z Olimpiadą.

(2) Uczestnikami Olimpiady mogą być również – za zgodą Komitetu Głównego – uczniowie szkół podstawowych, gimnazjów, zasadniczych szkół zawodowych i szkół zasadniczych, wykazujący zainteresowania, wiedzę i uzdolnienia wykraczające poza program właściwej dla siebie szkoły, pokrywające się z wymaganiami Olimpiady.

(3) By wziąć udział w Olimpiadzie, Uczestnik powinien zarejestrować się w Systemie Internetowym Olimpiady, zwanym dalej SIO, o adresie `http://oioioi.mimuw.edu.pl`.

(4) Uczestnicy zobowiązani są do:

- zapoznania się z zasadami organizacji zawodów,
- przestrzegania regulaminu i zasad organizacji zawodów,
- rozwiązywania zadań zgodnie z ich założeniami,
- informowania Komitetu Głównego o wszelkich kwestiach związanych z udziałem w Olimpiadzie – zwłaszcza w nagłych wypadkach lub w przypadku zastrzeżeń do organizacji lub przebiegu Olimpiady.

(5) Uczestnik ma prawo do:

- przystąpienia do zawodów I stopnia Olimpiady i otrzymania oceny swoich rozwiązań przekazanych Komitetowi Głównemu w sposób określony w zasadach organizacji zawodów,

- korzystania z komputera dostarczonego przez organizatorów w czasie rozwiązywania zadań w zawodach II i III stopnia, w przypadku zakwalifikowania do udziału w tych zawodach,
- zwolnienia z zajęć szkolnych na czas niezbędny do udziału w zawodach II i III stopnia, w przypadku zakwalifikowania do udziału w tych zawodach, a także do bezpłatnego zakwaterowania i wyżywienia oraz zwrotu kosztów przejazdu,
- złożenia odwołania od decyzji komitetu okręgowego lub Jury zgodnie z §6 niniejszego regulaminu.

§4 ORGANIZACJA ZAWODÓW

(1) Zawody Olimpiady mają charakter indywidualny.

(2) **Przebieg zawodów**

1. Zawody są organizowane przez Komitet Główny przy wsparciu komitetów okręgowych.
2. Zawody są trójstopniowe.
3. **Zawody I stopnia**
    1. Zawody I stopnia mają charakter otwarty i polegają na samodzielnym rozwiązywaniu przez uczestnika zadań ustalonych dla tych zawodów oraz przekazaniu rozwiązań w podanym terminie; sposób przekazania określony jest w zasadach organizacji zawodów danej edycji Olimpiady.
    2. Zawody I stopnia są przeprowadzane zdalnie przez Internet.
    3. Liczbę uczestników kwalifikowanych do zawodów II stopnia ustala Komitet Główny i podaje ją w zasadach organizacji zawodów. Komitet Główny kwalifikuje do zawodów II stopnia odpowiednią liczbę uczestników, których rozwiązania zadań I stopnia zostaną ocenione najwyżej, spośród uczestników, którzy uzyskali co najmniej 50% punktów możliwych do zdobycia w zawodach I stopnia.
    4. Komitet Główny może zmienić podaną w zasadach liczbę uczestników zakwalifikowanych do zawodów II stopnia co najwyżej o 25%. Komitet Główny ma prawo zakwalifikować do zawodów II stopnia uczestników, którzy zdobyli mniej niż 50% ogólnej liczby punktów, w kolejności zgodnej z listą rankingową, jeśli uzna, że poziom zakwalifikowanych jest wystarczająco wysoki.
4. **Zawody II stopnia**
    1. Zawody II stopnia są organizowane przez komitety okręgowe Olimpiady lub komisje zawodów powołane przez Komitet Główny i koordynowane przez Komitet Główny.
    2. Zawody II stopnia polegają na samodzielnym rozwiązywaniu zadań przygotowanych centralnie przez Komitet Główny. Zawody te odbywają się w ciągu dwóch sesji, przeprowadzanych w różnych dniach, w warunkach kontrolowanej samodzielności.

3. Rozwiązywanie zadań w zawodach II stopnia jest poprzedzone jednodniową sesją próbną umożliwiającą zapoznanie się uczestników z warunkami organizacyjnymi i technicznymi Olimpiady.
4. W czasie trwania zawodów nie można korzystać z żadnych książek ani innych pomocy takich jak: dyski, kalkulatory, notatki itp. Nie wolno mieć w tym czasie telefonu komórkowego ani innych własnych urządzeń elektronicznych.
5. Każdy uczestnik zawodów II stopnia musi mieć ze sobą legitymację szkolną.
6. Do zawodów III stopnia zostanie zakwalifikowanych 80 uczestników, których rozwiązania zadań II stopnia zostaną ocenione najwyżej, spośród uczestników, którzy uzyskali co najmniej 50% punktów możliwych do zdobycia w zawodach II stopnia.
7. Komitet Główny może zmienić liczbę uczestników zakwalifikowanych do zawodów III stopnia co najwyżej o 25%. Komitet Główny ma prawo zakwalifikować do zawodów III stopnia uczestników zawodów II stopnia, którzy zdobyli mniej niż 50% ogólnej liczby punktów, w kolejności zgodnej z listą rankingową, jeśli uzna, że poziom zakwalifikowanych jest wystarczająco wysoki.
8. Uczestnik zawodów II stopnia zakwalifikowany do zawodów III stopnia uzyskuje tytuł finalisty Olimpiady Informatycznej.

5. **Zawody III stopnia**

    1. Zawody III stopnia polegają na samodzielnym rozwiązywaniu zadań. Zawody te odbywają się w ciągu dwóch sesji, przeprowadzanych w różnych dniach, w warunkach kontrolowanej samodzielności.
    2. Rozwiązywanie zadań w zawodach III stopnia jest poprzedzone jednodniową sesją próbną umożliwiającą zapoznanie się uczestników z warunkami organizacyjnymi i technicznymi Olimpiady.
    3. W czasie trwania zawodów nie można korzystać z żadnych książek ani innych pomocy takich jak: dyski, kalkulatory, notatki itp. Nie wolno mieć w tym czasie telefonu komórkowego ani innych własnych urządzeń elektronicznych.
    4. Każdy uczestnik zawodów III stopnia musi mieć ze sobą legitymację szkolną.
    5. Na podstawie analizy rozwiązań zadań w zawodach III stopnia i listy rankingowej Komitet Główny przyznaje tytuły laureatów Olimpiady Informatycznej: I, II i III miejsca. Liczba laureatów nie przekracza połowy uczestników zawodów III stopnia. Zasady przyznawania tytułów laureata reguluje §8.1.
    6. W przypadku bardzo wysokiego poziomu zawodów III stopnia Komitet Główny może dodatkowo wyróżnić uczestników niebędących laureatami.
    7. Zwycięzcą Olimpiady Informatycznej zostaje każda osoba, która osiągnęła najlepszy wynik w zawodach III stopnia.

6. **Postanowienia ogólne**

    1. 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 organizacji zawodów.
    2. 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.
    3. Komitet Główny zastrzega sobie prawo do opublikowania rozwiązań zawodników, którzy zostali zakwalifikowani do następnego etapu, zostali wyróżnieni lub otrzymali tytuł laureata.
    4. Komitet Główny zawiadamia uczestnika oraz dyrektora jego szkoły o zakwalifikowaniu do zawodów stopnia II i III, podając jednocześnie miejsce i termin zawodów.

§5 PRZEPISY SZCZEGÓŁOWE

(1) Organizator dokona wszelkich starań, aby w miarę możliwości w danych warunkach organizować zawody w taki sposób i w takich miejscach, by nie wykluczały udziału osób niepełnosprawnych. W tym celu komitety okręgowe i Komitet Główny będą dążyły do organizacji zawodów w pomieszczeniach łatwo dostępnych oraz organizacji noclegu w miejscu łatwo dostępnym.

(2) Zawody Olimpiady Informatycznej odbywają się wyłącznie w terminie ustalonym przez Komitet Główny, bez możliwości powtarzania.

(3) Organizator dołoży starań i zrobi wszystko, co w danych warunkach jest możliwe, by umożliwić udział w olimpiadzie uczestnikowi, który równolegle bierze udział w innej olimpiadzie, a ich terminy się pokrywają.

(4) Rozwiązania zespołowe, niesamodzielne, niezgodne z zasadami organizacji zawodów 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.

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

(6) W szczególnie rażących wypadkach łamania regulaminu lub zasad organizacji zawodów, Komitet Główny może zdyskwalifikować zawodnika.

§6 TRYB ODWOŁAWCZY

(1) Po zakończeniu zawodów każdego stopnia każdy zawodnik będzie mógł zapoznać się ze wstępną oceną swojej pracy oraz zgłosić uwagi do tej oceny.

(2) Reklamacji nie podlega dobór testów, limitów czasowych, kompilatorów i sposobu oceny.

(3) Sposoby i terminy składania reklamacji są określone w Zasadach organizacji zawodów.

(4) Reklamacje rozpoznaje Komitet Główny. Decyzje podjęte przez Komitet Główny są ostateczne. Komitet Główny powiadamia uczestnika o wyniku rozpatrzenia reklamacji.

§7 REJESTRACJA PRZEBIEGU ZAWODÓW

(1) Komitet Główny prowadzi archiwum akt Olimpiady, przechowując w nim między innymi:

- zadania Olimpiady,
- rozwiązania zadań Olimpiady przez okres 5 lat,
- rejestr wydanych zaświadczeń i dyplomów laureatów,
- listy laureatów i ich nauczycieli,
- dokumentację statystyczną i finansową.

Rozdział III – Uprawnienia i nagrody

§8 UPRAWNIENIA I NAGRODY

(1) W klasyfikacji wyników uczestników Olimpiady stosuje się następujące terminy:

- finalista to zawodnik, który został zakwalifikowany do zawodów III stopnia,
- laureat to uczestnik zawodów III stopnia sklasyfikowany w pierwszej połowie uczestników tych zawodów i którego dokonania Komitet Główny uzna za zdecydowanie wyróżniające się wśród wyników finalistów. Laureaci dzielą się na laureatów I, II i III miejsca.

(2) Uprawnienia laureatów i finalistów określa rozporządzenie Ministra Edukacji Narodowej z dnia 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. 2007, nr 83, poz. 562, z późn. zm.).

(3) Potwierdzeniem uzyskania uprawnień oraz statusu laureata i finalisty jest zaświadczenie, którego wzór stanowi załącznik do rozporządzenia Ministra Edukacji Narodowej i Sportu z dnia 29 stycznia 2002 r. w sprawie organizacji oraz sposobu przeprowadzania konkursów, turniejów i olimpiad (Dz. U. 2002, nr 13, poz. 125, z późn. zm.). Komitet Główny prowadzi rejestr wydanych zaświadczeń.

(4) Komitet Główny nagradza laureatów I, II i III miejsca medalami, odpowiednio, złotymi, srebrnymi i brązowymi.

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

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

(7) 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.

(8) 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. 2005, nr 164, poz. 1365).

Rozdział IV – Olimpiada międzynarodowa

§9 UDZIAŁ W OLIMPIADZIE MIĘDZYNARODOWEJ

(1) Komitet Główny ustala skład reprezentacji Polski na Międzynarodową Olimpiadę Informatyczną oraz inne zawody międzynarodowe na podstawie wyników Olimpiady oraz regulaminów Międzynarodowej Olimpiady i tych zawodów.

(2) Organizator pokrywa koszty udziału zawodników w Międzynarodowej Olimpiadzie Informatycznej i zawodach międzynarodowych.

Rozdział V – Postanowienia końcowe

§10 POSTANOWIENIA KOŃCOWE

(1) Decyzje w sprawach nieobjętych powyższym regulaminem podejmuje Komitet Główny, ewentualnie jeśli sprawa tego wymaga w porozumieniu z Organizatorem.

(2) Komitet Główny rozsyła do szkół wymienionych w §3.1 oraz kuratoriów oświaty i koordynatorów edukacji informatycznej informację o rozpoczęciu danej edycji Olimpiady.

(3) Dyrektorzy szkół mają obowiązek dopilnowania, aby informacje dotyczące Olimpiady zostały przekazane uczniom.

(4) Komitet Główny zatwierdza sprawozdanie merytoryczne i przedstawia je Organizatorowi celem przedłożenia Ministerstwu Edukacji Narodowej.

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

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

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

(8) Niniejszy regulamin może zostać zmieniony przez Komitet Główny tylko przed rozpoczęciem kolejnej edycji zawodów Olimpiady.

§11 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.

Zasady organizacji zawodów XXI Olimpiady Informatycznej w roku szkolnym 2013/2014

§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 przeprowadzania konkursów, turniejów i olimpiad (Dz. U. 2002, nr 13, poz. 125, z późn. zm.). Organizatorem Olimpiady 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, Ośrodkiem Edukacji Informatycznej i Zastosowań Komputerów, a także z innymi środowiskami akademickimi, zawodowymi i oświatowymi działającymi w sprawach edukacji informatycznej.

§2 ORGANIZACJA OLIMPIADY

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

(2) Olimpiada Informatyczna jest trójstopniowa.

(3) W Olimpiadzie Informatycznej mogą brać indywidualnie udział uczniowie wszystkich typów szkół ponadgimnazjalnych. 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 300 uczestników, których rozwiązania zadań I stopnia zostaną ocenione najwyżej, spośród uczestników, którzy uzyskali co najmniej 50% punktów możliwych do zdobycia w zawodach I stopnia. Do zawodów III stopnia zostanie zakwalifikowanych 80 uczestników, których rozwiązania zadań II stopnia zostaną ocenione najwyżej, spośród uczestników, którzy uzyskali co najmniej 50% punktów możliwych do zdobycia w zawodach II stopnia.

(8) Komitet Główny może zmienić podane liczby zakwalifikowanych uczestników co najwyżej o 25%. Komitet Główny ma prawo zakwalifikować do zawodów wyższego stopnia uczestników zawodów niższego stopnia, którzy zdobyli mniej niż 50% ogólnej liczby punktów, w kolejności zgodnej z listą rankingową, jeśli uzna, że poziom zakwalifikowanych jest wystarczająco wysoki.

(9) 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.

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

(11) Terminarz zawodów:

- **zawody I stopnia** – 7 października – 4 listopada 2013 roku
  ogłoszenie wyników w witrynie Olimpiady – 29 listopada 2013 roku godz. 20.00
- **zawody II stopnia** – 11–13 lutego 2014 roku
  ogłoszenie wyników w witrynie Olimpiady – 21 lutego 2014 roku godz. 20.00
- **zawody III stopnia** – 1–4 kwietnia 2014 roku

§3 ROZWIĄZANIA ZADAŃ

(1) Jeśli rozwiązaniem zadania jest program, ocena rozwiązania jest określana na podstawie wyników testowania programu i uwzględnia poprawność oraz efektywność metody rozwiązania użytej w programie. Jeśli rozwiązaniem zadania jest plik z danymi, wówczas ocenia się poprawność danych.

(2) Rozwiązania zespołowe, niesamodzielne, niezgodne z zasadami organizacji zawodów 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://oioioi.mimuw.edu.pl`, do 4 listopada 2013 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 4 listopada 2013 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 18 listopada 2013 roku poprzez SIO każdy zawodnik będzie mógł zapoznać się ze wstępną oceną swojej pracy.

(10) Do 22 listopada 2013 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 22 listopada 2013 roku nie będą rozpatrywane.

§5 ZAWODY II I III STOPNIA

(1) Zawody II i III stopnia 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. Nie wolno mieć w tym czasie telefonu komórkowego ani innych własnych urządzeń elektronicznych.

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

§6 UPRAWNIENIA I NAGRODY

(1) Każdy zawodnik, który został zakwalifikowany do zawodów III stopnia, zostaje finalistą Olimpiady. Laureatem Olimpiady zostaje uczestnik zawodów III stopnia sklasyfikowany w pierwszej połowie uczestników tych zawodów, którego dokonania Komitet Główny uzna za zdecydowanie wyróżniające się wśród wyników finalistów. Laureaci dzielą się na laureatów I, II i III miejsca. W przypadku bardzo wysokiego poziomu zawodów III stopnia Komitet Główny może dodatkowo wyróżnić uczestników niebędących laureatami.

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

(3) 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.

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

(5) 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 roku „Prawo o szkolnictwie wyższym", na zasadach zawartych w tych uchwałach (Dz. U. 2005, nr 164, poz. 1365).

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

(7) Komitet Główny ustala skład reprezentacji Polski na XXVI Międzynarodową Olimpiadę Informatyczną w 2014 roku oraz inne zawody międzynarodowe na podstawie wyników Olimpiady oraz regulaminów Międzynarodowej Olimpiady i tych zawodów.

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

(9) Wyznaczeni przez Komitet Główny reprezentanci Polski na olimpiady międzynarodowe zostaną zaproszeni do nieodpłatnego udziału w XV Obozie Naukowo-Treningowym im. Antoniego Kreczmara, który odbędzie się w czasie wakacji 2014 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.

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

§7 PRZEPISY KOŃCOWE

(1) Komitet Główny zawiadamia wszystkich uczestników zawodów I i II stopnia o ich wynikach poprzez SIO. Wszyscy uczestnicy zawodów I stopnia 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 XXI 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) Każdy uczestnik zawodów II i III stopnia musi mieć ze sobą legitymację szkolną.

(4) 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.

(5) Komisja Regulaminowa powołana przez komitet okręgowy lub Komitet Główny czuwa nad prawidłowością przebiegu zawodów i pilnuje przestrzegania Regulaminu Olimpiady i Zasad organizacji zawodów.

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

(7) 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.

(8) 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.

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

(10) Podczas każdej sesji:

1. W trakcie pierwszych 60 minut nie wolno opuszczać przydzielonej sali zawodów. Zawodnicy spóźnieni więcej niż godzinę nie będą w tym dniu dopuszczeni do zawodów.
2. 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.

3. W SIO umieszczane będą 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.
4. Jakikolwiek inny sposób komunikowania się z członkami Jury co do treści i sposobów rozwiązywania zadań jest niedopuszczalny.
5. Komunikowanie się z innymi uczestnikami Olimpiady (np. ustnie, telefonicznie lub poprzez sieć) w czasie przeznaczonym na rozwiązywanie zadań jest zabronione pod rygorem dyskwalifikacji.
6. Każdy zawodnik ma prawo drukować wyniki swojej pracy w sposób opisany w Ustaleniach technicznych.
7. 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.
8. 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”).
9. Podczas zawodów III stopnia, w przypadku niektórych zadań, wskazanych przez Komitet Główny, zawodnicy będą mogli poznać wynik punktowy swoich pięciu wybranych zgłoszeń. Przez ostatnie 30 minut zawodów ta opcja nie będzie dostępna.
10. Rozwiązanie każdego zadania można zgłosić co najwyżej 10 razy. Spośród tych zgłoszeń oceniane jest jedynie najpóźniejsze poprawnie kompilujące się rozwiązanie.

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

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

(13) 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.

(14) Od 13 lutego 2014 roku od godz. 20.00 do 17 lutego 2014 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

Jacek Tomasiewicz Treść zadania | **Bartosz Tarnawski** Opracowanie | **Michał Adamczyk** Program

Dostępna pamięć: 128 MB. *OI, etap I, 7.10–4.11.2013*

Bar sałatkowy

Bajtotka wybrała się do baru sałatkowego. W barze na ladzie leży n owoców ułożonych w jednym rzędzie. Są to pomarańcze i jabłka. Bajtotka może wybrać pewien spójny fragment rzędu owoców, z którego zostanie przygotowana sałatka owocowa.

Wiadomo, że owoce z wybranego fragmentu będą dodawane do sałatki kolejno od lewej do prawej albo kolejno od prawej do lewej. Bajtotka uwielbia pomarańcze i ma dodatkowe wymaganie, aby w trakcie robienia sałatki liczba dodanych już pomarańczy nigdy nie była mniejsza od liczby dodanych jabłek, niezależnie od tego, czy owoce będą dodawane od lewej do prawej, czy odwrotnie. Pomóż Bajtotce i napisz program, który znajdzie jak najdłuższy fragment rzędu owoców spełniający jej wymagania.

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą n *(*$1 \leqslant n \leqslant 1\,000\,000$*), oznaczającą liczbę owoców. Kolejny wiersz zawiera napis złożony z* n *liter* $a_1 a_2 \ldots a_n$ *(*$a_i \in \{\texttt{p}, \texttt{j}\}$*). Jeśli* $a_i = \texttt{p}$*, to* i*-tym owocem w rzędzie jest pomarańcza, w przeciwnym przypadku jest to jabłko.*

Możesz założyć, że w testach wartych 50% punktów zachodzi $n \leqslant 10\,000$*, a w testach wartych 20% punktów zachodzi* $n \leqslant 1000$*.*

Wyjście

Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą równą liczbie owoców w najdłuższym spójnym fragmencie rzędu, który spełnia wymagania Bajtotki. Jeśli sałatka dla Bajtotki nie może zostać przyrządzona, prawidłowym wynikiem jest 0.

Przykład

Dla danych wejściowych:

```
6
jpjppj
```

poprawnym wynikiem jest:

```
4
```

Wyjaśnienie do przykładu: *Po odrzuceniu skrajnie lewego i skrajnie prawego jabłka Bajtotka może zamówić sałatkę z pozostałych owoców.*

Rozwiązanie

Analiza problemu

Będziemy rozważać słowa składające się wyłącznie z liter `p` oraz `j`. Długość słowa w oznaczamy przez $|w|$.

Słowo nazwiemy *legalnym*, jeśli każdy jego prefiks i każdy jego sufiks zawiera co najmniej tyle liter `p` co liter `j`. Mając dane słowo $w = a_1 a_2 \dots a_n$, chcemy znaleźć jego najdłuższe legalne podsłowo.

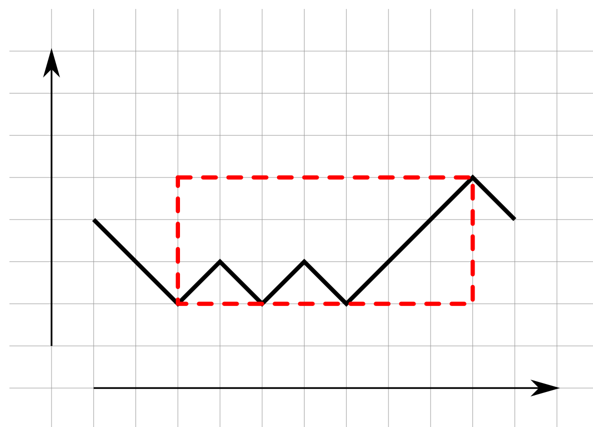
Trochę wygodniej będzie nam myśleć w kategoriach sum prefiksowych. Literze `p` przypiszmy wartość 1, a literze `j` wartość -1. Niech $pre[i]$ $(0 \leqslant i \leqslant n)$ będzie sumą wartości i pierwszych liter słowa w (w szczególności $pre[0] = 0$).

Możemy teraz sformułować równoważny warunek na legalność słowa:

Lemat 1. Podsłowo $a_i a_{i+1} \dots a_j$ słowa w jest legalne wtedy i tylko wtedy, gdy dla każdego $i-1 \leqslant k \leqslant j$ zachodzi

$$pre[i-1] \leqslant pre[k] \leqslant pre[j].$$

Pomyślmy teraz o tablicy pre jako o funkcji i przyjrzyjmy się jej wykresowi. Ta funkcja jest zdefiniowana wprawdzie tylko dla $0, 1, \dots, n$, ale żeby nasza wizualizacja była ładniejsza, możemy uzupełnić ją funkcjami liniowymi na każdym przedziale $[i-1, i]$, dla $1 \leqslant i \leqslant n$. Załóżmy, że $a_i a_{i+1} \dots a_j$ jest legalnym podsłowem. Wtedy wykres pre na przedziale $[i-1, j]$ jest w całości zawarty w prostokącie o lewym dolnym rogu $(i-1, pre[i-1])$ oraz prawym górnym $(j, pre[j])$.

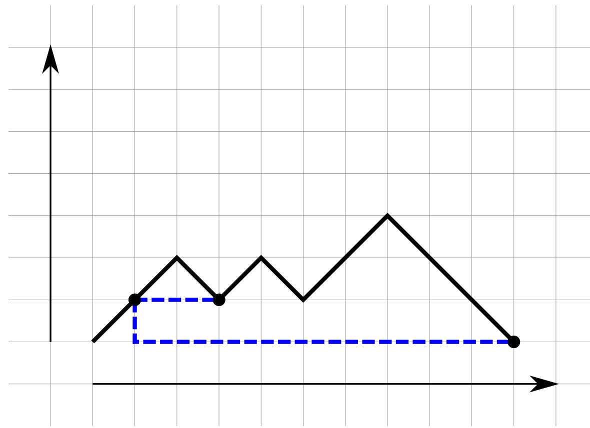


Rys. 1: Legalne podsłowo `pjpjppp` słowa `jjpjpjpppj` i odpowiadający mu prostokąt.

Rozwiązanie wzorcowe $O(n)$

Dla każdego $0 \leqslant i \leqslant n$ chcemy obliczyć i stablicować największe takie $j = end[i]$, że podsłowo $a_{i+1}a_{i+2}\ldots a_j$ jest legalne. Jeśli $a_{i+1} = \mathtt{j}$, to $end[i] = i$, bo żadne słowo zaczynające się od j nie może być legalne. Przyjmijmy, że $end[n] = n$.

Tablica *end* od razu da nam wynik, ale do jej obliczenia będziemy potrzebowali kilku innych tablic. Dla $0 \leqslant i \leqslant n$ niech $equal[i]$ będzie najmniejszym takim $j > i$, że $pre[i] = pre[j]$, a jeśli dla danego i takie j nie istnieje, to powiedzmy, że $equal[i] = n+1$. Podobnie definiujemy tablicę *below*, tym razem szukając najmniejszego $j > i$, aby $pre[j] < pre[i]$. Przykładowe wartości obu tablic dla pewnego i zostały przedstawione na rysunku.



Rys. 2: Zaznaczono punkty wykresu dla argumentów i, $equal[i]$ oraz $below[i]$.

Tablicę *equal* wyznaczamy w czasie liniowym, przeglądając kolejne pozycje od prawej do lewej i zapamiętując numer ostatniej pozycji o danej wartości *pre*. Mając tablicę *equal*, można łatwo wyznaczyć tablicę *below*; zresztą w tym rozwiązaniu wykorzystamy ją tylko na potrzeby dowodu.

Chcemy zdefiniować teraz pewien porządek na zbiorze $0, 1, \ldots, n$. Powiemy, że $i \prec j$, jeżeli $(pre[i], i) < (pre[j], j)$ w porządku leksykograficznym, tj. $pre[i] < pre[j]$ lub ($pre[i] = pre[j]$ oraz $i < j$). Okazuje się, że porządek ten ma bezpośredni związek z wartościami tablicy *end*:

Lemat 2. $end[i]$ jest równy tej liczbie spośród $i, i+1, \ldots, below[i] - 1$, która jest największa w porządku „$\prec$”.

Jego wyprowadzenie z lematu 1 pozostawiamy jako ćwiczenie.

Możemy teraz skonstruować tablicę *end*. Ponownie przeglądamy pozycje od prawej do lewej. Ustalmy $0 \leqslant i \leqslant n-1$. Jeżeli $a_{i+1} = \mathtt{j}$, to ustawiamy po prostu $end[i] = i$. Odtąd zakładamy, że $a_{i+1} = \mathtt{p}$. Rozważmy kilka przypadków.

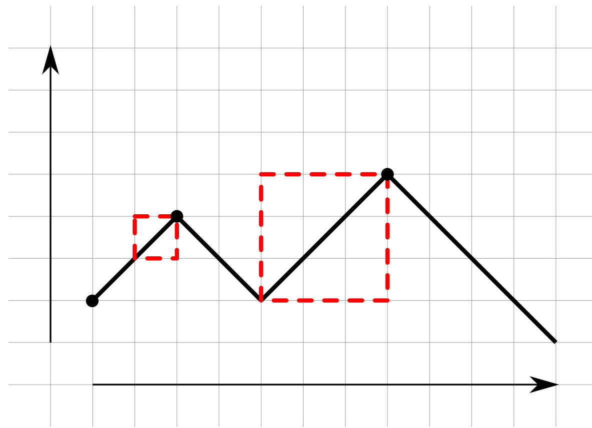
1. Jeśli $i = n-1$, to ustawiamy $end[i] = i + 1$.

2. Jeśli zaś $i < n-1$ oraz $equal[i] = n+1$, to powinniśmy ustawić $end[i] = end[i+1]$ (ćwiczenie).

3. Pozostaje przypadek $i < n - 1$ oraz $equal[i] \neq n + 1$ (nadal zakładamy, że $a_{i+1} = \mathtt{p}$).

   Niech $a = end[i + 1]$, $b = end[equal[i]]$. Jeśli $a \prec b$, to powinniśmy ustawić $end[i] = b$, a w przeciwnym przypadku $end[i] = a$.

Udowodnimy teraz poprawność obliczenia wyniku w punkcie 3. Wcześniej warto spojrzeć na rysunek przedstawiający przykładową konfigurację, w której $end[i + 1] \prec end[equal[i]]$.



Rys. 3: Zaznaczono punkty wykresu dla argumentów i, $end[i + 1]$ oraz $end[equal[i]]$.

Dzięki lematowi 2 wiemy, że szukamy liczby spośród $i, i+1, \ldots, below[i]-1$, która jest największa według porządku „$\prec$” – nazwijmy ją j. Nie jest to na pewno i, bo $pre[i + 1] > pre[i]$ (założyliśmy, że $a_{i+1} = \mathtt{p}$).

Może być zatem tak, że

- $i + 1 \leqslant j \leqslant equal[i] - 1$, wtedy $j = end[i + 1]$.

  Uzasadnienie: Zauważmy, że $equal[i] = below[i + 1]$. Skoro j jest największe w porządku „$\prec$” na przedziale $i, \ldots, below[i]-1$, to tym bardziej jest największe na przedziale $i + 1, \ldots, below[i + 1] - 1$. Czyli rzeczywiście $j = end[i + 1]$.

- $equal[i] \leqslant j \leqslant below[i] - 1$, wtedy $j = end[equal[i]]$.

  Uzasadnienie: Tym razem wystarczy zauważyć, że $below[i] = below[equal[i]]$. Ponownie, skoro j jest największe według „$\prec$” na przedziale $i, \ldots, below[i] - 1$, to jest największe także na $equal[i], \ldots, below[equal[i]] - 1$.

To już koniec opisu rozwiązania wzorcowego. Zostało ono zaimplementowane w plikach `bar.cpp` i `bar1.pas`.

Rozwiązanie alternatywne $O(n \log n)$

Tablica *below* w poprzednim rozwiązaniu przydawała się tylko w dowodzie. Tym razem wykorzystamy ją także w samym programie. Oprócz niej, będziemy potrzebowali drzewa przedziałowego podającego maksimum (według porządku „$\prec$") na zadanym przedziale.

Dla każdego i znajdujemy $end[i]$, odpytując nasze drzewo o największą liczbę w porządku „$\prec$" na przedziale $[i, below[i] - 1]$.

Takie rozwiązanie zostało zaimplementowane w pliku `bar3.pas`. Rozwiązanie `bar2.cpp` ma tę samą złożoność, ale opiera się na trochę innym pomyśle i korzysta ze struktury danych `set` z STL-a. Oba rozwiązania otrzymywały 100 punktów.

Rozwiązanie wolniejsze $O(n\sqrt{n})$

To eleganckie rozwiązanie zostało zaproponowane przez zawodników. Dosyć łatwo udowodnić jego poprawność, trochę trudniej jest z określeniem złożoności czasowej. Będziemy wielokrotnie przeglądać nasze słowo z lewa na prawo i z prawa na lewo, dzieląc je na coraz krótsze fragmenty.

Dla uproszczenia zapisu podsłowo w zaczynające się od i-tej litery, a kończące na j-tej oznaczymy przez $w[i..j]$. Niech $v = w[i..j]$. Powiemy, że v jest *lewostronnie legalne*, jeżeli dla każdego $i \leqslant k \leqslant j$ zachodzi $pre[i-1] \leqslant pre[k]$. Jeżeli v nie jest lewostronnie legalne, to najmniejsze takie $k \geqslant i$, że $pre[k] < pre[i-1]$, nazwiemy *lewostronnym uskokiem* v.

Analogicznie definiujemy słowo *prawostronnie legalne* (dla każdego $i \leqslant k \leqslant j$ $pre[k-1] \leqslant pre[j]$) oraz *prawostronny uskok* słowa, które takie nie jest (największe takie $k \leqslant j$, że $pre[k-1] > pre[j]$). Zgodnie z naszą poprzednią definicją, słowo lewostronnie i prawostronnie legalne jest po prostu legalne.

Skorzystamy z prostej obserwacji:

Lemat 3. Jeżeli k jest lewostronnym lub prawostronnym uskokiem słowa $v = w[i..j]$, zaś $u = w[i'..j']$ jest legalnym podsłowem v, to $j' < k$ lub $i' > k$.

Możemy teraz przejść do opisu algorytmu. Chcemy napisać funkcję $longest(i, j)$, która wyznaczy długość najdłuższego legalnego podsłowa słowa $w[i+1..j]$. Wtedy ostatecznym wynikiem będzie $longest(0, n)$.

Niech $v = w[i+1..j]$. Aby obliczyć $longest(i, j)$, wykonamy co następuje:

1. Jeśli v nie jest lewostronnie legalne, to znajdujemy najdłuższy taki podciąg $p_0, p_1, \ldots, p_r$, że $i - 1 = p_0 < p_1 < \ldots < p_r \leqslant j$ oraz p_{x+1} jest lewostronnym uskokiem słowa $w[p_x + 1..j]$ dla $x = 0, \ldots, r-1$. Na mocy lematu 3:

   $$longest(i, j) = \max(longest(i, p_1 - 1), longest(p_1, p_2 - 1), \ldots, longest(p_r, j)).$$

   Warto przy tym zauważyć, że każde ze słów $w[(p_x + 1)..(p_{x+1} - 1)]$ dla $x = 0, \ldots, r$ jest lewostronnie legalne (przyjmujemy $p_{r+1} = j + 1$).

2. Jeśli v jest lewostronnie, ale nie prawostronnie legalne, to robimy to samo co powyżej, tylko „w drugą stronę", wywołując się rekurencyjnie dla ciągu prawostronnie legalnych podsłów.

3. Jeśli v jest zarówno lewostronnie, jak i prawostronnie legalne, to wynik to $j - i$.

Poprawność algorytmu wynika bezpośrednio z lematu 3. Zastanówmy się, jaki jest jego czas działania. Widać, że na każdym poziomie rekurencji tracimy $O(n)$ czasu. Trudniej jest uzasadnić, że maksymalna liczba poziomów to $O(\sqrt{n})$. Przytoczony poniżej dowód jest dosyć złożony. Czytelnik, który nie jest szczególnie zainteresowany szacowaniem złożoności czasowej naszego algorytmu, może go śmiało pominąć.

Ustalmy pewien ciąg podsłów

$$w_0 = w[l_0 + 1, r_0], w_1 = w[l_1 + 1, r_1], \ldots, w_k = w[l_k + 1, r_k]$$

o następujących własnościach:

- $w_0 = w$,
- w_{x+1} jest podsłowem w_x i, co więcej, chcąc obliczyć wartość $longest(l_x, r_x)$, musieliśmy wywołać rekurencyjnie $longest(l_{x+1}, r_{x+1})$,
- żadne ze słów w_i nie jest legalne.

Możemy założyć, że słowo w nie było lewostronnie legalne (dowód dla przeciwnego przypadku różni się tylko szczegółami). Wtedy wykonaliśmy dla niego operację typu (1), czyli słowo w_1 było już lewostronnie legalne. Dla niego musieliśmy wykonać operację typu (2) i tak dalej.

Kontynuując to rozumowanie, dochodzimy do wniosku, że słowa w_{2l-1} są lewostronnie legalne, a w_{2l} – prawostronnie legalne dla $l \geqslant 1$.

Załóżmy też bez straty ogólności, że $k = 2m$. Skoncentrujmy uwagę na ciągu $l_1, l_2, \ldots, l_k$. Chcielibyśmy udowodnić kilka nierówności:

1. $pre[l_{2x-1}] < pre[l_{2x+1}]$ dla $1 \leqslant x < m$,
2. $pre[l_{2x}] > pre[l_{2x+2}]$ dla $1 \leqslant x < m$,
3. $pre[l_{2m-1}] < pre[l_{2m}]$.

Zanim przejdziemy do ich dowodu, zobaczmy, w jaki sposób wynika z nich ograniczenie $k = O(\sqrt{n})$. Niech $1 \leqslant x \leqslant m$. Mamy wtedy

$$pre[l_{2x-1}] < pre[l_{2x+1}] < \ldots < pre[l_{2m-1}] < pre[l_{2m}] < pre[l_{2m-2}] < \ldots < pre[l_{2x}].$$

Zatem $pre[l_{2x}] - pre[l_{2x-1}] \geqslant 2(m - x) + 1$. Zauważmy jeszcze, że dla każdych $0 \leqslant a, b \leqslant n$ mamy $|a - b| \geqslant |pre[a] - pre[b]|$, czyli $l_{2x} - l_{2x-1} \geqslant 2(m - x) + 1$.

Wiadomo, że $l_1 \leqslant l_2 \leqslant \ldots \leqslant l_k = l_{2m}$. Wobec tego

$$l_{2m} - l_1 \geqslant (l_{2m} - l_{2m-1}) + (l_{2m-1} - l_{2m-2}) + \ldots + (l_2 - l_1) \geqslant$$

$$\geqslant (l_{2m} - l_{2m-1}) + (l_{2m-2} - l_{2m-3}) + \ldots + (l_2 - l_1) \geqslant$$

$$\geqslant (2 \cdot 0 + 1) + (2 \cdot 1 + 1) + \ldots + (2 \cdot (m - 1) + 1) = m^2.$$

Wiemy jednak, że $l_k - l_1 \leqslant n$, stąd otrzymujemy szukane ograniczenie $m^2 \leqslant n$. Mamy więc nie więcej niż $O(\sqrt{n})$ poziomów.

Trzeba jeszcze tylko udowodnić nierówności (1), (2) i (3). Pozostawimy to jako ćwiczenie dla Czytelnika, podając tylko szkic dowodu w pierwszym przypadku.

Przypuśćmy, że $pre[l_{2x-1}] \geqslant pre[l_{2x+1}]$ dla pewnego $1 \leqslant x < m$. Słowo w_{2x-1} było lewostronnie legalne. Stąd $pre[l_{2x+1}] \geqslant pre[l_{2x-1}]$, czyli musi zachodzić $pre[l_{2x+1}] = pre[l_{2x-1}]$. Pokażemy, że wtedy słowo w_{2x+1} jest legalne, wbrew naszemu założeniu.

Skoro w_{2x-1} jest lewostronnie legalne, to dla każdego $l_{2x-1} \leqslant y \leqslant r_{2x-1}$ zachodzi $pre[y] \geqslant pre[l_{2x-1}] = pre[l_{2x+1}]$. Z kolei w_{2x} jest prawostronnie legalne, więc dla $l_{2x} \leqslant y \leqslant r_{2x}$ mamy $pre[y] \leqslant pre[r_{2x}]$. Wreszcie, wiemy, że $l_{2x} \leqslant l_{2x+1} \leqslant r_{2x}$. Po chwili zastanowienia dochodzimy do wniosku, że musi zachodzić $r_{2x+1} = r_{2x}$, więc słowo w_{2x+1} jest legalne. Otrzymana sprzeczność kończy dowód nierówności (1).

Na każdy z co najwyżej $O(\sqrt{n})$ poziomów poświęcamy $O(n)$ czasu, czyli ostateczna złożoność to $O(n\sqrt{n})$. To ograniczenie jest też optymalne – wartościowym ćwiczeniem może być samodzielne skonstruowanie wejścia, dla którego program musi poświęcić czas rzędu $\Omega(n\sqrt{n})$. Przykład takiego wejścia znajduje się w sekcji poświęconej testom.

Implementacja powyższego algorytmu znajduje się w pliku `bars10.cpp`. Taki program mógł zdobywać nawet do 100 punktów.

Rozwiązania niepoprawne

Zawodnicy zaproponowali dwa ciekawe typy rozwiązań niepoprawnych. Każde z nich opiera się na jakiejś „obserwacji”, która w ogólności jest niepoprawna, ale potrzeba danych o dosyć specyficznej strukturze, aby ją obalić.

1. „Jeżeli i jest lewym końcem najdłuższego legalnego podsłowa, to $pre[i-1]$ jest najmniejsze spośród liczb $pre[0], pre[1], \ldots, pre[i-1]$”.

   To rozwiązanie zbyt pochopnie odrzuca kandydatów na legalne podsłowo i może wypisać zbyt mały wynik. Zostało zaimplementowane w pliku `barb8.cpp`.

2. „Wystarczy znaleźć najdłuższe takie podsłowo $a_i a_{i+1} \ldots a_j$, aby dla każdego k $(i \leqslant k \leqslant j)$ istniały takie x, y $(x < k \leqslant y)$, że $pre[x] \leqslant pre[k]$ oraz $pre[k-1] \leqslant pre[y]$”.

   W oryginalnej wersji zadania mamy podobny warunek, ale wymagamy przy tym, aby $x = i - 1$ oraz $y = j$. Jeżeli pozbędziemy się tego założenia, to możemy znaleźć fałszywego kandydata na legalne podsłowo i wypisać zbyt dużą liczbę. Takie rozwiązanie zostało zaimplementowane w pliku `barb9.cpp`.

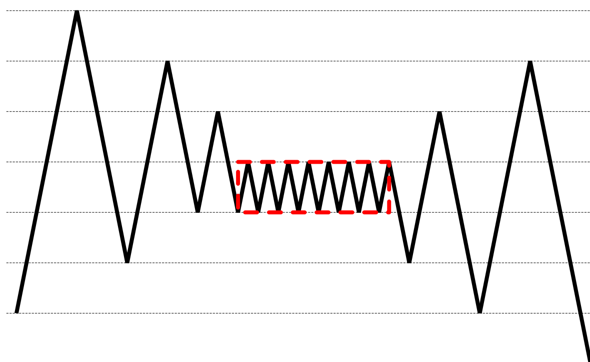
Testy

Przygotowano 10 grup testów. Pierwsza grupa zawiera małe testy poprawnościowe. Pozostałe grupy składają się z 3 lub 4 testów. Testy a i b są losowe z różnym prawdopodobieństwem występowania liter `p`. Test c jest dobrany spomiędzy następujących możliwości:

1. słowo Fibonacciego,

2. słowo Thuego-Morse'a,
3. słowo Bauma-Sweeta,
4. konkatenacja trzech słów, z których każde składa się odpowiednio z:
   - samych liter `j`,
   - samych liter `p`,
   - naprzemiennie ułożonych liter `p` i `j`.

Test *1g* oraz testy *d* występujące w grupach 3, 5, 7, 9 wszystkie mają strukturę podobną do przedstawionej na rysunku. Powodują błędne działanie rozwiązań `barb8.cpp`, `barb9.cpp` oraz maksymalny czas działania rozwiązania `bars10.cpp`.



Rys. 4: Wykres funkcji *pre* dla testu typu *d*; zaznaczono fragment odpowiadający najdłuższemu legalnemu podsłowu.

Jacek Tomasiewicz Treść zadania | **Bartosz Tarnawski** Opracowanie | **Piotr Smulewicz** Program

Dostępna pamięć: 64 MB. *OI, etap I, 7.10–4.11.2013*

Hotele

W Bajtocji jest n miast połączonych zaledwie $n-1$ drogami. Każda z dróg łączy bezpośrednio dwa miasta. Wszystkie drogi mają taką samą długość i są dwukierunkowe. 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 Bajtocji, Bajtazar, chce wybudować trzy luksusowe hotele, które będą gościć turystów z całego świata. Król chciałby, aby hotele znajdowały się w różnych miastach i były położone w tych samych odległościach od siebie.

Pomóż królowi i napisz program, który obliczy, na ile sposobów można wybudować takie trzy hotele w Bajtocji.

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą n ($1 \leqslant n \leqslant 5000$), oznaczającą liczbę miast w Bajtocji. Miasta są ponumerowane od 1 do n.

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

W testach wartych łącznie 50% punktów zachodzi dodatkowy warunek $n \leqslant 500$.

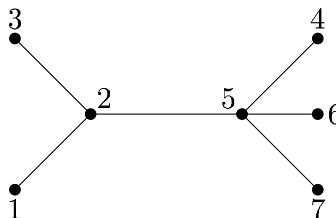
Wyjście

Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą równą liczbie sposobów wybudowania hoteli.

Przykład

Dla danych wejściowych:

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



poprawnym wynikiem jest:

```
5
```

Wyjaśnienie do przykładu: *Trójki (nieuporządkowane) miast, w których można wybudować hotele, to: $\{1,3,5\}$, $\{2,4,6\}$, $\{2,4,7\}$, $\{2,6,7\}$, $\{4,6,7\}$.*

Rozwiązanie

Mamy dane drzewo, nazwijmy je T. Niech $d(u, v)$ oznacza odległość pomiędzy wierzchołkami u i v w tym drzewie (liczbę krawędzi między nimi). Przez $u \rightsquigarrow v$ oznaczymy ścieżkę od u do v. Powiemy, że trzy różne wierzchołki x, y, z są *równoodległe*, jeśli $d(x, y) = d(x, z) = d(y, z)$.

Chcemy znaleźć liczbę nieuporządkowanych trójek równoodległych wierzchołków.

Rozwiązanie bardzo siłowe $O(n^4)$

Dla każdej trójki wierzchołków chcemy sprawdzić, czy są równoodległe. W tym celu wykonujemy przeszukiwanie naszego drzewa z każdego z nich (wszerz lub w głąb). To rozwiązanie zostało zaimplementowane w plikach `hots1.cpp`, `hots2.pas`. Na zawodach zdobywało ok. 30 punktów.

Rozwiązanie siłowe $O(n^3)$

Wykonujemy przeszukiwanie z każdego z wierzchołków i tablicujemy odległości między każdą parą wierzchołków. Przeglądamy wszystkie trójki wierzchołków. Dla każdej z nich sprawdzamy w czasie stałym, czy są one równoodległe.

To rozwiązanie, w odróżnieniu od poprzedniego, używa $O(n^2)$ pamięci. Zostało zaimplementowane w plikach `hotb1.cpp`, `hotb3.pas`. Zgodnie z informacją w treści zadania zdobywało na zawodach ok. 50 punktów.

Rozwiązanie wzorcowe $O(n^2)$

Zaczniemy od prostego lematu, który pomoże nam później lepiej scharakteryzować równoodległe trójki.

Lemat 1. Niech x, y, z będą wierzchołkami drzewa T. Ścieżki $x \rightsquigarrow y$, $y \rightsquigarrow z$ i $z \rightsquigarrow x$ mają dokładnie jeden wierzchołek wspólny.

Dowód: Dla danej trójki wierzchołków x, y, z określmy wierzchołek w jako pierwszy wierzchołek ze ścieżki $z \rightsquigarrow x$, który znajduje się także na ścieżce $y \rightsquigarrow x$. Wykażemy, że jest to dokładnie wierzchołek, którego szukamy.

Z definicji wierzchołka w mamy, że na ścieżce $y \rightsquigarrow w \rightsquigarrow z$ żaden wierzchołek nie powtarza się. (Ścieżka ta może być pojedynczym wierzchołkiem, gdy $y = z$). Jest to więc najkrótsza ścieżka z y do z. A zatem wierzchołek w leży na każdej ze ścieżek $y \rightsquigarrow x$, $z \rightsquigarrow x$ i $y \rightsquigarrow z$. Przecięciem dwóch pierwszych z tych ścieżek jest $w \rightsquigarrow x$, a jedynym punktem wspólnym tej ścieżki ze ścieżką $y \rightsquigarrow w \rightsquigarrow z$ jest w. Tak więc jest to jedyny wierzchołek leżący na przecięciu tych wszystkich ścieżek. ■

Wierzchołek w określony w lemacie 1 nazwiemy *wierzchołkiem spotkania*. W algorytmie wzorcowym będziemy ukorzeniać nasze drzewo po kolei w każdym jego wierzchołku i zliczać te równoodległe trójki, dla których jest on wierzchołkiem spotkania.

Lemat 2. Jeśli x, y, z jest trójką równoodległych wierzchołków drzewa T, to ich wierzchołek spotkania w spełnia $d(x,w) = d(y,w) = d(z,w)$.

Dowód: Oznaczmy $d_1 = d(x,w)$, $d_2 = d(y,w)$, $d_3 = d(z,w)$. Mamy $d(x,y) = d_1+d_2$, $d(x,z) = d_1 + d_3$ oraz $d(y,z) = d_2 + d_3$. Z przyrównania tych trzech odległości otrzymujemy, że $d_1 = d_2 = d_3$. ■

Ukorzeńmy więc T w jednym z jego wierzchołków r. Od tej chwili odległość dowolnego wierzchołka v od r nazywamy *głębokością* v. Ustawmy poddrzewa synów wierzchołka r w kolejności nierosnących maksymalnych głębokości. Oznaczmy je $S_1, S_2, \ldots, S_m$.

Będziemy korzystać z następującej obserwacji, będącej konsekwencją lematu 2.

Lemat 3. Niech x, y, z będą różnymi wierzchołkami na tej samej głębokości. Wtedy następujące warunki są równoważne:

1. x, y, z należą do poddrzew trzech różnych synów r,
2. x, y, z są równoodległe i r jest ich wierzchołkiem spotkania.

Niech najgłębszy wierzchołek w poddrzewie S_i będzie na głębokości D_i. Mamy zatem $D_1 \geqslant D_2 \geqslant \ldots \geqslant D_m$. Przeszukujemy każde z poddrzew S_i, wyznaczając tablicę $at_depth_i[1..D_i]$ taką, że $at_depth_i[l]$ jest liczbą wierzchołków w S_i o głębokości l. Zauważmy, że przeszukanie poddrzew wszystkich synów korzenia zajmie nam łącznie $O(n)$ czasu.

Ustalmy teraz l $(1 \leqslant l \leqslant D_1)$. Chcielibyśmy wyznaczyć liczbę trójek wierzchołków na głębokości l, należących do poddrzew różnych synów r (korzystamy z lematu 3).

Dla dwóch wierzchołków u, v napiszemy $u \prec v$, jeśli $u \in S_{i_u}$, $v \in S_{i_v}$ oraz $i_u < i_v$. Będziemy wybierać po kolei $i = 2, 3, \ldots, m-1$ i zliczać takie trójki wierzchołków x, y, z na głębokości l, że $x \prec y \prec z$ oraz $y \in S_i$. W tym celu wystarczy utrzymywać dwie zmienne $before_i[l]$ oraz $after_i[l]$, aby przy rozważaniu danego i zachodził niezmiennik:

$$before_i[l] = \sum_{a<i} at_depth_a[l], \quad after_i[l] = \sum_{i<b} at_depth_b[l].$$

Wówczas szukana liczba trójek wierzchołków x, y, z spełniających $d(x,r) = d(y,r) = d(z,r) = l$, $x \prec y \prec z$, $y \in S_i$ jest równa

$$before_i[l] \cdot at_depth_i[l] \cdot after_i[l]. \tag{1}$$

Sumując iloczyny (1) dla każdego korzenia r, każdego indeksu syna i oraz każdej głębokości $l \leqslant D_i$, otrzymujemy wynik.

Jaka jest złożoność czasowa rozwiązania przy ustalonym korzeniu r? Najważniejsze spostrzeżenie jest takie, że iloczyn (1) obliczamy tylko wtedy, gdy w poddrzewie S_i rzeczywiście jest jakiś wierzchołek położony na głębokości l. To oznacza, że liczbę obliczeń iloczynów możemy oszacować z góry przez liczbę wszystkich wierzchołków we wszystkich poddrzewach, czyli po prostu przez n. Łatwo zauważyć, że pozostałe obliczenia, tj. wyznaczanie wartości $before_i[l]$ oraz $after_i[l]$, możemy wykonać w takim samym czasie.

Dla ustalonego r zużywamy $O(n)$ czasu, więc ostateczna złożoność to $O(n^2)$. Warto zauważyć, że wynik jest rzędu $O(n^3)$, więc należy go pamiętać w zmiennej całkowitej 64-bitowej.

Powyższy algorytm pozwalał na zdobycie 100 punktów. Jego implementacje znajdują się w plikach `hot.cpp`, `hot2.pas`.

Szybsze rozwiązanie $O(n \log n)$

To rozwiązanie powstało dosyć późno – niemal rok po I etapie XXI Olimpiady. Pod pewnymi względami jest podobne do rozwiązania $O(n^2)$. Tutaj też będziemy ukorzeniać drzewo (ale tylko raz) i skorzystamy z lematu 1. Pewną nowością będzie wykorzystanie *centroidalnej dekompozycji* drzewa (ang. *centroid decomposition*) – przydatnej techniki, która pozwala na przetwarzanie drzewa metodą dziel i zwyciężaj.

Przygotowania

Ponownie ukorzeniamy nasze drzewo w r – jednym z jego wierzchołków. Rozbudujmy jeszcze trochę naszą terminologię. *Wysokością* wierzchołka v nazwiemy maksimum z odległości między v a jakimś wierzchołkiem w jego poddrzewie.

Synów każdego wierzchołka ustawiamy od lewej do prawej w kolejności nierosnących wysokości. Zaznaczmy, że robimy to tylko na potrzeby omówienia, sam algorytm nie musi korzystać z takiej reprezentacji. Jeżeli v ma synów, to *najstarszym synem* v nazwiemy jego pierwszego syna z lewej (jednego z najwyższych). Powiemy jeszcze, że w jest *pierworodny*, jeżeli jest najstarszym synem swojego ojca. Zauważmy, że korzeń nie jest pierworodny.

Zanim przejdziemy do opisu algorytmu, wprowadzimy kolejny lemat, tym razem o nie całkiem intuicyjnej treści, ale ciekawym dowodzie.

Lemat 4. Niech v będzie wierzchołkiem drzewa T. Przez $pot(v)$ (*potencjał* v) oznaczmy sumę wysokości wszystkich synów v oprócz najstarszego syna. Wówczas suma potencjałów wszystkich wierzchołków drzewa nie przekracza n.

Dowód: Na każdym wierzchołku postawmy jednego krasnoludka, w sumie n krasnoludków.

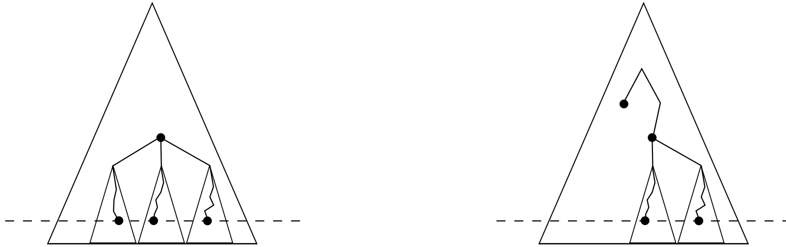
Następnie dla każdego wierzchołka v wykonujemy co następuje. Niech w będzie pierwszym wierzchołkiem na ścieżce z v do r, który nie jest pierworodny. Posyłamy krasnoludka z v do ojca w, a jeśli $w = r$, to ten krasnoludek opuszcza drzewo.

Pokażemy, że po zakończeniu tej procedury liczba krasnoludków na każdym wierzchołku będzie równa jego potencjałowi. Rozważmy bowiem dowolny wierzchołek v oraz jego syna w, który nie jest najstarszy. Wysokość w jest równa liczbie wierzchołków na ścieżce od w do jego skrajnie lewego liścia. Wszystkie wierzchołki na tej ścieżce oprócz w są pierworodne, czyli z każdego z nich przeszedł do v jeden krasnoludek. Natomiast każdy z pozostałych krasnoludków w poddrzewie w zatrzyma się wcześniej, gdyż na ścieżce od w do niego co najmniej raz nie szliśmy skrajnie w lewo. ■

Ten lemat bardzo przyda się w szacowaniu złożoności czasowej naszego algorytmu.

Podzielmy jeszcze równoodległe trójki w naszym drzewie na dwa rodzaje. Powiemy, że trójka x, y, z jest

- *zgodna*, jeśli wierzchołek spotkania x, y, z jest przodkiem wszystkich trzech wierzchołków,
- *niezgodna* w przeciwnym przypadku; patrz rys. 1.



Rys. 1: Po lewej: zgodna równoodległa trójka. Po prawej: niezgodna równoodległa trójka.

Nasz algorytm będzie dzielił się na dwie fazy:

1. Zliczanie zgodnych równoodległych trójek oraz przygotowanie zapytań związanych ze zliczaniem niezgodnych równoodległych trójek.
2. Przetworzenie zapytań za pomocą centroidalnej dekompozycji.

Faza 1

Będziemy postępować podobnie jak w rozwiązaniu $O(n^2)$, ale skorzystamy z lematu 4, aby złożoność tej fazy była liniowa.

Odrobinę rozszerzymy wprowadzoną wcześniej definicję głębokości. Kiedy v należy do poddrzewa wierzchołka w, to głębokością v *względem* w nazwiemy odległość $d(v, w)$.

Zliczanie równoodległych trójek

Przeszukamy nasze drzewo w głąb za pomocą rekurencyjnej funkcji *dfs*(). Chcemy, aby *dfs*(v) zwracało listę liczb at_depth_v zdefiniowaną w następujący sposób:

- długość at_depth_v równa się wysokości v,
- l-ty element (tym razem indeksujemy od 0) at_depth_v to liczba wierzchołków o głębokości l względem v (czyli zawsze zerowy element to 1).

Na razie skupimy się tylko na tym, później łatwo będzie rozszerzyć funkcję *dfs*(), aby zliczała zgodne równoodległe trójki.

Jeżeli v ma synów, to nie będziemy tworzyć listy-wyniku *dfs*(v) od początku, ale wykorzystamy wynik najstarszego syna v, zaktualizowany o wyniki pozostałych synów.

1. Ponumerujmy synów v jako $v_1, \ldots, v_k$ tak, aby najstarszy miał numer 1. Wywołujemy *dfs*() dla wszystkich synów v i zapisujemy wyniki. Niech $at_depth_{v_i}$ będzie listą obliczoną dla i-tego syna; dodatkowo oznaczmy $A = at_depth_{v_1}$.

2. Dodajemy liczbę 1 na początek listy A. Teraz A opisuje głębokości w drzewie składającym się z v i jego najstarszego syna.

3. Dla $i = 2, \ldots, k$ dodajemy elementy listy $at_depth_{v_i}$ do elementów A (j-ty element $at_depth_{v_i}$ do $(j+1)$-szego elementu A). Po i-tym kroku lista A opisuje głębokości w drzewie złożonym z v oraz poddrzew synów $v_1, v_2, \ldots, v_i$.

4. Na koniec zwracamy A jako at_depth_v.

Ten opis algorytmu wymaga jeszcze kilku komentarzy.

Po pierwsze, obiecywaliśmy, że nie będziemy korzystać z uporządkowania synów względem nierosnących wysokości. W algorytmie potrzebujemy jedynie stwierdzić, który syn jest najstarszy. Wystarczy wybrać jednego z synów o największej wysokości, tę zaś liczy się bardzo łatwo.

Po drugie, jeśli rozwiązanie piszemy w języku C++, to zamiast listy możemy zwracać dynamicznie alokowaną tablicę – STL-owy `vector`. Trzeba wtedy indeksować elementy od aktualnego końca tablicy, żeby możliwe było dostawienie nowego elementu o indeksie 1. Musimy też uważać, żeby nie kopiować całej tablicy podczas zwracania wyniku (można np. korzystać ze wskaźników).

Na mocy lematu 4 wywołanie *dfs*(r) zajmie $O(n)$ czasu.

Przetwarzając wierzchołek v, chcielibyśmy zliczyć zgodne równoodległe trójki o wierzchołku spotkania v. Robimy to, korzystając z list-wyników dla synów v oraz dwóch dodatkowych zmiennych, tak samo jak w rozwiązaniu $O(n^2)$.

Przygotowanie zapytań dla fazy 2

Chcemy również zrobić coś z niezgodnymi równoodległymi trójkami, dla których v jest wierzchołkiem spotkania. W tej fazie nie wykonamy samego zliczania tych trójek, ale zredukujemy je do innego problemu.

Podczas wywołania funkcji *dfs*(), tworzymy dodatkową listę $pairs_at_depth_v$.

> l-ty element $pairs_at_depth_v$ to liczba nieuporządkowanych par x, y wierzchołków o głębokości l względem v, należących do poddrzew różnych synów v.

Chcemy przy tym, aby długość tej listy była równa wysokości drugiego najwyższego syna v. Wobec tego możemy obliczać ją dla każdego wierzchołka, nadal zachowując złożoność $O(n)$. Można ją łatwo wyliczyć podczas aktualizowania at_depth_v; szczegóły pozostawiamy Czytelnikowi jako ćwiczenie.

Zobaczmy, jak wygląda niezgodna równoodległa trójka o wierzchołku spotkania v. Dwa wierzchołki należą do poddrzew dwóch różnych synów v i mają tę samą głębokość względem v, powiedzmy l. Trzeci wierzchołek znajduje się poza poddrzewem v i jest odległy od niego o l.

Gdybyśmy potrafili w czasie stałym odpowiadać na następujące pytanie:

„Ile jest wierzchołków odległych od v o zadane l?”,

to wówczas nie mielibyśmy problemu ze zliczeniem tych trójek. Oznaczmy odpowiedź na nasze pytanie przez $at_distance(v, l)$. Wtedy liczba niezgodnych równoodległych trójek o wierzchołku spotkania v to

$$pairs_at_depth_v[l] \cdot (at_distance(v, l) - at_depth_v[l]).$$

Nie jesteśmy w stanie obliczać funkcji $at_distance()$ w czasie stałym. Na szczęście przebieg naszego algorytmu nie zależy od wyników tej funkcji – możemy zapisać wszystkie jej wywołania jako zapytania, odpowiemy na nie zbiorczo w fazie 2. Wraz z zapytaniami zapamiętamy odpowiadające im wartości $pairs_at_depth_v[l]$ oraz $at_depth_v[l]$, aby po przebiegu fazy 2 uaktualnić wynik o odpowiednie iloczyny.

Zauważmy, że w ten sposób wyprodukujemy $O(n)$ zapytań, bowiem suma rozmiarów tablic $pairs_at_depth_v$, na mocy lematu 4, szacuje się z góry przez n.

Faza 2

Naszym celem będzie jak najszybsze odpowiedzenie na $O(n)$ zapytań postaci $at_distance(v, l)$. Wykorzystamy do tego centroidalną dekompozycję drzewa. O samej technice można więcej przeczytać na algorytmicznym blogu prowadzonym przez Pawła Gawrychowskiego[1]. Tutaj przedstawimy ją tylko w zakresie potrzebnym do rozwiązania zadania.

Jeśli po ukorzenieniu T w wierzchołku v poddrzewo każdego syna v ma co najwyżej $\frac{n}{2}$ wierzchołków, to mówimy, że v jest *centroidem* T. Każde drzewo ma jeden lub dwa centroidy. Dowód tego faktu wraz z liniowym algorytmem na znajdowanie centroidów można znaleźć w omówieniu zadania *Polaryzacja* z XX Olimpiady Informatycznej [20].

Nasz algorytm będzie rekurencyjny. Dla każdego zapytania będziemy pamiętać dotychczas obliczoną częściową odpowiedź. Początkowo jest ona równa zero. Chcemy, aby zachodził następujący niezmiennik:

> Częściowa odpowiedź na zapytanie $at_distance(v, l)$ jest równa liczbie wierzchołków odległych od v o l i znajdujących się w największym dotychczas przetworzonym poddrzewie zawierającym v.

Jeżeli nasz algorytm zostanie wywołany w pewnym momencie dla poddrzewa S naszego wyjściowego drzewa T, to robimy co następuje.

1. Znajdujemy centroid c drzewa S.

2. Usuwamy go. Nasze drzewo S rozspójnia się na poddrzewa $S_1, \ldots, S_k$, które przetwarzamy rekurencyjnie.

3. Przeszukujemy drzewo S z wierzchołka c. Dla każdego poddrzewa S_i obliczamy dwie tablice:

   - $at_depth_i[]$ – gdzie $at_depth_i[l]$ to liczba wierzchołków odległych od c o l w poddrzewie S_i,

[1] `http://fajnezadania.wordpress.com/2013/02/19/ile-sciezek/`

- $queries_i[]$ – wszystkie zapytania w poddrzewie S_i.

A także jedną tablicę $at_depth_c[]$, gdzie $at_depth_c[l]$ to liczba wierzchołków w całym drzewie S odległych od c o l. Ponadto dla każdego wierzchołka obliczamy jego odległość od c.

4. Korzystając z tablicy $at_depth_c[]$, uaktualniamy odpowiedzi na zapytania postaci $at_distance(c, \ldots)$.

5. Następnie dla każdego drzewa S_i i zapytania postaci $at_distance(v, l)$, $v \in S_i$, zliczamy wierzchołki odległe od v o l, które należą do S, a nie należą do S_i. Ta liczba to po prostu:

$$at_depth_c[l - d(c, v)] - at_depth_i[l - d(c, v)].$$

Zwiększamy aktualną odpowiedź na to zapytanie o powyższą wartość.

Wywołujemy nasz algorytm dla całego drzewa T. Nasz niezmiennik gwarantuje nam, że uzyskamy w ten sposób pełne odpowiedzi na wszystkie zapytania.

Pozostaje pytanie o złożoność czasową naszego programu. Skorzystamy z argumentu „obliczeń na kolejnych poziomach”, który pojawia się często w dowodzie złożoności sortowania przez scalanie. Będziemy rozważać następujące poziomy:

- całe drzewo T,
- poddrzewa powstałe z T po usunięciu jego centroidu,
- poddrzewa powstałe po usunięciu ich centroidów,
- ...
- pojedyncze wierzchołki.

Na każdym z poziomów mamy w sumie n wierzchołków oraz $O(n)$ zapytań dotyczących tych wierzchołków. Nasz algorytm zużywa zatem $O(n)$ czasu na poziom. Poziomów może być co najwyżej $\log n$, ponieważ usuwając centroid, dwukrotnie zmniejszamy rozmiar największego drzewa.

Zakończyliśmy w ten sposób fazę 2, a tym samym cały algorytm. Jego czas działania to $O(n + n \log n) = O(n \log n)$. Został zaimplementowany w pliku `hot4.cpp`.

Czytelników zainteresowanych metodą centroidalnej dekompozycji zachęcamy do zmierzenia się z zadaniem *Këbab* z XIV Obozu im. A. Kreczmara. Jest ono dostępne w serwisie `main.edu.pl`.

Testy

Przygotowano dziesięć grup testów. W pierwszej grupie znajdują się małe testy – od 1 do 5 wierzchołków. W każdej z pozostałych grup znajdują się po dwa testy. Testy typu *a* są losowe, zaś w testach typu *b* jeden z wierzchołków ma zagwarantowany duży stopień.

Test *10b* to gwiazda złożona z 5000 wierzchołków – test z największym możliwym do uzyskania wynikiem.

Wojciech Rytter, Bartosz Szreder — Treść zadania, Opracowanie
Karol Farbiś, Szymon Stankiewicz — Program

Dostępna pamięć: 64 MB. *OI, etap I, 7.10–4.11.2013*

Klocki

Mały Bitek i jego koledzy cały wczorajszy dzień w przedszkolu spędzili na zabawie kolorowymi klockami. Tworzyli przeróżne budowle, ale szybko im się to znudziło. Postanowili więc poustawiać klocki jeden za drugim. Unikali oni przy tym sytuacji, w której pewien klocek stoi tuż przed klockiem tego samego koloru. Po dłuższym czasie udało im się ustawić w ten sposób wszystkie klocki. Wtedy zajęcia się skończyły i rodzice odebrali dzieci z przedszkola.

Dzisiaj Bitek przybył do przedszkola pół godziny przed zajęciami. Zobaczył, że dzieło zbudowane poprzedniego dnia nadal istniało. Niestety, Bitek przewrócił się i wszystkie klocki się wymieszały. Chłopiec natychmiast uporządkował klocki i zaczął zastanawiać się, jak szybko odbudować to, co zepsuł. Przypomniał sobie, jakiego koloru klocki stały na końcach rzędu.

Pomóż małemu Bitkowi i podpowiedz, jak może ustawić klocki, żeby żaden klocek nie stał przed klockiem tego samego koloru oraz żeby klocki na końcach miały takie kolory, jakie podał Bitek. Jeśli Bitkowi coś się pomyliło lub chłopiec przypadkowo zagubił niektóre klocki podczas upadku, tak że teraz ustawienie zgodne z jego warunkami nie jest możliwe, pomóż mu to stwierdzić.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się trzy liczby całkowite k, p i q ($1 \leqslant k \leqslant 1\,000\,000$, $1 \leqslant p, q \leqslant k$), pooddzielane pojedynczymi odstępami i oznaczające liczbę kolorów klocków oraz kolory pierwszego i ostatniego klocka w szukanym ustawieniu. W drugim wierszu znajduje się k liczb całkowitych $i_1, i_2, \ldots, i_k$, pooddzielanych pojedynczymi odstępami ($1 \leqslant i_j \leqslant 1\,000\,000$). Liczba i_j oznacza, że Bitek ma dokładnie i_j klocków koloru j. Możesz założyć, że liczba wszystkich klocków nie przekracza miliona, tzn. $n = i_1 + i_2 + \ldots + i_k \leqslant 1\,000\,000$.

Wyjście

Twój program powinien wypisać na standardowe wyjście n liczb całkowitych pooddzielanych pojedynczymi odstępami reprezentujących kolory kolejnych klocków w szukanym ułożeniu. Jeśli takie ułożenie nie istnieje, Twój program powinien wypisać tylko jedną liczbę całkowitą: 0.

Jeśli jest wiele poprawnych odpowiedzi, Twój program może podać dowolną z nich.

Przykład

Dla danych wejściowych:

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

jednym z poprawnych wyników jest:

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

a dla danych wejściowych:	*poprawnym wynikiem jest:*
`3 3 1`	`0`
`2 4 2`	

Wyjaśnienie do przykładów:
Innym poprawnym ułożeniem klocków w pierwszym przykładzie jest `3 1 2 3 2 3 2 1`.
W drugim przykładzie Bitek musiał się pomylić – nie można ułożyć klocków tak, żeby spełnione były warunki z treści zadania.

Rozwiązanie

Wstęp

W zadaniu *Klocki* naszym celem jest ułożyć z podanego zestawu kolorowych klocków *ciąg alternujący*, czyli taki, w którym żadna liczba (reprezentująca kolor klocka) nie występuje dwa razy pod rząd. Ponadto ustalony jest element początkowy oraz końcowy ciągu.

Niech i_1 oznacza liczbę dostępnych klocków koloru 1, i_2 liczbę klocków koloru 2 itd. Kolor, którym mamy rozpocząć ciąg, oznaczmy przez p, natomiast kolor „końcowy" oznaczmy przez q. Ponadto k to liczba różnych kolorów klocków, a n to liczba wszystkich klocków, tzn. $n = i_1 + i_2 + \ldots + i_k$. Dla uproszczenia opisu rozwiązania przyjmujemy od teraz, że liczności klocków poszczególnych kolorów są posortowane niemalejąco, czyli i_1 to liczba klocków najrzadziej występujących w podanym zbiorze, a i_k to liczba klocków najpopularniejszych w zestawie. Kolory elementu początkowego i końcowego nazwiemy kolorami *krańcowymi*.

Ciągi alternujące

Rozwiążmy najpierw odrobinę prostsze zadanie: z podanego zestawu klocków chcemy ułożyć **jakikolwiek** ciąg alternujący – bez ustalonego elementu początkowego i końcowego. Okazuje się, że w tym przypadku łatwo sprawdzić, czy jest to w ogóle możliwe i, jeśli jest, ułożyć taki ciąg. Warunek konieczny i wystarczający do istnienia ciągu alternującego prezentuje się następująco:

$$i_k \leqslant \frac{n+1}{2}.$$

Przypomnijmy, że i_k oznacza liczbę najczęściej występujących klocków.

Konieczność wynika z zasady szufladkowej Dirichleta: gdyby powyższa nierówność nie była spełniona, to nawet jeśli wstawilibyśmy klocek koloru k na co drugiej pozycji w ciągu, zabrakłoby nam klocków o innych kolorach i musielibyśmy wstawić obok siebie dwa klocki koloru k.

Dostateczność można wykazać poprzez podanie algorytmu konstrukcji ciągu alternującego:

1. Załóżmy, że $n = i_1 + i_2 + \ldots + i_k$ jest nieparzyste.

2. Znajdź j i taki podział i_j na $i'_j + i''_j = i_j$, żeby

$$i_1 + i_2 + \ldots + i'_j + 1 = i''_j + i_{j+1} + \ldots + i_k.$$

3. Niech S_1 będzie ciągiem i_1 klocków koloru 1, i_2 klocków koloru 2, $\ldots$, i'_j klocków koloru j.

4. Niech S_2 będzie ciągiem i''_j klocków koloru j, i_{j+1} klocków koloru $j+1$, $\ldots$, i_k klocków koloru k.

5. Zbuduj docelowy ciąg, przeplatając elementy ciągów S_2 i S_1.

Dla n parzystych jest tak samo, tylko bez $+1$ w szukaniu podziału (drugi podpunkt algorytmu).

Właściwy problem

Okazuje się, że warunek wystarczający do skonstruowania ciągu alternującego dla **dowolnej** pary kolorów krańcowych (oznaczanych dalej p i q) jest bardzo podobny do warunku koniecznego istnienia dowolnego ciągu alternującego:

$$i_k \leqslant \frac{n-1}{2}.$$

Jeśli $p = q$, dochodzi do tego prosty warunek brzegowy, polegający na sprawdzeniu, czy $i_p \geqslant 2$.

Konstrukcję docelowego ciągu zaczynamy od usunięcia dwóch elementów ze zbioru klocków: jednego w kolorze p i jednego w kolorze q. Nawet jeśli $p \neq k$ i $q \neq k$, to nadal spełniony jest warunek wystarczający do zbudowania jakiegokolwiek ciągu alternującego, co też czynimy. Na końcach tego ciągu należy następnie dodać dwa wcześniej wyjęte klocki p i q.

Teraz zadanie polega na tym, żeby nie doprowadzić do powtórzenia koloru na lewym albo prawym brzegu ciągu. Oznaczmy kolory klocków na końcach otrzymanego ciągu alternującego jako $\hat{p}$ i $\hat{q}$. Mamy następujące przypadki:

1. $\{p, q\} \cap \{\hat{p}, \hat{q}\} = \emptyset$. Przypadek trywialny do rozwiązania: dostawiamy klocki p i q jakkolwiek i zawsze będzie poprawnie.

2. $p \neq q \wedge \hat{p} \neq \hat{q}$. Przypadek niewiele trudniejszy: może być tak, że p lub q równe jest $\hat{p}$ lub $\hat{q}$, więc możemy być zmuszeni wstawić p (albo q) po konkretnej stronie, żeby nie wystąpiło powtórzenie koloru.

3. $\hat{p} = \hat{q}$. Podany wcześniej algorytm konstrukcji ciągu alternującego doprowadzi do takiej sytuacji tylko wtedy, gdy kolor $\hat{p}$ jest najliczniejszy w zredukowanym ciągu oraz kolor ten występuje maksymalną dopuszczalną liczbę razy. Ale z tego wynika, że ani p, ani q nie są tego samego koloru co $\hat{p}$, bo wtedy byłoby $i_k > \frac{n-1}{2}$. To oznacza, że jest to ten sam przypadek co (1).

4. Możemy teraz przyjąć, że $\hat{p} \neq \hat{q}$, $p = q$ oraz p jest takie samo jak $\hat{p}$ albo $\hat{q}$. Bez straty ogólności przyjmijmy $p = \hat{p}$. Mamy więc trzy elementy koloru p i jeden koloru $\hat{q}$. Jeden klocek koloru p oraz klocek koloru $\hat{q}$ ustawiamy po jednej stronie. Teraz musimy przepchnąć jedno p w takie miejsce w środku ciągu, w którym sąsiadują dwa elementy koloru innego od p. Takie miejsce zawsze istnieje – gdyby nie istniało, to element koloru p musiałby występować na co drugiej pozycji w zredukowanym ciągu, a zatem doliczając dwa dodatkowe elementy koloru p (tzn. elementy p i q), zaburzylibyśmy nierówność z warunku koniecznego.

Jeśli warunek wystarczający do ułożenia ciągu alternującego dla dowolnych p i q nie jest spełniony, nie wszystko stracone. Istnieje jeszcze kilka przypadków, gdy poprawny ciąg da się ułożyć:

- Dla n nieparzystego oraz $i_k = \frac{n+1}{2}$ musimy rozmieszczać klocki koloru k na co drugiej pozycji, także początkowej i końcowej, aby ciąg był alternujący. Rozwiązanie zatem istnieje dla $p = q = k$.

- Dla n parzystego oraz $i_k = \frac{n}{2}$ przynajmniej jeden z klocków p, q musi być koloru k. Co więcej, jeśli oba są koloru k, to poprawny ciąg da się ułożyć wtedy i tylko wtedy, gdy $i_{k-1} < \frac{n}{2}$, czyli gdy są przynajmniej trzy różne kolory klocków. Jeśli jednakże $k = 2$ i $i_1 = i_2 = \frac{n}{2}$, to jedyny ciąg alternujący przeplata kolory 1 i 2, więc początek i koniec muszą mieć takie właśnie kolory.

Rozwiązanie wzorcowe

Ponieważ musimy posortować klocki według liczności ich kolorów, złożoność czasowa rozwiązania wzorcowego wyznaczona jest przez złożoność zastosowanego algorytmu sortowania. Wszystkie późniejsze operacje wykonywane są w czasie stałym (sprawdzanie warunków istnienia ciągu alternującego) bądź liniowym (konstrukcja ciągu alternującego).

Jako wzorcowe uznano programy działające w czasie $O(k \log k + n)$, stosujące efektywny algorytm sortowania przez porównania, lub $O(n)$, używające sortowania kubełkowego. Rozwiązanie o złożoności $O(k \log k + n)$ zaimplementowano w plikach `klo.cpp`, `klo1.c` i `klo2.pas`.

Jacek Tomasiewicz Treść zadania, Opracowanie

Karol Pokorski Program

Dostępna pamięć: 128 MB. *OI, etap I, 7.10–4.11.2013*

Kurierzy

Bajtazar pracuje w firmie BAJ sprzedającej gry komputerowe. Firma BAJ współpracuje z wieloma firmami kurierskimi, które dostarczają sprzedawane gry klientom firmy BAJ. Bajtazar prowadzi kontrolę tego, jak przebiegała współpraca firmy BAJ z firmami kurierskimi. Ma on listę kolejno wysłanych paczek, wraz z informacją o tym, która firma kurierska dostarczyła którą paczkę. Interesuje go, czy któraś z firm kurierskich nie uzyskała niezasłużonej przewagi nad innymi firmami kurierskimi.

Jeżeli w jakimś przedziale czasu określona firma kurierska dostarczyła więcej niż połowę wysłanych wówczas paczek, to powiemy, że firma ta **dominowała** *w tym czasie. Bajtazar chce stwierdzić, czy w określonych przedziałach czasu jakieś firmy kurierskie dominowały, a jeśli tak, to które to były firmy.*

Pomóż Bajtazarowi! Napisz program, który będzie znajdował dominującą firmę lub stwierdzi, że żadna firma nie dominowała.

Wejście

Pierwszy wiersz standardowego wejścia zawiera dwie liczby całkowite n i m ($1 \leqslant n, m \leqslant 500\,000$), oddzielone pojedynczym odstępem i oznaczające liczbę wysłanych przez firmę BAJ przesyłek oraz liczbę przedziałów czasowych, dla których chcemy poznać dominujące firmy. Firmy kurierskie są ponumerowane od 1 do n.

Drugi wiersz wejścia zawiera n liczb całkowitych $p_1, p_2, \ldots, p_n$ ($1 \leqslant p_i \leqslant n$), pooddzielanych pojedynczymi odstępami; p_i oznacza numer firmy kurierskiej, która dostarczyła i-tą (w kolejności chronologicznej) wysłaną paczkę.

Kolejne m wierszy zawiera opisy kolejnych zapytań, po jednym w wierszu. Opis każdego zapytania składa się z dwóch liczb całkowitych a i b ($1 \leqslant a \leqslant b \leqslant n$), oddzielonych pojedynczym odstępem, oznaczających, że szukamy firmy dominującej w okresie między wysłaniem a-tej a b-tej paczki włącznie.

W testach wartych łącznie 65% punktów zachodzi dodatkowy warunek $n, m \leqslant 50\,000$, a w testach wartych 30% punktów zachodzi $n, m \leqslant 5000$.

Wyjście

Standardowe wyjście powinno zawierać m wierszy, w których powinny znaleźć się odpowiedzi na kolejne zapytania, po jednej w wierszu. W każdym wierszu powinna znaleźć się jedna liczba całkowita, równa numerowi firmy, która zdominowała rynek w rozważanym przedziale czasu, lub 0, jeśli takiej firmy nie było.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`7 5`	`1`
`1 1 3 2 3 4 3`	`0`
`1 3`	`3`
`1 4`	`0`
`3 7`	`4`
`1 7`	
`6 6`	

Rozwiązanie

Mamy dany ciąg składający się z n elementów. Musimy odpowiedzieć na m zapytań o lidera w pewnym fragmencie ciągu, przy czym *liderem* nazywamy element, który występuje w danym fragmencie więcej niż połowę razy.

Wyszukiwanie lidera

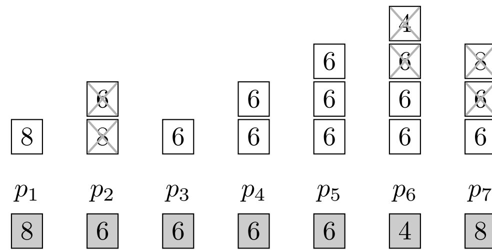
Zauważmy, że jeżeli ciąg długości d ma lidera, to jest on tylko jeden. Gdyby było dwóch liderów, to liczba ich wystąpień musiałaby być większa niż $2 \cdot \frac{d}{2} = d$, a mamy tylko d elementów.

Jeśli dla każdego elementu policzymy jego wystąpienia, przeglądając cały ciąg, to znalezienie lidera zajmie nam czas $O(d^2)$. Jednak lidera możemy znaleźć szybciej. Wystarczy posortować cały ciąg, a następnie zliczać kolejne pakiety elementów o tych samych wartościach. Możemy też to zrobić sprytniej. Zauważmy, że jeśli ciąg ma lidera, to w naszym posortowanym ciągu lider na pewno znajdzie się pod indeksem o numerze $\lceil \frac{d}{2} \rceil$, gdyż nie jesteśmy w stanie zmieścić wszystkich elementów o wartości lidera przed ani za środkowym indeksem. W ten sposób lidera znajdziemy w czasie $O(d \log d)$, ze względu na sortowanie danych.

Spróbujmy uzyskać jeszcze szybsze rozwiązanie. W tym celu przyda nam się pewne proste, ale ważne spostrzeżenie. Zauważmy, że jeżeli ciąg posiada lidera, to usunięcie z niego pary **różnych** elementów da ciąg z tym samym liderem. Faktycznie, ponieważ zawsze usuwamy dwa różne elementy, to najwyżej jeden z nich jest liderem. To oznacza, że lider oryginalnego ciągu w nowym ciągu występuje więcej niż $\frac{d}{2} - 1 = \frac{d-2}{2}$ razy. A zatem jest też liderem nowego ciągu, mającego $d-2$ elementy.

Usuwanie z ciągu par różnych elementów nie jest zupełnie trywialne. Stworzymy początkowo pusty stos, na który będziemy wkładać kolejne elementy ciągu. Po każdym wstawieniu elementu sprawdzamy, czy na szczycie stosu znajdują się dwie różne wartości. Jeśli tak, to zdejmujemy je ze stosu – jest to równoważne z usunięciem pary różnych liczb z ciągu (patrz rys. 1).

Zauważmy jednak, że tak naprawdę nie musimy trzymać całego stosu, gdyż wszystkie wartości poniżej szczytu będą zawsze takie same. Wystarczy więc przechowywać wartość tych elementów oraz rozmiar stosu (zmienne *wartość* i *krotność*). W ten sposób otrzymujemy następujący pseudokod rozwiązania.



Rys. 1: Przykład działania algorytmu wyznaczania kandydata na lidera za pomocą stosu dla ciągu 8, 6, 6, 6, 6, 4, 8.

```
 1: function kandydatNaLidera(P[1..d])
 2: begin
 3:    wartosc := −1;
 4:    krotnosc := 0;
 5:    for k := 1 to d do
 6:       if krotnosc = 0 then begin
 7:          krotnosc := 1;
 8:          wartosc := P[k];
 9:       end else begin
10:          if wartosc ≠ P[k] then
11:             krotnosc := krotnosc − 1;
12:          else
13:             krotnosc := krotnosc + 1;
14:       end
15:    kandydat := −1;
16:    if krotnosc > 0 then
17:       kandydat := wartosc;
18:    return kandydat;
19: end
```

Wcześniej zauważyliśmy, że jeśli początkowy ciąg posiada lidera, to po usunięciu pary różnych elementów lider się nie zmieni. Po usunięciu wszystkich par różnych elementów otrzymamy ciąg stały. Dowolny z elementów tego ciągu na pewno jest kandydatem na lidera, ale nie mamy jeszcze pewności, czy rzeczywiście jest liderem. Aby się o tym przekonać, na koniec powinniśmy przejrzeć cały ciąg i policzyć wystąpienia kandydata – jeśli jest ich więcej niż $\frac{d}{2}$, to znaleźliśmy lidera, a w przeciwnym przypadku ciąg nie posiada lidera.

Złożoność takiego rozwiązania to $O(d)$, ponieważ każdy element rozpatrywany jest tylko raz, a końcowe zliczenie wystąpień kandydata działa również w czasie $O(d)$.

Rozwiązanie wolne $O(n \cdot m)$

Dla każdego zapytania możemy zastosować wprost wyżej opisany algorytm wyszukiwania lidera. W pesymistycznym przypadku będziemy musieli odpowiedzieć na m

pytań o lidera we fragmentach długości $O(n)$, więc uzyskamy złożoność $O(n \cdot m)$. Takie rozwiązanie pozwalało otrzymać około 30% punktów. Przykładowa implementacja znajduje się w plikach `kurs5.cpp` i `kurs6.pas`.

Rozwiązanie wzorcowe $O(n + m \log n)$

Zauważmy, że jeżeli mamy pewne dwa sąsiadujące ze sobą fragmenty ciągu, dla których wyznaczyliśmy wartość i krotność kandydata na lidera, to możemy w łatwy sposób obliczyć analogiczne wyniki dla sumy tych fragmentów. Wartością i krotnością kandydata w scalonym fragmencie będą odpowiednio wartość częściej występującego elementu z fragmentów składowych oraz suma lub różnica krotności, w zależności od tego, czy kandydaci byli równi.

Chcielibyśmy wybrać strukturę danych, która przechowa wartości i krotności dla pewnych *bazowych* fragmentów ciągu (przykładowo, fragmentów o długościach będących potęgami dwójki). Struktura powinna umożliwiać szybkie rozbicie dowolnego fragmentu z zapytania na fragmenty bazowe, dla których są już obliczone wartości i krotności. Dodatkowo, liczba fragmentów bazowych w rozbiciu powinna być nieduża, gdyż scalenie ich wszystkich zajmuje czas liniowy względem ich liczby.

Struktura danych

Idealną strukturą danych do wykonywania żądanych operacji jest **drzewo przedziałowe** (opisane dokładnie w rozwiązaniach zadań *Tetris 3D* z XIII Olimpiady Informatycznej [13] oraz *Koleje* z IX Olimpiady Informatycznej [9], a także w *Wykładach z Algorytmiki Stosowanej*, `http://was.zaa.mimuw.edu.pl`). Tę strukturę danych dla całego ciągu zbudujemy w czasie $O(n)$, a rozbicie dowolnego fragmentu na fragmenty bazowe wykonamy za jej pomocą w czasie $O(\log n)$, uzyskując maksymalnie $O(\log n)$ fragmentów bazowych.

Przetworzenie kandydatów

Drzewo przedziałowe umożliwi szybkie znalezienie kandydata na lidera w każdym fragmencie. W ten sposób uzyskamy m fragmentów z przypisanymi kandydatami na lidera. Chcielibyśmy szybko obliczyć liczbę wystąpień każdego kandydata w jego fragmencie. Jeśli wykonamy to siłowo, uzyskamy złożoność $O(n \cdot m)$.

Wszystkich kandydatów możemy jednak przetworzyć hurtowo. Będziemy przeglądać cały ciąg od lewej do prawej i zliczać wystąpienia poszczególnych liczb. Niewielki zakres wartości elementów pozwala nam utworzyć tablicę, w której każda z liczb może być zliczona pod indeksem odpowiadającym jej wartości.

Jak wykorzystać powyższą tablicę do wyznaczenia liczby wystąpień kandydata w dowolnym fragmencie $[x..y]$? Jeśli napotykamy koniec fragmentu w miejscu y, to w tym momencie pod odpowiednim indeksem w tablicy mamy informację o liczbie wystąpień kandydata we fragmencie $[1..y]$, czyli trochę za dużo. Wcześniej, napotykając początek fragmentu, moglibyśmy zapamiętać liczbę wystąpień kandydata we fragmencie $[1..(x-1)]$. Odejmując liczby wystąpień kandydata w powyższych fragmentach uzyskamy liczbę wystąpień kandydata w szukanym fragmencie $[x..y]$.

Przetworzenie wszystkich kandydatów zajmie czas $O(n+m)$. Ostatecznie, złożoność czasowa całego algorytmu wyniesie $O(n+m\log n)$, przy czym $O(m\log n)$ to czas rozbicia m fragmentów na fragmenty bazowe. Implementacja tego rozwiązania znajduje się w plikach `kur.cpp` i `kur2.pas`.

Rozwiązanie randomizowane $O(k\cdot(n+m))$

Spróbujmy wymyślić inny sposób znajdowania kandydatów na lidera. Dla pojedynczego fragmentu możemy takiego kandydata po prostu wylosować. Zauważmy, że wylosowany element będzie poprawnym kandydatem z prawdopodobieństwem większym od 50%. Rzeczywiście, jeśli fragment zawiera lidera, to lider ten występuje we fragmencie więcej niż połowę razy, natomiast jeśli we fragmencie lidera nie ma, to każdy kandydat jest równie dobry.

Tak więc prawdopodobieństwo tego, że się pomylimy, jest mniejsze niż $\frac{1}{2}$. Możemy je poprawić, jeśli wylosujemy k kandydatów i dla każdego z nich sprawdzimy, czy rzeczywiście jest liderem. Prawdopodobieństwo błędu wynosi wtedy mniej niż $(\frac{1}{2})^k$. Dla ustalonego k, powiedzmy $k=10$, prawdopodobieństwo błędu dla pojedynczego fragmentu wyniesie maksymalnie $(\frac{1}{2})^{10}=\frac{1}{1024}\approx 0{,}1\%$, czyli stosunkowo mało. Tak więc prawdopodobieństwo sukcesu będzie co najmniej $1-(\frac{1}{2})^{10}\approx 99{,}9\%$.

Zauważmy, że oszacowaliśmy prawdopodobieństwo dla pojedynczego zapytania, a w zadaniu mamy m zapytań. Jeśli dla każdego fragmentu wylosujemy oddzielnie k kandydatów, to prawdopodobieństwo sukcesu wyniesie co najmniej $(1-(\frac{1}{2})^k)^m$. Okazuje się, że już dla $k=26$ prawdopodobieństwo sukcesu dla maksymalnego $m=500\,000$ to w najgorszym razie około 99%, co możemy uznać za satysfakcjonujące.

W ten sposób zgromadziliśmy po k kandydatów dla każdego fragmentu. Musimy teraz sprawdzić, czy któryś z nich jest faktycznie liderem. Wszystkich kandydatów możemy przetworzyć za pomocą sposobu opisanego w rozwiązaniu wzorcowym: wybieramy po jednym kandydacie z każdego fragmentu i wykonujemy algorytm przetwarzania. Następnie całość powtarzamy jeszcze $k-1$ razy.

Złożoność czasowa otrzymanego rozwiązania to $O(k\cdot(n+m))$, ze względu na k-krotne przetwarzanie wszystkich fragmentów. Takie rozwiązanie dla odpowiednio dobranego k także uzyskiwało maksymalną liczbę punktów. Implementacja znajduje się w plikach `kur3.cpp` i `kur4.pas`.

Testy

Przygotowano 15 grup testów. Pierwsza grupa składa się z małych testów poprawnościowych, wygenerowanych ręcznie. Wszystkie pozostałe grupy zawierają testy następujących typów:

- naprzemienne ciągi złożone tylko z dwóch wartości z drobnymi losowymi zmianami;
- ciąg posortowany złożony z liczb od 1 do $\sqrt{n}$ z drobnymi zmianami;
- ciągi generowane losowo.

Jakub Radoszewski — Treść zadania, Opracowanie
Jakub Radoszewski — Program

Dostępna pamięć: 512 MB. *OI, etap I, 7.10–4.11.2013*

Wąż

Na planszy o rozmiarze $3 \times n$ leży wąż. Kolejne fragmenty węża są ponumerowane od 1 do $3n$. Fragmenty o kolejnych numerach (tj. 1 i 2, 2 i 3, 3 i 4...) znajdują się na polach planszy sąsiadujących bokiem. Przykładowo, na planszy rozmiaru 3×9 wąż może leżeć tak:

7	6	5	4	17	18	19	20	21
8	1	2	3	16	15	26	25	22
9	10	11	12	13	14	27	24	23

Niektóre spośród pól zajmowanych przez węża zostały zamazane. Czy potrafisz odtworzyć układ węża?

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita n ($1 \leqslant n \leqslant 1000$), oznaczająca długość planszy. Kolejne trzy wiersze zawierają opis planszy; i-ty z nich zawiera n liczb całkowitych a_{ij} ($0 \leqslant a_{ij} \leqslant 3n$ dla $1 \leqslant j \leqslant n$). Jeśli $a_{ij} > 0$, to a_{ij} oznacza numer fragmentu węża znajdującego się na j-tym polu i-tego wiersza planszy. Jeśli natomiast $a_{ij} = 0$, to numer fragmentu węża znajdującego się na rozważanym polu nie jest znany.

W testach wartych łącznie 15% punktów zachodzi warunek $n \leqslant 10$, w testach wartych łącznie 40% punktów zachodzi warunek $n \leqslant 40$, a w testach wartych łącznie 70% punktów zachodzi warunek $n \leqslant 300$.

Wyjście

Twój program powinien wypisać na standardowe wyjście trzy wiersze. W i-tym wierszu powinno znajdować się n liczb całkowitych dodatnich b_{ij} (dla $1 \leqslant j \leqslant n$). Wszystkie liczby b_{ij} łącznie powinny stanowić permutację liczb od 1 do $3n$. Układ liczb na wyjściu powinien odtwarzać możliwe położenie węża zgodne z danymi wejściowymi.

Możesz założyć, że istnieje co najmniej jeden sposób odtworzenia położenia węża na planszy. Jeśli jest więcej niż jedno rozwiązanie, Twój program może wypisać dowolne z nich.

Przykład

Dla danych wejściowych:
```
9
0 0 5 0 17 0 0 0 21
8 0 0 3 16 0 0 25 0
0 0 0 0 0 0 0 0 23
```

jednym z poprawnych wyników jest:
```
7 6 5 4 17 18 19 20 21
8 1 2 3 16 15 26 25 22
9 10 11 12 13 14 27 24 23
```

Rozwiązanie

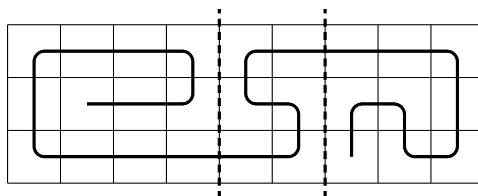
Spróbujmy na początek sformułować treść zadania w języku teorii grafów. Planszę $3 \times n$ możemy wyobrazić sobie jako graf nieskierowany, którego wierzchołkami są pola planszy. Krawędź między wierzchołkami występuje wtedy, gdy pola odpowiadające tym wierzchołkom sąsiadują bokiem. Ułożenie węża na planszy odpowiada wówczas pewnej *ścieżce Hamiltona* w tym grafie, czyli ścieżce przechodzącej przez każdy wierzchołek grafu dokładnie raz.

Niektóre pola planszy mają przypisane parami różne numery z zakresu od 1 do $3n$. Numer pola oznacza, którym z kolei wierzchołkiem na szukanej ścieżce Hamiltona jest to pole. Poszukujemy zatem jakiejkolwiek ścieżki Hamiltona *zgodnej z numeracją wierzchołków.* W treści zadania znajdujemy zapewnienie, że taka ścieżka istnieje.

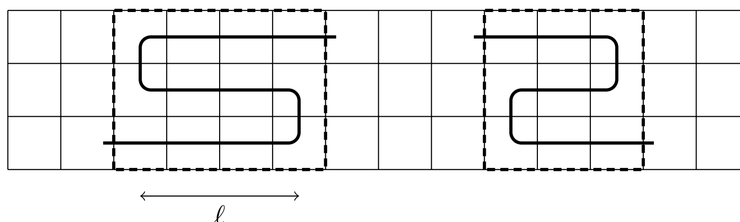
Jak może leżeć wąż?

Aby zbliżyć się do rozwiązania, warto przyjrzeć się temu, jak wyglądają różne ścieżki Hamiltona na kratce $3 \times n$, na razie nie biorąc pod uwagę numeracji wierzchołków.

Najpierw rozważmy sytuację, w której planszę da się podzielić pionowym cięciem wzdłuż linii kratki na fragmenty, między którymi ścieżka przechodzi tylko raz. Poniższy rysunek pokazuje przykład takiej ścieżki zaczerpnięty z treści zadania, z dwiema możliwymi pozycjami cięcia.

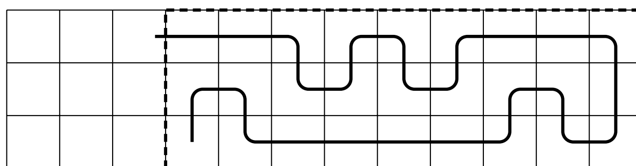


Po wykonaniu wszystkich takich cięć ścieżka Hamiltona całej planszy rozpada się na ścieżki Hamiltona krótszych fragmentów planszy (w powyższym przykładzie są to trzy ścieżki). Mamy zatem dwa skrajne fragmenty planszy oraz pewne fragmenty planszy w środku. Te drugie spełniają dodatkowy warunek, że początek ścieżki znajduje się w nich na lewym boku fragmentu, a koniec na prawym boku. Po chwili rysowania widać, że taka ścieżka Hamiltona musi mieć postać *zygzaka*:

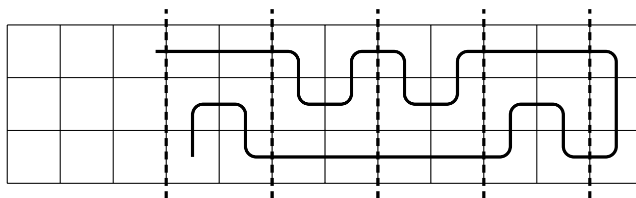


Na rysunku przedstawiono dwa jedyne możliwe ułożenia zygzaka. Drugim parametrem zygzaka jest jego długość (czyli liczba pól zajmowanych w poziomie), oznaczona na rysunku przez ℓ. W skrajnym przypadku, gdy $\ell = 1$, zygzak ograniczony do swojego fragmentu planszy jest po prostu pionowym odcinkiem.

Zostały nam jeszcze do przeanalizowania dwa skrajne fragmenty planszy. W przypadku każdego z nich o ścieżce Hamiltona wiemy tylko tyle, że jej początek znajduje się na lewym (odpowiednio prawym) boku fragmentu. W przykładzie z treści zadania tak się szczęśliwie złożyło, że w prawym skrajnym fragmencie planszy drugi koniec ścieżki znalazł się na tym samym boku co jej początek w tym fragmencie. O takiej ścieżce powiemy, że jest to ścieżka typu *tam i z powrotem*. Ma ona całkiem regularną postać, np.:

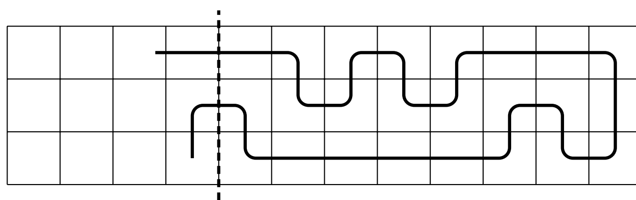


Jak opisać taką ścieżkę? Możemy podzielić fragment planszy na pary kolejnych kolumn (z wyłączeniem ostatniej kolumny, w której ścieżka zawraca). W każdej takiej parze cztery górne pola albo cztery dolne pola tworzą *garb*:

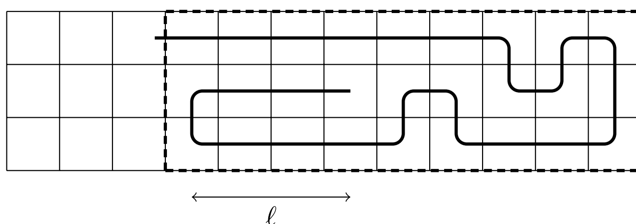


Zaznaczmy tylko, że jeśli liczba kolumn we fragmencie jest parzysta, to skrajnie lewy garb jest ucięty w połowie.

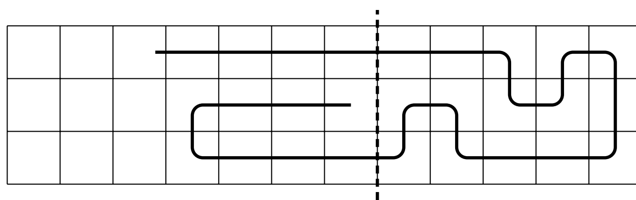
Jeszcze wygodniejszy dla nas sposób patrzenia na taką ścieżkę jest następujący. Jeśli obciąć kolumnę, w której znajdują się oba końce ścieżki, to w wyniku otrzymamy ścieżkę Hamiltona dla krótszego fragmentu planszy, również o tej własności, że oba jej końce leżą na tym samym boku fragmentu (czyli również ścieżkę *tam i z powrotem*):



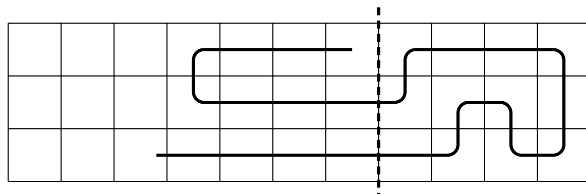
To wszystko pisaliśmy przy założeniu, że ścieżka Hamiltona w skrajnym fragmencie jest ścieżką typu *tam i z powrotem*. Jeśli ścieżka kończy się gdzieś wewnątrz skrajnego fragmentu, to żeby dojść do tego punktu, musi najpierw wrócić do tego samego boku, przy którym się zaczęła (żeby odwiedzić znajdujące się tam pola fragmentu planszy). Faktycznie, gdyby ścieżka nie musiała już wracać do początkowego boku, oznaczałoby to, że musiałaby odwiedzać te pola za pomocą *zygzaka*, co jednak przeczyłoby temu, że fragment planszy jest skrajny po podziale. Po dotarciu do początkowego boku ścieżka zawraca w kierunku swojego końca. Ponieważ mamy do dyspozycji tylko trzy wiersze, więc opisane tu zawracanie odpowiada *zawijasowi*, w którym ścieżka biegnie jednym wierszem w jedną stronę, a drugim w drugą stronę:



Na rysunku zaznaczyliśmy długość *zawijasa* ℓ. A co znajduje się za *zawijasem*? Mamy tam nic innego jak ścieżkę Hamiltona, której oba końce znajdują się na tym samym boku krótszego fragmentu – jest to dokładnie ścieżka typu *tam i z powrotem*, którą rozważaliśmy wcześniej!

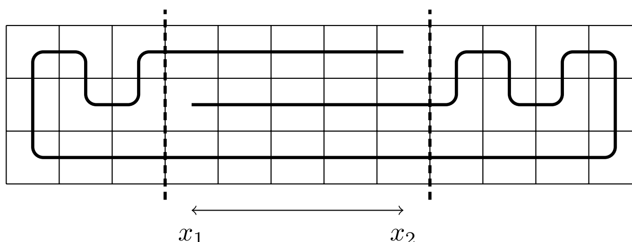


Warto jeszcze zwrócić uwagę na dwie rzeczy. Po pierwsze, *zawijas* nie musi mieć dokładnie takiego kształtu jak ten powyżej. Jedynie część „zawracająca” musi składać się z dwóch kolejnych wierszy. Oto inny, zupełnie poprawny *zawijas*:



Po drugie, może się tak zdarzyć, że zawijas będzie biegł przez całą długość skrajnego fragmentu. W takim przypadku cała ścieżka będzie miała dobrze nam znaną postać *zygzaka* rozważanego na początku tej sekcji.

Na sam koniec zostawiliśmy sobie ostatni przypadek, którego można było nawet w ogóle nie zauważyć. Jest to taka ścieżka Hamiltona, której w żadnym miejscu nie można podzielić pionowym cięciem na ścieżki Hamiltona krótszych fragmentów. Jeśli taka ścieżka zaczyna się na lewym albo na prawym brzegu planszy, to jest to po prostu pojedynczy *zygzak* lub jedna ścieżka typu *tam i z powrotem*, z *zawijasem* lub bez. Jeśli nie, to okazuje się, że jest tylko jeden rodzaj takiej ścieżki, złożony z *zawijasa drugiego typu* i dwóch ścieżek typu *tam i z powrotem*:



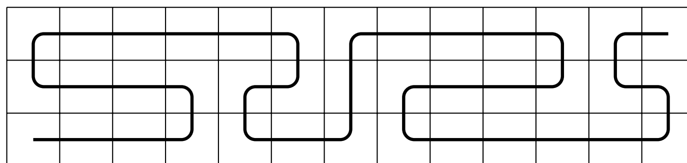
Zawijas drugiego typu ma dwa parametry, x_1 i x_2, określające pozycje jego początku i końca. Składa się z dwóch ścieżek połączonych tylko z jednym fragmentem planszy i ścieżki łączącej oba fragmenty planszy; możliwe są wszystkie rozmieszczenia tych trzech ścieżek w trzech wierszach planszy. Taką ścieżkę Hamiltona będziemy nazywać *zakręconą*. Pozwolimy sobie już pominąć dokładne uzasadnienie, że każda „niepodzielna" ścieżka jest ścieżką zakręconą.

W podsumowaniu całej sekcji możemy napisać, że każda ścieżka Hamiltona na kratce $3 \times n$ albo jest *zakręcona*, albo składa się z co najwyżej dwóch ścieżek typu *tam i z powrotem*, z których każda może mieć dodatkowy *zawijas*, i z dowolnej liczby *zygzaków* w środku.

Pierwsza przymiarka

Nadeszła pora, aby przypomnieć sobie, że nasze zadanie do pewnego stopnia precyzuje, jaką ścieżkę Hamiltona mamy znaleźć. Mamy mianowicie daną dwuwymiarową tablicę $a[1..n, 1..3]$ określającą numery poszczególnych pól na ścieżce, przy czym 0 w tablicy oznacza, że numer danego pola nie jest znany (celowo zamieniliśmy wymiary tablicy w stosunku do treści zadania, tak aby były zgodne z osiami układu współrzędnych). Chcielibyśmy wykorzystać dotychczasowe rozważania kombinatoryczne, aby stworzyć efektywny algorytm znajdowania ścieżki Hamiltona pasującej do zadanych numerów pól. Podstawą algorytmu będzie metoda programowania dynamicznego.

W tej sekcji pozwolimy sobie na pewne uproszczenie, co pomoże nam uchwycić główne pomysły zawarte w rozwiązaniu wzorcowym. Założymy mianowicie, że szukana ścieżka Hamiltona składa się tylko i wyłącznie z *zygzaków* (przykład takiej ścieżki znajduje się na rysunku na następnej stronie). W kolejnej sekcji pokażemy, jak rozszerzyć to rozwiązanie tak, aby poradzić sobie z pozostałymi typami ścieżek. Będziemy celowali w rozwiązanie o złożoności $O(n^2)$.



Spróbujemy zastosować naturalny pomysł, aby budować naszą ścieżkę Hamiltona, dokładając kolejne zygzaki od lewej do prawej. Jeśli mamy już pokryty jakiś fragment planszy, to na kolejnej pozycji będziemy próbowali przykładać różnej długości zygzaki i sprawdzać, czy pasują one do zadanej numeracji pól. W przypadku, gdy któryś z tych zygzaków będzie zgodny z numeracją pól, zapamiętamy informację, że udało nam się pokryć nowy, dłuższy fragment planszy. Na koniec sprawdzimy, czy którymś ze sposobów udało nam się pokryć całą planszę.

Na początku algorytmu musimy jeszcze podjąć kilka decyzji dotyczących początku ścieżki. Skrajnie lewy zygzak może zaczynać się w lewym dolnym albo w lewym górnym rogu planszy; początek każdego kolejnego zygzaka na ścieżce jest już wyznaczony jednoznacznie. Ponadto musimy ustalić, z której strony chcemy umieścić początek, a z której koniec ścieżki Hamiltona, innymi słowy – które pole ma mieć numer 1, a które $3n$. W tym opisie założymy, że pierwszy zygzak zaczyna się w lewym dolnym lub lewym górnym polu planszy, któremu to polu chcemy przypisać numer 1.

Napiszemy pomocniczą funkcję

$$zygzak(x_1, x_2, y_1)$$

sprawdzającą dla $1 \leqslant x_1 \leqslant x_2 \leqslant n$ oraz $y_1 \in \{1, 3\}$, czy fragment planszy od x_1-tej do x_2-tej kolumny włącznie można poprawnie pokryć zygzakiem, którego początek znajduje się na polu $P = (x_1, y_1)$, a koniec na polu $K = (x_2, 4 - y_1)$. Kształt takiego zygzaka jest wyznaczony jednoznacznie. Okazuje się, że – przy przyjętych przed chwilą założeniach – numeracja pól w zygzaku również jest zadana jednoznacznie! Faktycznie, numery kolejnych pól zygzaka rosną, począwszy od numeru $3(x_1 - 1) + 1$ na polu P, a skończywszy na numerze $3x_2$ na polu K.

Załóżmy, że mielibyśmy już tę funkcję. Wówczas całe rozwiązanie moglibyśmy zrealizować z użyciem jednej tablicy wartości logicznych $pokryte[0..n, 1..3]$, przy czym $pokryte[i, j]$ przechowuje informację, czy udało nam się pokryć fragment planszy złożony z pierwszych i kolumn za pomocą ścieżki kończącej się na polu (i, j). Zakładamy, że na początku cała tablica jest wypełniona wartościami **false**. Oto pseudokod:

```
1: pokryte[0,1] := pokryte[0,3] := true;
2: for i := 0 to n − 1 do
3:   for j ∈ {1,3} do
4:     if pokryte[i,j] then begin
5:       for k := i + 1 to n do
6:         if zygzak(i + 1, k, j) then
7:           pokryte[k, 4 − j] := true;
8:     end
9: return pokryte[n,1] or pokryte[n,3];
```

Jeśli będziemy umieli obliczać wartości funkcji *zygzak* w czasie stałym, to powyższy pseudokod będzie działał w pożądanym czasie $O(n^2)$.

Zajmijmy się więc funkcją *zygzak*. Oszczędzimy sobie dużo pracy, jeśli zauważymy, że każdy zygzak składa się z trzech *odcinków* położonych w poszczególnych wierszach planszy. A konkretnie, dla wywołania $\mathit{zygzak}(x_1, x_2, y_1)$ są to następujące odcinki:

- (x_1, y_1) – (x_2, y_1) o kolejnych numerach pól od $3x_1 - 2$ do $2x_1 + x_2 - 2$,
- $(x_2, 2)$ – $(x_1, 2)$ o kolejnych numerach pól od $2x_1 + x_2 - 1$ do $x_1 + 2x_2 - 1$,
- $(x_1, 4 - y_1)$ – $(x_2, 4 - y_1)$ o kolejnych numerach pól od $x_1 + 2x_2$ do $3x_2$.

Dla każdego z tych odcinków chcemy sprawdzić w czasie stałym, czy zadana numeracja pól planszy jest zgodna z numerami, które próbujemy przydzielić. Wykonamy to za pomocą dodatkowej tablicy *odcinek*, której wartości wyznaczymy zawczasu, znów za pomocą programowania dynamicznego. Aby zmieścić się w czasie i pamięci $O(n^2)$, musimy zaprojektować tę tablicę dosyć sprytnie.

Zrobimy to tak, aby $\mathit{odcinek}[x, y, k, v]$ oznaczało liczbę kolejnych pól planszy, począwszy od pola (x, y) i idąc w kierunku $k \in \{\texttt{lewo}, \texttt{prawo}\}$, których numery są zgodne z numeracją $v, v + 1, v + 2, \ldots$, przy czym pole (x, y) otrzymuje numer v. Formalnie, dla $k = \texttt{prawo}$ chcemy mieć:

$$\mathit{odcinek}[x, y, k, v] = \max\{l \geqslant 0 \,:\, \mathit{pasuje}(a[x, y], v) \wedge \ldots \wedge \mathit{pasuje}(a[x+l-1, y], v+l-1)\}$$

przy czym $\mathit{pasuje}(p, q)$ jest prawdziwe wtedy i tylko wtedy, gdy $p = 0$ lub $p = q$. Dla $k = \texttt{lewo}$ definiujemy to analogicznie.

Tak określona tablica ma $O(n^2)$ pól i możemy ją wypełnić wartościami w czasie $O(n^2)$. Należy tylko nie pogubić się w dużej liczbie jej wymiarów. Jeśli $k = \texttt{prawo}$, to elementy tablicy wyznaczamy od prawej do lewej:

```
1: for x := n downto 1 do
2:   for y := 1 to 3 do
3:     for v := 1 to 3n do
4:       if not pasuje(a[x, y], v) then
5:         odcinek[x, y, prawo, v] := 0
6:       else
7:         odcinek[x, y, prawo, v] := 1 + odcinek[x + 1, y, prawo, v + 1];
```

Przyjmujemy przy tym, że dla $x = n + 1$ mamy zawsze $\mathit{odcinek}[x, y, k, v] = 0$. Dla $k = \texttt{lewo}$ obliczenia wykonujemy analogicznie, tylko od lewej do prawej.

Mając wypełnioną tablicę *odcinek*, wynik funkcji *zygzak* obliczamy tak:

```
1: function zygzak(x1, x2, y1)
2: begin
3:   return (odcinek[x1, y1, prawo, 3x1 − 2] ⩾ x2 − x1 + 1) and
4:          (odcinek[x2, 2, lewo, 2x1 + x2 − 1] ⩾ x2 − x1 + 1) and
5:          (odcinek[x1, 4 − y1, prawo, x1 + 2x2] ⩾ x2 − x1 + 1);
6: end
```

Rozwiązanie wzorcowe

Aby otrzymać kompletne rozwiązanie, musimy jeszcze, bagatela, rozważyć wszystkie pozostałe konfiguracje ścieżki Hamiltona na kratce $3 \times n$. Na szczęście najtrudniejsze mamy już właściwie za sobą.

Po tym, jak szczegółowo rozpisaliśmy przypadek zygzaka, Czytelnik łatwo uwierzy, że dokładnie w ten sam sposób jesteśmy w stanie w czasie stałym sprawdzić każdy *zawijas*, a także *zawijasy drugiego typu* występujące w zakręconych ścieżkach Hamiltona.

Drugim ważnym elementem są ścieżki typu *tam i z powrotem*. Występują one albo samodzielnie, albo z zawijasem, albo wreszcie jako element składowy ścieżek zakręconych. Wprowadzimy dla nich dość ogólną tablicę wartości logicznych $tizp[x, y_1, y_2, k, v]$, której elementy oznaczają: czy istnieje ścieżka typu *tam i z powrotem* o końcach na polach (x, y_1) i (x, y_2), zawierająca wszystkie pola planszy położone w kierunku k od pól końcowych, przydzielająca polu (x, y_1) numer v i dalej numery rosnąco. Mamy: $x \in \{1, \ldots, n\}$, $y_1, y_2 \in \{1, 2, 3\}$, $y_1 \neq y_2$, $k \in \{\texttt{lewo}, \texttt{prawo}\}$ oraz $v \in \{1, \ldots, 3n\}$.

Tablica ta ma rozmiar $O(n^2)$. Aby wypełnić ją w czasie $O(n^2)$, skorzystamy z poczynionej już wcześniej obserwacji, że jeśli z fragmentu pokrytego ścieżką typu *tam i z powrotem* usuniemy skrajną kolumnę (dla $k = \texttt{prawo}$ będzie to skrajnie lewa kolumna), to otrzymamy ścieżkę tego samego typu dla fragmentu o jedną kolumnę krótszego. Aby zatem wyznaczyć wartości tablicy dla $k = \texttt{prawo}$, poruszamy się od prawej do lewej ($x = n, \ldots, 1$) i każdą wartość typu $tizp[x, y_1, y_2, k, v]$ obliczamy na podstawie jednej lub dwóch wartości typu $tizp[x+1, y'_1, y'_2, k, v']$. Wybór dwóch wartości mamy tylko wtedy, gdy $\{y_1, y_2\} = \{1, 3\}$, co odpowiada podjęciu decyzji, w którą stronę ma pójść *garb*. Dla $k = \texttt{lewo}$ postępujemy podobnie, tylko obliczenia wykonujemy od lewej do prawej.

W ten sposób omówiliśmy już wszystkie elementy składowe ścieżek. Na koniec pozostaje nam połączyć to wszystko w jedną całość. Przykładowo, pokażemy, w jaki sposób można rozważyć wszystkie ścieżki złożone z *zygzaków* i ścieżek typu *tam i z powrotem*:

```
 1: for x := 1 to n do
 2:    foreach y1, y2 ∈ {1, 2, 3}, y1 ≠ y2 do
 3:       if tizp[x, y1, y2, lewo, 1] then
 4:          pokryte[x, y2] := true;
 5: pokryte[0, 1] := pokryte[0, 3] := true;
 6: { Wypełnij resztę tablicy pokryte (programowanie dynamiczne dla zygzaków) }
 7: wynik := false;
 8: for x := 1 to n do
 9:    foreach y1, y2 ∈ {1, 2, 3}, y1 ≠ y2 do
10:       if tizp[x, y1, y2, prawo, 3x − 2] and pokryte[x − 1, y1] then
11:          wynik := true;
12: for y := 1 to 3 do
13:    wynik := wynik or pokryte[n, y];
14: return wynik;
```

Jeśli do powyższego pseudokodu dodać rozpatrywanie *zawijasów* w ścieżkach *tizp* (można to zrobić za pomocą analogicznej funkcji jak w przypadku *zygzaków*, wykorzystującej tablicę *odcinek*), otrzymamy rozwiązanie nieuwzględniające jeszcze tylko ścieżek *zakręconych*. Te ostatnie już tradycyjnie potraktujemy trochę po macoszemu i przemyślenie sposobu ich rozpatrzenia pozostawimy Czytelnikowi.

Na sam koniec rozwiązania pozostała nam ostatnia trudność, bardziej implementacyjna niż koncepcyjna, a mianowicie odtworzenie wyniku. Dotychczas poszukiwaliśmy jedynie odpowiedzi na pytanie, czy ścieżka istnieje (co skądinąd mamy zagwarantowane), a teraz musimy się zastanowić, jak tę ścieżkę znaleźć. Zastosujemy tu standardową metodę odtwarzania wyniku w programowaniu dynamicznym. Z każdą tablicą w rozwiązaniu przechowującą wartości logiczne zwiążemy dodatkową tablicę pomocniczą, tzw. tablicę ojców. Jeśli w jakimś polu tablicy logicznej wystąpi wartość **true**, to w odpowiadającym mu polu tablicy pomocniczej umieścimy informację o tym, na jakiej podstawie tę wartość wyznaczyliśmy. Przykładowo, w przypadku tablicy *pokryte* będzie to para liczb określająca indeksy pola tablicy, na podstawie którego uzyskaliśmy w danym polu wartość **true** (lub informacja, że pojawiła się ona z powodu ścieżki typu *tam i z powrotem*), natomiast w przypadku tablicy *tizp* może to być albo analogiczna piątka liczb określająca indeksy poprzedniego pola w tablicy, albo po prostu jedna liczba 0 lub 1 wskazująca, w którą stronę był skierowany ostatni *garb*. W każdym przypadku odpowiednią informację zapamiętamy przy okazji wykonywania głównego algorytmu programowania dynamicznego. Gdy teraz będziemy chcieli odtworzyć wynikową ścieżkę, wystarczy wycofać się po wartościach ojców.

Implementację rozwiązania wzorcowego można znaleźć w pliku `waz.cpp` (rekurencja ze spamiętywaniem) i `waz1.cpp` (programowanie dynamiczne). W plikach `wazs2.cpp` i `wazs7.cpp` zapisano niewiele gorsze rozwiązanie, o złożoności $O(n^3)$, za to nieco prostsze w implementacji. Pominięto w nim tablicę *odcinek*, a wszystkie *zygzaki* i *zawijasy* sprawdzano kolejno pole po polu.

Inne rozwiązania

Z racji podobieństwa zadania do problemu znajdowania ścieżki Hamiltona w grafie, w rozwiązaniu można próbować stosować techniki standardowe dla tego problemu.

Najprostsze rozwiązanie wykładnicze (`wazs1.cpp`) rekurencyjnie konstruuje ścieżkę, startując z każdego możliwego wierzchołka. Rozwiązanie to w pewnych przypadkach działa zupełnie nieźle, np. gdy plansza jest w całości wypełniona (wtedy jest to zwykłe przeszukiwanie) lub gdy jest pusta. Tego typu rozwiązania zdobywały na zawodach 20-30 punktów.

Inne podejście wykorzystuje fakt, że graf podany w zadaniu ma małą szerokość (równą 3). Jest to programowanie dynamiczne, w którym pojedynczy *stan* opisuje konfigurację na przekroju grafu, czyli w trzech wierzchołkach położonych w jednej kolumnie planszy. W takim stanie zakładamy, że ponumerowaliśmy już odpowiednio wszystko od tego stanu na lewo, i pamiętamy wszystkie informacje istotne z punktu widzenia konstruowania dalszej części ścieżki. Mogą to być np. następujące informacje:

- numer kolumny planszy (x),
- jakie trzy liczby wpisaliśmy w tę kolumnę planszy (trzyelementowa tablica t),

- jaki jest zbiór liczb A użytych dotychczas na planszy – nietrudno zauważyć, że jeśli już wpisane liczby dają się rozszerzyć do rozwiązania, to zbiór A wyraża się jako suma co najwyżej dwóch rozłącznych przedziałów,
- ilu sąsiadów na ścieżce Hamiltona brakuje każdemu z rozważanych trzech wierzchołków, po uwzględnieniu tej kolumny i wszystkiego, co było wcześniej – każdy wierzchołek ma co najwyżej dwóch sąsiadów na ścieżce Hamiltona (tablica *deg*).

Poniższej przedstawiamy fragment planszy wraz z odpowiadającym mu stanem:

17	18	19	20	21
16	13	12	1	2
15	14	11	10	9

- $x = 5$
- $t[1] = 9$, $t[2] = 2$, $t[3] = 21$
- $A = [1, 2] \cup [9, 21]$
- $deg[1] = deg[2] = deg[3] = 1$

Przygotowano kilka przykładowych implementacji tego typu podejścia (`wazs[3-6].cpp`), różniących się liczbą informacji pamiętanych w pojedynczym stanie (a co za tym idzie, liczbą wymiarów programowania dynamicznego) i sposobem przechodzenia między stanami. We wszystkich rozwiązaniach pamiętamy jedynie faktycznie dopuszczalne stany. W zależności od optymalizacji tego typu rozwiązania zdobywały na zawodach od 30 do nawet 60 punktów.

Warto jeszcze wyróżnić rozwiązanie `wazb1.cpp`, które przekraczało limit pamięciowy do tego zadania. Rozwiązanie wzorcowe zużywa mniej niż 200 MB pamięci. Jest tak jednak dzięki temu, że tablice stosowane w tym rozwiązaniu są zadeklarowane rozważnie, z użyciem typów całkowitych 1- i 2-bajtowych (tj. `char` i `short` w C++). Gdyby zamiast tego wszędzie używać rozrzutnie typu 4-bajtowego (`int`), można zużyć nawet ponad 600 MB pamięci, co właśnie ma miejsce w rozwiązaniu `wazb1.cpp`. Limit pamięciowy w zadaniu (512 MB) był dobrany tak, aby zmieszczenie się w nim wymagało jedynie elementarnej rozwagi w tym zakresie.

Testy

W tym zadaniu zaproponowano dość dużą liczbę testów, jednak niezgrupowanych, z których każdy był wart od 1 do 4 punktów. Każdy z testów można w pewnym sensie traktować jako osobną łamigłówkę.

W większości testy były generowane zgodnie z konfiguracjami z rozwiązania wzorcowego poprzez wyzerowanie pewnych losowych pól z planszy. Najtrudniejsze testy charakteryzują się niewielką liczbą pól ze znanymi numerami fragmentów węża (np. ścieżkę *zakręconą* można wymusić już za pomocą wartości trzech pól). W innych testach złośliwe ułożenie pól o znanych wartościach miało pomóc sprawdzić efektywność rozwiązań. Przykładowo, w przypadku ścieżki typu *tam i z powrotem* podanie wartości pól tylko w górnym lub tylko w dolnym wierszu determinuje całą konfigurację, jednak jeśli pójdzie się od drugiej strony, rozwiązanie wykładnicze będzie długo błądzić. Wreszcie w zestawie znalazły się testy z pustą lub prawie pustą planszą i testy z planszą całkowicie wypełnioną.

Zawody II stopnia

opracowania zadań

Adam Polak — Treść zadania, Opracowanie
Lech Duraj — Program

Dostępna pamięć: 64 MB. *OI, etap II, dzień próbny, 11.02.2014*

Karty

Na stole leży n kart ułożonych w pewnej kolejności. Na każdej karcie zapisane są dwie liczby całkowite: jedna z nich na jednej stronie karty (na awersie), a druga na drugiej stronie karty (na rewersie). Początkowo wszystkie karty leżą na stole awersem do góry. Iluzjonista Bajtazar zamierza przedstawić (i to wielokrotnie!) swój popisowy Wielki Trik z Wyszukiwaniem Binarnym. Aby jednak mógł go zaprezentować, ciąg liczb widocznych na stole musi być niemalejący. W tym celu Bajtazar być może będzie musiał odwrócić niektóre karty tak, aby widoczne były liczby znajdujące się na ich rewersach.

Trik Bajtazara wymaga udziału osoby z publiczności. Jednak niektórzy zgłaszający się na ochotnika widzowie są podstawieni przez konkurentów Bajtazara. Każdy z nich, wchodząc na scenę, błyskawicznym ruchem zamieni ze sobą miejscami dwie spośród leżących na stole kart. Po każdej z takich zamian Bajtazar może znowu odwrócić niektóre karty na drugą stronę, ale nawet mimo tego może nie być w stanie wykonać triku. Będzie wtedy zmuszony wrócić do tradycyjnych metod zabawiania widzów, z udziałem królików i kapeluszy.

Napisz program, który określi, po każdej zamianie kart miejscami, czy Bajtazar może wykonać swoją sztuczkę.

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita n ($2 \leqslant n \leqslant 200\,000$), oznaczająca liczbę kart. W kolejnych n wierszach opisane są karty, po jednej w wierszu, w takiej kolejności, w jakiej leżą na stole. W i-tym z tych wierszy znajdują się dwie liczby całkowite x_i i y_i ($0 \leqslant x_i, y_i \leqslant 10^7$) oddzielone pojedynczym odstępem. Są to liczby zapisane na i-tej karcie: x_i oznacza liczbę zapisaną na awersie tej karty, a y_i oznacza liczbę zapisaną na jej rewersie. Początkowy ciąg kart nie musi pozwalać na wykonanie triku.

W następnym wierszu wejścia znajduje się jedna liczba całkowita m ($1 \leqslant m \leqslant 1\,000\,000$), oznaczająca liczbę zamian. W kolejnych m wierszach opisane są zamiany: j-ty z tych wierszy zawiera dwie liczby całkowite a_j i b_j ($1 \leqslant a_j, b_j \leqslant n$) oddzielone pojedynczym odstępem, oznaczające, że j-ty z zaproszonych na scenę widzów zamieni miejscami a_j-tą i b_j-tą kartę.

W testach wartych 30% punktów zachodzi dodatkowy warunek: liczby po obu stronach kart są identyczne. W (być może innych) testach wartych 38% punktów zachodzi dodatkowy warunek $n \leqslant 2000$.

Wyjście

Twój program powinien wypisać na standardowe wyjście m wierszy, każdy zawierający pojedyncze słowo `TAK` *lub* `NIE`*. W j-tym wierszu powinno znaleźć się słowo* `TAK`*, jeśli Bajtazar może, po j-tej zamianie kart, ułożyć ciąg niemalejący, odwracając niektóre karty. W przeciwnym wypadku w tym wierszu powinno znaleźć się słowo* `NIE`*.*

Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
NIE
TAK
```

Testy „ocen”:

`1ocen`: *$n = 6$, $m = 6$, mały test poprawnościowy;*

`2ocen`: *$n = 7$, $m = 9$, liczby po obu stronach kart są identyczne;*

`3ocen`: *$n = 200\ 000$, $m = 1\ 000\ 000$, każda zamiana o parzystym numerze cofa poprzednią.*

Rozwiązanie

W zadaniu mamy do czynienia z ciągiem n par liczb (x_i, y_i). Na ciągu tym m razy wykonywana jest operacja zamiany miejscami dwóch (niekoniecznie sąsiednich) par. Po każdej operacji musimy odpowiedzieć na pytanie, czy da się z każdej pary wybrać po jednej liczbie tak, by powstały w ten sposób ciąg liczb był niemalejący.

Rozwiązanie wzorcowe

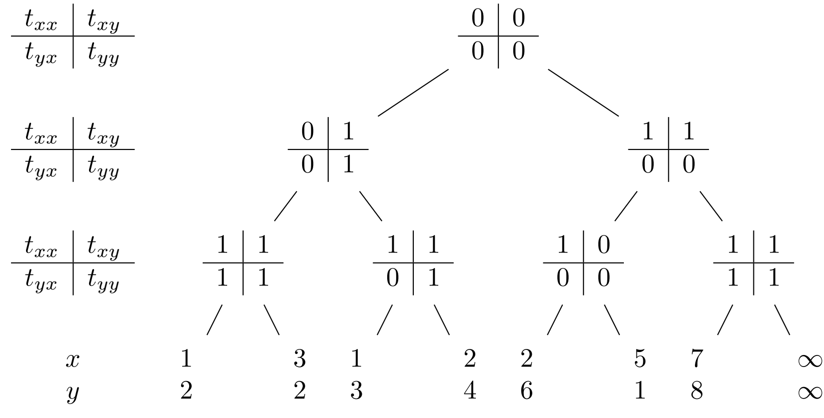
Rozwiązanie opiera się na strukturze danych nazywanej statycznym drzewem przedziałowym[1]. Jest to pełne drzewo binarne, którego liście odpowiadają wyrazom rozważanego przez nas ciągu, zaś węzły wewnętrzne – pewnym jego spójnym podciągom. Węzeł wewnętrzny zawsze reprezentuje podciąg powstały przez sklejenie podciągów reprezentowanych przez synów tego węzła.

Dla uproszczenia załóżmy, że długość ciągu n jest potęgą dwójki. Jeśli tak nie jest, możemy dopisać na końcu ciągu odpowiednią liczbę par, których oba elementy są równe nieskończoność (w praktyce, przy ograniczeniach z treści zadania, za nieskończoność wystarczy przyjąć 10^7). Jak łatwo zauważyć, wydłuży to ciąg co najwyżej dwukrotnie oraz nie zmieni odpowiedzi na postawione w zadaniu pytania.

W rozwiązaniu wzorcowym z każdym węzłem drzewa w skojarzone są cztery zmienne logiczne $t_{xx}[w]$, $t_{xy}[w]$, $t_{yx}[w]$, $t_{yy}[w]$. Załóżmy, że węzeł w reprezentuje podciąg od i-tego do j-tego wyrazu ciągu włącznie. Wówczas zmienna $t_{xx}[w]$ przyjmuje

[1] Statyczne drzewa przedziałowe pojawiały się już wielokrotnie na Olimpiadzie Informatycznej. Znane są również pod nazwą drzew licznikowych oraz drzew turniejowych. Można się o nich co nieco dowiedzieć na stronie `http://was.zaa.mimuw.edu.pl`.

wartość *prawda* wtedy i tylko wtedy, gdy z par w podciągu reprezentowanym przez w da się wybrać po jednej liczbie tak, by powstały ciąg był niemalejący, przy czym z pierwszej pary (x_i, y_i) wybieramy x_i, a z ostatniej pary (x_j, y_j) wybieramy x_j. Pozostałe trzy zmienne są zdefiniowane analogicznie.



Rys. 1: Przykład ilustrujący strukturę danych wykorzystywaną w rozwiązaniu. Wartości logiczne *prawda* są oznaczane jako 1, a *fałsz* jako 0.

Znając wartości zmiennych dla synów, możemy szybko (w czasie stałym) obliczyć je również dla ojca. Na przykład, jeśli synami (odpowiednio lewym i prawym) węzła w są węzły l i p, przedział reprezentowany przez węzeł l kończy się na wyrazie i-tym, a przedział reprezentowany przez węzeł p zaczyna się na wyrazie $(i+1)$-szym, to

$$\begin{aligned} t_{xx}[w] = & (t_{xx}[l] \wedge t_{xx}[p] \wedge x_i \leqslant x_{i+1}) \vee \\ & (t_{xx}[l] \wedge t_{yx}[p] \wedge x_i \leqslant y_{i+1}) \vee \\ & (t_{xy}[l] \wedge t_{xx}[p] \wedge y_i \leqslant x_{i+1}) \vee \\ & (t_{xy}[l] \wedge t_{yx}[p] \wedge y_i \leqslant y_{i+1}). \end{aligned}$$

Pozostałe trzy zmienne możemy obliczyć analogicznie.

Na początku rozwiązania konstruujemy drzewo dla wyjściowego ciągu, odwiedzając węzły w kolejności od dołu do góry i obliczając skojarzone z nimi zmienne. Konstrukcja drzewa odbywa się zatem w czasie $O(n)$, gdyż taka jest liczba węzłów drzewa.

Każda operacja zamiany, którą mamy do wykonania, powoduje zmianę wartości dwóch wyrazów ciągu. Po wykonaniu takiej operacji niektóre węzły w drzewie mogą mieć nieaktualne wartości zmiennych – są to jednak wyłącznie węzły leżące na ścieżkach od zmodyfikowanych liści do korzenia drzewa. Możemy „naprawić" te węzły, odwiedzając je od dołu do góry i obliczając na nowo wartości skojarzonych z nimi zmiennych, co zajmuje czas $O(\log n)$, gdyż taka jest wysokość drzewa.

Po wykonaniu operacji możemy odpowiedzieć na postawione w zadaniu pytanie, sprawdzając, czy którakolwiek z czterech zmiennych w korzeniu ma wartość *prawda*.

Przedstawione rozwiązanie działa w czasie $O(n + m \log n)$ i pamięci $O(n)$. Jego implementacje znajdują się w plikach `kar.cpp`, `kar1.pas` oraz `kar2.cpp`.

W trakcie zawodów żaden z zawodników nie nadesłał poprawnego rozwiązania, które byłoby istotnie różne od powyższego.

Rozwiązania powolne

Najprostsze rozwiązanie po wykonaniu każdej operacji zamiany sprawdza wszystkie 2^n sposobów wyboru elementów par w poszukiwaniu takiego, który utworzy ciąg niemalejący. Takie rozwiązanie działa w czasie $O(mn2^n)$ i dostaje 15 punktów. Implementacja znajduje się w pliku `kars6.cpp`.

Nieco bardziej skomplikowane, ale za to dużo szybsze rozwiązanie również oblicza odpowiedzi na pytania niezależnie po każdej operacji zamiany. Czyni to jednak w czasie liniowym. Przegląda ciąg od lewej do prawej, obliczając dla i-tego wyrazu ciągu liczbę p_i – najmniejszą liczbę, jaką może kończyć się ciąg niemalejący wybrany z początkowego fragmentu wyjściowego ciągu kończącego się i-tym wyrazem. Wartości p_i obliczamy zgodnie z wzorami:

$$p_1 = \min(x_1, y_1),$$

$$p_i = \begin{cases} \min(x_i, y_i) & \text{jeśli } p_{i-1} \leqslant \min(x_i, y_i), \\ \max(x_i, y_i) & \text{jeśli } \min(x_i, y_i) < p_{i-1} \leqslant \max(x_i, y_i), \\ \text{odpowiedź } \texttt{NIE} & \text{w przeciwnym przypadku.} \end{cases}$$

Jeżeli dla pewnego i obie liczby x_i i y_i są mniejsze niż poprzednio obliczone p_{i-1}, to wiemy już, że nie da się wybrać ciągu niemalejącego; możemy więc przerwać algorytm i odpowiedzieć `NIE`. Jeżeli natomiast przejrzymy cały ciąg, nie napotykając powyższego problemu, odpowiedź brzmi `TAK`. Takie rozwiązanie działa w czasie $O(mn)$ i dostaje 38 punktów. Implementacja znajduje się w plikach `kars1.cpp` i `kars4.pas`.

Rozwiązanie wariantu z dodatkowym warunkiem $x_i = y_i$

Zgodnie z treścią zadania, w testach wartych 30 punktów zachodzi dodatkowy warunek $x_i = y_i$. W takiej wersji zadanie można rozwiązać dużo prościej, jednak rozwiązanie to ma niewiele wspólnego z rozwiązaniem oryginalnego problemu.

Z tym dodatkowym warunkiem zadanie sprowadza się do następującego: na danym ciągu liczb wykonujemy operacje zamiany dwóch wyrazów i po każdej takiej operacji musimy stwierdzić, czy ciąg jest posortowany niemalejąco. W rozwiązaniu wystarczy pamiętać liczbę takich wyrazów, które są większe od następnego. Ciąg jest posortowany wtedy i tylko wtedy, gdy liczba ta jest równa zero. Zamiana wyrazów o numerach a i b powoduje zmianę statusu – z „większy od następnego” na „nie większy od następnego” lub na odwrót – co najwyżej czterech wyrazów: $a-1$, a, $b-1$ oraz b. Łatwo więc zaktualizować pamiętaną liczbę w czasie stałym.

Przedstawione rozwiązanie działa w czasie $O(n+m)$ i jest zaimplementowane w pliku `karb1.cpp`.

Testy

Przygotowano 13 zestawów testów. Zestawy, w których zachodzi dodatkowy warunek $x_i = y_i$, nazywane są dalej *jednostronnymi*, zaś pozostałe *dwustronnymi*. Testy były tworzone z użyciem następujących technik.

Testy sortujące: Dla każdej karty wybierana jest jedna z dwóch stron, a następnie przez kolejne zamiany ciąg jest sortowany tak, aby był niemalejący względem wartości kart na tych stronach. Sortowanie przebiega jednym z klasycznych algorytmów: przez selekcję, wstawianie, kopcowanie lub sortowanie szybkie. Operacja ta jest powtarzana kilkukrotnie, z różnym wyborem stron kart. W testach sortujących dominują zdecydowanie odpowiedzi `NIE`. W każdym zestawie znajdują się dwa testy sortujące – z małym i dużym zakresem wartości kart – odpowiednio testy a i b.

Testy jednostronne specjalne: Znajdują się w zestawach jednostronnych, jako testy c. Składają się w większości z zamian jednakowych kart, a także krótkich ciągów wracających do punktu wyjścia – wszystko po to, aby mieć stosunkowo dużo odpowiedzi `TAK`.

Testy typu „flipper”: Znajdują się w zestawach dwustronnych, jako testy c. Zamieniają ze sobą tylko trzy karty – początkową, środkową i końcową. Test jest tak skonstruowany, aby zamiany powodowały konieczność obrócenia dużej liczby pozostałych kart.

Testy permutacyjne: Znajdują się w zestawach dwustronnych, jako testy d. Ciągi wartości po jednej i po drugiej stronie kart stanowią permutację $\{1, 2, \ldots, n\}$; zamiany dokonywane są po cyklach permutacji.

W zestawie 1 znajduje się dodatkowo ręcznie utworzony test e.

Paweł Parys — Treść zadania, Opracowanie
Karol Pokorski — Program

Dostępna pamięć: 64 MB. *OI, etap II, dzień pierwszy, 12.02.2014*

Przestępcy

Bajtogród jest pięknym miastem położonym nad rzeką. Znajduje się w nim n domów, ponumerowanych kolejnymi liczbami naturalnymi od 1 do n, i w tej kolejności są ułożone wzdłuż brzegu rzeki.

Bajtogród był bardzo spokojnym miastem, w którym wszystkim dobrze się powodziło. Niestety, od niedawna grasuje tam gang dwóch groźnych przestępców – Bitka i Bajtka. Dokonali już wielu napadów, przez co mieszkańcy boją się wychodzić z domów.

W trakcie rabunku, Bitek i Bajtek wychodzą ze swoich domów i idą naprzeciw siebie – żaden z nich się nie cofa. Bitek idzie w kierunku rosnących numerów, a Bajtek w kierunku malejących numerów domów. Po drodze (zanim się spotkają) każdy z nich wybiera kilka domów, do których się włamuje i kradnie cenne przedmioty (i ważne dane). Po dokonaniu napadów spotykają się w pewnym domu i dzielą się łupami. Mieszkańcy Bajtogrodu mają tego dość – wszyscy chcieliby, aby w mieście ponownie zapanował spokój. Poprosili o pomoc detektywa Bajtoniego.

Detektyw ustalił, że bandyci mieszkają w domach tego samego koloru, niestety nie wiadomo dokładnie w których, ani jaki to kolor. Dodatkowo, anonimowy informator powiedział detektywowi, że bandyci są właśnie w trakcie kolejnego skoku. Z obawy o swoje bezpieczeństwo, informator nie podał dokładnie, w których domach ma nastąpić włamanie. Podał jednak kolory kolejno rabowanych domów. Okazało się, że bandyci są dość przesądni – każdy bandyta obrabuje dom każdego koloru co najwyżej raz.

Bajtoni nie może już dłużej czekać. Chce urządzić zasadzkę w miejscu spotkania przestępców. Pomóż Bajtoniemu, pisząc program, który znajdzie wszystkie możliwe miejsca spotkania przestępców.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie liczby całkowite n oraz k ($3 \leqslant n \leqslant 1\,000\,000$, $1 \leqslant k \leqslant 1\,000\,000$, $k \leqslant n$) oddzielone pojedynczym odstępem, oznaczające liczbę domów oraz liczbę kolorów domów w Bajtogrodzie. Kolory są ponumerowane kolejnymi liczbami naturalnymi od 1 do k. W drugim wierszu wejścia znajduje się ciąg n liczb całkowitych $c_1, c_2, \ldots, c_n$ ($1 \leqslant c_i \leqslant k$), pooddzielanych pojedynczymi odstępami. Są to kolory kolejnych domów w Bajtogrodzie.

W trzecim wierszu wejścia znajdują się dwie liczby całkowite m oraz l ($1 \leqslant m, l \leqslant n$, $m + l \leqslant n - 1$) oddzielone pojedynczym odstępem i określające liczbę domów odwiedzonych kolejno przez Bitka i Bajtka. W czwartym wierszu wejścia znajduje się m parami różnych liczb całkowitych $x_1, x_2, \ldots, x_m$ ($1 \leqslant x_i \leqslant k$), pooddzielanych pojedynczymi odstępami. Są to kolory domów odwiedzonych przez Bitka podane w kolejności odwiedzania (wyłączając dom Bitka). W piątym i ostatnim wierszu wejścia znajduje się l parami różnych liczb całkowitych $y_1, y_2, \ldots, y_l$ ($1 \leqslant y_i \leqslant k$), pooddzielanych pojedynczymi odstępami. Są to kolory domów odwiedzonych przez Bajtka podane w kolejności odwiedzania (wyłączając dom Bajtka). Zachodzi warunek $x_m = y_l$, jest to kolor domu, w którym bandyci będą dzielić łupy.

Wyjście

Twój program powinien wypisać na standardowe wyjście dokładnie dwa wiersze. W pierwszym wierszu powinna znaleźć się liczba domów, w których zgodnie z warunkami zadania mogą spotkać się bandyci. W drugim wierszu powinien znaleźć się rosnący ciąg numerów tych domów, pooddzielanych pojedynczymi odstępami. Jeśli bandyci nie mogą się spotkać w żadnym miejscu, to pierwszy wiersz ma zawierać liczbę 0, a drugi wiersz ma być pusty.

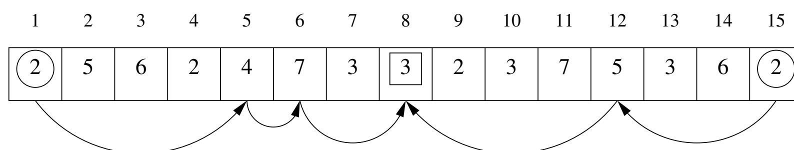
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
3
7 8 10
```



Wyjaśnienie do przykładu: *W powyższym przykładzie bandyci mogą mieszkać w domach o kolorach 2 (Bitek mieszkałby wtedy w domu o numerze 1 lub 4, a Bajtek w domu numer 15) lub 6 (Bitek mieszkałby w domu numer 3, natomiast Bajtek w domu numer 14). Przykładowy scenariusz drogi Bitka (z domu numer 1) jest następujący: odwiedza domy numer: 5 (koloru 4), 6 (koloru 7), a potem 7, 8 lub 10 (koloru 3). Bajtek może wtedy rabować dom numer 12 (koloru 5), a następnie spotkać się z Bitkiem w domu 7, 8 lub 10 (koloru 3). Powyższy rysunek przedstawia sytuację, w której bandyci spotykają się w domu numer 8.*

Testy „ocen”:

`1ocen`: $n = 7$, $k = 3$, $m = 2$, $l = 3$, *bandyci mogą dzielić łupy tylko w jednym domu;*

`2ocen`: $n = 10$, $k = 3$, $m = 3$, $l = 2$, *bandyci nie mogą się spotkać w żadnym miejscu;*

`3ocen`: $n = 1000$, $k = 1$, $m = 1$, $l = 1$, *wszystkie domy są tego samego koloru;*

`4ocen`: $n = 1\,000\,000$, $k = 1000$, $m = l = 100$, *ciąg domów składa się z 1000 identycznych fragmentów po 1000 domów, pokolorowanych kolejnymi kolorami od 1 do 1000; ciągi domów odwiedzanych przez obu bandytów są postaci* $1, 2, 3, \ldots, 100$.

Rozwiązanie

Rozwiązanie wzorcowe

Siłowym sposobem obliczenia wyniku byłoby sprawdzanie wszystkich możliwych par domów, w których mogą mieszkać Bitek i Bajtek, a dla każdej z nich wszystkich możliwych ciągów kolejno napadanych domów, spełniających warunki zadania. Jednak program działający w ten sposób byłby oczywiście bardzo wolny, dlatego powinniśmy najpierw zastanowić się, jak zmniejszyć liczbę możliwości, które musimy przeanalizować. Jak można się domyślać z rozmiaru danych na wejściu, rozwiązanie powinno działać w czasie liniowym względem n.

Po pierwsze zauważmy, że jeśli bandyci mieszkają w domach o jakimś kolorze c, to możemy założyć, że Bitek mieszka w pierwszym od lewej domu tego koloru, a Bajtek w ostatnim z nich (stosujemy tutaj konwencję, że dom o numerze 1 znajduje się po lewej stronie, a dom o numerze n po prawej). Istotnie, jeśli Bitek może napaść na pewien ciąg domów, zaczynając z domu o numerze i, to może napaść na ten sam ciąg domów, także zaczynając z domu o mniejszym numerze; analogicznie, lecz odwrotnie, jest z Bajtkiem.

Podobnie rzecz ma się z kolejnymi domami odwiedzanymi przez bandytów: jeśli Bitek jest w domu o numerze i i ma napaść na dom o ustalonym kolorze c, to może wybrać pierwszy dom po i mający kolor c; analogicznie dla Bajtka. Istotnie, możliwości kolejnych rabunków z tego domu mogą być tylko większe niż z dalszych domów o tym samym kolorze. Nie dotyczy to oczywiście ostatniego napadanego domu (czyli miejsca spotkania), gdyż w jego przypadku interesuje nas wyznaczenie wszystkich możliwości.

Widzimy więc, że chcemy rozważać ciągi $s_0, s_1, \ldots, s_{m-1}$ numerów domów odwiedzanych przez Bitka (przy czym s_0 to miejsce jego zamieszkania, natomiast miejsce spotkania jest pominięte) spełniające następujące warunki:

- $1 \leqslant s_0 < s_1 < \ldots < s_{m-1} \leqslant n$, czyli Bitek idzie tylko w prawo;
- $c_{s_i} = x_i$ dla każdego $i \in \{1, \ldots, m-1\}$, czyli napadane domy mają kolory jak podano na wejściu;
- $c_j \neq c_{s_0}$ dla każdego $j \in \{1, \ldots, s_0 - 1\}$, czyli Bitek mieszka w pierwszym domu pewnego koloru;
- $c_j \neq x_i$ dla wszystkich $i \in \{1, \ldots, m-1\}$ oraz $j \in \{s_{i-1}+1, \ldots, s_i - 1\}$, czyli napadany jest zawsze pierwszy możliwy dom danego koloru (nie dotyczy to ostatniego napadu – miejsca spotkania).

Ciąg taki nazwiemy *dobrym ciągiem Bitka*. Podobnie, *dobrym ciągiem Bajtka* nazwiemy ciąg $r_0, r_1, \ldots, r_{l-1}$ taki, że:

- $1 \leqslant r_{l-1} < r_{l-2} < \ldots < r_0 \leqslant n$;
- $c_{r_i} = y_i$ dla każdego $i \in \{1, \ldots, l-1\}$;
- $c_j \neq c_{r_0}$ dla każdego $j \in \{r_0 + 1, \ldots, n\}$;

- $c_j \neq y_i$ dla wszystkich $i \in \{1, \ldots, l-1\}$ oraz $j \in \{r_i + 1, \ldots, r_{i-1} - 1\}$.

Bandyci mogą spotkać się w domu numer a, jeśli istnieje dobry ciąg Bitka $s_0, s_1, \ldots, s_{m-1}$ oraz dobry ciąg Bajtka $r_0, r_1, \ldots, r_{l-1}$ takie, że

- $c_{s_0} = c_{r_0}$, czyli bandyci mieszkają w domu o tym samym kolorze;
- $c_a = x_m$, czyli miejsce spotkania ma podany kolor;
- $s_{m-1} < a < r_{l-1}$, czyli Bitek przychodzi na miejsce spotkania z lewej, a Bajtek z prawej.

Wprost z powyższej definicji wynika, że dla ustalonego koloru c istnieje co najwyżej jeden dobry ciąg Bitka zaczynający się od domu o kolorze c (tzn. taki, że $c_{s_0} = c$). Łączna długość wszystkich takich ciągów, dla wszystkich c, jest rzędu $O(k \cdot m)$, więc wyznaczenie ich wszystkich w całości byłoby zbyt wolne. Widzimy jednak, że cały ciąg nie jest nam potrzebny, wystarczy znać ostatni jego element, tzn. s_{m-1}; oznaczmy go $d[c]$. Podobnie, przez $e[c]$ oznaczmy ostatni element dobrego ciągu Bajtka $r_0, r_1, \ldots, r_{l-1}$ takiego, że $c_{r_0} = c$. (Dla niektórych kolorów c wartość $d[c]$ lub $e[c]$ może być niezdefiniowana, jeśli żaden dobry ciąg nie istnieje). Daje to nam podział zadania na dwa kroki:

1. Dla każdego c oblicz $d[c]$ oraz $e[c]$.
2. Znajdź wszystkie a takie, że $c_a = x_m$ oraz $d[c] < a < e[c]$ dla pewnego c.

Zacznijmy od prostszej części, czyli od rozwiązania kroku 2. Mamy tutaj dane pewne przedziały: dla każdego c przedział od $d[c]$ do $e[c]$. Chcemy znaleźć wszystkie a należące któregokolwiek z tych przedziałów (spośród nich odfiltrujemy w trywialny sposób tylko takie, że $c_a = x_m$). Na początek odrzućmy takie c, że $d[c]$ lub $e[c]$ jest niezdefiniowane lub $d[c]$ nie jest mniejsze niż $e[c]$. Następnie posortujmy wszystkie początki oraz końce przedziałów (czyli liczby $d[c]$ oraz $e[c]$). Aby uzyskać złożoność liniową można użyć sortowania kubełkowego, aczkolwiek w praktyce spokojnie wystarczy QuickSort z biblioteki standardowej. Teraz będziemy iść od lewej do prawej po kolejnych numerach domów, trzymając licznik ilości otwartych przedziałów. Gdy zaczyna się nowy przedział, licznik zwiększamy; gdy jakiś przedział się kończy – zmniejszamy; jeśli licznik ma wartość dodatnią, to numer analizowanego domu możemy wziąć jako a: znajduje się on wewnątrz pewnego przedziału. Nie musimy się przy tym zastanawiać nad faktem czy przedziały są otwarte czy domknięte, gdyż końce przedziałów mają kolor x_{m-1} lub y_{l-1}, który z założenia jest inny niż x_m.

Wróćmy teraz do kroku 1. Tutaj potrzeba minimalnej ilości sprytu, gdyż liczenie dobrych ciągów dla każdego koloru z osobna byłoby za wolne. Skoncentrujmy się na dobrych ciągach Bitka, czyli na liczbach $d[c]$; liczby $e[c]$ oblicza się analogicznie. Będziemy wyznaczać dobre ciągi Bitka po kolei dla wszystkich możliwych s_0, idąc od lewej do prawej. Dla $s_0 = 1$ robimy to zupełnie wprost: znajdujemy najmniejszy numer domu $s_1 > s_0$ mającego kolor x_1, następnie najmniejszy numer domu $s_2 > s_1$ mającego kolor x_2, itd. W ten sposób obliczyliśmy $d[c_1]$. Następnie (w pętli) przesuwamy s_0 na kolejny dom, nie zapominając ani nie zmieniając liczb $s_1, \ldots, s_{m-1}$. Interesują nas przy tym tylko takie s_0, że żaden dom wcześniej nie ma takiego samego koloru jak

dom numer s_0. Oczywiście po takim zwiększeniu liczby s_0 nasz ciąg być może nie jest już dobrym ciągiem Bitka; trzeba go wówczas poprawić. Robimy to tak: dla kolejnych i (zaczynając od 1, aż do $m-1$) sprawdzamy, czy $s_{i-1} < s_i$. Jeśli dla pewnego i jest to prawdą, to przerywamy poprawianie: dostaliśmy dobry ciąg Bitka, czyli jako $d[c_{s_0}]$ powinniśmy wziąć s_{m-1}. A jeśli nie, to przesuwamy s_i na kolejny dom tego samego koloru, aż do momentu, gdy liczba s_i stanie się większa od s_{i-1}. Po takim przesuwaniu musimy sprawdzić kolejne i, gdyż teraz być może przestało być prawdą, że $s_i < s_{i+1}$.

Dość łatwo przekonać się o poprawności tej metody. W momencie, gdy sprawdzamy czy $s_{i-1} < s_i$, to dzięki wcześniejszemu poprawianiu wiemy, że $s_0 < s_1 < \ldots < s_{i-1}$, natomiast z poprzedniego kroku dużej pętli wiemy, że $s_i < \ldots < s_{m-1}$; jeśli dodatkowo okazuje się, że $s_{i-1} < s_i$, to kolejność wszystkich elementów ciągu staje się prawidłowa. W dodatku widzimy, że dzięki naszej „zachłanności”, czyli wybieraniu zawsze pierwszej możliwości od lewej, pozostałe warunki dobrego ciągu będą spełnione.

Pozostaje drobny szczegół implementacyjny. Otóż, aby nasz algorytm działał szybko, powinniśmy umieć w czasie stałym przesunąć s_i na numer kolejnego domu tego samego koloru. Aby to było możliwe, musimy po prostu wyliczyć sobie na początku dla każdego domu wskaźnik na kolejny dom tego samego koloru. Łatwo to zrobić idąc od lewej do prawej i pamiętając dla każdego koloru ostatni dotychczas widziany dom tego koloru. Wtedy dla kolejno analizowanego domu mamy numer poprzedniego domu w takim samym kolorze; tworzymy między nimi wskaźnik. Przy okazji dla każdego domu możemy sobie zaznaczyć, czy istnieje wcześniejszy dom tego samego koloru – ta informacja jest przydatna podczas przesuwania s_0.

Przeanalizujmy teraz kwestię złożoności naszego algorytmu (dla kroku 1). Kluczowe jest tu założenie z treści zadania, że liczby $x_1, \ldots, x_m$ są parami różne. Zatem jako s_i próbujemy brać numery domu innego koloru niż jako s_j dla $j \neq i$ (gdzie $i, j \geqslant 1$). Innymi słowy, każdy numer domu pojawi się najwyżej raz jako s_0, oraz najwyżej raz jako pewien s_i dla $i \geqslant 1$. Ponadto dla każdej wartości s_0 być może dla wielu wartości i zwiększamy liczbę s_i, ale tylko dla jednego i wykonujemy porównanie s_i z s_{i-1} niekończące się przesunięciem s_i. Wynika z tego, że nasz algorytm działa w czasie liniowym, czyli tak jak chcieliśmy.

Rozwiązanie wzorcowe można znaleźć w plikach `prz.cpp` i `prz1.pas`. Wydaje się, że rozwiązania optymalne muszą działać z grubsza według powyższego opisu.

Rozwiązania nieoptymalne i błędne

Można próbować oczywiście naiwnie poprawiać dobre ciągi Bitka i Bajtka. Po pierwsze, bez użycia tablicy wskaźników do kolejnego domu tego samego koloru – idąc po prostu po kolejnych domach i sprawdzając, czy mają właściwy kolor. Wówczas każdym z m elementów ciągu odwiedzimy praktycznie wszystkie n domów. Po drugie, możemy nie przerywać poprawiania, gdy okaże się, że $s_{i-1} < s_i$ – czyli sprawdzać ten warunek zawsze dla wszystkich i. W tym przypadku też prawie dla każdego z n domów (jako elementu s_0) przejrzymy cały ciąg długości m. Oba te rozwiązania prowadzą do istotnie gorszej, kwadratowej złożoności obliczeniowej. Testy zostały tak dobrane, aby takie rozwiązania otrzymywały co najwyżej 40% maksymalnej punktacji. Rozwią-

zania pierwszego typu można znaleźć w plikach `przs1.cpp` i `przs2.pas`, a drugiego typu w plikach `przs3.cpp` i `przs4.pas`.

Podstawowym błędem może być pominięcie warunku o tym, że kolory domów bandytów muszą być jednakowe. Takie rozwiązania pomijają chyba najważniejszą trudność tego zadania, dlatego intencją Jury było, aby otrzymały 0 punktów. Przykładowa implementacja znajduje się w pliku `przb1.cpp`.

Michał Włodarczyk — Treść zadania, Opracowanie
Karol Pokorski — Program

Dostępna pamięć: 256 MB. *OI, etap II, dzień pierwszy, 12.02.2014*

Superkomputer

Bajtazar stworzył superkomputer o bardzo nowatorskiej konstrukcji. Może on zawierać wiele (takich samych) procesorów. Każdy procesor może w jednostce czasu wykonać jedną instrukcję.

Programy nie są w tym komputerze wykonywane sekwencyjnie, lecz mają strukturę drzewa. Każda instrukcja programu może mieć zero, jedną lub wiele **instrukcji następujących** *po niej. Po wykonaniu każdej instrukcji należy wykonać instrukcje następujące po niej, jednak można je wykonać w dowolnej kolejności; w szczególności można je wykonywać równolegle na różnych procesorach. Jeśli w superkomputerze jest k procesorów, to w każdej jednostce czasu będzie można wykonać równolegle co najwyżej k instrukcji.*

Bajtazar ma do uruchomienia pewien program. Ponieważ chce optymalnie wykorzystywać swoje zasoby, zastanawia się, jak liczba procesorów wpłynie na szybkość obliczeń. Prosi Cię o określenie, dla danego programu i liczby procesorów, najkrótszego możliwego czasu wykonania tego programu z użyciem superkomputera o tylu procesorach.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie liczby całkowite n i q ($1 \leqslant n, q \leqslant 1\,000\,000$) oddzielone pojedynczym odstępem, oznaczające liczbę instrukcji programu Bajtazara oraz liczbę zapytań.

W drugim wierszu wejścia znajduje się ciąg q liczb całkowitych $k_1, k_2, \ldots, k_q$ ($1 \leqslant k_i \leqslant 1\,000\,000$) pooddzielanych pojedynczymi odstępami: k_i oznacza liczbę procesorów, którymi dysponuje Bajtazar w i-tym zapytaniu.

W trzecim i ostatnim wierszu wejścia znajduje się ciąg $n-1$ liczb całkowitych $a_2, a_3, \ldots, a_n$ ($1 \leqslant a_i < i$) pooddzielanych pojedynczymi odstępami: a_i określa numer instrukcji, po której następuje instrukcja numer i. Instrukcje numerowane są kolejnymi liczbami naturalnymi od 1 do n, przy czym instrukcja numer 1 to pierwsza instrukcja programu.

W testach wartych łącznie 35% punktów zachodzi dodatkowo warunek: $n \leqslant 30\,000$, $q \leqslant 50$. W podzbiorze tych testów wartym łącznie 20% punktów zachodzi dodatkowo warunek: $n \leqslant 1000$, $q \leqslant 10$.

Wyjście

Twój program powinien wypisać na standardowe wyjście jeden wiersz zawierający ciąg q liczb całkowitych pooddzielanych pojedynczymi odstępami: i-ta z tych liczb powinna określać minimalny czas wykonania programu przy założeniu, że superkomputer Bajtazara posiada k_i procesorów.

Przykład

Dla danych wejściowych:

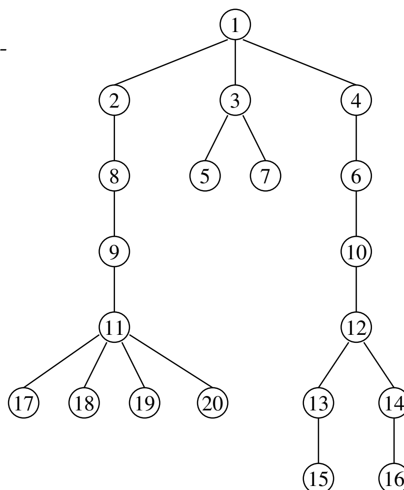
```
20 1
3
1 1 1 3 4 3 2 8 6 9 10 12 12 13 14 11 11 11 11
```

poprawnym wynikiem jest:

```
8
```

Wyjaśnienie do przykładu:
Wykonanie programu może wyglądać następująco:

Czas	*Instrukcje*		
1	*1*		
2	*2*	*3*	*4*
3	*5*	*6*	*7*
4	*8*	*10*	
5	*9*	*12*	
6	*11*	*13*	*14*
7	*15*	*16*	*17*
8	*18*	*19*	*20*



Testy „ocen”:

`1ocen`: $n = 10$, $q = 2$, *drzewo instrukcji jest ścieżką;*

`2ocen`: $n = 10$, $q = 3$, *mały losowy test;*

`3ocen`: $n = 100$, $q = 3$, *wszystkie instrukcje następują po instrukcji nr 1;*

`4ocen`: $n = 127$, $q = 1$, *instrukcje tworzą pełne drzewo binarne;*

`5ocen`: $n = 1\,000\,000$, $q = 31$, *drzewo instrukcji jest długą ścieżką.*

Rozwiązanie

W niniejszym opracowaniu będziemy używać zarówno terminologii z treści zadania, jak i z teorii grafów. Oznaczmy przez $d(x)$ wysokość poddrzewa zaczepionego w wierzchołku-instrukcji x (czyli długość najdłuższej ścieżki biegnącej z x w dół poddrzewa). *Głębokością* x będziemy nazywać długość ścieżki od korzenia (czyli początkowej instrukcji) do x. W szczególności głębokość korzenia wynosi 1. Niech $s(t)$ równa się liczbie wierzchołków na głębokości t. Instrukcję bezpośrednio poprzedzającą x nazwiemy jej *ojcem*, zaś w przypadku pośrednim będziemy mówić o *przodku* x. Aby wyrazić, że zdarzenie miało miejsce w j-tej jednostce czasu, powiemy, że nastąpiło w *turze* j (zamiennie: w momencie bądź w chwili j).

Rozwiązanie dla ustalonej liczby procesorów

Zastanówmy się na początek, jak rozwiązać nasz problem, gdy liczba procesorów jest ustalona – oznaczmy ją przez k. Jeśli $k = 1$, to w każdej jednostce czasu możemy wykonać dokładnie jedną instrukcję, zatem odpowiedzią będzie n. *W dalszej części opracowania będziemy już zakładać, że $k > 1$, co pozwoli uprościć późniejsze rozumowania.*

W pierwszej turze możemy wykonać tylko jedną instrukcję. Natomiast już w kolejnej liczba dostępnych instrukcji może przekroczyć k. Które z nich powinniśmy wykonać od razu, a które mogą poczekać? Okazuje się, że warto w pierwszym rzędzie wykonywać te instrukcje, które mają najdłuższą ścieżkę instrukcji oczekujących po nich.

Twierdzenie 1. *Algorytm zachłanny, który w pierwszej kolejności wybiera instrukcje o największym $d(x)$, generuje plan wykonania programu o minimalnej długości.*

Polecamy Czytelnikowi zastanowić się samemu nad dowodem. Jest to dobre ćwiczenie na szukanie precyzyjnych argumentów do wykazania twierdzenia, które intuicyjnie wydaje się prawdziwe, ale nie jest tak łatwo to uzasadnić. Twierdzenie 1 można udowodnić na kilka sposobów, jednak my skorzystamy z lematu, który przyda się również w dalszej części opracowania. Mianowicie wykażemy, że jeśli w pewnej chwili choć jeden procesor pozostaje bezczynny, to znaczy, że wąskim gardłem jest kształt drzewa instrukcji.

Oznaczmy przez $r_k(t)$ liczbę instrukcji wykonywanych przez algorytm zachłanny w turze t dla k procesorów. Zauważmy, że r_k może zależeć od tego, w jaki sposób algorytm rozstrzyga remisy w sytuacji, gdy gotowych do wykonania jest więcej niż k instrukcji maksymalizujących d. Nie przejmujmy się tym na razie. Ustalmy dowolny algorytm zachłanny i odnośmy się do planu generowanego przez niego – później okaże się, że w istocie jest to bez znaczenia.

Lemat 1. Jeśli $r_k(t) < k$, to wszystkie instrukcje na głębokościach nie większych niż t zostają wykonane do chwili t.

Dowód: Dowód przeprowadzimy przez indukcję względem t. Dla $t = 1$ teza jest oczywista. Przyjmijmy zatem $t > 1$.

Jeśli $r_k(t-1) < k$, to na mocy założenia indukcyjnego, ojcowie wszystkich instrukcji na głębokości t zostają wykonani do chwili $t-1$. Skoro $r_k(t) < k$, to znaczy, że algorytm zachłanny wykonał wszystkie dostępne instrukcje.

Rozważmy teraz przypadek $r_k(t-1) = k$. Niech b oznacza liczbę jednostek czasu bezpośrednio poprzedzających t, w których wykonywanych jest po k instrukcji (patrz rys. 1). Wiemy, że $b \leqslant t-2$, ponieważ w pierwszej turze gotowa jest tylko jedna instrukcja.

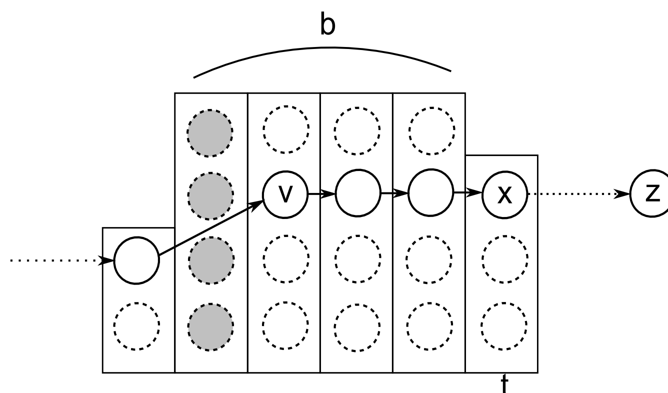
Przypuśćmy, że teza indukcyjna nie zachodzi. To oznacza, że pewna instrukcja o głębokości t zostaje wykonana później. W takim razie algorytm zachłanny przypisze do chwili t pewnego jej przodka (być może pośredniego) – oznaczmy go przez x. Zauważmy, że $d(x) \geqslant 2$.

Czy w chwili $t-1$ zostaje wykonany ojciec x? Jeśli nie, to znaczy, że został wykonany już wcześniej, zatem w chwili $t-1$ instrukcja x była gotowa do wykonania.

Skoro algorytm zachłanny jej nie wybrał, to znaczy, że gotowych było k innych instrukcji $y_1, \ldots, y_k$ spełniających $d(y_i) \geqslant 2$. To by oznaczało, że w chwili t na wykonanie czekało co najmniej k instrukcji będących synami $y_1, \ldots, y_k$, co przeczy założeniu, że $r_k(t) < k$.

Skoro więc ojciec x został wykonany w chwili $t-1$, to cofamy się w czasie z naszym rozumowaniem i pytamy się, czy „dziadek" x został wykonany w chwili $t-2$, czy też wcześniej. Jeśli wcześniej, to w chwili $t-2$ gotowych było k instrukcji $y_1, \ldots, y_k$ spełniających $d(y_i) \geqslant 3$, co prowadzi do sprzeczności analogicznie jak w poprzednim przypadku. Powtarzamy ten argument dla kolejnych przodków x.

Na mocy przeprowadzonego rozumowania przodek x stopnia b (oznaczmy go przez y) zostaje wykonany w chwili $t-b$. Przypomnijmy, że x leży na głębokości mniejszej niż t, zatem głębokość y to maksymalnie $t-b-1$. Ale $r_k(t-b-1) < k$, zatem na mocy założenia indukcyjnego y powinien być wykonany do chwili $t-b-1$. Wykazana sprzeczność dowodzi tezy indukcyjnej. ■



Rys. 1: Ilustracja do dowodu lematu 1 dla $k = b = 4$. Instrukcja na głębokości t, która nie zostaje wykonana na czas, jest oznaczona przez z. Przodek x stopnia 4 zostaje wykonany przed momentem $t-4$, zatem przodek stopnia 3 (oznaczony przez v) jest gotowy w turze $t-4$. Z właściwości algorytmu zachłannego wynika, że wszystkie instrukcje pokolorowane na szaro mają następników stopnia co najmniej 4, co jest niemożliwe.

Możemy już teraz udowodnić twierdzenie 1:

Dowód: Niech $t_0 = \max\{t \geqslant 1 : r_k(t) < k \wedge r_k(t+1) > 0\}$. Takie t_0 istnieje zawsze, gdy drzewo ma co najmniej dwa wierzchołki (w przypadku jednego wierzchołka teza jest trywialna). Skoro wszystkie instrukcje na głębokościach nie większych niż t_0 są wykonane do chwili t_0, a algorytm się jeszcze nie zakończył, to musi również zachodzić $s(t_0+1) > 0$. Instrukcje na głębokościach większych niż t_0 są gotowe do realizacji w chwili t_0+1 (oczywiście nie jest możliwe, by były gotowe wcześniej) i są wykonywane po k w turze, poza być może ostatnią turą działania programu. Jest jasne, że żaden plan nie jest w stanie wykonać ich szybciej, zatem plan generowany przez algorytm zachłanny jest optymalny. ■

Przedstawiony algorytm można zaimplementować w złożoności czasowej $O(n \log n)$. Wystarczy dysponować strukturą danych, która pozwala szybko dodawać elementy oraz znajdować i usuwać ten o największym d. Założenia te spełnia *kopiec*[1], na którym oba typy operacji działają w czasie logarytmicznym.

Możemy w ten sposób udzielić odpowiedzi dla każdego zapytania po kolei. Otrzymamy rozwiązanie działające w czasie $O(qn \log n)$, które zdobywało na zawodach 35% punktów. Zostało ono zaimplementowane w plikach `sups3.cpp` i `sups4.pas`.

Rozwiązanie wzorcowe

Jako że potencjalnie liczba zapytań może być tak duża jak n, to rozsądne wydaje się obliczenie najpierw odpowiedzi dla wszystkich $k = 1, \ldots, n$, zapamiętanie ich w tablicy oraz wypisanie na koniec tych pozycji, o które nas proszono. Jest jasne, że dla $k > n$ odpowiedzią jest wysokość drzewa.

Zanim jednak będziemy na to gotowi, musimy lepiej zrozumieć działanie algorytmu zachłannego. Następny lemat daje użyteczne kryterium, mówiące, kiedy wszystkie procesory będą zajęte.

Lemat 2. Dla dowolnych t, i, takich że $\sum_{j=t}^{t+i} s(j) \geqslant (i+1)k$, zachodzi $r_k(t+i) = k$.

Dowód: Załóżmy nie wprost, że $r_k(t+i) < k$. Wiemy z lematu 1, że wszystkie instrukcje znajdujące się na głębokościach $t, t+1, \ldots, t+i$ (oraz być może także niektóre instrukcje z mniejszych głębokości) zostaną wykonane w przedziale $[t, t+i]$. Pociąga to

$$\sum_{j=t}^{t+i} s(j) \leqslant \sum_{j=t}^{t+i} r_k(j) < (i+1)k.$$

Otrzymana sprzeczność dowodzi tezy lematu. ∎

Wykorzystując oba lematy, można wykazać, że wartości $r_k(t)$ zależą jedynie od ciągu $s(t)$. Nieistotne są dokładna struktura drzewa oraz sposób, w jaki algorytm zachłanny rozstrzyga remisy. Nie będziemy jednak dowodzić tutaj tego faktu, ponieważ zrobimy to niejako przy okazji podczas konstruowania algorytmu wzorcowego.

Przyda nam się jeszcze jedna prosta obserwacja. Mianowicie jeśli w pewnym momencie wszystkie procesory są zajęte, to na pewno nie zmieni się to po zmniejszeniu k.

Lemat 3. Jeśli $r_k(t) = k$, to $r_{k-1}(t) = k-1$.

Dowód: Oznaczmy $t_0 = \max\{i < t : r_k(i) < k\}$. Z lematu 1 wiadomo, że instrukcje na głębokościach nie większych niż t_0 są zakończone do chwili t_0. Zatem w przedziale $[t_0{+}1, t]$ komputer może wykonywać tylko instrukcje o głębokościach z tego przedziału, co implikuje

$$(t-t_0)(k-1) < (t-t_0)k = \sum_{j=t_0+1}^{t} r_k(j) \leqslant \sum_{j=t_0+1}^{t} s(j).$$

Powyższa nierówność w połączeniu z lematem 2 daje tezę lematu. ∎

[1] Można o nim poczytać w książce [25].

Algorytm

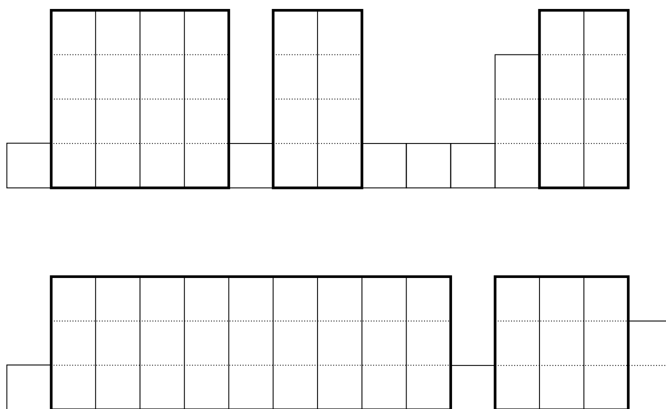
Bogatsi o nową wiedzę, możemy zabrać się do konstruowania efektywnego algorytmu wyliczania $r_k(t)$. Znamy $r_n = s$ i będziemy rozpatrywać kolejne $k = n-1, n-2, \ldots$ Chcielibyśmy wyznaczyć $r_k(1), r_k(2), \ldots$ znając $r_{k+1}(1), r_{k+1}(2), \ldots$ Ponadto musimy zrobić to tak szybko dla wszystkich k, aby całkowity czas działania był $o(n^2)$. Początkowo cel wydaje się nieosiągalny, skoro liczba wszystkich wartości do obliczenia może być rzędu n^2. Trudność tę można ominąć, rozpatrując każdy maksymalny spójny ciąg, na którym $r_k(t) = k$, jako pojedynczy obiekt. Nazwiemy taki ciąg k-*blokiem* lub po prostu blokiem, jeśli wiadomo, o jakie k chodzi. Jeśli $r_k(t) < k$, to przedział jednoelementowy $[t,t]$ będziemy nazywali małym blokiem. Dla symetrii pozostałe bloki będziemy nazywać dużymi.

Załóżmy, że znamy strukturę $(k+1)$-bloków. Aby wyznaczyć k-bloki, zaczynamy od $t = 1$ i przesuwamy się w prawo. Przyjmijmy, że przeprowadziliśmy obliczenia do t_0 i wszystkie instrukcje na głębokościach nie większych niż t_0 zostały już przydzielone.

Jeśli $s(t_0+1) \leqslant k$, to oczywiście $r_k(t_0+1) = s(t_0+1)$. A co w przypadku, gdy $s(t_0+1) \geqslant k+1$? Wówczas w tym miejscu rozpoczyna się duży $(k+1)$-blok. Z lematu 1 wiemy, że nowy k-blok będzie sumą $(k+1)$-bloków. Warunek z lematu 2 gwarantuje, że kolejne bloki możemy scalać zachłannie.

Jeśli bezpośrednio na prawo znajduje się mały $(k+1)$-blok $[t,t]$ oraz $\sum_{j=t_0+1}^{t} s(j) \geqslant (t-t_0)k$, to $r_k(t) = k$ i dodajemy go do tworzonego k-bloku. W przeciwnym razie musimy wykonać $\sum_{j=t_0+1}^{t} s(j) - k\cdot(t-t_0-1) < k$ instrukcji i pozostawiamy mały blok o większej wartości. Jeśli natomiast napotkamy duży blok, to na pewno będziemy musieli go scalić.

Kiedy zakończymy procedurę scalania, traktujemy prawy koniec bloku jako nowe t_0 i powtarzamy cały proces.



Rys. 2: Przykładowa struktura 4-bloków i 3-bloków dla tych samych danych.

Implementacja i analiza złożoności

Implementacja powyższego algorytmu wymaga zastanowienia, choć gotowy kod okaże się stosunkowo krótki. Należy w szczególności dobrać odpowiednie struktury danych do przechowywania informacji o blokach, aby uniknąć wykonywania liniowej liczby operacji dla pojedynczego k.

Zauważmy, że nie warto interesować się małym blokiem o wysokości s, dopóki nie będziemy analizować $k = s$ (wtedy zmieni się on w duży blok) lub nie będziemy go scalać z innym blokiem. Zamiana w duży blok może oczywiście nastąpić tylko raz dla konkretnego bloku, więc operacji tego typu będzie co najwyżej n. Również sytuacja, w której blok zostaje scalony z blokiem położonym z lewej strony, może zdarzyć się tylko n razy, bo za każdym razem maleje liczba bloków. Pozostaje przeanalizować przypadek, w którym uaktualniamy stan małego bloku, ale nie zostaje on scalony, czyli dalej zachodzi $r_k(t) < k$. Zauważmy, że może mieć to miejsce jedynie na końcu procedury scalania bloków, kiedy pewien duży blok zmniejsza swoją wysokość o 1. Suma wysokości wszystkich dużych bloków, początkowo równa zeru, w trakcie działania algorytmu zwiększy się co najwyżej o n, przy zamianach małych bloków w duże. Suma ta nie może spaść poniżej zera, więc liczba operacji ostatniego typu amortyzuje się do $O(n)$.

Poniższy pseudokod pokazuje inicjalizację struktury danych pozwalającej efektywnie zaimplementować przedstawiony pomysł. Tablica *tasks* (początkowo wypełniona zerami) pamięta wartości r_k dla aktualnego k. Jeśli *first_in_block*$[i]$ równa się **true**, to i jest początkiem bloku o długości *block_size*$[i]$. Jeżeli *first_in_block*$[i]$ = **false**, to pomijamy i w dalszych obliczeniach (w szczególności, nie dbamy już o aktualizowanie *tasks*$[i]$).

Z kolei *by_tasks* jest tablicą list. Lista *by_tasks*$[i]$ pamięta te głębokości, dla których r_k równa się i. Zakładamy, że można dodawać do niej elementy w czasie stałym (operacja *push*) oraz przeglądać je w czasie proporcjonalnym do długości listy. Aby ominąć problem usuwania elementów z listy, przed przetworzeniem bloku i będziemy sprawdzać, czy wartość *tasks*$[i]$ jest aktualna.

```
 1: begin
 2:   for i := 1 to n do
 3:     tasks[d[i]] := tasks[d[i]] + 1;
 4:   for i := 1 to n do begin
 5:     if tasks[i] > 0 then
 6:       by_tasks[tasks[i]].push(i);
 7:     block_size[i] := 1;
 8:     first_in_block[i] := true;
 9:   end
10: end
```

Poniżej widzimy główną część programu. W momencie, w którym wysokość bloku przekracza k, obliczana jest zmienna *delta* równa liczbie instrukcji, które nie mieszczą się w bloku. Powiększamy blok w prawo, dopóki *delta* nie równa się 0. Za każdym razem, gdy powiększamy pewien blok, sprawdzamy, czy czas zakończenia nie wzrasta.

```
 1: begin
 2:   wynik[n] := d[1];
 3:   for k := n − 1 downto 1 do
 4:     foreach x ∈ by_tasks[k + 1] do
 5:       if first_in_block[x] and tasks[x] = k + 1 then begin
 6:         delta := block_size[x];
 7:         tasks[x] := k;
 8:         by_tasks[k].push(x);
 9:         while delta > 0 do begin
10:           y := x + block_size[x];
11:           wynik[k] := max(wynik[k], y + block_size[y] − 1);
12:           if block_size[y] = 1 and tasks[y] + delta < k then begin
13:             tasks[y] := tasks[y] + delta;
14:             delta := 0;
15:             by_tasks[tasks[y]].push(y);
16:           end
17:           else begin
18:             { Gdy tasks[y] = k + 1, delta może wzrosnąć! }
19:             delta := delta − (k − tasks[y]) · block_size[y];
20:             block_size[x] := block_size[x] + block_size[y];
21:             first_in_block[y] := false;
22:           end
23:         end
24:       end
25:   return wynik;
26: end
```

Taka struktura danych nie gwarantuje, że bloki analizowane będą od lewej do prawej. Zachęcamy jednak Czytelnika do wykazania, że nie ma to wpływu na wynik algorytmu. Przedstawione rozwiązanie można znaleźć w pliku `sup2.cpp`.

Inna metoda implementacji to użycie struktury *find-union*. Czas działania algorytmu jest w tym przypadku minimalnie gorszy od $O(n)$, jednak różnica jest niewykrywalna w testach. Rozwiązania korzystające z tego pomysłu znajdują się w plikach `sup.cpp` oraz `sup1.pas`.

Gdyby pominąć pomysł z dużymi blokami i za każdym razem analizować każdą głębokość osobno, otrzymalibyśmy algorytm o złożoności pesymistycznej $O(qn)$, ale radzący sobie z dużymi losowymi testami. Na zawodach mógłby on zdobyć do 60% punktów. Pomysł ten zaimplementowano w plikach `sups5.cpp` oraz `sups6.pas`.

Rozwiązanie alternatywne

Okazuje się, że istnieje zupełnie inne rozwiązanie, sprowadzające problem do zagadnienia geometrycznego. Jest ono godne uwagi, gdyż okazuje się prostsze w implementacji. Główna obserwacja orzeka, że wąskie gardło każdego programu znajduje się na jego

końcu. Zwrócił na to uwagę Paweł Gawrychowski na swoim blogu[2] (uzasadnienie poprawności w opisie na blogu różni się od przedstawionego tutaj).

Twierdzenie 2. *Optymalny czas zakończenia dla k procesorów to*

$$H_k = \max_{h:s(h)>0} \left(h + \left\lceil \frac{\sum_{i>h} s(i)}{k} \right\rceil \right),$$

gdzie przez $\lceil r \rceil$ oznaczamy najmniejszą liczbę całkowitą nie mniejszą niż r.

Dowód: Łatwo zauważyć, że czas zakończenia żadnego planu nie może być mniejszy niż H_k. Dla każdego h wierzchołki znajdujące się na głębokościach większych niż h mogą być przetwarzane nie wcześniej niż w momencie $h+1$ oraz ich wykonanie zajmie co najmniej $\lceil \sum_{i>h} s(i)/k \rceil$ jednostek czasu.

Zgodnie z dowodem twierdzenia 1, czas zakończenia wykonywania instrukcji programu przez algorytm zachłanny to $h + \lceil \sum_{i>h} s(i)/k \rceil$ dla $h = t_0 = \max\{t \geqslant 1 : r_k(t) < k \wedge r_k(t+1) > 0\}$. W połączeniu z poprzednią obserwacją implikuje to, że czas ten jest równy H_k. ■

Jako że potrzebujemy szybko obliczać H_k dla różnych k, chcielibyśmy liczyć maksimum z jak najprostszych funkcji.

Lemat 4. Oznaczmy $f_h(k) = kh + \sum_{i>h} s(i)$. H_k można wyrazić równoznacznie jako

$$\left\lceil \frac{\max_{h:s(h)>0} f_h(k)}{k} \right\rceil.$$

Dowód: Zauważmy, że

$$h + \left\lceil \frac{\sum_{i>h} s(i)}{k} \right\rceil = \lceil f_h(k)/k \rceil.$$

Chcemy się przekonać, że

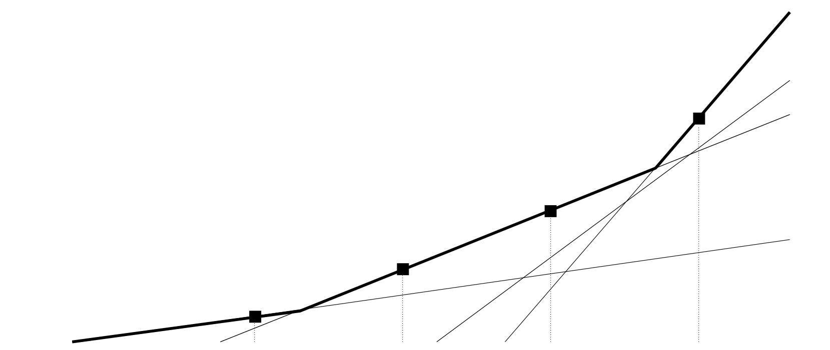
$$\left\lceil \frac{\max_{h:s(h)>0} f_h(k)}{k} \right\rceil = \max_{h:s(h)>0} \lceil f_h(k)/k \rceil.$$

Równość zachodzi, gdyż funkcja $x \longrightarrow \lceil x/k \rceil$ jest monotoniczna, zatem zachowuje maksimum zbioru. ■

Niech m oznacza wysokość wejściowego drzewa. Sprowadziliśmy problem do wyznaczania maksimum m funkcji liniowych w punktach $1, \ldots, n$. Zauważmy, że jeśli $f_h(k_0) > f_i(k_0)$ dla każdego $i < h$, to taka sama zależność będzie zachodzić dla dowolnego $k > k_0$. Innymi słowy, gdy funkcja f_h „przegoni” funkcje o niższych indeksach, to pozostanie już zawsze od nich większa, gdyż rośnie szybciej.

Daje to prosty algorytm wyznaczania górnej otoczki rodziny funkcji liniowych. Po przetworzeniu $f_1, f_2, \ldots, f_h$ trzymamy informacje o kolejnych funkcjach na stosie. Dla każdego i pamiętamy, na jakim maksymalnym przedziale $[left_i, right_i]$ funkcja f_i jest

[2] `http://fajnezadania.wordpress.com/2014/04/13/superkomputer/`



Rys. 3: Górna otoczka czterech funkcji liniowych.

większa od pozostałych. Jeżeli wspomniany przedział jest pusty, zapominamy o tej funkcji.

Jak dodać do struktury funkcję f_{h+1}? Przypuśćmy, że na szczycie stosu znajduje się f_j. Szukamy w czasie stałym najmniejszego k, dla którego $f_{h+1}(k) > f_j(k)$. Jeśli $k \leqslant \mathit{left}_j$, to usuwamy ze stosu f_j i powtarzamy procedurę. W przeciwnym razie uaktualniamy right_j i dodajemy f_{h+1} na czubek stosu.

Oczywiście każdą funkcję usuniemy ze stosu maksymalnie raz, zatem całkowita złożoność obliczeniowa przedstawionego algorytmu jest liniowa. Rozwiązanie alternatywne można znaleźć w pliku `sup3.cpp`.

Jacek Tomasiewicz — Treść zadania, Opracowanie
Aleksander Łukasiewicz, Karol Pokorski — Program

Dostępna pamięć: 128 MB. *OI, etap II, dzień drugi, 13.02.2014*

Ptaszek

W Bajtockim Lesie rośnie w rzędzie n drzew. Na czubku pierwszego drzewa siedzi ptaszek, który chciałby dolecieć na czubek ostatniego drzewa. Ponieważ ptaszek jest jeszcze mały, więc niekoniecznie będzie miał siłę pokonać całą drogę bez zatrzymywania się. Jeśli ptaszek siedzi na czubku drzewa o numerze i, to podczas pojedynczego lotu może on dolecieć do dowolnego z drzew o numerach $i+1,\ i+2,\ \ldots,\ i+k$, po czym musi odpocząć.

Ponadto lot w górę jest dla niego o wiele trudniejszy niż lot w dół. Innymi słowy, ptaszek się męczy, jeśli leci z niższego na wyższe, bądź równe drzewo. W przeciwnym przypadku ptaszek się nie męczy.

Należy tak wybrać drzewa, na których będzie lądował ptaszek, aby zmęczył się jak najmniejszą liczbę razy. Dodatkowo, ptaszek ma kilku kolegów, którzy także chcieliby przedostać się z pierwszego do ostatniego drzewa – mogą być oni bardziej lub mniej wytrzymali od niego (a zatem mogą mieć oni różne wartości k). Pomóż także kolegom ptaszka.

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą n ($2 \leqslant n \leqslant 1\,000\,000$), oznaczającą liczbę drzew w Bajtockim Lesie. Drugi wiersz wejścia zawiera n liczb całkowitych $d_1, d_2, \ldots, d_n$ ($1 \leqslant d_i \leqslant 10^9$) pooddzielanych pojedynczymi odstępami: d_i oznacza wysokość i-tego drzewa.

Trzeci wiersz wejścia zawiera jedną liczbę całkowitą q ($1 \leqslant q \leqslant 25$), określającą liczbę ptaszków, dla których należy rozważyć przelot. Kolejne q wierszy zawierają opisy ptaszków: w i-tym z tych wierszy znajduje się liczba całkowita k_i ($1 \leqslant k_i \leqslant n-1$), oznaczająca wytrzymałość i-tego ptaszka. Innymi słowy, maksymalna liczba drzew, które może minąć i-ty ptaszek, zanim będzie zmuszony odpocząć, wynosi $k_i - 1$.

W testach wartych łącznie 70% punktów zachodzi dodatkowy warunek: $n \leqslant 100\,000$. W podzbiorze tych testów wartym łącznie 30% punktów zachodzi dodatkowy warunek: $n \leqslant 1000$.

Wyjście

Twój program powinien wypisać na standardowe wyjście dokładnie q wierszy. W i-tym wierszu powinna się znaleźć odpowiedź dla i-tego ptaszka z wejścia. Odpowiedź dla każdego ptaszka składa się z jednej liczby całkowitej, równej minimalnej liczbie razy, w których ptaszek musi podlecieć z niższego na wyższe, bądź równe drzewo.

Przykład

Dla danych wejściowych:

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

```
2
2
5
```

poprawnym wynikiem jest:

```
2
1
```

Wyjaśnienie do przykładu: *Pierwszy ptaszek może zatrzymać się kolejno na drzewach o numerach 1, 3, 5, 7, 8, 9. Ptaszek będzie się męczył, lecąc z trzeciego drzewa na piąte i z siódmego na ósme.*

Testy „ocen”:

`1ocen`: *$n = 11$, $q = 1$, $k_1 = 5$, wysokości wszystkich drzew są równe, odpowiedzią jest 2 (wystarczy międzylądowanie na szóstym drzewie);*

`2ocen`: *$n = 100$, $q = 2$, $k_1 = 5$, $k_2 = 6$, wysokości drzew są na przemian 1 i 2 – dla obu ptaszków optymalna strategia wybiera odpoczynek co 5 drzew, co daje 11-krotne zmęczenie obu ptaszków;*

`3ocen`: *$n = 100$, $q = 1$, $k_1 = 10$, ciąg wysokości drzew to: $100, 99, \ldots, 1$, odpowiedzią jest 0;*

`4ocen`: *$n = 1\,000\,000$, $q = 25$, $k_i = i$, $d_{1000} = d_{2000} = d_{3000} = \ldots = d_{1\,000\,000} = 2$, zaś pozostałe $d_i = 1$.*

Rozwiązanie

We wszystkich rozwiązaniach wynik będziemy obliczać oddzielnie dla każdego ptaszka. Na potrzeby opisu przyjmijmy więc, że rozważamy jednego ptaszka o wytrzymałości k. Dodatkowo, oznaczmy przez w_i minimalną liczbę zmęczeń ptaszka, zakładając, że kończy on podróż na i-tym drzewie.

Rozwiązanie wolne $O(q \cdot n \cdot k)$

Spróbujemy obliczyć wartości $w_1, w_2, \ldots, w_n$ za pomocą programowania dynamicznego. Załóżmy, że obliczamy wartość w_i oraz że mamy już obliczone wyniki dla wszystkich wcześniejszych drzew, czyli wartości $w_1, w_2, \ldots, w_{i-1}$.

Obliczając liczbę zmęczeń potrzebnych na dostanie się do i-tego drzewa, możemy przejrzeć wszystkie drzewa, z których mógł bezpośrednio przylecieć do niego ptaszek. Będzie to tylko k poprzednich drzew, ponieważ tylko tyle może przelecieć ptaszek bez robienia odpoczynku.

Załóżmy, że ptaszek leci z j-tego drzewa na i-te. Jeśli leci z niższego drzewa na wyższe bądź równe ($d_j \leqslant d_i$), to się zmęczy. W związku z tym, łączna liczba zmęczeń wzrośnie o jeden, czyli $w_i = w_j + 1$. W przeciwnym przypadku, gdy ptaszek leci z wyższego drzewa na niższe ($d_j > d_i$), to się nie zmęczy. Łączna liczba zmęczeń

pozostanie bez zmian, czyli $w_i = w_j$. Ostatecznie powinniśmy wybrać minimalną liczbę zmęczeń, jaką możemy uzyskać.

```
1:  function obliczZmeczenie(d[1..n])
2:  begin
3:    w[1] := 0;
4:    for i := 2 to n do begin
5:      w[i] := ∞;
6:      for j := max(1, i − k) to i − 1 do begin
7:        zmeczenie := 0;
8:        if d[j] ⩽ d[i] then
9:          zmeczenie := 1;
10:       w[i] := min(w[i], w[j] + zmeczenie);
11:     end
12:   end
13:   return w[n];
14: end
```

Złożoność takiego rozwiązania dla pojedynczego ptaszka wynosi $O(n \cdot k)$, ponieważ dla każdego drzewa przeglądamy k poprzednich drzew. Powtarzając to dla wszystkich ptaszków, uzyskamy złożoność czasową $O(q \cdot n \cdot k)$.

Za takie rozwiązanie można było uzyskać około 30% punktów. Implementacje znajdują się w plikach `ptas1.cpp` i `ptas5.pas`.

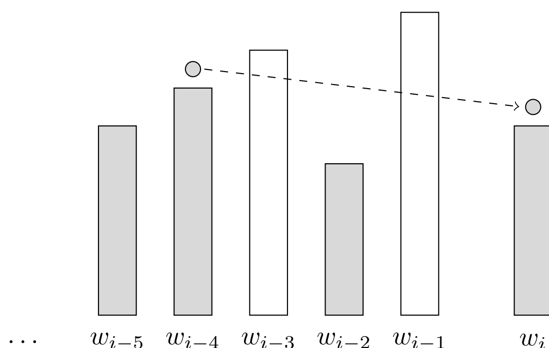
Rozwiązanie wzorcowe $O(q \cdot n)$

Zauważmy, że jeśli mamy dwa drzewa $j < i$ oddalone maksymalnie o k, to liczba zmęczeń ptaszka dla tych drzew nie różni się o więcej niż jeden, czyli $|w_i - w_j| \leqslant 1$.

Załóżmy, że tak nie jest. Jeśli $w_i > w_j$, to moglibyśmy z j-tego drzewa polecieć do i-tego i w najgorszym przypadku uzyskalibyśmy $w_i = w_j + 1$. Jeśli zaś $w_j > w_i$, to moglibyśmy znaleźć takie drzewo $p \leqslant j$, że $w_p \leqslant w_i$ i z p-tego drzewa dałoby się dolecieć do j-tego. W ten sposób uzyskalibyśmy $w_j = w_p + 1 \leqslant w_i + 1$, więc obserwacja jest prawdziwa.

Z powyższą, kluczową obserwacją możemy przejść do właściwego rozwiązania. Załóżmy, że obliczamy wartość w_i oraz że mamy już obliczone wyniki dla wszystkich wcześniejszych drzew. Tak naprawdę, interesuje nas tyko k poprzednich drzew, czyli tych, z których można bezpośrednio dolecieć do i-tego drzewa. Dodatkowo, drzewa te możemy podzielić na dwa zbiory. Niech A będzie zbiorem drzew o najmniejszej wartości w – dokładniej, o wartości $x = \min(w_{i-1}, w_{i-2}, \ldots, w_{i-k})$. Natomiast B to zbiór pozostałych drzew, czyli takich, w których liczba zmęczeń ptaszka będzie o jeden większa (zauważmy, że zbiór B może być pusty, natomiast zbiór A będzie zawsze zawierał co najmniej jedno drzewo).

Chcielibyśmy teraz znaleźć poprzednika i, czyli takie drzewo, z którego opłaca się bezpośrednio dolecieć do i-tego drzewa. Najlepszym miejscem będzie najwyższe drzewo ze zbioru A lub ze zbioru B, ponieważ każde niższe drzewo może dać tylko nie lepszy wynik. Dodatkowo zauważmy, że jeżeli miałoby to być drzewo ze zbioru B, to moglibyśmy równie dobrze wybrać dowolne drzewo ze zbioru A i taki przelot byłby



Rys. 1: Obliczanie wartości w_i (dla $k = 5$). Szarym kolorem oznaczono zbiór A, czyli drzewa z liczbą zmęczeń x. Białym kolorem oznaczono zbiór B, czyli drzewa z liczbą zmęczeń $x + 1$.

nie gorszy. Podsumowując, poprzednikiem i będzie najwyższe drzewo ze zbioru A, patrz rys. 1.

W naszym algorytmie powinniśmy przechowywać oddzielnie oba zbiory. Gdy wraz ze zwiększaniem i będziemy przesuwać się do kolejnych drzew, za każdym razem będziemy usuwać jeden element z któregoś ze zbiorów A, B i wstawiać jeden element do któregoś ze zbiorów. Dodatkowo, gdy napotkamy drzewo o największej dotąd wartości w, wszystkie elementy ze zbioru B trafią do zbioru A. Chcemy więc dobrać strukturę danych z wyróżnionym początkiem i końcem, która umożliwi operacje: wstawiania na koniec, zdejmowania z początku oraz odczytywania maksimum.

Obliczanie maksimów

Najtrudniejszym elementem jest efektywne znajdywanie najwyższego drzewa. Możemy użyć do tego kolejki priorytetowej, struktury `set` z biblioteki STL lub drzewa przedziałowego, opisywanych wielokrotnie w opracowaniach poprzednich zadań olimpijskich. Takie programy będą miały złożoność $O(q \cdot n \log n)$. Za takie rozwiązania można było uzyskać 70% punktów, a przykładowe implementacje trzech wariantów zawarte są w plikach `ptas3.cpp`, `ptas6.cpp` i `ptas7.cpp`.

Najwyższe drzewa możemy znajdywać szybciej, w zamortyzowanym czasie $O(n)$. Taka struktura danych, zwana *K-max* kolejką, wystąpiła już w rozwiązaniach zadań olimpijskich: w zadaniu *Temperatura* z II etapu XVIII Olimpiady [18] lub w zadaniu *Piloci* z III etapu XVII Olimpiady [17].

To kończy opis rozwiązania wzorcowego, działającego w czasie $O(q \cdot n)$. Implementacje znajdują się w plikach `pta.cpp`, `pta1.cpp`, `pta2.cpp` oraz `pta3.pas`, `pta4.pas`.

Testy

Testy zostały podzielone na cztery główne grupy: *losowe* – wygenerowane losowo, zarówno małe testy poprawnościowe, jak i duże testy wydajnościowe; *niskie drzewa* – losowe z małymi wysokościami drzew; *ustalona wytrzymałość* – małe lub duże wytrzymałości ptaszków; *rosnące wysokości* – przedziały drzew tworzą rosnące wysokości.

Bartosz Tarnawski
Treść zadania, Opracowanie

Karol Farbiś
Program

Dostępna pamięć: 128 MB. *OI, etap II, dzień drugi, 13.02.2014*

Rajd

W Bajtogrodzie niedługo odbędzie się doroczny rajd rowerzystów. Bajtogrodzcy rowerzyści są urodzonymi długodystansowcami. Przedstawiciele lokalnej społeczności motorowerzystów, od dawna zwaśnieni z rowerzystami, postanowili sabotować to wydarzenie.

W Bajtogrodzie znajduje się n skrzyżowań, połączonych jednokierunkowymi ulicami. Co ciekawe, w sieci ulic nie występują cykle – jeżeli ze skrzyżowania u można dojechać do v, to na pewno z v nie da się w żaden sposób dostać do u.

Trasa rajdu będzie prowadziła przez bajtogrodzkie ulice. Motorowerzyści postanowili w dniu wyścigu z samego ranka przyjechać na swoich lśniących maszynach na jedno ze skrzyżowań i zupełnie je zablokować. Co prawda wówczas związek kolarski szybko wytyczy alternatywną trasę, ale być może nie będzie ona taka długa i rowerzyści nie będą mogli wykazać się swoimi możliwościami. Na to właśnie liczą motorowerzyści – chcą zablokować takie skrzyżowanie, żeby najdłuższa trasa, która je omija, była możliwie krótka.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie liczby całkowite n i m ($2 \leqslant n \leqslant 500\,000$, $1 \leqslant m \leqslant 1\,000\,000$) oddzielone pojedynczym odstępem, oznaczające liczbę skrzyżowań i ulic w Bajtogrodzie. Skrzyżowania numerujemy liczbami od 1 do n. Kolejne m wierszy zawiera opis sieci drogowej: w i-tym z tych wierszy znajdują się dwie liczby całkowite a_i, b_i ($1 \leqslant a_i, b_i \leqslant n$, $a_i \neq b_i$) oddzielone pojedynczym odstępem, oznaczające, że istnieje jednokierunkowa ulica od skrzyżowania o numerze a_i do skrzyżowania o numerze b_i.

W testach wartych łącznie 33% punktów dla każdej ulicy zachodzi dodatkowy warunek: $a_i < b_i$.

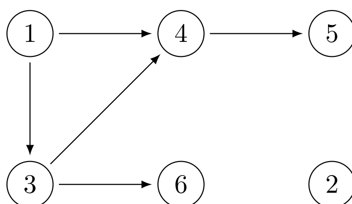
Wyjście

Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać dwie liczby całkowite oddzielone pojedynczym odstępem. Pierwsza z tych liczb ma oznaczać numer skrzyżowania, które powinni zablokować motorowerzyści, druga zaś – maksymalną liczbę ulic, którymi mogą przejechać wówczas rowerzyści. W przypadku, gdy istnieje wiele poprawnych rozwiązań, Twój program może wypisać dowolne z nich.

Przykład

Dla danych wejściowych:
```
6 5
1 3
1 4
3 6
3 4
4 5
```

poprawnym wynikiem jest:
```
1 2
```



Testy „ocen":

`1ocen`: *$n = 10$, $m = 9$, ścieżka, najlepiej zablokować ją w środku;*

`2ocen`: *$n = 100$, $m = 4950$, istnieją wszystkie drogi ze skrzyżowań o mniejszych numerach do skrzyżowań o większych numerach;*

`3ocen`: *$n = 500\,000$, $m = 749\,999$, ze skrzyżowania i wychodzi droga do $i-1$ (o ile $i \geqslant 2$) oraz $\frac{i}{2}$ (o ile $2 \mid i$).*

Rozwiązanie

Dany jest graf skierowany bez cykli (ang. *directed acyclic graph*, w skrócie *DAG*). Należy znaleźć wierzchołek, którego usunięcie minimalizuje długość najdłuższej ścieżki.

Nasz graf będziemy oznaczać przez G, jego zbiór wierzchołków przez $V(G)$, zaś zbiór krawędzi – przez $E(G)$. Mamy $|V(G)| = n$, $|E(G)| = m$. Będziemy utożsamiać wierzchołki z ich etykietami, czyli napiszemy $V(G) = \{1, 2, \ldots, n\}$. Przez (a, b) oznaczymy skierowaną krawędź od a do b.

Dla uproszczenia możemy założyć, że graf nie ma izolowanych wierzchołków; wówczas $n = O(m)$.

W podzadaniu wartym 33% punktów można założyć, że zachodzi następująca własność:

$$\text{dla każdej krawędzi } (a, b) \in E(G) \quad a < b. \tag{*}$$

Po odpowiednim przenumerowaniu wierzchołków taka własność może zachodzić dla dowolnego DAG-u. Algorytm, który wykonuje takie przenumerowanie, nazywamy sortowaniem topologicznym. Zajmuje on $O(n + m)$ czasu (można go zrealizować za pomocą przeszukiwania w głąb). Jest on opisany w opracowaniu zadania *Licytacja* z XIX Olimpiady [19] oraz w książce [25].

Ponieważ przyda się nam to we wszystkich algorytmach, ustalmy, że zaraz po wczytaniu grafu sortujemy wierzchołki topologicznie. Od tej chwili możemy zakładać, że własność (*) zachodzi dla naszego grafu G.

Zanim przejdziemy do właściwego rozwiązania, zastanówmy się (lub przypomnijmy sobie), jak szybko wyznaczyć najdłuższą ścieżkę w naszym grafie G. Osiągniemy to, obliczając tablicę $longestStart[1..n]$. Wartość na i-tej pozycji ma być równa długości (liczbie krawędzi) najdłuższej ścieżki zaczynającej się w wierzchołku i. Na początku inicjujemy ją zerami, a potem wykonujemy pętlę:

```
1: for i := n downto 1 do
2:    for (i, j) ∈ E do
3:       longestStart[i] := max(longestStart[i], longestStart[j] + 1);
```

Teraz największa wartość w tablicy jest długością najdłuższej ścieżki. To prowadzi nas już do jednego z wolnych rozwiązań.

Rozwiązanie wolne $O(nm)$

Usuwamy po kolei każdy z wierzchołków, a następnie obliczamy długość najdłuższej ścieżki.

To rozwiązanie zostało zaimplementowane w plikach `rajs3.cpp`, `rajs4.pas`. Zdobywało około 30 punktów.

Ścieżki omijające wierzchołek

Powiemy, że ścieżka p *omija* wierzchołek v, gdy v nie należy do p. Żeby zdobyć więcej punktów, trzeba przyjrzeć się temu, jak może wyglądać najdłuższa ścieżka omijająca pewien wierzchołek.

Lemat 1. Każda ścieżka p omijająca wierzchołek v:

1. kończy się w wierzchołku w, który jest wcześniej niż v w porządku topologicznym ($w < v$), albo

2. rozpoczyna się w wierzchołku w, który jest później niż v w porządku topologicznym ($v < w$), albo

3. istnieje krawędź (u, w) na ścieżce p, taka że u jest wcześniej niż v, a w jest później niż v w porządku topologicznym ($u < v < w$).

Dowód: Z własności (*) wiemy, że jeśli p składa się z kolejnych krawędzi $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$, to $v_1 < v_2 < \ldots < v_k$. Stąd już wynika nasz lemat. ■

Wystarczy, że dla każdego wierzchołka rozpatrzymy te trzy przypadki.

Obliczmy teraz, tak jak to zostało opisane wyżej, tablicę $longestStart[1..n]$, a także tablicę jej maksimów sufiksowych $longestAfter[1..n]$, to znaczy

$$longestAfter[i] = \max_{j \geqslant i} longestStart[j].$$

W analogiczny sposób obliczamy jeszcze tablicę $longestEnd[1..n]$ (długość najdłuższej ścieżki kończącej się w danym wierzchołku) oraz tablicę jej maksimów prefiksowych $longestBefore[1..n]$.

Chcielibyśmy jeszcze obliczyć tablicę $longestBypass[1..n]$ taką, aby zachodziło

$$longestBypass[i] = \max_{a<i<b,\ (a,b)\in E(G)} (longestEnd[a] + 1 + longestStart[b]) . \qquad (**)$$

Jeśli to nam się uda, to poznamy długość najdłuższej ścieżki omijającej dowolny wierzchołek v. Na mocy lematu 1 będzie to

$$\max(longestBefore[v-1], longestAfter[v+1], longestBypass[v]).$$

Wtedy możemy już rozwiązać całe zadanie, znajdując minimum z tej wartości dla wszystkich wierzchołków.

Żeby efektywnie wyliczyć tablicę $longestBypass[1..n]$, możemy użyć drzewa przedziałowego. Doprowadzi nas to do pierwszego rozwiązania wzorcowego.

Rozwiązanie wzorcowe $O(m \log n)$

Rozwiązanie to nie jest najszybszym znanym, nie prowadzi też do najkrótszego kodu, ale za to jest intuicyjne. Przez to cieszyło się dużą popularnością wśród zawodników.

Skorzystamy z drzewa przedziałowego, którego liście będą odpowiadały kolejnym wierzchołkom $1, 2, \ldots, n$.

Nasze drzewo każdemu liściowi v będzie przypisywało pewną wartość – $value(v)$. Początkowo $value(v) = 0$ dla każdego v. Chcemy, aby po wykonaniu naszego algorytmu $value(v)$ równało się docelowej wartości $longestBypass[v]$ zdefiniowanej jak w (**).

Drzewo powinno udostępniać operacje ustawienia maksimum na przedziale oraz odczytania wartości w punkcie, czyli

1. $set(l, r, max_v)$ – dla każdego i takiego, że $l \leqslant i \leqslant r$, ustaw $value(i) = \max(value(i), max_v)$,
2. $query(i)$ – odczytaj wartość $value(i)$.

Zawodnicy często spotykają drzewa przedziałowe tego rodzaju. O takich i podobnych drzewach przedziałowych można dowiedzieć się dużo ciekawych rzeczy, oglądając *Wykłady z Algorytmiki Stosowanej* (`http://was.zaa.mimuw.edu.pl`).

Niech $(a, b) \in E(G)$ będzie dowolną krawędzią. Oznaczmy długość najdłuższej ścieżki zawierającej tę krawędź przez $longestWith(a, b)$. Wartość tę możemy obliczyć w czasie stałym, ponieważ

$$longestWith(a, b) = longestEnd[a] + 1 + longestStart[b].$$

Teraz dla każdej krawędzi $(a, b) \in E(G)$ i $l := longestWith(a, b)$ wykonamy operację $set(a+1, b-1, l)$. Możemy o tym myśleć jak o „informowaniu” wierzchołków między a i b, że istnieje omijająca je ścieżka o długości l.

Następnie odpytujemy wszystkie wierzchołki za pomocą operacji *query*(), uzyskując tablicę *longestBypass*$[1..n]$. Ponieważ każda operacja na drzewie działa w czasie $O(\log n)$, całość zajmie nam $O(m \log n)$ czasu.

Takie rozwiązanie zostało zaimplementowane w plikach `raj.cpp`, `raj1.pas`. Zdobywało 100 punktów.

Autor zadania nie miał żadnego pomysłu na szybsze rozwiązanie. Taki pomysł miał za to jeden z zawodników.

Rozwiązanie $O(m \log^* n)$

To rozwiązanie, choć asymptotycznie szybsze od wzorcowego, w praktyce działa podobnie szybko. Opiera się ono na lemacie 1, ale korzysta z niego w inny sposób. Dla wygody wprowadźmy kolejny lemat, który jest prostym wnioskiem z poprzedniego.

Lemat 2. Ustalmy dowolny wierzchołek $v \in V(G)$. Ścieżka o długości l omijająca v istnieje wtedy i tylko wtedy, gdy spełniony jest choć jeden z następujących warunków.

1. Dla pewnego $w < v$ zachodzi *longestEnd*$[w] \geqslant l$.
2. Dla pewnego $w > v$ zachodzi *longestStart*$[w] \geqslant l$.
3. Dla pewnej krawędzi (u, w), $u < v < w$, zachodzi *longestWith*$(u, w) \geqslant l$.

Stworzymy jeszcze jedną tablicę, tym razem indeksowaną od 0 do n. Jej elementami będą listy par liczb całkowitych. Nazwiemy ją *valIntervals*$[0..n]$. Konstruujemy ją w taki sposób, aby para (a, b) $(1 \leqslant a \leqslant b \leqslant n)$ należała do listy *valIntervals*$[l]$, jeżeli

1. $b = n$ oraz *longestEnd*$[a - 1] = l$, lub
2. $a = 1$ oraz *longestStart*$[b + 1] = l$, lub
3. $(a - 1, b + 1) \in E(G)$ oraz *longestWith*$(a - 1, b + 1) = l$.

Zauważmy, że mając już *longestEnd*$[1..n]$ oraz *longestStart*$[1..n]$, prosto obliczamy *valIntervals*$[0..n]$ w czasie $O(n + m)$.

Dzięki temu, jak sformułowaliśmy lemat 2, otrzymujemy od razu kolejną równoważność.

Lemat 3. Ustalmy dowolny wierzchołek $v \in V(G)$. Ścieżka o długości l omijająca v istnieje wtedy i tylko wtedy, gdy istnieją takie liczby całkowite a, b, $1 \leqslant a \leqslant v \leqslant b \leqslant n$, że para (a, b) należy do pewnej listy *valIntervals*$[l']$, przy czym $l' \geqslant l$.

Zdefiniujmy jeszcze zbiory A_l, B_l dla $l = 0, 1, \ldots, n$. Będą to podzbiory zbioru $V(G) = \{1, \ldots, n\}$. Niech

$$A_l = \{v : a \leqslant v \leqslant b \text{ dla którejś pary } (a, b) \in \textit{valIntervals}[l]\}$$

$$B_l = A_l \cup A_{l+1} \cup \ldots \cup A_n.$$

Wtedy na mocy lematu 3 B_l jest zbiorem tych wierzchołków, które omija choć jedna ścieżka długości l.

Z definicji wiemy, że zbiór B_{l+1} zawiera się w zbiorze B_l dla każdego $l = 0, \ldots, n-1$.

Potrzebujemy jeszcze ostatniej obserwacji.

Lemat 4. Istnieje takie $0 \leqslant l_0 \leqslant n-1$, że $B_{l_0} = V(G)$ oraz $B_{l_0+1} \neq V(G)$.

Dowód: Łatwo zobaczyć, że $B_n = \emptyset$ (w grafie nie ma ścieżki długości n), zaś $B_0 = \{1, \ldots, n\} = V(G)$. ■

Jesteśmy już w stanie przedstawić nasz algorytm w ogólnym zarysie. Będziemy obliczać zbiory B_l kolejno dla $l = n, n-1, n-2, \ldots$ Zatrzymamy się w momencie, gdy $l = l_0$ jak w lemacie 4.

Wówczas optymalnym rozwiązaniem będzie usunięcie dowolnego wierzchołka z $V(G) \setminus B_{l_0+1}$, a najdłuższa ścieżka w powstałym grafie będzie miała długość l_0.

Obliczanie zbiorów B_l

Pozostaje jeszcze pytanie – w jaki sposób można efektywnie obliczać kolejne zbiory B_l (dla malejących l) oraz sprawdzać, czy są równe $V(G)$.

Aby wprowadzić powiew świeżości, możemy zapomnieć o tym, że nasze zbiory mają coś wspólnego z wierzchołkami, i sformułować ten podproblem na nowo.

> Mamy n szklanek ustawionych w rzędzie od lewej do prawej i ponumerowanych liczbami od 1 do n. Początkowo wszystkie szklanki są puste. Bajtek m razy wybiera dwie szklanki a, b ($1 \leqslant a \leqslant b \leqslant n$), a następnie napełnia wodą szklanki $a, a+1, \ldots, b$. Raz napełniona szklanka pozostaje pełna, ponowne napełnianie już nic nie zmienia. Pytamy się, w którym momencie po raz pierwszy wszystkie szklanki są pełne.

Siłowe rozwiązanie zajmuje $O(nm)$ czasu, ponieważ przy każdym napełnianiu być może trzeba sprawdzić $O(m)$ szklanek. My jednak nie chcemy tracić czasu na przeglądanie już napełnionych szklanek.

Niech $full[1..n]$ będzie tablicą reprezentującą to, co sugeruje nazwa – $full[i] = \textbf{true}$, jeżeli w danym momencie i-ta szklanka jest pełna. Dobrze by było, gdybyśmy w każdym momencie oraz dla każdego i mogli szybko obliczyć poniższą funkcję

$$firstEmpty(i) = \min\{j \geqslant i : full[j] = \textbf{false}\}.$$

Załóżmy na chwilę, że wywołanie tej funkcji zajmuje nam $f(n)$ czasu. Wówczas będziemy w stanie rozwiązać nasz problem w czasie $O((m+n)f(n))$. Kiedy napełniając kubki pomiędzy a i b, natrafimy na pełny kubek, to będziemy mogli przeskoczyć do najbliższego pustego na prawo w czasie $f(n)$. Każdemu takiemu przeskoczeniu (być może poza ostatnim) możemy przyporządkować napełnienie pustego kubka. Pusty kubek będziemy napełniać co najwyżej n razy, do tego jeszcze trzeba doliczyć co najwyżej m „ostatnich przeskoczeń”, po których nic nie napełniliśmy.

Pozostaje więc kwestia obliczania funkcji *firstEmpty*(). Z pomocą przychodzi nam struktura reprezentująca zbiory rozłączne (tzw. *Find-Union*; patrz [25], [35]). Pełne kubki poprzydzielamy do *bloków*, tak aby zachodził niezmiennik:

> Kubki a oraz b ($a \leqslant b$) należą do tego samego bloku wtedy i tylko wtedy, gdy wszystkie kubki $a, a+1, \ldots, b$ są pełne.

Przynależność do bloków reprezentujemy właśnie za pomocą struktury Find-Union. Dodatkowo dla każdego bloku pamiętamy numer jego najbardziej prawego kubka. Dzięki temu możemy obliczyć *firstEmpty*() za pomocą pojedynczej operacji *find*(), czyli w zamortyzowanym czasie $O(\log^* n)$. Szczegóły związane z uaktualnianiem bloków podczas napełniania kubków pozostawiamy Czytelnikowi do samodzielnego przemyślenia.

Powyższe rozwiązanie zostało zaimplementowane w pliku `raj8.cpp`. Dostawało oczywiście 100 punktów.

Testy

Przygotowano 24 testy rozdzielone pomiędzy 9 grup. Większość małych testów to testy poprawnościowe. Testy duże to w większości testy losowe. Wykorzystano następujące rodzaje grafów:

- ścieżka,
- turniej (czyli DAG-klika),
- graf z losowymi krawędziami wygenerowanymi w taki sposób, aby wierzchołki o niskich numerach miały dużo krawędzi wychodzących, a te o wysokich numerach – dużo krawędzi wchodzących,
- graf z wyróżnionymi dwoma wierzchołkami s i t, zawierający dla każdego innego wierzchołka v krawędzie (s, v) i (v, t),
- ścieżka z usuniętymi niektórymi krawędziami i dodanymi krawędziami w losowych miejscach,
- graf z jednoznacznym rozwiązaniem (wyrzucenie jednego z wierzchołków znacznie zmniejsza długość najdłuższej ścieżki, a każdego z pozostałych – tylko o 1),
- oraz kilka małych grafów wygenerowanych na kartce.

Wykorzystano także grafy powstałe z powyższych na drodze następujących operacji:

- rozłączna suma dwóch grafów A i B (stawiamy grafy A, B obok siebie, nic więcej nie robimy),
- suma krawędzi grafów A i B (A i B muszą mieć tyle samo wierzchołków, bierzemy najpierw graf A, a potem jeszcze dodajemy do niego wszystkie krawędzie z grafu B),
- „połączenie” grafów A i B (stawiamy grafy A, B obok siebie, następnie dla każdej pary $a \in V(A), b \in V(B)$ dodajemy krawędź (a, b)).

Zawody III stopnia

opracowania zadań

Jacek Tomasiewicz Treść zadania
Marek Sommer Opracowanie
Krzysztof Kiljan Program

Dostępna pamięć: 128 MB. *OI, etap III, dzień próbny, 1.04.2014*

FarmerCraft

W wiosce o nazwie Bajtowice znajduje się n domów połączonych za pomocą $n-1$ dróg. Z każdego domu można dojechać do każdego innego na dokładnie jeden sposób. Domy są ponumerowane liczbami od 1 do n. W domu o numerze 1 mieszka sołtys Bajtazar. W ramach programu zapewniania mieszkańcom wsi dostępu do najnowszych technologii, do domu Bajtazara dostarczona została paczka zawierająca n komputerów. Do każdego domu we wsi ma trafić jeden z komputerów. Mieszkańcy Bajtowic zgodnie postanowili, że jak tylko otrzymają komputery, zagrają razem przez sieć w najnowszą wersję gry „FarmerCraft”.

Bajtazar załadował paczkę do swojego samochodu z kratką i za chwilę wyruszy, aby rozwieźć komputery po domach. Benzyny starczy mu tylko na przejechanie każdą drogą co najwyżej dwukrotnie. W każdym domu Bajtazar zostawia jeden komputer i od razu rusza w dalszą drogę. Mieszkańcy każdego domu, natychmiast po otrzymaniu komputera, zabierają się za instalację gry FarmerCraft. Dla każdego domu znany jest czas, jaki jest na to konieczny (zależny od stopnia sprawności informatycznej jego mieszkańców). Po rozwiezieniu wszystkich komputerów Bajtazar wraca do siebie i również zabiera się za instalację gry. Czas przejechania każdą drogą bezpośrednio łączącą dwa domy wynosi dokładnie 1 minutę, a czas wypakowywania komputerów jest pomijalny.

Pomóż Bajtazarowi ustalić taką kolejność rozwożenia komputerów, aby wszyscy mieszkańcy wsi (wraz z Bajtazarem) mogli jak najszybciej rozpocząć grę. Dokładniej, interesuje nas najwcześniejszy moment, w którym wszyscy będą mieli zainstalowaną grę FarmerCraft.

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą n ($2 \leqslant n \leqslant 500\,000$), oznaczającą liczbę domów w Bajtowicach. Drugi wiersz zawiera n liczb całkowitych $c_1, c_2, \ldots, c_n$ ($1 \leqslant c_i \leqslant 10^9$) pooddzielanych pojedynczymi odstępami; c_i oznacza czas instalacji gry (w minutach) przez mieszkańców domu o numerze i.

Kolejne $n-1$ wierszy opisuje drogi łączące domy. Każdy z tych wierszy zawiera dwie dodatnie liczby całkowite a i b ($1 \leqslant a < b \leqslant n$) oddzielone pojedynczym odstępem. Oznaczają one, że istnieje bezpośrednia droga pomiędzy domami o numerach a i b.

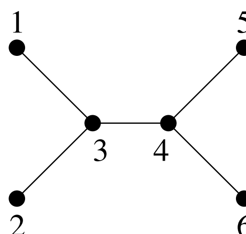
Możesz założyć, że w testach wartych łącznie 40% punktów zachodzi dodatkowy warunek $n \leqslant 7000$.

Wyjście

Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą, równą minimalnemu czasowi (w minutach), po którym wszyscy mieszkańcy będą mogli rozpocząć wspólną grę w FarmerCrafta.

Przykład

Dla danych wejściowych:
```
6
1 8 9 6 3 2
1 3
2 3
3 4
4 5
4 6
```
poprawnym wynikiem jest:
```
11
```



Wyjaśnienie do przykładu: *Bajtazar powinien zawieźć komputery kolejno do domów o numerach: 3, 2, 4, 5, 6 i 1. Gra będzie zainstalowana na komputerach odpowiednio (w kolejności numeracji domów) po: 11, 10, 10, 10, 8 i 9 minutach. Wszyscy mogą rozpocząć grę po 11 minutach.*

Gdyby Bajtazar zawiózł komputery kolejno do domów: 3, 4, 5, 6, 2 i 1, to gra byłaby na nich zainstalowana odpowiednio po: 11, 16, 10, 8, 6 i 7 minutach, a wszyscy mogliby rozpocząć grę dopiero po 16 minutach.

Rozwiązanie

W zadaniu mamy dane drzewo o n wierzchołkach, ukorzenione w wierzchołku nr 1. Każdy wierzchołek i ma przypisaną pewną wartość c_i – czas instalacji gry. Przejazd dowolną krawędzią trwa jedną minutę.

Szukamy trasy o najmniejszym koszcie, która zaczyna się w korzeniu, odwiedza każdy wierzchołek i kończy się w korzeniu. Trasa nie może przechodzić żadną krawędzią więcej niż dwa razy. *Kosztem* trasy nazywamy wartość:

$$\max\left(t_1 + c_1, t_2 + c_2, \ldots, t_n + c_n\right),$$

gdzie t_i oznacza czas rozpoczęcia instalacji gry w i-tym wierzchołku – dla korzenia jest to czas przejścia całej trasy, a dla pozostałych wierzchołków jest to czas, po którym zostaną one odwiedzone po raz pierwszy.

Sposób obchodzenia drzewa

Najistotniejszą informacją w zadaniu jest to, że przez każdą krawędź możemy przejść co najwyżej dwukrotnie. To oznacza, że gdy wejdziemy do pewnego poddrzewa, to nie będziemy mogli z niego wyjść, dopóki nie odwiedzimy wszystkich wierzchołków, które się w tym poddrzewie znajdują. Gdybyśmy wyszli z tego poddrzewa wcześniej, to nigdy byśmy do niego nie wrócili, ponieważ jedyna krawędź łącząca to poddrzewo z resztą drzewa zostałaby już odwiedzona dwukrotnie (podczas wejścia i podczas wyjścia). Trasa, której szukamy, jest zatem obejściem drzewa algorytmem DFS. Dowolny

porządek DFS odpowiada poprawnej trasie, jednak w zadaniu trzeba jeszcze zminimalizować koszt trasy. W tym celu należy w każdym wierzchołku wybrać optymalną kolejność, w jakiej będziemy odwiedzali jego synów.

Instalacja gry w korzeniu

Sposób instalacji gry w korzeniu drzewa (w domu Bajtazara) jest niepotrzebnie wyróżniony. Czas, po którym Bajtazar będzie miał zainstalowaną grę, jest równy czasowi obejścia całego drzewa (ten czas jest równy dwukrotności liczby krawędzi), powiększonemu o c_1. Możemy wyznaczyć tę wartość na samym początku, po czym obliczyć odpowiedź dla uproszczonej wersji zadania, w której w każdym wierzchołku gra jest instalowana w chwili pierwszego odwiedzenia tego wierzchołka (czyli w korzeniu jest instalowana od razu). Końcową odpowiedzią będzie większy z tych dwóch czasów.

Programowanie dynamiczne

Wszystkie podejścia do rozwiązania tego zadania opierają się na metodzie programowania dynamicznego. Aby móc skorzystać z tej metody, niezbędna jest następująca obserwacja:

Obserwacja 1. Optymalna kolejność odwiedzania synów wierzchołka v nie zależy od tego, w jakiej kolejności odwiedzane były inne poddrzewa przed dojściem do v.

Powyższe stwierdzenie jest raczej naturalne – kolejność odwiedzania wierzchołków przed dotarciem do v wpływa tylko na to, po jakim czasie Bajtazar dotrze do poddrzewa v. Niech t będzie czasem instalacji ostatniej gry w poddrzewie v, przy założeniu, że Bajtazar dociera do wierzchołka v w czasie t_1 i przechodzi dane poddrzewo optymalną trasą. Gdyby Bajtazar odwiedził wierzchołek v w czasie t_2, to sposób obejścia tego poddrzewa nie powinien się zmienić, a czas, po którym zostałaby zainstalowana ostatnia gra, byłby równy $t - t_1 + t_2$.

Spróbujmy zatem dla każdego poddrzewa v obliczyć minimalny czas instalacji ostatniej gry w tym poddrzewie, gdyby Bajtazar dotarł do wierzchołka v po czasie 0. Oznaczmy tę wartość przez T_v. Na mocy tego, co powiedzieliśmy do tej pory, taka wartość pozwala obliczyć minimalny czas instalacji, jeśli Bajtazar odwiedzi poddrzewo v w dowolnej innej chwili.

Oprócz czasów T_v, przydadzą nam się jeszcze wartości K_v, które będą oznaczać czas obejścia całego poddrzewa v. Jest to zawsze dokładnie dwukrotność liczby krawędzi, które znajdują się w poddrzewie v.

We wszystkich rozwiązaniach będziemy wyznaczać wartości T_v i K_v dla każdego poddrzewa, zaczynając od liści i kończąc w korzeniu. Po zakończeniu obliczeń, minimalny czas instalacji ostatniej gry w całym drzewie będzie można odczytać z T_1.

Dynamiczne obliczanie wartości T_v i K_v

Niech v będzie liściem. Jeśli wejdziemy do niego po czasie 0, to gra zostanie tam zainstalowana po czasie c_v. Zatem $T_v = c_v$. Skoro v jest liściem, to nie wychodzą z niego żadne krawędzie, więc $K_v = 0$.

Niech teraz v nie będzie liściem. Oznaczmy jego synów przez s_1, s_2, …, s_k. Czas obejścia poddrzewa v jest sumą czasów obejścia wszystkich poddrzew, które z niego wychodzą. Należy do tego doliczyć czas na wejście do syna (1 minuta) i czas na powrót do ojca (1 minuta):

$$K_v = \sum_{i=1}^{k} (1 + K_{s_i} + 1).$$

Aby obliczyć wartość T_v należy ustalić optymalną kolejność odwiedzania synów. Załóżmy na razie, że udało się ją znaleźć: s_{p_1}, s_{p_2}, …, s_{p_k}. Niech $X_{s_{p_i}}$ będzie czasem, po którym Bajtazar wejdzie do poddrzewa s_{p_i}:

$$X_{s_{p_i}} = \sum_{j=1}^{i-1} \left(1 + K_{s_{p_j}} + 1\right) + 1.$$

Na powyższą sumę składają się czasy obejścia wszystkich wcześniejszych poddrzew i czas na przejście krawędzią od wierzchołka v do wierzchołka s_{p_i}.

Na podstawie wartości $X_{s_{p_i}}$ można prosto wyznaczyć wartość T_v:

$$T_v = \max\left(X_{s_{p_1}} + T_{s_{p_1}}, X_{s_{p_2}} + T_{s_{p_2}}, \ldots, X_{s_{p_k}} + T_{s_{p_k}}\right).$$

Powyższe wzory pozwalają w czasie $O(k)$ obliczyć wartości K_v i T_v na podstawie wyników dla poszczególnych synów v. Brakuje jeszcze tylko sposobu na znalezienie optymalnego uporządkowania poddrzew.

Optymalna kolejność odwiedzania poddrzew

Najprostszym rozwiązaniem byłoby sprawdzenie wszystkich możliwych permutacji synów, a spośród nich wybranie tej najtańszej. To rozwiązanie działa w złożoności $O(k!)$ dla pojedynczego wierzchołka, dając w rezultacie algorytm o złożoności $O(n!)$, co dla danych z wejścia (n jest rzędu 500 000) jest zdecydowanie za dużo. Implementacja tego algorytmu znajduje się w pliku `fars2.cpp`. Podczas zawodów za takie rozwiązanie można było otrzymać 20 punktów.

Zastanówmy się, który z synów powinien zostać odwiedzony jako ostatni. Niech G_{s_i} oznacza czas, po którym ostatnia gra zostałaby zainstalowana w poddrzewie s_i, gdyby syn s_i został odwiedzony na samym końcu. Wówczas:

$$\begin{aligned} G_{s_i} &= \sum_{j\in\{1,2,\ldots,k\}\setminus\{i\}} \left(1 + K_{s_j} + 1\right) + 1 + T_{s_i} = \\ &= \sum_{j\in\{1,2,\ldots,k\}} \left(1 + K_{s_j} + 1\right) - (1 + K_{s_i} + 1) + 1 + T_{s_i} = \\ &= K_v - 1 + T_{s_i} - K_{s_i}. \end{aligned}$$

Lemat 1. Istnieje optymalna kolejność przechodzenia poddrzew, w której poddrzewo o najmniejszej wartości G_{s_i} jest odwiedzane jako ostatnie.

Dowód: Ustalmy dowolną optymalną kolejność przechodzenia poddrzew: s_{p_1}, s_{p_2}, $\ldots$, s_{p_k}. Jeśli $G_{s_{p_k}}$ jest najmniejsze, to teza jest spełniona. Załóżmy przeciwnie: niech h będzie takim indeksem, że $G_{s_{p_h}}$ przyjmuje najmniejszą wartość spośród wszystkich G_{s_i} (wówczas $G_{s_{p_h}} < G_{s_{p_k}}$).

Wykażemy, że jeśli przeniesiemy syna s_{p_h} z h-tej pozycji na sam koniec, to kolejność ciągle będzie optymalna.

Spójrzmy, jak zmienią się największe czasy instalacji gier w poszczególnych poddrzewach. W poddrzewach od s_{p_1} do $s_{p_{h-1}}$ nic się nie zmieni. Poddrzewa od $s_{p_{h+1}}$ do s_{p_k} zostaną odwiedzone o $(1 + K_{s_{p_h}} + 1)$ minut wcześniej, więc czasy instalacji mogą się tylko zmniejszyć. Natomiast w poddrzewie s_{p_h}, które zostało przeniesione na sam koniec, największy czas instalacji mógł się zwiększyć. Wiemy jednak, że ten czas jest równy dokładnie $G_{s_{p_h}}$, zatem przeniesienie nie mogło pogorszyć wyniku, który był równy co najmniej $G_{s_{p_k}}$.

Po wykonaniu przeniesienia h-tego elementu na koniec, koszt trasy okazał się nie większy niż przed modyfikacją. Zatem kolejność, w której poddrzewo s_{p_h} jest na końcu, również jest optymalna. ■

Korzystając z lematu 1, można skonstruować algorytm działający w złożoności $O(k^2)$. Algorytm ten będzie obliczał wartości G_{s_i} dla wszystkich synów, wybierał tego z najmniejszym wynikiem i ustawiał go na końcu, a następnie powtarzał całą procedurę dla pozostałych synów (wybierając przedostatniego syna, potem przedprzedostatniego, itd.). Złożoność całego rozwiązania to $O(n^2)$. Implementacja znajduje się w pliku `fars1.cpp`. Na zawodach za takie rozwiązanie można było otrzymać 40 punktów, obiecanych w treści zadania.

Do otrzymania rozwiązania wzorcowego wystarczy już tylko jedna drobna obserwacja. Zamiast k razy wybierać kandydata na ostatni element, możemy posortować wszystkich synów względem różnicy $T_{s_i} - K_{s_i}$. Syn, dla którego wartość $T_{s_i} - K_{s_i}$ jest najmniejsza, ma również najmniejszą wartość G_{s_i}, niezależnie od wartości K_v, która pojawia się we wzorze na G_{s_i}. Nie trzeba zatem przeliczać wartości G_{s_i} na nowo za każdym razem gdy wybierzemy ostatnie poddrzewo – kolejność poddrzew się nie zmieni. W każdym wierzchołku v można więc znaleźć optymalną kolejność w czasie $O(k \log k)$. Całkowita złożoność tego rozwiązania jest równa $O(n \log n)$. Jego implementacje znajdują się w plikach `far.cpp`, `far2.cpp` i `far3.cpp`.

Testy

Do oceny rozwiązań przygotowano 10 grup testów. Występowały następujące rodzaje testów:

- *1a-10a* – testy, w których to Bajtazar ma największy czas instalacji gry,
- *1b-10b*, *1f-6f* – testy obalające rozwiązania zachłanne: istnieje poddrzewo, w którym opłaca się najpierw iść do liścia o mniejszym czasie instalacji,
- *1c-10c* – testy, w których Bajtazar ma stosunkowo mały czas instalacji w porównaniu do innych,

- *1d-10d* – testy, w których z korzenia wyrasta $\sqrt{n}$ poddrzew, z których każde ma po $\sqrt{n}$ węzłów,
- *1e-10e* – testy całkowicie losowe,
- *3g*, *5g*, *6g*, *7f-10f* – testy, w których drzewo ma strukturę gwiazdy.

Jan Kanty Milczek, Jakub Pachocki
Treść zadania, Opracowanie

Jan Kanty Milczek
Program

Dostępna pamięć: 24 MB. *OI, etap III, dzień pierwszy, 2.04.2014*

Dookoła świata

Bajtazar, po wielu latach starań, otrzymał wreszcie upragnioną licencję pilota. Z radości postanowił kupić sobie samolot i oblecieć dookoła całą planetę 3-SATurn (jest to planeta, na której znajduje się Bajtocja). Bajtazar planuje lecieć wzdłuż równika, tak by przelecieć nad każdym jego punktem. Niestety, całą sprawę utrudnia fakt, że samoloty trzeba tankować. Dla każdego samolotu wiadomo, ile co najwyżej kilometrów może on przelecieć, jeśli zaczynał z pełnym bakiem. Paliwo można uzupełnić na dowolnym lotnisku położonym na równiku, co niestety wiąże się z lądowaniem na tym lotnisku.

Ponieważ kupno samolotu nie jest łatwą sprawą, Bajtazar prosi Cię o pomoc. Przedstawił Ci propozycje różnych modeli samolotów – mogą one różnić się pojemnością baku. Dla każdego z tych modeli należy wyznaczyć minimalną liczbę lądowań (włącznie z finalnym lądowaniem), które będzie musiał wykonać Bajtazar w celu ukończenia podróży dookoła świata. Dla każdego modelu podróż można zaczynać od dowolnego lotniska.

Wejście

Pierwszy wiersz standardowego wejścia zawiera dwie liczby całkowite n i s ($2 \leqslant n \leqslant 1\,000\,000$, $1 \leqslant s \leqslant 100$) oddzielone pojedynczym odstępem, oznaczające liczbę lotnisk położonych na równiku i liczbę samolotów, jakich kupno rozważa Bajtazar.

Drugi wiersz zawiera n dodatnich liczb całkowitych $l_1, l_2, \ldots, l_n$ ($l_1 + l_2 + \ldots + l_n \leqslant 10^9$) pooddzielanych pojedynczymi odstępami, opisujących odległości między kolejnymi lotniskami na równiku. Liczba l_i oznacza, że odległość między i-tym a $(i+1)$-szym (lub n-tym i pierwszym, jeśli $i = n$) lotniskiem wynosi l_i kilometrów.

Trzeci wiersz zawiera s liczb całkowitych $d_1, d_2, \ldots, d_s$ ($1 \leqslant d_i \leqslant l_1 + l_2 + \ldots + l_n$) pooddzielanych pojedynczymi odstępami. Liczba d_i oznacza, że i-ty samolot jest w stanie przelecieć d_i kilometrów, zanim będzie musiał lądować i zatankować.

W testach wartych 50% punktów zachodzi warunek $n \leqslant 100\,000$, a w podzbiorze tych testów wartym 20% punktów zachodzi dodatkowy warunek $n \leqslant 1000$.

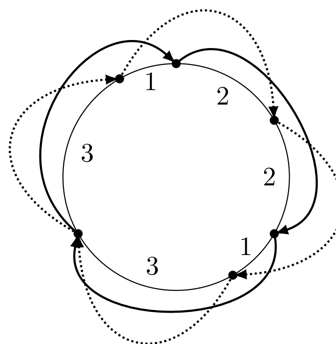
W innych testach (nie wymienionych powyżej) wartych 18% punktów zachodzi warunek: $s \leqslant 5$.

Wyjście

Na standardowe wyjście należy wypisać s wierszy: i-ty z nich powinien zawierać jedną liczbę całkowitą oznaczającą minimalną liczbę lotów, które trzeba wykonać i-tym samolotem, aby oblecieć planetę 3-SATurn dookoła wzdłuż równika, zaczynając z dowolnego lotniska, lub słowo `NIE`*, jeśli nie da się tym samolotem odbyć podróży.*

Przykład

Dla danych wejściowych:
```
6 4
2 2 1 3 3 1
3 2 4 11
```
poprawnym wynikiem jest:
```
4
NIE
3
2
```



Wyjaśnienie do przykładu: *Na rysunku pogrubiona ciągła linia reprezentuje optymalną podróż samolotu o pojemności baku 4, zaś linią przerywaną oznaczono podróż samolotu o pojemności baku 3.*

Testy „ocen”:

`1ocen`: *$n = 8$, $s = 3$, mały test sprawdzający kilka przypadków brzegowych;*

`2ocen`: *$n = 24$, $s = 8$, test, w którym każde kolejne dwa lotniska są odległe o 1;*

`3ocen`: *$n = 1\,000\,000$, $s = 100$, test maksymalnego rozmiaru.*

Rozwiązanie

Co widzimy

Widząc ograniczenia z treści zadania, dochodzimy do wniosku, że oczekuje się od nas obliczenia odpowiedzi dla każdego samolotu w czasie liniowym – jest to słuszny wniosek.

Pierwsza oczywista obserwacja

Zawsze opłaca się latać w tym samym kierunku. Jeśli jakaś sekwencja lotów w prawo (na wschód) rozwiązuje zadanie, to można ją odwrócić, otrzymując rozwiązanie składające się z lotów w lewo (na zachód). Dla zupełnej jasności dodajmy jeszcze, że nie ma żadnego sensu zawracać. Od tej chwili zakładamy, że wykonujemy loty tylko w prawo.

Druga oczywista obserwacja

Nie ulega wątpliwości, że istnieje optymalna podróż, w której każdy lot – być może oprócz ostatniego – jest tak długi, jak tylko się da. Rozpatrywanie danego samolotu

warto zacząć od obliczenia dla każdego lotniska, jakie jest najdalsze lotnisko po prawej, na które jesteśmy w stanie dolecieć na jednym baku. Możemy to prosto wykonać tzw. *funkcją gąsienicową* – korzystając z obserwacji, że jeśli z lotniska a najdalej można dolecieć na lotnisko b ($b \neq a$), to z lotniska $a+1$ na pewno można dolecieć na lotnisko b, a może i do lotnisk o większych numerach (modulo n). Zwiększając a od 1 do n, indeks b również tylko zwiększamy i wykonamy nim łącznie najwyżej $2n$ zwiększeń, bo przecież z ostatniego lotniska nie możemy lecieć dalej niż o n lotnisk.

Rozwiązanie $O(sn \log n)$ w pamięci $O(n \log n)$

Wiedząc, jak daleko można dolecieć jednym lotem, możemy obliczyć w czasie i pamięci $O(n \log n)$, jak daleko można z każdego lotniska dolecieć 2^i lotami dla dowolnego $i = 0, 1, \ldots, \lfloor \log n \rfloor$. W tych obliczeniach korzystamy z prostej obserwacji, że aby przemieścić się z danego lotniska o 2^i lotów, wystarczy dwukrotnie przemieścić się o 2^{i-1} lotów.

Znając dla każdego lotniska „zasięg” 2^0 lotami, 2^1 lotami, 2^2 lotami... możemy dla danego lotniska obliczyć (w czasie $O(\log n)$), ilu lotów potrzeba, aby oblecieć świat. Aby to zrobić, wybieramy największe takie i, że 2^i lotów *nie* wystarczy do oblecenia świata, i wykonujemy te loty. Następnie sprawdzamy w podobny sposób, ile jeszcze lotów trzeba, aby dokończyć podróż. Zauważmy, że kolejne wybierane i będą coraz mniejsze (zamiast wybrać i i i mogliśmy wybrać $i+1$).

Problem

Moglibyśmy tak rozwiązać zadanie, gdyby nie bezlitosne ograniczenie pamięci – 24 MB. Powyższe rozwiązanie, zaimplementowane w pliku `doos3.cpp`, zgodnie z sugestią w treści zadania uzyskiwało na zawodach połowę punktów.

Nieoczywista obserwacja

Obliczmy (w czasie liniowym) wynik dla jakiegoś lotniska (np. lotniska nr 1) – oznaczmy ten wynik x. Zauważmy, że jeśli zaczynając z jakiegoś lotniska, potrzebujemy x lotów, to z każdego innego potrzebujemy x, $x-1$ albo $x+1$ lotów. Jest tak dlatego, że z dowolnego lotniska możemy wlecieć na jakieś lotnisko należące do (jakiegoś) rozwiązania optymalnego, później wykonać wszystkie loty wchodzące w skład rozwiązania optymalnego poza ostatnim, a na koniec wrócić na lotnisko startowe.

Rozwiązanie $O(sn \log n)$ w pamięci liniowej

Oznacza to, że musimy jedynie sprawdzić, czy da się z jakiegoś lotniska oblecieć świat w $x-1$ lotów. Możemy w takim razie obliczyć dla każdego lotniska, jak daleko dolecimy z niego $x-1$ lotami. W tym celu przedstawimy $x-1$ jako sumę potęg dwójki i znów zastosujemy pomysł z obliczaniem zasięgów dla lotnisk w 2^0 lotów, 2^1 lotów itd. Tym razem jednak wystarczy nam pamięć $O(n)$. Będziemy przemieszczać się o 2^i lotów ze wszystkich lotnisk naraz, dla tych potęg dwójki, które występują w przedstawieniu

binarnym $x - 1$. Natomiast przy obliczaniu zasięgów w 2^{i+1} lotów wystarczy nam pamiętać zasięgi w 2^i lotów.

Rozwiązanie takie jest zaimplementowane w pliku `doos1.cpp`. Treść zadania sugeruje, że takie rozwiązanie zdobywa ok. 70% punktów.

Rozwiązanie wzorcowe $O(sn)$

Zastanówmy się, kiedy punkt startowy potrzebuje o jeden więcej lot niż rozwiązanie optymalne. Dzieje się tak w dwóch przypadkach:

- w pewnym momencie podczas okrążania świata „wlatujemy" na jakieś lotnisko zawarte w rozwiązaniu optymalnym;
- cały czas latamy pomiędzy lotniskami z rozwiązania optymalnego, przenosząc się do o jeden dalszego „odcinka" (tj. fragmentu pomiędzy „optymalnymi" lotniskami), aż w końcu wracamy do pierwszego odcinka, ale w jego wcześniejszej części.

Zauważmy, że jeśli jakieś lotnisko należy do jakiegokolwiek rozwiązania optymalnego, to po wykonaniu z niego lotu o maksymalnej długości znajdziemy się w innym lotnisku należącym do jakiegoś rozwiązania optymalnego. To ważne spostrzeżenie oznacza, że jeśli nasze startowe lotnisko jest lotniskiem o wyniku nieoptymalnym pierwszego typu, to po dokładnie n lotach znajdziemy się na pewno w optymalnym punkcie startowym. Jeśli za to startowe lotnisko jest lotniskiem nieoptymalnym typu drugiego, to po powrocie do startowego odcinka, możemy rozpocząć loty od lotniska, na jakie trafiliśmy. Jeśli jest ono nieoptymalne typu pierwszego, to wiemy już, co zrobić (co więcej, do chwili znalezienia się w jakimś optymalnym punkcie startowym nie odwiedzimy żadnego z już odwiedzonych lotnisk, więc wystarczy nam n lotów minus liczba już wykonanych lotów). Jeśli natomiast jest typu drugiego, to znowu wykonamy jakieś loty i wrócimy na ten sam odcinek – w jeszcze wcześniejszej części. Kontynuujmy latanie. W pewnym momencie powtórzy się jakiś wierzchołek (wtedy mamy pewność, że jest on lotniskiem optymalnym – przecież za każdym razem zmienialiśmy odcinek pomiędzy takimi). Wierzchołek powtórzy się po najwyżej n lotach.

Podsumowując – po wykonaniu n lotów z dowolnego miejsca zawsze znajdujemy w optymalnym punkcie startowym. Obliczenie wyniku jest już tylko formalnością. Rozwiązanie takie jest zaimplementowane w pliku `doo.cpp`.

Testy

Większość testów ma z góry ustalony wynik dla pewnej długości lotu, składa się z lotnisk parami oddalonych o mniej więcej tyle, a oprócz tego zawiera lotniska „wabiki", które mają różne, atrakcyjne dla rozwiązań heurystycznych, cechy. Testy z grup a mają najwięcej wabików, testy b średnio wiele, a testy c najmniej (gdzie najmniej to około $\frac{9}{10}$ lotnisk, a najwięcej to ok. $\frac{29999}{30000}$). Testy d to testy całkowicie losowe.

Jacek Tomasiewicz **Bartłomiej Dudek**
Treść zadania, Opracowanie Program

Dostępna pamięć: 128 MB. *OI, etap III, dzień pierwszy, 2.04.2014*

Mrowisko

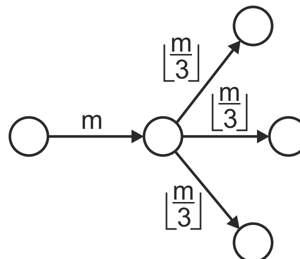
Mrówki plądrują opuszczone mrowisko w poszukiwaniu jedzenia. Mrowisko składa się z n komór oraz łączących je $n-1$ korytarzy. Wiemy, że z każdej komory do każdej innej komory można przejść w dokładnie jeden sposób. Inaczej mówiąc, komory i korytarze tworzą drzewo.

We wszystkich komorach, do których prowadzi tylko jeden korytarz, znajdują się wejścia do mrowiska. Przy każdym z wejść ustawiło się po g grup mrówek składających się kolejno z $m_1, m_2, \ldots, m_g$ mrówek. Grupy będą wchodziły do mrowiska pojedynczo (kolejna grupa wchodzi dopiero wtedy, gdy w mrowisku nie ma już żadnych mrówek). Wewnątrz mrowiska mrówki poruszają się w określony sposób:

- *Po wejściu do komory, w której zbiega się d nieodwiedzonych jeszcze przez daną grupę mrówek korytarzy, grupa ta dzieli się na d równolicznych grup. Każda powstała w ten sposób grupa porusza się dalej jednym z tych d korytarzy. Jeśli $d = 0$, to grupa mrówek po prostu opuszcza mrowisko.*

- *Jeśli mrówki nie mogą podzielić się na równoliczne grupy, to silniejsze osobniki zjadają słabsze, do momentu, aż będzie możliwy podział na grupy równej wielkości (w szczególności, jedna mrówka może zjeść samą siebie i zniknąć).*

Poniższy rysunek przedstawia grupę m mrówek, które wchodzą do komory, w której zbiegają się trzy nieodwiedzone jeszcze korytarze, a następnie dzielą się na trzy równe grupy o licznościach $\lfloor m/3 \rfloor$.



Nad jednym z korytarzy znajduje się otwór, przez który do środka dostał się długi język głodnego mrówkojada. Wiemy, że mrówkojad zjada każdą przechodzącą tym korytarzem grupę mrówek, która składa się z dokładnie k mrówek. Chcemy wiedzieć, ile mrówek zje mrówkojad.

Wejście

Pierwszy wiersz standardowego wejścia zawiera trzy liczby całkowite n, g, k ($2 \leqslant n, g \leqslant 1\,000\,000$, $1 \leqslant k \leqslant 10^9$) pooddzielane pojedynczymi odstępami. Oznaczają one odpowiednio liczbę komór, liczbę grup mrówek oraz liczność grup mrówek zjadanych przez mrówkojada. Komory są ponumerowane liczbami od 1 do n.

Drugi wiersz zawiera g liczb całkowitych $m_1, m_2, \ldots, m_g$ $(1 \leqslant m_i \leqslant 10^9)$ *pooddzielanych pojedynczymi odstępami, gdzie* m_i *oznacza liczność i-tej grupy mrówek czekającej przy każdym z wejść do mrowiska. Kolejnych* $n-1$ *wierszy opisuje korytarze mrowiska; i-ty z nich zawiera dwie liczby całkowite* a_i, b_i $(1 \leqslant a_i, b_i \leqslant n)$ *oddzielone pojedynczym odstępem, oznaczające, że komory o numerach* a_i *i* b_i *są połączone korytarzem. Język mrówkojada znajduje się w korytarzu, którego opis pojawia się jako pierwszy na wejściu.*

W testach wartych 50% punktów liczba wszystkich grup mrówek wchodzących do mrowiska nie przekroczy $1\,000\,000$*. Ponadto w podzbiorze tych testów wartym 20% punktów zachodzi dodatkowy warunek* $n, g \leqslant 100$*.*

Wyjście

Twój program powinien wypisać na standardowe wyjście jeden wiersz zawierający jedną liczbę całkowitą, oznaczającą liczbę mrówek, które zje mrówkojad.

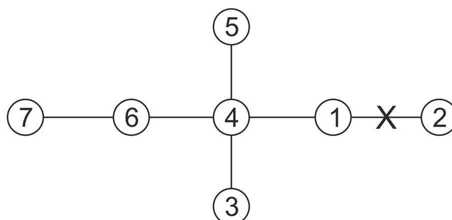
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
21
```



Wyjaśnienie do przykładu: *Przy każdej z komór o numerach 2, 3, 5 i 7 ustawia się po 5 grup mrówek. Mrówkojad zje 3 mrówki z pierwszej grupy startującej z komory 2 oraz po 3 mrówki z czwartych i piątych grup startujących z komór 3, 5 i 7.*

Testy „ocen”:

`1ocen`: $n = 20$, $g = 20$, $k = 5$, *komory w mrowisku są połączone korytarzami w jeden długi tunel. Język mrówkojada znajduje się w krańcowym korytarzu. Rozmiary grup to* $1, \ldots, 20$*.*

`2ocen`: $n = 2^{19} + 1$, $g = 20$, $k = 1$, *budowa mrowiska jest następująca: komora 1 jest połączona z komorą n (w tym korytarzu znajduje się mrówkojad), a komora i-ta (dla* $i = 2, 3, \ldots, n-1$*) jest połączona z komorą* $\left\lfloor \frac{i}{2} \right\rfloor$*. Do mrowiska wchodzą grupy mrówek o rozmiarach będących kolejnymi potęgami dwójki:* $2^0, 2^1, \ldots, 2^{19}$*.*

Rozwiązanie

Mrowisko w zadaniu opisane jest jako graf będący drzewem, po którym chodzą grupy mrówek. Każda z grup startuje kolejno w każdym z liści i w każdym wierzchołku dzieli się na x grup, gdzie $x+1$ to stopień tego wierzchołka. Wyjątkiem są wierzchołki startowe, w których grupy nie dzielą się. Dzielenie jest całkowitoliczbowe i każda z mniejszych grup wyrusza jedną z krawędzi, z pominięciem tej, którą przyszła. Gdy jakaś grupa dojdzie do liścia, to znika.

W drzewie zaznaczamy jedną krawędź i musimy policzyć, ile grup o liczności k przez nią przeszło. Oznaczmy końce tej krawędzi przez a i b.

Zanim zaczniemy myśleć nad różnymi rozwiązaniami, zauważmy, że dla każdej grupy mrówek wchodzącej do mrowiska, co najwyżej jedna grupa mrówek pochodząca z tej początkowej grupy przejdzie obok mrówkojada. Jest tak dlatego, iż mrówki z danej grupy idą zawsze nieodwiedzoną krawędzią, do momentu aż ich grupa będzie za mała, żeby się podzielić, lub dojdą do liścia. Z tej obserwacji będą korzystały wszystkie przedstawione rozwiązania.

Liczba wierzchołków drzewa to n, a wszystkich grup mrówek jest g. Dodatkowo, przez m oznaczymy górne ograniczenie na liczność grupy, czyli dokładniej $m = \max(m_1, m_2, \ldots, m_g)$.

Rozwiązanie siłowe $O(n^2 \cdot g)$

Najprostszym rozwiązaniem jest symulacja ruchu każdej grupy mrówek z każdego liścia. Wszystkich grup jest g, liczba liści i rozmiar drzewa jest rzędu $O(n)$, więc złożoność czasowa wyniesie $O(n^2 \cdot g)$.

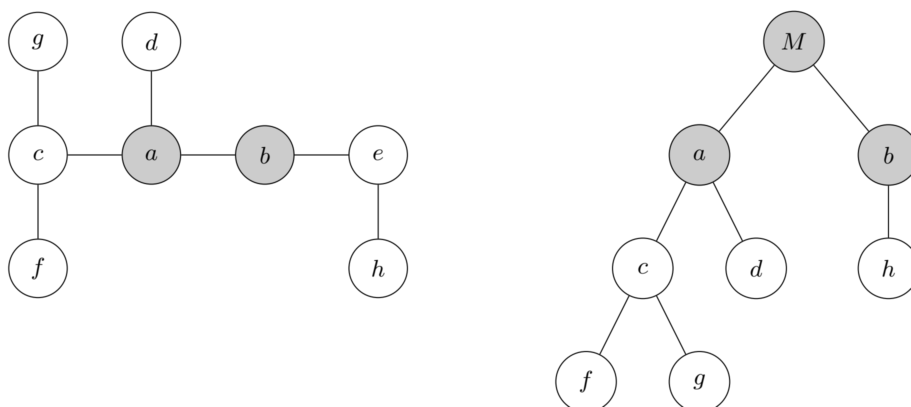
Za takie rozwiązanie można było uzyskać około 20% punktów. Implementacje znajdują się w plikach `mros1.cpp` oraz `mros4.pas`.

Rozwiązanie wolne $O(n \cdot g \cdot \log m)$

Spróbujmy skompresować wejściowe drzewo. Zauważmy, że możemy usunąć wierzchołki (różne od a i b), z których wychodzą tylko dwie krawędzie. Takie wierzchołki nie zmieniają wyniku, ponieważ gdy grupa przez nie przechodzi, jej liczność nie zmienia się (dzielimy przez 1).

Następnie dla każdego wierzchołka powinniśmy umieć stwierdzić, którą krawędzią iść, aby dojść do mrówkojada (będzie tylko jedna taka krawędź, a wszystkie inne nas nie interesują, ponieważ nie wpływają na wynik). Takie kierunki możemy wyznaczyć, przykładowo, ukorzeniając nasze drzewo w miejscu mrówkojada (rys. 1). W ten sposób, poruszając się w górę drzewa, będziemy iść zawsze w kierunku mrówkojada.

Przypomnijmy, że m oznacza górne ograniczenie na liczność grupy. Chodzenie z każdego liścia, zawsze w stronę mrówkojada (dopóki rozmiar grupy jest niezerowy), będzie miało złożoność $O(\log m)$, gdyż w każdym kroku zmniejszamy rozmiar grupy przynajmniej dwukrotnie. Ostatecznie, złożoność całego rozwiązania wyniesie $O(n \cdot g \cdot \log m)$.



Rys. 1: Drzewo po prawej stronie powstaje przez ukorzenienie drzewa po lewej stronie w pozycji mrówkojada. W drzewie po prawej stronie usunęliśmy wierzchołek e, ponieważ wychodziły z niego tylko dwie krawędzie.

Za takie rozwiązanie można było uzyskać około 50% punktów, a implementacje znajdują się w plikach `mros2.cpp` oraz `mros5.pas`.

Rozwiązanie wzorcowe $O((n+g)\log g)$

Zastanówmy się, jak duża grupa mrówek powinna znaleźć się w wierzchołku a, aby mrówki, które udadzą się w kierunku wierzchołka b, zostały zjedzone przez mrówkojada.

Wyznaczanie przedziałów

Niech p_u oznacza liczbę grup, na które podzielą się mrówki, będąc w wierzchołku u. Wiemy, że mrówkojad zjada grupy złożone z k mrówek, a więc jeśli w wierzchołku a będzie dokładnie $p_a \cdot k$ mrówek, to po podziale zostanie ich dokładnie k. Ponieważ dzielimy całkowitoliczbowo, to możemy dodać do tego dowolną liczbę mniejszą od p_a, czyli wartości $p_a \cdot k + 1$, $p_a \cdot k + 2$, $\ldots$, $p_a \cdot k + p_a - 1$ również będą poprawne. Ogólniej, dostajemy przedział liczb całkowitych od wartości $x = p_a \cdot k$ (włącznie) do $y = p_a \cdot (k+1)$ (wyłącznie). Taki przedział oznaczmy jako $[x..y)$.

Znaleźliśmy przedział określający liczbę mrówek, które powinny być w wierzchołku a. Cofnijmy się o jeden wierzchołek wcześniej, czyli weźmy pod uwagę sąsiadów wierzchołka a. Następnie dla nich spróbujmy znaleźć przedziały, takie że (po podzieleniu) do wierzchołka a przybędzie szukana liczba mrówek, czyli wartość z przedziału $[x..y)$ (co w efekcie spowoduje, że do mrówkojada dojdzie dokładnie k mrówek).

Dla ustalenia uwagi, rozpatrzmy jednego z sąsiadów, wierzchołek c. W tym wierzchołku mrówki podzielą się na p_c grup. W związku z tym, każda wartość z przedziału $[p_c \cdot x..p_c \cdot y)$ będzie poprawna. Zauważmy, że ponownie jest to spójny przedział.

W ten sposób uzyskujemy schemat konstrukcji przedziałów, który możemy powtarzać, dla coraz to bardziej oddalonych wierzchołków, aż dojdziemy do liści. Koszt

wyznaczenia przedziału dla każdego wierzchołka jest stały. Jeśli będziemy się poruszać od położenia mrówkojada do wszystkich innych wierzchołków (na przykład przeszukiwaniem w głąb), to każdy z nich odwiedzimy raz, więc w czasie $O(n)$ uzyskamy przedziały wartości dla wszystkich liści. Warto zauważyć, że jeśli w jakimś wierzchołku początek wyznaczonego przedziału jest większy niż m, to nie musimy już obliczać wyników dla wierzchołków w jego poddrzewie, gdyż w takim przypadku i tak żadna grupa mrówek z tego poddrzewa nie dotrze do mrówkojada.

Sprawdzanie grup mrówek

Znając te przedziały, umiemy odpowiadać w czasie stałym, czy grupa o rozmiarze w, startując z ustalonego liścia, zostanie zjedzona przez mrówkojada. Wystarczy sprawdzić, czy wartość w zawiera się w przedziale obliczonym dla tego liścia.

Jeśli wykonamy to dla wszystkich grup i wszystkich liści, to złożoność całego rozwiązania wyniesie $O(n \cdot g)$. Takie rozwiązanie otrzymywało 50% punktów i zostało zaimplementowane w plikach `mros3.cpp` oraz `mros6.pas`.

Przyspieszenie rozwiązania

Powyższe rozwiązanie można jeszcze usprawnić. Wystarczy posortować wszystkie rozmiary grup mrówek, a następnie dla każdego liścia wyszukać binarnie spójny podciąg grup mrówek, które zostaną zjedzone przez mrówkojada.

Sortowanie zajmuje czas $O(g \log g)$, a wyszukiwanie binarne $O(n \log g)$. Całkowita złożoność czasowa wyniesie zatem $O((n+g) \log g)$. Takie rozwiązanie zostało zaimplementowane w plikach `mro.cpp` oraz `mro1.pas`. Przy implementacji należy pamiętać, że wynik powinniśmy przechowywać w 64-bitowym typie zmiennych całkowitych. Ponadto polecamy Czytelnikowi zastanowienie się nad tym, dlaczego taki typ danych jest wystarczający (na pierwszy rzut oka nie jest to wcale oczywiste).

Testy

Każda grupa składała się z 5 testów. We wszystkich testach drzewo po jednej stronie mrówkojada zostało wygenerowane losowo, a po drugiej stronie znajdowało się:

- `a` – pełne drzewo binarne o korzeniu przy mrówkojadzie i z kilkoma małymi drzewami doczepionymi wewnątrz,
- `b` – pełne drzewo binarne, doczepione liściem do krawędzi z mrówkojadem,
- `c` – ścieżka długości $\frac{n}{3}$ z doczepionymi gwiazdami o rozmiarach około $\frac{n}{20}$,
- `d` – ścieżka długości $\frac{n}{6}$ z doczepionymi poddrzewami,
- `e` – $n-2$ wierzchołki doczepione bezpośrednio do krawędzi z mrówkojadem.

Marcin Pilipczuk — Treść zadania, Opracowanie
Marcin Andrychowicz, Adam Karczmarz — Program

Dostępna pamięć: 256 MB. *OI, etap III, dzień pierwszy, 2.04.2014*

Turystyka

Król Bajtazar jest przekonany, że pełna turystycznych atrakcji Bajtocja powinna przyciągać rzesze turystów, a ci powinni wydawać pieniądze, przyczyniając się do wypełnienia królewskiego skarbca. Tak się jednak nie dzieje. Król polecił swojemu doradcy przyjrzeć się bliżej sytuacji. Ten odkrył, że przyczyną małej popularności królestwa wśród obcokrajowców jest niedostatecznie rozwinięta sieć dróg.

Nadmieńmy, że w Bajtocji jest n miast i m dwukierunkowych dróg, z których każda łączy dwa różne miasta. Drogi mogą prowadzić tunelami i estakadami. Nie ma gwarancji, że z każdego miasta da się dojechać do każdego innego.

Doradca zauważył, że obecna sieć dróg nie pozwala na zorganizowanie żadnej długiej wycieczki. Zaczynając wycieczkę w dowolnym z miast i poruszając się drogami, nie jesteśmy w stanie odwiedzić więcej niż 10 miast bez przejeżdżania przez to samo miasto dwukrotnie.

Z braku funduszy, zamiast budować nowe drogi, Bajtazar postanowił zbudować w Bajtocji sieć punktów informacji turystycznej (PIT), w których odpowiednio przeszkoleni pracownicy będą reklamować zalety krótkich wycieczek. Dla każdego miasta, PIT powinien znajdować się albo w tym mieście, albo w którymś z miast bezpośrednio połączonych z nim drogą. Dla każdego miasta znany jest koszt wybudowania PIT-u w tym mieście. Pomóż Bajtazarowi znaleźć najtańszy sposób zbudowania sieci PIT-ów.

Wejście

Pierwszy wiersz standardowego wejścia zawiera dwie liczby całkowite n, m ($2 \leqslant n \leqslant 20\,000$, $0 \leqslant m \leqslant 25\,000$) oddzielone pojedynczym odstępem, oznaczające odpowiednio liczbę miast i dróg w Bajtocji. Miasta są ponumerowane liczbami od 1 do n. Drugi wiersz wejścia zawiera n liczb całkowitych $c_1, c_2, \ldots, c_n$ ($0 \leqslant c_i \leqslant 10\,000$) pooddzielanych pojedynczymi odstępami; liczba c_i oznacza koszt zbudowania PIT-u w mieście o numerze i.

Dalej następuje opis dróg w Bajtocji. W i-tym z kolejnych m wierszy znajdują się dwie liczby całkowite a_i, b_i ($1 \leqslant a_i < b_i \leqslant n$) oddzielone pojedynczym odstępem, oznaczające, że miasta o numerach a_i i b_i są połączone drogą. Pomiędzy każdą parą miast istnieje co najwyżej jedna droga.

W testach wartych łącznie 20% punktów zachodzi dodatkowy warunek $n \leqslant 20$.

Wyjście

Twój program powinien wypisać na standardowe wyjście jedną liczbę całkowitą, oznaczającą łączny koszt budowy wszystkich PIT-ów.

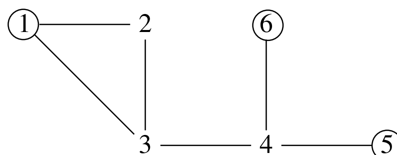
Przykład

Dla danych wejściowych:

```
6 6
```

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

poprawnym wynikiem jest:
```
7
```



Wyjaśnienie do przykładu: *Aby uzyskać minimalny koszt budowy, PIT-y powinny zostać zbudowane w miastach o numerach 1, 5 i 6 (koszt wyniesie 3 + 2 + 2 = 7).*

Testy „ocen”:

`1ocen`: *$n = 10$, $m = 9$, sieć dróg tworzy ścieżkę długości 10;*

`2ocen`: *$n = 10$, $m = 45$, między każdą parą miast jest droga;*

`3ocen`: *$n = 20\,000$, $m = 19\,998$, ze wszystkich miast z wyjątkiem 1 i 2 wychodzi dokładnie jedna droga – do miasta 1 lub 2.*

Rozwiązanie

Na początek przetłumaczmy treść zadania na język teorii grafów. Mamy dany na wejściu nieskierowany graf $G = (V, E)$ o n wierzchołkach (reprezentujących miasta) i m krawędziach (reprezentujących drogi), wraz z wagami $c : V \to \mathbb{Z}_{\geqslant 0}$. Naszym celem jest znalezienie podzbioru $X \subseteq V$ o minimalnym możliwym koszcie tak, by każdy wierzchołek $v \in V$ należał do X lub miał sąsiada należącego do X. Zbiór X będzie oczywiście zawierał wierzchołki reprezentujące miasta, w których zbudowane zostaną punkty informacji turystycznej.

W języku teorii grafów mówimy, że taki zbiór X jest *zbiorem dominującym*: wierzchołek v *dominuje* wierzchołek w jeśli $v = w$ lub $vw \in E$, zbiór X dominuje zbiór Y jeśli dla każdego $y \in Y$ istnieje $x \in X$ dominujący y, a zbiór dominujący to taki podzbiór V, który dominuje wszystkie wierzchołki grafu. Podsumowując, naszym zadaniem jest znalezienie najtańszego (względem wag c) zbioru dominującego w danym grafie.

Zauważmy, że naiwne rozwiązanie, sprawdzające wszystkie możliwe podzbiory $X \subseteq V$, działa w czasie $O(2^n(n + m))$: sprawdzenie, czy dany zbiór X jest zbiorem dominującym, i wyznaczenie jego kosztu można łatwo wykonać w czasie $O(n+m)$. To rozwiązanie, zaimplementowane w plikach `turs3.cpp` i `turs6.pas`, będzie skuteczne dla $n \leqslant 20$, otrzymując około 20% punktów, lecz na pewno nie poradzi sobie dla większych grafów. Co więcej, problem znalezienia najtańszego zbioru dominującego

jest problemem bardzo trudnym i jedyne znane algorytmy rozwiązujące go działają w czasie wykładniczym od liczby wierzchołków w grafie ($O(\gamma^n)$ dla pewnej stałej $1 < \gamma < 2$). Jest to szybciej niż algorytm naiwny, ale wciąż niewystarczająco.

Pozostaje nam wykorzystać ostatnią informację zawartą w treści zadania, którą być może część czytelników uznała tylko za tło fabularne: w Bajtocji nie można odwiedzić więcej niż $t = 10$ miast, nie przejeżdżając przez jedno miasto dwukrotnie. W języku teorii grafów oznacza to, że w grafie G nie ma ścieżki prostej (takiej, na której nie powtarzają się wierzchołki) dłuższej niż $t = 10$.[1] Zachęcam do próby narysowania sobie przykładu „skomplikowanego” grafu, który nie ma długiej ścieżki – jest to dość trudne. Pozwala to mieć nadzieję, że ta własność grafu G powoduje, że musi mieć on określoną strukturę, przez co problem znajdowania najtańszego zbioru dominującego staje się istotnie prostszy.

Załóżmy, że dany graf G jest spójny – w przeciwnym razie, możemy osobno rozważyć każdą spójną składową G. Weźmy dowolny wierzchołek $r \in V$ i przeszukajmy graf w głąb (DFS), zaczynając z wierzchołka r. Niech T będzie drzewem rozpinającym grafu G, wygenerowanym przez to przeszukiwanie (tzn. drzewo T jest ukorzenione w wierzchołku r i dla każdego wierzchołka $v \in V$ kolejne dzieci v w drzewie T odpowiadają kolejnym rekurencyjnym wywołaniom procedury przeszukiwania grafu w głąb, wywoływanym w czasie rozpatrywania wierzchołka v). Następująca obserwacja okazuje się kluczowa:

Obserwacja 1. Głębokość drzewa T (liczba wierzchołków na najdłuższej ścieżce od liścia do korzenia r) jest **nie większa niż** t.

Dowód: Ścieżka od liścia do korzenia w drzewie T jest również ścieżką prostą w grafie G. ■

Zastanówmy się teraz, jak wykorzystać to spostrzeżenie. Pomyślmy o przeszukiwaniu grafu w głąb jako o procedurze, która, rozpatrując wierzchołek v, posiada pewien zbiór wierzchołków S na stosie (są to przodkowie wierzchołka v w drzewie T, wraz z wierzchołkiem v), odwiedziła już całkowicie pewne wierzchołki (oznaczmy ich zbiór przez A) i jeszcze nie odwiedziła wierzchołków $B = V \setminus (A \cup S)$. Warto myśleć o zbiorze A jako o zbiorze wierzchołków *na lewo* od ścieżki S w drzewie T, a o zbiorze B jako o zbiorze wierzchołków *na prawo* od tej ścieżki; przykładową konfigurację zbiorów S, A i B przedstawia rysunek 1. Poczyńmy następującą obserwację:

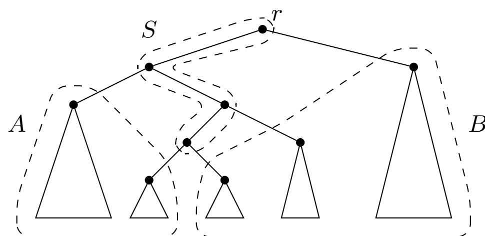
Obserwacja 2. Nie ma żadnej krawędzi między zbiorem A a zbiorem B.

Dowód: Gdyby taka krawędź ab istniała dla pewnych wierzchołków $a \in A$, $b \in B$, to algorytm przeszukiwania w głąb poszedłby tą krawędzią i odwiedziłby wierzchołek b, zanim by całkowicie opuścił wierzchołek a. ■

Z obserwacji 1 wynika, że $|S| \leqslant t$, gdyż zbiór S odpowiada ścieżce od wierzchołka v do korzenia r w drzewie T. Inaczej mówiąc, zbiór S jest małym *separatorem*, którego usunięcie, zgodnie z obserwacją 2, dzieli graf na dwie niepołączone części A i B. Co więcej, *każda* ścieżka z wierzchołka do korzenia w drzewie T jest takim separatorem.

---

[1] Dla zachowania ogólności rozumowania, w dalszej części będziemy używać litery t na oznaczenie ograniczenia na długość ścieżki w grafie G i na koniec „przypomnimy sobie”, że $t = 10$.



Rys. 1: Przykładowy zbiór S, dzielący graf na część już odwiedzoną A i część jeszcze nie odwiedzoną B. Po lewej: zgodna równoodległa trójka. Po prawej: niezgodna równoodległa trójka.

Ideą rozwiązania jest, by wykorzystać te separatory do zaprojektowania algorytmu opartego o programowanie dynamiczne, niejako „zamiatając” drzewo T od lewej do prawej.

Programowanie dynamiczne

Przypomnijmy, że mamy ustalone drzewo T wyznaczone przez algorytm przeszukiwania grafu w głąb. Stan algorytmu przeszukiwania grafu w głąb po każdym jego kroku (tj. po nowym wywołaniu rekurencyjnym lub po zakończeniu aktualnego wywołania) odpowiada pewnej trójce zbiorów (S, A, B), zdefiniowanej wcześniej. Niech $(S_1, A_1, B_1), (S_2, A_2, B_2), \ldots, (S_p, A_p, B_p)$ będzie ciągiem kolejnych takich trójek, które wystąpiły w czasie przeszukiwania w głąb. Zauważmy, że $(S_1, A_1, B_1) = (\{r\}, \emptyset, V \setminus \{r\})$ to początkowa trójka, powstała po pierwszym wywołaniu procedury przeszukiwania w głąb dla wierzchołka r, zaś $(S_p, A_p, B_p) = (\emptyset, V, \emptyset)$ to ostatnia trójka, powstała po zakończeniu wywołania rekurencyjnego dla wierzchołka r. Co więcej, $p = 2n$, gdyż dla każdego wierzchołka $v \in V$, powstają dwie trójki (S_i, A_i, B_i) – pierwsza gdy przeszukiwanie w głąb wchodzi do tego wierzchołka, a druga jak z niego wychodzi. Do powyższych trójek dołóżmy jeszcze trójkę $(S_0, A_0, B_0) = (\emptyset, \emptyset, V)$, odpowiadającą stanowi sprzed pierwszego wywołania procedury przeszukiwania w głąb. Będziemy rozpatrywać każdą trójkę (S_i, A_i, B_i) po kolei i znajdywać pewną liczbę kandydatów na częściowe rozwiązania $Y \subseteq S_i \cup A_i$.

Przyjrzyjmy się jednej trójce (S_i, A_i, B_i). Z naszych dwóch obserwacji z początku rozdziału wynika, że mamy $|S_i| \leqslant t$ oraz nie ma żadnej krawędzi łączącej A_i i B_i. Weźmy jakiegoś „dobrego kandydata” na część rozwiązania $Y \subseteq S_i \cup A_i$, do którego będziemy później dobierać wierzchołki ze zbioru B_i. Jakie własności powinien mieć taki zbiór Y i co powinniśmy o nim pamiętać?

Po pierwsze, zauważmy, że żaden wierzchołek ze zbioru B_i nie dominuje żadnego wierzchołka ze zbioru A_i. Tak więc, nasi kandydaci Y na część zbioru dominującego muszą dominować cały zbiór A_i. Po drugie, ważne jest dla nas zachowanie zbioru Y na zbiorze S_i:

1. Pełen zbiór dominujący musi dominować zbiór B_i. Część elementów B_i może być dominowana przez wierzchołki z S_i (ale nie z A_i). Zatem istotne jest dla nas przecięcie $S_i \cap Y$, gdyż mówi nam ono, co w zbiorze B_i mamy już zdominowane.

2. Musimy też zdominować zbiór S_i. Część wierzchołków z S_i może być zdominowana przez wierzchołki z B_i, tak więc nie możemy wymagać, by Y dominował też cały zbiór S_i, ale musimy zapamiętać, które wierzchołki pozostały do zdominowania.

Powyższe rozważania prowadzą nas do następującej definicji. Wprowadźmy symbol $\mathtt{T}$, oznaczający należenie do zbioru Y, symbol $\mathtt{N}_1$, oznaczający nienależenie, ale bycie już zdominowanym, oraz symbol $\mathtt{N}_0$, oznaczający zarówno nienależenie do Y, jak i nie bycie zdominowanym przez Y.

Definicja 1. Dla ustalonej trójki (S_i, A_i, B_i) powiemy, że zbiór $Y \subseteq S_i \cup A_i$ jest *częściowym rozwiązaniem*, jeśli Y dominuje cały zbiór A_i. Co więcej, jeśli dodatkowo mamy daną funkcję $\sigma : S_i \to \{\mathtt{T}, \mathtt{N}_1, \mathtt{N}_0\}$, to powiemy, że częściowe rozwiązanie Y *ma interfejs* σ, jeśli zachodzą następujące warunki dla każdego $v \in S_i$:

- $\sigma(v) = \mathtt{T}$ wtedy i tylko wtedy gdy $v \in Y$,
- $\sigma(v) = \mathtt{N}_1$ wtedy i tylko wtedy gdy $v \notin Y$, ale v jest zdominowany przez Y, oraz
- $\sigma(v) = \mathtt{N}_0$ wtedy i tylko wtedy gdy $v \notin Y$ i v nie jest zdominowany przez Y.

Nasze dotychczasowe rozważania można podsumować następującym lematem:

Lemat 1. Ustalmy $0 \leqslant i \leqslant p$. Jeśli X jest zbiorem dominującym w grafie G, to zbiór $Y = X \cap (S_i \cup A_i)$ jest częściowym rozwiązaniem. Co więcej, jeśli X jest najtańszym zbiorem dominującym w G, a Y ma interfejs σ, to Y jest najtańszym częściowym rozwiązaniem o interfejsie σ.

Dowód: Pierwsza część lematu wynika wprost z obserwacji 2: elementy zbioru A_i mogą być zdominowane tylko przez elementy $S_i \cup A_i$. By udowodnić drugą część lematu, załóżmy nie wprost, że $Y' \subseteq S_i \cup A_i$ jest częściowym rozwiązaniem o interfejsie σ i o koszcie mniejszym niż koszt rozwiązania Y. Rozważmy zbiór $X' = (X \setminus Y) \cup Y' = (X \cap B_i) \cup Y'$. Skoro Y' ma mniejszy koszt niż Y, to X' jest tańszy od X. By otrzymać sprzeczność, pokażemy, że X' jest też zbiorem dominującym w grafie G. Rozważmy dowolny wierzchołek $v \in V$ i niech $w \in X$ będzie dowolnym wierzchołkiem dominującym wierzchołek v w rozwiązaniu X. Rozważmy następujące przypadki:

1. Jeśli $v \in A_i$, to v jest zdominowany przez Y', gdyż Y' jest częściowym rozwiązaniem.
2. Jeśli $w \in A_i$, lecz $v \notin A_i$, to mamy $v \in S_i$ oraz $\sigma(v) \neq \mathtt{N}_0$. Wówczas Y' dominuje v, gdyż ma również interfejs σ.
3. Jeśli $w \in S_i$, to $w \in S_i \cap X = S_i \cap Y = S_i \cap Y' \subseteq X'$, gdyż Y' też ma interfejs σ.
4. Jeśli $w \in B_i$, to $w \in B_i \cap X = B_i \cap X' \subseteq X'$.

W każdym przypadku otrzymujemy, że v jest zdominowany przez X'. Z dowolności wyboru v wynika, że X' jest zbiorem dominującym w G. ∎

Definicja tabeli i obliczenia

W naszym programowaniu dynamicznym dla każdej trójki (S_i, A_i, B_i) oraz interfejsu $\sigma : S_i \to \{\mathtt{T}, \mathtt{N}_1, \mathtt{N}_0\}$, obliczamy wartość $M[i, \sigma]$ będącą minimalnym możliwym kosztem częściowego rozwiązania $Y \subseteq S_i \cup A_i$ o interfejsie σ. Dla $i = 0$, gdy $(S_0, A_0, B_0) = (\emptyset, \emptyset, V)$, początkową wartością jest $M[0, \emptyset] = 0$: jedynym częściowym rozwiązaniem jest zbiór pusty[2]. Wartość najtańszego zbioru dominującego w grafie G odczytamy z pola $M[p, \emptyset]$: stan $(S_p, A_p, B_p) = (\emptyset, V, \emptyset)$ ma tylko jeden możliwy interfejs $\sigma = \emptyset$ i częściowe rozwiązania dla tej trójki to dokładnie zbiory dominujące w grafie G.

Przeanalizujmy teraz, jak obliczyć pojedynczą wartość $M[i + 1, \sigma]$, mając dane wszystkie wartości $M[j, \sigma']$ dla $0 \leqslant j \leqslant i$. Po pierwsze, zauważmy, że nie wszystkie interfejsy σ mają sens: jeśli dwa wierzchołki $u, v \in S_{i+1}$ są połączone krawędzią w grafie G, ale $\sigma(u) = \mathtt{T}$ oraz $\sigma(v) = \mathtt{N}_0$, to wówczas nie ma żadnego częściowego rozwiązania o tym interfejsie, i w tym przypadku $M[i + 1, \sigma] = +\infty$. W przeciwnym przypadku, mamy dwa podprzypadki: stan $(S_{i+1}, A_{i+1}, B_{i+1})$ powstał albo w wyniku nowego wywołania rekurencyjnego procedury przeszukiwania w głąb, albo w wyniku zakończenia aktualnego wywołania.

Nowe wywołanie. Załóżmy, że stan $(S_{i+1}, A_{i+1}, B_{i+1})$ powstał w wyniku wywołania procedury przeszukiwania rekurencyjnego na wierzchołku v. Zauważmy, że wówczas $S_{i+1} = S_i \cup \{v\}$, $A_{i+1} = A_i$ oraz $B_{i+1} = B_i \setminus \{v\}$. Niech Y będzie dowolnym częściowym rozwiązaniem o interfejsie σ. Rozważmy trzy przypadki, zależnie od wartości $\sigma(v)$.

Załóżmy wpierw, że $\sigma(v) = \mathtt{T}$, czyli $v \in Y$, i zastanówmy się, jaki interfejs σ' ma zbiór $Y' = Y \setminus \{v\}$ względem trójki (S_i, A_i, B_i). Oczywiście, dla każdego $w \in S_i$ mamy $\sigma'(w) = \mathtt{T}$ wtedy i tylko wtedy gdy $w \in Y$, co jest równoważne $\sigma(w) = \mathtt{T}$. Podobnie, jeśli $\sigma(w) = \mathtt{N}_0$, czyli Y nie dominuje w, to również Y' nie dominuje w i $\sigma'(w) = \mathtt{N}_0$. Jeśli zaś $\sigma(w) = \mathtt{N}_1$, to albo Y' dominuje w, albo Y' nie dominuje w, lecz v dominuje w ($vw \in E$). Tak więc, $\sigma'(w) = \mathtt{N}_1$ lub $\sigma'(w) = \mathtt{N}_0$, ale ten drugi przypadek może zajść tylko wtedy, gdy $vw \in E$. Oznaczmy więc $\sigma' \prec \sigma$, jeśli dla każdego $w \in S_i$ mamy:

1. jeśli $\sigma(w) = \mathtt{T}$, to $\sigma'(w) = \mathtt{T}$;
2. jeśli $\sigma(w) = \mathtt{N}_0$, to $\sigma'(w) = \mathtt{N}_0$;
3. jeśli $\sigma(w) = \mathtt{N}_1$, to $\sigma'(w) = \mathtt{N}_1$ lub $\sigma'(w) = \mathtt{N}_0$ oraz $vw \in E$.

Wówczas z powyższych rozważań wynika, że:

$$M[i + 1, \sigma] = c(v) + \min_{\sigma' \prec \sigma} M[i, \sigma'].$$

Rozważmy teraz przypadek $\sigma(v) = \mathtt{N}_1$. Przez $\sigma|_{S_i}$ oznaczmy funkcję σ ograniczoną do dziedziny S_i. Zauważmy, że v nie ma żadnych sąsiadów w zbiorze A_i, tak więc Y musi dominować v jakimś wierzchołkiem w $S_i = S_{i+1} \setminus \{v\}$. Jeśli istnieje $w \in S_i$ taki, że $vw \in E$ oraz $\sigma(w) = \mathtt{T}$, to każde częściowe rozwiązanie dla trójki (S_i, A_i, B_i) o interfejsie $\sigma|_{S_i}$ będzie dominowało v, gdyż będzie zawierało wierzchołek w. Z drugiej

[2]Zastosowaliśmy tutaj konwencję, że $\emptyset$ oznacza też pustą funkcję.

strony, jeśli taki wierzchołek w nie istnieje, to żadne częściowe rozwiązanie dla trójki (S_i, A_i, B_i) o interfejsie $\sigma|_{S_i}$ nie będzie dominowało v, co oznacza, że nie istnieje żadne częściowe rozwiązanie dla trójki $(S_{i+1}, A_{i+1}, B_{i+1})$ o interfejsie σ.

Mamy więc, że w pierwszym przypadku $M[i+1, \sigma] = M[i, \sigma|_{S_i}]$, zaś w drugim przypadku $M[i+1, \sigma] = +\infty$.

Pozostał nam najprostszy przypadek $\sigma(v) = \mathtt{N}_0$. Przypomnijmy, że rozpatrujemy tylko takie interfejsy σ, dla których żadne dwa sąsiadujące wierzchołki w S_{i+1} nie mogą mieć wartości $\mathtt{T}$ i $\mathtt{N}_0$. Razem z faktem, że v nie ma sąsiadów w A_i, oznacza to, że rodziny częściowych rozwiązań dla trójki (S_i, A_i, B_i) o interfejsie $\sigma|_{S_i}$ i dla trójki $(S_{i+1}, A_{i+1}, B_{i+1})$ o interfejsie σ są dokładnie takie same. Mamy więc $M[i+1, \sigma] = M[i, \sigma|_{S_i}]$.

Zakończenie wywołania. Załóżmy, że stan $(S_{i+1}, A_{i+1}, B_{i+1})$ powstał w wyniku zakończenia procedury przeszukiwania rekurencyjnego na wierzchołku v. Zauważmy, że wówczas $S_{i+1} = S_i \setminus \{v\}$, $A_{i+1} = A_i \cup \{v\}$ oraz $B_{i+1} = B_i$. Niech Y będzie dowolnym częściowym rozwiązaniem o interfejsie σ.

Jeśli $v \in Y$, to Y jest częściowym rozwiązaniem zgodnym z interfejsem $\sigma_{v \mapsto \mathtt{T}}$ względem trójki (S_i, A_i, B_i), gdzie $\sigma_{v \mapsto \mathtt{T}}$ jest interfejsem σ rozszerzonym o przyporządkowanie $v \mapsto \mathtt{T}$. Koszt najtańszego takiego częściowego rozwiązania przechowywany jest w komórce $M[i, \sigma_{v \mapsto \mathtt{T}}]$.

Jeśli $v \notin Y$, to z faktu, że Y jest częściowym rozwiązaniem, wynika, że Y dominuje v i Y jest częściowym rozwiązaniem zgodnym z interfejsem $\sigma_{v \mapsto \mathtt{N}_1}$ względem (S_i, A_i, B_i). Koszt najtańszego takiego częściowego rozwiązania przechowywany jest w komórce $M[i, \sigma_{v \mapsto \mathtt{N}_1}]$. Zatem otrzymujemy następującą zależność:

$$M[i+1, \sigma] = \min\left(M[i, \sigma_{v \mapsto \mathtt{T}}], M[i, \sigma_{v \mapsto \mathtt{N}_1}]\right).$$

Analiza złożoności

Tabela M ma $O(3^t n)$ komórek, gdyż indeks i przebiega zakres $0 \leqslant i \leqslant p = 2n$, a $|S_i| \leqslant t$ dla każdego i. Taka też jest złożoność pamięciowa powyższego algorytmu. Zastanówmy się teraz nad złożonością czasową.

Obliczenie nowej wartości $M[i+1, \sigma]$ prawie zawsze wymaga sięgnięcia tylko do kilku poprzednich wartości oraz obejrzenia krawędzi incydentnych z „kluczowym” wierzchołkiem v. Wydawać się może, że więcej czasu wymaga obliczenie $M[i+1, \sigma]$ w przypadku, gdy v jest nowym wierzchołkiem ($S_{i+1} = S_i \cup \{v\}$) i $\sigma(v) = \mathtt{T}$, gdyż wtedy musimy przejrzeć wszystkie interfejsy $\sigma' \prec \sigma$. Zauważmy jednak, że dla każdego interfejsu σ' względem (S_i, A_i, B_i), istnieje co najwyżej jeden interfejs σ, dla którego zachodzi $\sigma' \prec \sigma$: by otrzymać σ z σ', należy położyć $\sigma(v) = \mathtt{T}$ oraz zamienić wszystkie wartości $\mathtt{N}_0$ na $\mathtt{N}_1$ wśród sąsiadów wierzchołka v w zbiorze S_i. Tak więc, przy uważnej implementacji, powyższy algorytm programowania dynamicznego można wykonać w czasie $O(3^t(n+m))$; to rozwiązanie zostało zaimplementowane w plikach `turs1.cpp` i `turs4.pas`.

Możliwe jest jednak pewne dodatkowe usprawnienie. Przypomnijmy, że możemy pomijać obliczenia dla interfejsów σ, w których istnieją dwa wierzchołki u, v połączone krawędzią, dla których $\sigma(u) = \mathtt{T}$ i $\sigma(v) = \mathtt{N}_0$. Co więcej, przypomnijmy, że każdy

zbiór S_i odpowiada ścieżce w drzewie T, a więc i w grafie G. Okazuje się, że te dwie obserwacje istotnie zmniejszają liczbę interfejsów σ, które musimy rozważać.

Dla ustalonej trójki (S_i, A_i, B_i), zakodujmy interfejs σ jako ciąg $|S_i|$ symboli nad alfabetem $\{\mathtt{T}, \mathtt{N}_1, \mathtt{N}_0\}$, oznaczających wartości $\sigma(v)$ dla kolejnych wierzchołków na ścieżce S_i. W tym ciągu, symbole $\mathtt{T}$ i $\mathtt{N}_0$ nie mogą wystąpić obok siebie. To sprowadza nas do następującego pytania dotyczącego kombinatoryki na słowach:

> Ile jest t-literowych słów nad alfabetem $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ takich, że litery $\mathtt{b}$ i $\mathtt{c}$ nie występują obok siebie?

Nazwijmy powyższe słowa *dobrymi* i niech $F(t)$ oznacza liczbę dobrych słów o t literach. W dobrym słowie, dla każdych dwóch kolejnych pozycji, mamy tylko 7 możliwości wyboru liter na tych pozycjach: kombinacje `bc` i `cb` są zabronione. Łącząc litery kolejno w pary, otrzymujemy następującą obserwację:

Obserwacja 3.

$$F(2t) \leqslant 7^t < 2{,}65^{2t}.$$

Oznacza to, że dobrych słów jest istotnie mniej niż wszystkich słów o określonej długości: w powyższej obserwacji mamy ograniczenie $2{,}65^{2t}$ zamiast naiwnego 3^{2t}.

W dowodzie obserwacji 3 jest jednak dużo luzu. Niestety do precyzyjnego określenia wartości $F(2t)$ potrzebne są trochę bardziej zaawansowane narzędzia. Zamiast nich zadowolimy się tutaj następującym, dość dokładnym, oszacowaniem.

Lemat 2.

$$F(t) \leqslant \sqrt{2} \cdot (1 + \sqrt{2})^t.$$

Dowód: Dla $\alpha \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, przez $F_\alpha(t)$ oznaczmy liczbę dobrych t-literowych słów, kończących się na literę α. Oczywiście, $F(t) = F_\mathtt{a}(t) + F_\mathtt{b}(t) + F_\mathtt{c}(t)$. Zdefiniujmy $G(t) = \sqrt{2} \cdot F_\mathtt{a}(t) + F_\mathtt{b}(t) + F_\mathtt{c}(t)$. Będziemy, przez indukcję po wartości t, dowodzić następującej równości, która natychmiast daje tezę lematu (gdyż $F(t) \leqslant G(t)$):

$$G(t) = \sqrt{2} \cdot (1 + \sqrt{2})^t.$$

Dla $t = 1$ mamy $F_\mathtt{a}(1) = F_\mathtt{b}(1) = F_\mathtt{c}(1) = 1$ i powyższa równość zachodzi.

Załóżmy, że równość ta jest spełniona dla pewnego $t \geqslant 1$. Niech w będzie $(t+1)$-literowym dobrym słowem, niech α będzie ostatnią literą w, i niech w' będzie słowem w z obciętą ostatnią literą. Oczywiście, w' jest t-literowym dobrym słowem. Jeśli $\alpha = \mathtt{a}$, to w' może kończyć się na dowolną literę. Jeśli $\alpha = \mathtt{b}$, to w' nie może kończyć się na $\mathtt{c}$. Podobnie, jeśli $\alpha = \mathtt{c}$, to w' nie może kończyć się na $\mathtt{b}$. Uzyskujemy więc następujące zależności:

$$\begin{aligned}
F_\mathtt{a}(t+1) &= F_\mathtt{a}(t) + F_\mathtt{b}(t) + F_\mathtt{c}(t),\\
F_\mathtt{b}(t+1) &= F_\mathtt{a}(t) + F_\mathtt{b}(t),\\
F_\mathtt{c}(t+1) &= F_\mathtt{a}(t) + F_\mathtt{c}(t).
\end{aligned}$$

Podsumowując:

$$\begin{aligned} G(t+1) &= \sqrt{2}\cdot F_{\mathtt{a}}(t+1) + F_{\mathtt{b}}(t+1) + F_{\mathtt{c}}(t+1) \\ &= \sqrt{2}\big(F_{\mathtt{a}}(t) + F_{\mathtt{b}}(t) + F_{\mathtt{c}}(t)\big) + \big(F_{\mathtt{a}}(t) + F_{\mathtt{b}}(t)\big) + \big(F_{\mathtt{a}}(t) + F_{\mathtt{c}}(t)\big) \\ &= (2+\sqrt{2})F_{\mathtt{a}}(t) + (1+\sqrt{2})F_{\mathtt{b}}(t) + (1+\sqrt{2})F_{\mathtt{c}}(t) \\ &= (1+\sqrt{2})(\sqrt{2}\cdot F_{\mathtt{a}}(t) + F_{\mathtt{b}}(t) + F_{\mathtt{c}}(t)) \\ &= (1+\sqrt{2})G(t) \\ &= \sqrt{2}\cdot(1+\sqrt{2})^{t+1}. \end{aligned}$$ ■

Tak więc przy uważnej implementacji omówionego algorytmu programowania dynamicznego, rozważającego tylko „istotne” interfejsy σ, otrzymujemy rozwiązanie o złożoności czasowej $O((1+\sqrt{2})^t(n+m))$. To rozwiązanie zostało zaimplementowane w plikach `tur.cpp` i `tur0.pas`. Zauważmy, że dla ograniczenia $t = 10$ mamy $(1+\sqrt{2})^t < 7000$ i rozwiązanie powinno spokojnie mieścić się w limitach czasowych.

Wolniejsze implementacje

Rozwiązania zaimplementowane w plikach `turs2.cpp` i `turs5.pas` mają nieoptymalnie zaimplementowane interfejsy – niezależnie dla każdego wierzchołka S_i wybieramy, czy jest on wzięty do częściowego rozwiązania i czy jest on już zdominowany. Prowadzi to do złożoności czasowej $O(4^t(n+m))$.

Rozwiązania wolne w pozostałych plikach stosują trochę inną strukturę zamiatania drzewa T. Oparte one są o następujące spostrzeżenie: dla każdego wierzchołka v, jeśli oznaczymy przez C_v zbiór wszystkich potomków wierzchołka v w drzewie T, a przez S_v zbiór jego przodków (wliczając w to v), to S_v separuje C_v od $V \setminus (C_v \cup S_v)$ oraz oczywiście $|S_v| \leqslant t$. Niestety, w tym przypadku nie mamy tak ładnych przejść między kolejnymi stanami, jak w rozwiązaniu wzorcowym, i te algorytmy działają istotnie wolniej od rozwiązania wzorcowego.

Testy

Wygenerowanie ciekawych testów dla tego zadania okazało się pewnym wyzwaniem. Testy były podzielone na następujące kategorie:

1. drzewo o wysokości 3 i średnicy 5; do niektórych wierzchołków dodano krawędzie do niektórych przodków;
2. pełen graf dwudzielny, posiadający po jednej stronie 4 wierzchołki, a po drugiej stronie dużo wierzchołków; dodatkowo jedna z krawędzi została zastąpiona ścieżką długości 2;
3. losowy graf o 10 wierzchołkach;
4. losowy graf o 8 wierzchołkach, do którego podoczepiano liście;
5. losowy graf o 6 wierzchołkach, do którego podoczepiano poddrzewa o wysokości 2 lub trójkąty;

6. drzewo o średnicy 10.

Warto zwrócić uwagę, że weryfikacja, czy graf wejściowy spełnia warunki zadania (czy nie posiada ścieżki długości dłuższej niż $t = 10$), jest również bardzo trudnym problemem. Można go rozwiązać przez podobny algorytm programowania dynamicznego jak rozwiązanie wzorcowe, lecz jest to bardziej techniczne i żmudne. Zainteresowanych czytelników zachęcamy do zastanowienia się, jak taki algorytm mógłby wyglądać – w szczególności, jaka byłaby definicja interfejsu i jak by wyglądała tablica programowania dynamicznego.

Szersze spojrzenie: strukturalne miary grafu

W szerszym ujęciu opisane rozwiązanie wzorcowe jest tak naprawdę algorytmem programowania dynamicznego na dość specyficznej *dekompozycji ścieżkowej* (ang. *path decomposition*) wejściowego grafu. Bez podawania formalnej definicji, dekompozycja ścieżkowa grafu G opisuje, jak „zamieść" graf G, używając małej miotły. Obserwację 1 można przeformułować następująco: jeśli graf nie posiada ścieżki dłuższej niż t, to *głębokość drzewiasta* (ang. *treedepth*) grafu G wynosi co najwyżej t; głębokość drzewiasta spójnego grafu G jest najmniejszą możliwą wysokością drzewa T o tych samych wierzchołkach co G, takiego, że każda krawędź G łączy przodka i potomka w T (nie wymagamy tu, by T był podgrafem G).

Dużo bardziej znaną miarą strukturalną grafów jest *szerokość drzewiasta* (ang. *treewidth*), która odpowiada zamiataniu grafu G z użyciem małej miotły, ale tak, że w poszczególnych krokach zamiatanie może „rozgałęzić się" na różne części grafu G (czyli struktura zamiatania przypomina drzewo, a nie ścieżkę). Mając daną dekompozycję drzewiastą (czyli taką receptę, jak zamiatać graf) o małej szerokości (czyli małej miotle), można rozwiązać efektywnie wiele problemów, używając programowania dynamicznego w podobny sposób, jak to zrobiliśmy w rozwiązaniu wzorcowym. Umiejętność rozwiązywania trudnych problemów na grafach o ograniczonej szerokości drzewiastej okazuje się istotnym elementem w wielu zastosowaniach, zarówno w teoretycznej jak i praktycznej informatyce.

Zainteresowanych czytelników odsyłamy do rozdziału 12 w książce R. Diestla [41], gdzie znajduje się dokładny opis dekompozycji drzewiastych i szerokości drzewiastej, oraz do rozdziału 7 w nowo powstałej książce o algorytmach parametryzowanych [42], który opisuje różne algorytmy programowania dynamicznego na dekompozycjach[3].

[3]Książka powinna się ukazać na początku 2015 roku.

Wojciech Nadara — Treść zadania, Opracowanie
Adam Polak — Program

Dostępna pamięć: 128 MB. *OI, etap III, dzień drugi, 3.04.2014*

Lampy słoneczne

Bajtazar ma bardzo duży i piękny ogród, lecz gdy zapadnie zmrok, nie może podziwiać jego uroków. W związku z tym postanowił zaopatrzyć się w lampy, których światło pozwoli na rozkoszowanie się widokiem ogrodu również w nocy.

Zakupione przez Bajtazara lampy nie oświetlają wszystkiego wokół, a jedynie obszar zawarty w pewnym kącie. Dla wszystkich lamp kąt ten jest taki sam, co więcej, lampy muszą być zamontowane tak, aby wszystkie świeciły w tym samym kierunku. Ponadto są to lampy słoneczne. Co prawda w nocy słońca nie ma, ale wystarczy, że na lampę będzie świecić odpowiednia liczba innych lamp, a ona także zacznie świecić. Oczywiście, lampy zapalają się również po podłączeniu ich do prądu.

Bajtazar zamontował lampy w swoim ogrodzie i ustalił, w jakiej kolejności będzie podłączał do nich prąd. Dla uproszczenia ponumerujmy lampy liczbami od 1 do n w tej właśnie kolejności, tzn. Bajtazar w momencie i podłączy prąd do lampy o numerze i. Bajtazar zastanawia się, kiedy zaczną świecić poszczególne lampy. Pomóż mu i napisz program, który odpowie na to pytanie.

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą n ($1 \leqslant n \leqslant 200\,000$) oznaczającą liczbę lamp, które zakupił Bajtazar. W drugim wierszu wejścia znajdują się cztery liczby całkowite X_1, Y_1, X_2, Y_2 ($-10^9 \leqslant X_i, Y_i \leqslant 10^9$, $(X_i, Y_i) \neq (0, 0)$) pooddzielane pojedynczymi odstępami, wyznaczające obszar oświetlany przez każdą z lamp. Jeśli pewna lampa stoi w punkcie (x, y), to oświetla obszar (wraz z brzegiem), który znajduje się w mniejszym z kątów wyznaczonych przez dwie półproste o początkach w (x, y), z których i-ta (dla $i = 1, 2$) przechodzi także przez punkt $(x + X_i, y + Y_i)$. Kąt ten jest zawsze różny od 180 stopni.

Kolejne n wierszy wejścia opisuje rozstawienie lamp: i-ty z tych wierszy zawiera dwie liczby całkowite x_i, y_i ($-10^9 \leqslant x_i, y_i \leqslant 10^9$) oddzielone pojedynczym odstępem, oznaczające, że lampa o numerze i jest umieszczona w punkcie (x_i, y_i). Możesz założyć, że żadne dwie lampy nie stoją w tym samym punkcie.

Ostatni wiersz wejścia zawiera n liczb całkowitych $k_1, k_2, \ldots, k_n$ ($1 \leqslant k_i \leqslant n$) pooddzielanych pojedynczymi odstępami, oznaczających, że jeżeli lampa o numerze i znajdzie się w obszarze oświetlanym przez co najmniej k_i innych lamp, to ona także zacznie świecić.

W testach wartych łącznie 30% punktów zachodzi dodatkowy warunek $n \leqslant 1000$.

Wyjście

Twój program powinien wypisać na standardowe wyjście jeden wiersz zawierający n liczb całkowitych $t_1, \ldots, t_n$ pooddzielanych pojedynczymi odstępami. Liczba t_i ma oznaczać moment zaświecenia się lampy o numerze i.

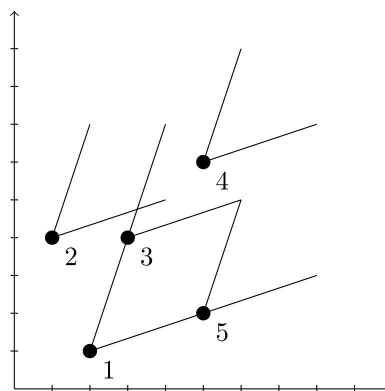
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
1 2 1 2 5
```



Wyjaśnienie do przykładu: *W chwili 1 Bajtazar podłącza prąd do lampy 1, co powoduje zaświecenie się również lampy 3. Po podłączeniu prądu do lampy 2, zaczyna świecić również lampa 4 (oświetlona przez lampy 1, 2 i 3).*

Testy „ocen”:

`1ocen`: $n = 7$, *mały test losowy;*

`2ocen`: $n = 6$, *lampy oświetlają kąt zero stopni (Bajtazar zainwestował w lasery);*

`3ocen`: $n = 160\,000$, *lampy ustawione w kratę o wymiarach* 400×400, *oświetlają kąt 90 stopni o ramionach równoległych do osi układu współrzędnych.*

Rozwiązanie

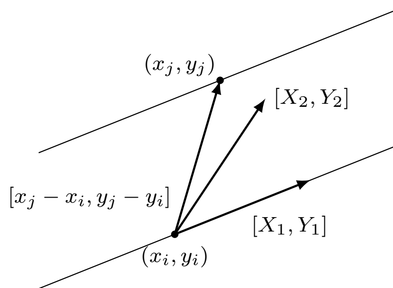
Na początek zauważmy, że przedstawione zadanie pod względem ideologicznym nie różni się wiele od zadania, w którym wszystkie lampy oświetlałyby obszar zadany przez wektory $[1, 0]$ oraz $[0, 1]$ (na razie w naszych rozważaniach załóżmy, że kąt, który oświetlają lampy, jest niezerowy). I rzeczywiście okazuje się, że dla każdego zestawu danych wejściowych można skonstruować zestaw n lamp, z których każda będzie odpowiadała pewnej lampie z oryginalnego zestawu i oświetlać będzie obszar zadany przez wektory $[1, 0]$ oraz $[0, 1]$, a ponadto i-ta lampa z nowego zestawu będzie oświetlać j-tą lampę z nowego zestawu wtedy i tylko wtedy, gdy i-ta lampa z oryginalnego zestawu oświetlała j-tą lampę z oryginalnego zestawu. Skupimy się teraz na wykonaniu tej konstrukcji.

Niektórzy mogą zauważyć, że można zastosować tutaj tzw. przekształcenie afiniczne płaszczyzny, w którym nowe pozycje lamp wyrażają się pewnymi funkcjami liniowymi od ich oryginalnych pozycji. My jednak zastosujemy tutaj inne podejście; będzie ono miało różne zalety, o których przekonamy się później.

Rozważmy rodzinę prostych równoległych do wektora $[X_1, Y_1]$ z zadania. Zauważmy, że każda taka prosta ma dokładnie jeden punkt wspólny z prostą przechodzącą przez punkty $(0, 0)$ oraz (X_2, Y_2) (założyliśmy na razie, że każda lampa

oświetla niezerowy kąt), zatem przecina tę prostą w punkcie (tX_2, tY_2) dla pewnej liczby rzeczywistej t. Każdej takiej prostej przypiszmy odpowiadającą jej liczbę t. Intuicyjnie rzecz ujmując, możemy sobie wyobrazić prostą równoległą do wektora $[X_1, Y_1]$, przemieszczającą się ze stałą prędkością w kierunku zgodnym ze zwrotem wektora $[X_2, Y_2]$, tak że w chwili 0 przechodzi przez punkt $(0, 0)$, a w chwili 1 przez punkt (X_2, Y_2). Wówczas w chwili t (t może być ujemne) będzie się ona pokrywać z tą prostą, której przyporządkowaliśmy liczbę t. Możemy też rozważyć analogiczną rodzinę prostych równoległych do $[X_2, Y_2]$ i przyporządkować im liczby u, względem prostej przechodzącej przez punkty $(0, 0)$ i (X_1, Y_1). Teraz, każda lampa leży na dokładnie jednej prostej należącej do pierwszej rodziny i na dokładnie jednej prostej należącej do drugiej rodziny. Lampie przyporządkujemy parę liczb (t_i, u_i), gdzie t_i to liczba t przyporządkowana prostej należącej do pierwszej rodziny, analogicznie u_i. Łatwo teraz zauważyć, że lampa o numerze i oświetla lampę o numerze j wtedy i tylko wtedy, gdy $t_i \leqslant t_j$ oraz $u_i \leqslant u_j$.

Przekształcenie $(x_i, y_i) \to (t_i, u_i)$ jest zatem przekształceniem, które wspomnieliśmy wcześniej. W szczególności, można je zapisać konkretnymi wzorami. Zauważmy jednak, że nie interesują nas wcale dokładne wartości t_i dla lamp, a jedynie ich porządek. Możemy zatem posortować lampy według tych wartości i każdej lampie przydzielić jako t_i pozycję, na której stoi po takim posortowaniu. Podobnie możemy zrobić z wartościami u_i. Aby posortować lampy po liczbach t_i, wcale nie trzeba tych liczb *explicite* wyznaczać. Wystarczy umieć stwierdzać, dla danych indeksów i, j, która z liczb t_i oraz t_j jest większa. W tym celu sprawdzamy, czy wektor $[x_j - x_i, y_j - y_i]$ leży po tej samej stronie wektora $[X_1, Y_1]$, co wektor $[X_2, Y_2]$ – jeśli po tej samej stronie, to $t_i < t_j$, a w przeciwnym razie $t_i > t_j$ (rys. 1). Do tego celu służy nam iloczyn wektorowy: znak iloczynu wektorowego $[a, b] \times [c, d] = ad - bc$ jest dodatni, jeśli wektor $[c, d]$ leży na lewo od wektora $[a, b]$, a ujemny, jeśli leży on na prawo[1].



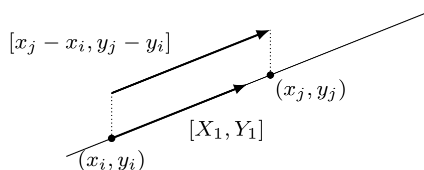
Rys. 1: Mamy $t_i < t_j$, gdyż oba wektory $[x_j - x_i, y_j - y_i]$, $[X_2, Y_2]$ leżą po tej samej stronie wektora $[X_1, Y_1]$ (w tym przypadku oba na lewo). Innymi słowy, prosta równoległa do wektora $[X_1, Y_1]$, przemieszczająca się zgodnie ze zwrotem wektora $[X_2, Y_2]$, odwiedzi najpierw lampę (x_i, y_i), a potem lampę (x_j, y_j).

W ten sposób każdej lampie możemy przyporządkować parę (a_i, b_i) liczb całkowitych z przedziału $[1, n]$, taką że a_i jest pozycją liczby t_i w posortowanym ciągu t, a b_i jest pozycją u_i w posortowanym ciągu u. Każda z liczb z przedziału $[1, n]$ zostanie

[1] Więcej o iloczynie wektorowym można przeczytać np. w książce [25].

na każdej współrzędnej wykorzystana dokładnie raz oraz to przyporządkowanie ma tę własność, że lampa o numerze i oświetla lampę o numerze j wtedy i tylko wtedy, gdy $a_i \leqslant a_j$ oraz $b_i \leqslant b_j$.

Do tej pory jednak przemilczeliśmy pewną kwestię: mianowicie co powinniśmy zrobić, jeżeli dwóm lampom odpowiadałaby ta sama liczba t? Jeżeli chcemy otrzymać przyporządkowanie liczb takie, jak przed chwilą, to nie możemy ustalić między nimi dowolnego porządku. Dla przykładu, gdybyśmy mieli do czynienia z sytuacją, w której $[X_1, Y_1] = [1, 0]$ i $[X_2, Y_2] = [0, 1]$, oraz dwiema lampami o współrzędnych odpowiednio $(0, 0)$ i $(1, 0)$, to ich liczby t wynosiłyby 0, jednak pierwsza lampa oświetla drugą, zatem musielibyśmy mieć $a_1 = 1$ oraz $a_2 = 2$, a nie na odwrót. W przypadku równości liczb t (które wykrywamy na podstawie tego, że opisany wcześniej iloczyn wektorowy się zeruje) musimy ten porządek ustalić zgodnie ze zwrotem wektora $[X_1, Y_1]$. Do tego celu idealnie nadaje się iloczyn skalarny (rys. 2).



Rys. 2: Mamy $t_i = t_j$. Iloczyn skalarny $[X_1, Y_1] \cdot [x_j - x_i, y_j - y_i]$ jest dodatni, więc i-ta lampa otrzyma mniejszy numer niż j-ta.

Podsumujmy: porównując lampy i-tą oraz j-tą, należy sprawdzić, czy iloczyn wektorowy $[X_1, Y_1] \times [x_i - x_j, y_i - y_j]$ jest dodatni, czy ujemny, i na tej podstawie (oraz na podstawie znaku iloczynu $[X_1, Y_1] \times [X_2, Y_2]$) ustalić porządek między liczbami t_i oraz t_j. W przypadku, gdy $t_i = t_j$, czyli gdy dany iloczyn wektorowy wynosi 0, należy porządek między tymi lampami ustalić zgodnie ze znakiem iloczynu skalarnego tych dwóch wektorów (jeżeli iloczyn wektorowy dwóch niezerowych wektorów jest równy 0, to ich iloczyn skalarny nie może wynosić 0).

Otrzymaliśmy zatem ostateczną wersję wstępnego przetwarzania współrzędnych. Lampy należy posortować za pomocą opisanej funkcji porównującej, w której używamy wektora $[X_1, Y_1]$, przypisać im ich pozycje w tym porządku, a potem zrobić to samo dla analogicznej funkcji porównującej, w której zamiast wektora $[X_1, Y_1]$ będziemy używać wektora $[X_2, Y_2]$. Cała konstrukcja działa w czasie $O(n \log n)$.

Jakie są zalety tego rozwiązania? Po pierwsze, operuje ono jedynie na liczbach całkowitych. Po drugie, wynikowe nowe „współrzędne" są od razu przenumerowane, tzn. są to liczby całkowite z przedziału $[1, n]$, co będzie pomocne, gdy będziemy nimi potem indeksować pewne struktury danych. Po trzecie, jest ono dość łatwe w implementacji. Poza już wymienionymi ma jeszcze jedną najistotniejszą zaletę – zawiera ono w sobie także poprawną analizę przypadku kątów zerowych! Zdejmuje to z nas obowiązek specjalnego traktowania tego, mogłoby się wydawać, bardzo niewygodnego przypadku. Sprawdzenie, że także i w takich przypadkach zaproponowany algorytm ma żądane własności, jest łatwym ćwiczeniem.

Rozwiązanie wzorcowe

Odtąd zakładamy, że wszystkie pierwsze oraz wszystkie drugie współrzędne lamp są różne i należą do przedziału $[1, n]$.

Pokażemy najpierw, że jeżeli ustalimy konkretną chwilę s, to jesteśmy w stanie w rozsądnej złożoności czasowej stwierdzić, które z lamp będą w tym momencie zapalone. Będziemy przetwarzać lampy w kolejności rosnących pierwszych współrzędnych. Dzięki temu, jeżeli ustalimy pewną lampę, to wszystkie lampy, które mogą potencjalnie na nią świecić, będą przetworzone przed przetworzeniem jej samej. Lampa będzie świecić w momencie s, jeżeli jej numer i jest nie większy niż s lub jeśli znajduje się ona w obszarze oświetlanym przez co najmniej k_i innych lamp (równoważnie: znajduje się w obszarze oświetlanym przez co najmniej k_i z lamp przetworzonych przed nią). Aby efektywnie sprawdzać drugi z tych warunków, będziemy utrzymywali drzewo przedziałowe. W istocie, ze wszystkich przetworzonych do tej pory lamp interesuje nas wyłącznie liczba zapalonych lamp spośród tych, które mają drugą współrzędną w przedziale $[1, u_i - 1]$, gdzie i jest numerem aktualnie rozpatrywanej lampy. Wystarczy więc drzewo przedziałowe indeksować drugimi współrzędnymi lamp i cały algorytm działa w czasie $O(n \log n)$.

Jeżeli rozpatrywalibyśmy łatwiejszą wersję naszego zadania, mianowicie, wyznaczenie momentu zapalenia konkretnej lampy, a nie wyznaczenie tego dla wszystkich lamp, to używając opisanego przed chwilą algorytmu oraz wyszukiwania binarnego, rozwiązalibyśmy ten problem efektywnie. Jednak nasz problem w istocie jest trudniejszy. Aby go rozwiązać, sprawdźmy najpierw, które lampy będą zapalone w momencie $\lfloor \frac{n}{2} \rfloor$. W ten sposób lampy podzielą nam się na dwa zbiory. Nazwijmy je: A – lampy, które świeciły w momencie $\lfloor \frac{n}{2} \rfloor$, oraz B – wszystkie pozostałe. Zauważmy, że dla lamp ze zbioru A możemy uzyskać odpowiedzi za pomocą wywołania rekurencyjnego – lampy ze zbioru B nie wpływają na momenty zapalenia się lamp ze zbioru A, zatem rzeczywiście możemy tu po prostu ograniczyć przedział poszukiwania z $[1, n]$ do $[1, \lfloor \frac{n}{2} \rfloor]$ i zapomnieć o lampach ze zbioru B. Ze zbiorem B jest podobnie. Tutaj przedział poszukiwania zawęża się do $[\lfloor \frac{n}{2} \rfloor + 1, n]$, ale nie możemy tak po prostu zapomnieć o lampach ze zbioru A, ponieważ wpływają one na momenty zapalenia się lamp z B. Przed wywołaniem rekurencyjnym dla lamp ze zbioru B i zapomnieniem w tym wywołaniu o istnieniu lamp ze zbioru A musimy dla każdej lampy ze zbioru B stwierdzić, ile lamp ze zbioru A na nią świeci, i odjąć tę liczbę od zapotrzebowania tej lampy na liczbę lamp, które muszą na nią świecić, aby ona również się zaświeciła. Jednak tę informację już mamy – wyznaczaliśmy ją przecież w sposób jawny w trakcie stwierdzania, które lampy są zapalone w momencie $\lfloor \frac{n}{2} \rfloor$! Ten pomysł wystarcza, aby rozwiązać nasze zadanie. Kolejne wywołania rekurencyjne będą przebiegały na tej samej zasadzie – dla ustalonego przedziału odpowiedzi oraz zbioru lamp, które w jego trakcie zapalą się, po prostu sprawdzamy, które lampy zapalą się do połowy tego przedziału, a które zapalą się później, aktualizujemy zapotrzebowanie lamp, które zapalą się później, i wywołujemy się rekurencyjnie dla obu części. Rekurencję przerywamy oczywiście wtedy, kiedy przedział odpowiedzi będzie jednopunktowy. Opisane rozwiązanie jest rozwiązaniem wzorcowym.

Zastanówmy się teraz nad złożonością takiego rozwiązania. Na pierwszy rzut oka nie wygląda to jak klasyczny algorytm „dziel i zwyciężaj”, ponieważ lampy mogą się

w każdym momencie wywołania rekurencyjnego podzielić bardzo nierówno, a zazwyczaj wymagane jest, aby podzieliły się one po połowie! Zauważmy jednak, że to, co dzieli się na pół, to przedział czasu – dzięki temu nasze drzewo rekurencji będzie miało wysokość co najwyżej $O(\log n)$. Ponadto zauważmy, że na każdym poziomie rekurencji każda lampa pojawia się w co najwyżej jednym wywołaniu, a czas, którego potrzebujemy na stwierdzenie dla danego zbioru lamp o rozmiarze r, które z nich będą świeciły w ustalonym momencie czasu, to $O(r \log n)$ (musimy tylko pamiętać inteligentnie zerować drzewo przedziałowe wykorzystywane w każdym wywołaniu). Tak więc skoro na danym poziomie rekurencji sumaryczna liczność zbiorów lamp nie przekracza n, to na każdym poziomie rekurencji wykonamy pracę, która zajmie nam co najwyżej $O(n \log n)$ czasu. Jako że poziomów rekurencji jest co najwyżej $O(\log n)$, to oznacza to, że przedstawione rozwiązanie działa w złożoności czasowej $O(n \log^2 n)$ oraz pamięciowej $O(n)$. Podobny algorytm (przypominający równoczesne wyszukiwanie binarne dla wielu elementów naraz) wystąpił w rozwiązaniu zadania *Meteory* z XVIII Olimpiady Informatycznej [18].

Rozwiązanie wzorcowe jest zaimplementowane w plikach `lam.cpp`, `lam2.pas` i `lam3.cpp`.

Rozwiązanie alternatywne

Rozwiązanie alternatywne opiera się na dwuwymiarowym drzewie przedziałowym. Jest trudniejsze implementacyjnie od rozwiązania wzorcowego, ale dla zawodnika posiadającego odpowiedni warsztat może być znacząco prostsze koncepcyjnie.

Struktura danych, której będziemy używać, to statyczne drzewo przedziałowe, którego liście odpowiadają kolejnym jednostkom czasu. Węzeł reprezentujący przedział $[l, r]$ zawiera zrównoważone drzewo BST (w dostarczonej implementacji jest to tzw. drzepiec, z ang. *treap*), w którym będziemy przechowywać drugie współrzędne lamp, które zaświecą się w chwili $s \in [l, r]$.

Na początku wykonajmy przekształcenie współrzędnych do postaci (t_i, u_i) oraz posortujmy lampy według pierwszej współrzędnej. Będziemy obliczać wyniki dla lamp w tej właśnie kolejności. Dzięki temu, gdy przetwarzamy daną lampę, wszystkie lampy, które ją oświetlają, mają już obliczone wyniki i znajdują się w naszej strukturze danych. Skoro w strukturze danych znajdują się tylko lampy, które mają mniejszą pierwszą współrzędną niż aktualna lampa, to jest ona oświetlana przez dokładnie te lampy ze struktury, które mają mniejszą drugą współrzędną. Zatem obliczenie wyniku dla lampy i sprowadza się do zejścia w dół drzewa przedziałowego w poszukiwaniu najwcześniejszej chwili s takiej, że do niej włącznie zaświeciło się przynajmniej k_i spośród już przetworzonych lamp, które mają drugą współrzędną mniejszą niż u_i. Minimum z tej liczby oraz i to wynik dla i-tej lampy. Kiedy już go obliczymy, możemy dodać tę lampę do drzewa i przejść do kolejnej.

Złożoność czasowa tego rozwiązania to $O(n \log^2 n)$, a pamięciowa $O(n \log n)$. Takie rozwiązanie dostawało około 90 punktów. Implementacja znajduje się w pliku `lams3.cpp`.

Rozwiązanie wolne

Rozwiązanie wolne to prosta symulacja działająca w czasie $O(n^2)$, która dla każdej kolejno zapalanej lampy przegląda wszystkie pozostałe, żeby stwierdzić, które z nich są przez nią oświetlane. Implementacja tego rozwiązania znajduje się w plikach `lams1.cpp` i `lams2.pas`. Na zawodach otrzymywała ok. 30 punktów.

Rozwiązania błędne

Zostały przygotowane cztery błędne rozwiązania będące niepoprawnymi wersjami rozwiązania wzorcowego:

`lamb1.cpp` – przy obliczaniu iloczynów wektorów nie używa typów całkowitych 64-bitowych – dostaje 60 punktów;

`lamb2.cpp` – nie obsługuje przypadku kąta równego zero stopni – dostaje 80 punktów;

`lamb3.cpp` – ignoruje lampy leżące na brzegu oświetlanego obszaru – dostaje 30 punktów;

`lamb4.cpp` – działa tak, jakby lampa zaczynała oświetlać inne lampy dopiero po podłączeniu jej do prądu, nawet jeśli zaświeciła się wcześniej – dostaje 0 punktów (takie zadanie można by rozwiązać znacząco prościej).

Testy

Zestaw testów zawierał dziesięć grup, po dwa testy w każdej grupie. W każdej grupie pierwszy test jest losowy, natomiast drugi został wygenerowany według jednej z metod opisanych poniżej:

1b, 6b, 8b, 10b – po przekształceniu współrzędnych pozycje lamp tworzą regularną kratkę,

2b, 7b – lampy ustawione są w linii, każda oświetla każdą następną, wszystkie zapalają się w tym samym momencie,

3b, 9b – kąt ma zero stopni,

4b, 5b – lampy ustawione są w dwóch rzędach – każda lampa z pierwszego oświetla każdą lampę z drugiego.

Bartosz Tarnawski Treść zadania, Opracowanie
Adam Karczmarz, Tomasz Syposz Program

Dostępna pamięć: 128 MB. *OI, etap III, dzień drugi, 3.04.2014*

Panele słoneczne

Bajtazar postanowił zainwestować w odnawialne źródła energii i założył fabrykę paneli słonecznych. Okazało się to trafnym posunięciem – już po kilku dniach do Bajtazara zgłosiło się n klientów. Każdy z nich zamówił jeden prostokątny panel, podając przy tym dopuszczalny zakres jego wysokości i szerokości.

Produkowane panele składają się z kwadratowych ogniw fotowoltaicznych. Dostępne są ogniwa dowolnych całkowitych rozmiarów, ale wszystkie ogniwa danego panelu muszą być jednakowe. Stosowany proces technologiczny powoduje, że im większe są ogniwa, z których składa się panel, tym jest on bardziej wydajny. Bajtazar chciałby zatem dla każdego z zamówionych paneli poznać maksymalną długość boku ogniw, z których można go wyprodukować.

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą n ($1 \leqslant n \leqslant 1000$), oznaczającą liczbę zamówionych paneli. W kolejnych n wierszach znajdują się opisy poszczególnych paneli: i-ty z nich zawiera cztery liczby całkowite $s_{min}, s_{max}, w_{min}, w_{max}$ ($1 \leqslant s_{min} \leqslant s_{max} \leqslant 10^9$, $1 \leqslant w_{min} \leqslant w_{max} \leqslant 10^9$) pooddzielane pojedynczymi odstępami, oznaczające odpowiednio minimalną i maksymalną szerokość oraz minimalną i maksymalną wysokość i-tego panelu.

W testach wartych 75% punktów dla każdego panelu zachodzi dodatkowy warunek $s_{max}, w_{max} \leqslant 10^7$. W podzbiorze tych testów wartym 20% punktów zachodzi dodatkowy warunek $n \leqslant 10$.

Wyjście

Twój program powinien wypisać na standardowe wyjście dokładnie n wierszy, zawierających odpowiedzi do kolejnych przypadków testowych z wejścia: w i-tym wierszu ma znajdować się liczba całkowita oznaczająca maksymalną długość boku ogniw, z których można wyprodukować i-ty panel.

Przykład

Dla danych wejściowych:

```
4
3 9 8 8
1 10 11 15
4 7 22 23
2 5 19 24
```

poprawnym wynikiem jest:

```
8
7
2
5
```

Wyjaśnienie do przykładu: *Bajtazar wyprodukuje cztery panele słoneczne o następujących rozmiarach: 8×8 (złożony z jednego ogniwa), 7×14 (złożony z dwóch ogniw), 4×22 lub 6×22 (złożony z 22 lub 33 ogniw) oraz 5×20 (złożony z czterech ogniw).*

Testy „ocen”:

`1ocen`: *$n = 1000$, $s_{max}, w_{max} \leqslant 10^7$; dla każdego z zamówionych paneli, maksymalna długość boku ogniw, z których można go wyprodukować, jest równa s_{max}.*

Rozwiązanie

Powiemy, że liczba całkowita k jest *zgodna* z parą przedziałów o całkowitych końcach $([a, b], [c, d])$, jeśli istnieją dla niej całkowite x, y takie, że $kx \in [a, b]$ oraz $ky \in [c, d]$.

Dane jest n par przedziałów o współrzędnych końców pomiędzy 1 a m. Dla każdej z nich chcemy znaleźć największą liczbę zgodną z tą parą przedziałów.

Rozwiązanie wzorcowe $O(n\sqrt{m})$

Nasze rozwiązanie będzie działało *on-line* (tzn. na zapytania będziemy odpowiadać niezależnie) i wystarczy mu stała ilość pamięci.

Ustalmy parę przedziałów $([a, b], [c, d])$, nazwijmy ją P. Będziemy potrzebowali kilku obserwacji.

Lemat 1. Liczba k jest zgodna z P wtedy i tylko wtedy, gdy $\lceil \frac{a}{k} \rceil \leqslant \lfloor \frac{b}{k} \rfloor$ oraz $\lceil \frac{c}{k} \rceil \leqslant \lfloor \frac{d}{k} \rfloor$.

Dowód: Wystarczy zauważyć, że dla liczb całkowitych dodatnich k, x, b mamy równoważność: $kx \leqslant b \iff x \leqslant \lfloor \frac{b}{k} \rfloor$. Podobnie $kx \geqslant a \iff x \geqslant \lceil \frac{a}{k} \rceil$. ■

Lemat 2. Załóżmy, że k jest największą liczbą zgodną z P, $kx \in [a, b]$ oraz $ky \in [c, d]$. Wtedy $k = \lfloor \frac{b}{x} \rfloor$ lub $k = \left\lfloor \frac{d}{y} \right\rfloor$.

Dowód: Wiemy, że $k+1$ nie jest dobra, stąd $(k+1)x > b$ lub $(k+1)y > d$. Bez straty ogólności możemy założyć, że $(k + 1)x > b$, a wtedy nierówności $k \leqslant \frac{b}{x}$ i $(k + 1) > \frac{b}{x}$ implikują $k = \lfloor \frac{b}{x} \rfloor$. ■

Trzecia obserwacja wydaje się całkiem prosta, a jednak jest kluczem do rozwiązania zadania.

Lemat 3. Niech k, x, y będą jak w założeniach lematu 2 oraz niech $m = \max(b, d)$. Wówczas (1) $k \leqslant \sqrt{m}$ lub (2) $x, y \leqslant \sqrt{m}$.

Dowód: W przeciwnym razie $kx > b$ lub $ky > d$. ■

Teraz możemy już opisać wzorcowy algorytm:

1: **function** najlepsze_ogniwo($P = ([a, b], [c, d])$)
2: **begin**
3: $\quad m := \max(b, d)$;
4: $\quad wynik := 1$;
5: $\quad$ { Przypadek 1: $wynik \leqslant \sqrt{m}$ }

```
 6:    for k := 2 to ⌊√m⌋ do
 7:       if k zgodna z P then   { użyj lematu 1 }
 8:          wynik := k;
 9:
10:    { Przypadek 2: wynik ⩾ √m }
11:    for l := 1 to ⌊√m⌋ do begin
12:       k1 := ⌊b/l⌋;
13:       k2 := ⌊d/l⌋;
14:       for i in {1, 2} do
15:          if ki zgodna z P then   { użyj lematu 1 }
16:             wynik := max(wynik, ki);
17:    end
18:    return wynik;
19: end
```

Został on zaimplementowany w plikach `pan.cpp`, `pan1.pas`.

Rozwiązanie alternatywne $O(n\sqrt{m})$

Przypadek 2 możemy obsługiwać także w trochę inny sposób. Nie musimy korzystać z charakteryzacji największej zgodnej liczby z lematu 2. W zamian przychodzi nam z pomocą:

Lemat 4. Niech x, y będą liczbami całkowitymi. Oznaczmy $I_x = \left[\left\lceil \frac{a}{x} \right\rceil, \left\lfloor \frac{b}{x} \right\rfloor\right]$, $J_y = \left[\left\lceil \frac{c}{y} \right\rceil, \left\lfloor \frac{d}{y} \right\rfloor\right]$. Wówczas zbiór liczb całkowitych k takich, że $kx \in [a, b]$ oraz $ky \in [c, d]$, można zapisać jako przecięcie $I_x \cap J_y$, ograniczone do liczb całkowitych.

Będziemy szukać największej liczby całkowitej, która należy do przecięcia $I_x \cap J_y$ dla pewnych $x, y \leqslant \sqrt{m}$. Poszukiwanie możemy zrealizować w czasie $O(\sqrt{m})$.

```
 1: x := 1; y := 1;
 2: while x ⩽ √m and y ⩽ √m do begin
 3:    if Ix ∩ Jy ≠ ∅ then
 4:       return max Ix ∩ Jy;
 5:    else if Ix leży na lewo od Jy then begin
 6:       y := y + 1;
 7:       while y ⩽ √m and Jy = ∅ do
 8:          y := y + 1;
 9:    end else begin
10:       x := x + 1;
11:       while x ⩽ √m and Ix = ∅ do
12:          x := x + 1;
13:    end
14: end
15: return wynik mniejszy od √m, czyli zachodzi przypadek 1;
```

To rozwiązanie, tak samo jak wzorcowe, zużywa $O(1)$ pamięci. Zostało zaimplementowane w pliku `pan4.cpp`.

Rozwiązanie wolniejsze $O(n\sqrt{m}\log m)$

Również to rozwiązanie różni się od wzorcowego tylko sposobem obsługi przypadku 2. W pewnym sensie robi to samo co rozwiązanie alternatywne, ale korzystając z abstrakcji „zamiatania”.

Naszymi zdarzeniami będą lewe i prawe końce niepustych przedziałów typu I_x oraz J_y (zdefiniowanych w lemacie 4) dla $x, y \leqslant \sqrt{m}$. Mamy zatem cztery rodzaje zdarzeń (2 typy przedziałów $\times$ 2 końce odcinka).

Wrzucamy punkty do tablicy w dowolnej kolejności, a potem ją sortujemy; to zajmie nam $O(\sqrt{m}\log\sqrt{m}) = O(\sqrt{m}\log m)$ czasu. Następnie przeglądamy końce w kolejności malejących współrzędnych aż do wykrycia pierwszego przecięcia przedziałów dwóch różnych typów.

To rozwiązanie działa w czasie $O(n\sqrt{m}\log m)$ i na zawodach zdobywało około 70 punktów. Przykładowa implementacja znajduje się w pliku `pans11.cpp`. W rozwiązaniu tym można by osiągnąć czas $O(n\sqrt{m})$, gdyby punkty każdego rodzaju generować od razu w dobrej kolejności, a następnie w celu uzyskania posortowanej listy zdarzeń wykonać trzy scalenia.

Testy

Przygotowano 17 testów. Nie były grupowane, za to w obrębie pojedynczego testu występowały zapytania różnych rodzajów. Wykorzystano następujące typy zapytań:

- całkowicie losowe
- wynik jest duży
- przedziały są krótkie
- przedziały są krótkie, wynik jest mały
- wynik jest blisko początku jednego z przedziałów
- wynik na pewno nie należy do żadnego z przedziałów
- jeden przedział jest krótki, drugi długi
- $([cp, cp], [a, b])$, gdzie p jest liczbą pierwszą, $p > a, b$
- $([cp, cp], [a, b])$, gdzie $c \leqslant 8$, p jest liczbą pierwszą, $cp < a, b$ lub $cp > a, b$
- $([a, b], [c, d])$, gdzie $a \leqslant c \leqslant b \leqslant d$ (przedziały mają wspólny punkt)
- $([a, b + t], [c, d + t])$, gdzie b jest wynikiem, b nie dzieli żadnej z liczb $d + 1, d + 2, \ldots, d + t$.

Michał Włodarczyk — Treść zadania, Opracowanie
Bartosz Kostka, Paweł Parys — Program

Dostępna pamięć: 64 MB. *OI, etap III, dzień drugi, 3.04.2014*

Załadunek

Stacje kolejowe w Bajtołach Górnych i Bajtołach Dolnych połączone są jednym torem kolejowym. Pociąg pokonuje trasę w jedną stronę w ciągu s minut. Ze stacji pociągi nie mogą wyruszać częściej niż co minutę. Jeśli na torze znajduje się więcej niż jeden pociąg, to wszystkie muszą jechać w tym samym kierunku.

Wiemy, że na stację w Bajtołach Górnych zajedzie n pociągów, i znamy czasy ich przyjazdów. Każdy z pociągów musi dotrzeć na stację w Bajtołach Dolnych, gdzie załadowany zostanie towarem, a następnie wrócić na stację w Bajtołach Górnych. Dla uproszczenia zakładamy, że załadunek trwa pomijalnie krótko.

Należy wyznaczyć minimalny czas powrotu ostatniego pociągu na stację w Bajtołach Górnych.

Wejście

Pierwszy wiersz standardowego wejścia zawiera dwie liczby całkowite n, s ($1 \leqslant n \leqslant 1\,000\,000$, $1 \leqslant s \leqslant 10^9$) oddzielone pojedynczym odstępem, oznaczające liczbę pociągów i czas przejazdu w jedną stronę. W drugim wierszu znajduje się n liczb całkowitych $t_1, t_2, \ldots, t_n$ ($0 \leqslant t_1 \leqslant t_2 \leqslant \ldots \leqslant t_n \leqslant 10^9$) pooddzielanych pojedynczymi odstępami, oznaczających czasy przyjazdu kolejnych pociągów na stację w Bajtołach Górnych.

W testach wartych 50% punktów spełniony jest warunek $n \leqslant 5000$, a w podzbiorze tych testów wartym 25% punktów spełniony jest warunek $n \leqslant 400$.

Wyjście

Twój program powinien wypisać na standardowe wyjście jeden wiersz zawierający jedną liczbę całkowitą, oznaczającą minimalny czas, w którym wszystkie pociągi powrócą do stacji w Bajtołach Górnych.

Przykład

Dla danych wejściowych:

```
3 4
1 8 11
```

poprawnym wynikiem jest:

```
20
```

Wyjaśnienie do przykładu: *Aby osiągnąć optymalny czas, pociągi mogą wyruszyć ze stacji w Bajtołach Górnych w chwilach 1, 9 i 11, a ze stacji w Bajtołach Dolnych w chwilach 5, 15 i 16.*

Testy „ocen”:

`1ocen`: $n = 7$, $s = 10$, *prosty test poprawnościowy; warto pierwsze dwa pociągi wysłać razem i kolejne pięć razem;*

`2ocen`: $n = 100$, $s = 5$, *pociągi przyjeżdżają co 10 minut, każdy zdąży pokonać całą trasę zanim przyjedzie kolejny;*

`3ocen`: $n = 1000$, $s = 3$, *pociągi przyjeżdżają co minutę; najpierw wszystkie pociągi po kolei wysyłamy w jedną stronę, a potem w drugą.*

Rozwiązanie

Dla uproszczenia nazwijmy stację w Bajtołach Górnych stacją A, zaś tę w Bajtołach Dolnych – stacją B. Naszym celem jest wyznaczenie takiego planu przejazdu pociągów, aby ostatni z nich powrócił do stacji A jak najwcześniej. Próba wygenerowania wszystkich możliwych planów jest skazana na niepowodzenie z powodu ogromnej liczby kombinacji, nawet dla niewielkiej liczby pociągów. Warto zastanowić się, jaką strukturę powinno mieć rozwiązanie optymalne i jak przekuć to na efektywny algorytm. W tym celu pokażemy kilka obserwacji, które pozwalają odrzucić nieciekawe plany i upraszczają poszukiwanie rozwiązania.

Analiza zadania

Zauważmy na początek, że żadne dwa pociągi nie mogą odjechać w tym samym czasie. Biorąc to pod uwagę, ciąg (t_i) możemy przekształcić w ciąg (a_i), gdzie a_i oznacza najwcześniejszy moment, w którym pociąg i może teoretycznie odjechać ze stacji A. Jeśli w (t_i) kilka pociągów przyjeżdża w tym samym czasie, to w (a_i) każdy z nich przyjedzie minutę później od poprzedniego. Konstrukcję ciągu (a_i) przedstawia poniższy pseudokod.

1: $a_1 := t_1$;
2: **for** $i := 2$ **to** n **do**
3: $\quad a_i := \max(t_i,\, a_{i-1} + 1)$;

Podamy teraz dwie proste obserwacje, które zaowocują pierwszym algorytmem rozwiązującym problem w czasie wielomianowym.

Fakt 1. *Istnieje rozwiązanie optymalne, w którym pociągi odjeżdżają ze stacji A oraz ze stacji B w tej samej kolejności, w której przybywają do A.*

Dowód: Rozważmy plan, według którego pociąg p przybywa przed pociągiem q do A, jednak q odjeżdża pierwszy z A. Możemy zamienić ich czasy odjazdów, ponieważ w momencie odjazdu q pociąg p czeka już na stacji. Analogicznie możemy zamienić kolejność odjazdów z B.

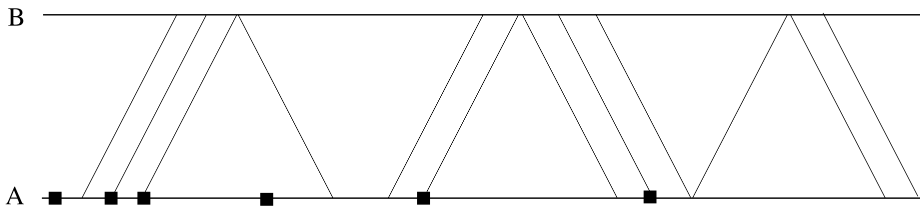
Powyższa zamiana daje poprawny plan o takim samym czasie zakończenia. Powtarzamy tę operację, dopóki istnieje taka para (p, q), aż ostatecznie otrzymamy nowe rozwiązanie optymalne o takiej samej kolejności odjazdów co przyjazdów.

Czytelnik może dociekać, czy aby na pewno po skończonej liczbie operacji otrzymamy plan spełniający powyższe warunki. Tak jest w istocie, ponieważ za każdym razem maleje liczba par pociągów (p, q), które odjeżdżają w innym porządku, niż przyjeżdżają[1]. ■

Fakt 2. *Można ograniczyć się do takich rozwiązań, w których jeśli dwa kolejne odjazdy są w tym samym kierunku, to dzieli je dokładnie jedna minuta.*

Dowód: Rozumujemy podobnie jak poprzednio: wybieramy rozwiązanie optymalne niespełniające założenia i staramy się je poprawić. Przypuśćmy zatem, że pociąg p odjeżdża w chwili t, pociąg q odjeżdża w chwili $u > t+1$ w tym samym kierunku oraz żaden pociąg nie odjeżdża w przedziale czasowym $[t+1, u-1]$. Modyfikujemy ten plan, opóźniając odjazd pociągu p do minuty $u-1$. Nie przeszkodzi to pociągowi q, jako że jedzie on w tę samą stronę. Również nie przeszkodzi to żadnym innym pociągom, gdyż pociąg p i tak dojedzie do celu wcześniej niż pociąg q.

Ponownie chcielibyśmy być pewni, że po pewnej liczbie modyfikacji otrzymamy plan, którego nie da się już poprawić. Tym razem zachęcamy Czytelnika do samodzielnego uzasadnienia tego faktu. ■



Rys. 1: Przykładowy plan przemieszczania się pociągów na osi czasu. Ukośne kreski reprezentują przejazdy pociągów pomiędzy stacjami. Czarne kwadraty oznaczają chwile pojawiania się kolejnych pociągów w A.

Pierwszy algorytm

Od teraz będziemy skupiać się tylko na rozwiązaniach spełniających warunki z faktów 1 i 2. Możemy zatem myśleć, że pociągi poruszają się w spójnych *blokach*; przykładowo, pociągi o numerach $i, i+1, \ldots, j$ (numeracja pochodząca od kolejności pojawiania się w A) wyruszają z jednej stacji w momentach $t, t+1, \ldots, t+j-i$. Pozwala to rozwiązać problem przy pomocy programowania dynamicznego[2].

Niech $dp_{i,j}$ oznacza najwcześniejszy moment, w którym i pierwszych pociągów może dotrzeć do B, a ponadto $j \leqslant i$ z nich może powrócić już do A. Oczywiście poszukiwaną przez nas wartością jest $dp_{n,n}$. Wiemy na początek, że $dp_{0,0} = 0$.

[1] Własność, że pewna dodatnia zmienna całkowita maleje przy każdej operacji, nazywa się *półniezmiennikiem*. Tego typu własności pomagają dowodzić, że dany program zakończy się po skończonej liczbie kroków.

[2] Programowanie dynamiczne to powszechnie stosowana technika rozwiązywania problemów algorytmicznych przez dzielenie ich na podproblemy. Można o niej poczytać w książce [25].

Jak obliczyć $dp_{i,j}$? Jeśli w optymalnym rozwiązaniu odpowiadającym tej sytuacji poprzedni blok pociągów przyjechał z B do A i były to pociągi o numerach $k, \ldots, j$, to wartość $dp_{i,j}$ wynosiłaby

$$dp_{i,k-1} + (j-k) + s. \tag{1}$$

Jak wyglądałby wtedy plan? Do chwili $t = dp_{i,k-1}$ postępujemy zgodnie z optymalnym planem przerzucenia i pociągów do B i powrotu $k-1$ do A. Zaczynając od chwili t, wysyłamy $j-k+1$ pociągów z B do A jeden po drugim. Ostatni z nich dojedzie do A w chwili $t + (j-k) + s$.

Jeśli natomiast ostatni blok o numerach $k, \ldots, i$ $(k > j)$ przemieszczał się w kierunku B, to wartość $dp_{i,j}$ byłaby równa

$$\max(dp_{k-1,j} + (i-k),\ a_i) + s. \tag{2}$$

Różnica we wzorach bierze się stąd, że w drugim przypadku musimy „poczekać” z wypuszczeniem bloku pociągów, aż i-ty pociąg pojawi się na stacji A. Niech $t = \max(dp_{k-1,j} + (i-k), a_i)$. Jako że ciąg (a_i) jest ściśle rosnący, to w momentach $t-i+k, \ldots, t$ możemy wysłać $i-k+1$ pociągów do B. Pierwszy z nich odjedzie w chwili $t-i+k \geqslant dp_{k-1,j}$, więc nie będzie miał miejsca konflikt z wcześniejszymi przejazdami, zaś ostatni wyjedzie nie wcześniej niż w chwili a_i.

Powyższy schemat programowania dynamicznego ma $O(n^2)$ stanów, a rozpatrzenie pojedynczego stanu wymaga obliczenia minimum z $O(n)$ wartości. Daje to algorytm o złożoności czasowej $O(n^3)$, który jest wystarczająco szybki, by przejść niewielkie testy, i zgodnie z treścią zadania pozwalał zdobyć na zawodach 25% punktów. Jego implementacje można znaleźć w plikach `zals2.cpp` i `zals5.pas`.

Główna obserwacja i rozwiązanie kwadratowe

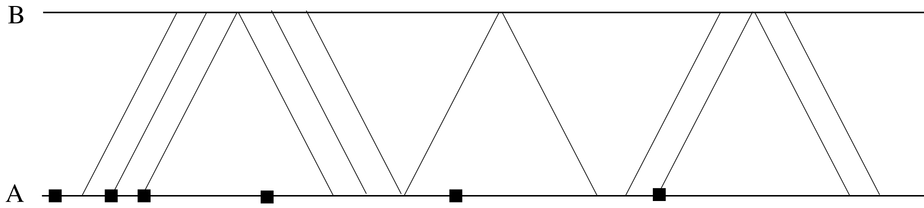
Poniższy lemat jest kluczem do poprawienia złożoności czasowej pierwszego algorytmu.

Lemat 1. Istnieje rozwiązanie optymalne, w którym za każdym razem, gdy pewien pociąg wraca z B, to bezpośrednio po nim wracają wszystkie pozostałe pociągi będące wtedy w B.

Dowód: Rozważmy rozwiązanie, w którym w chwili t z B odjeżdża pociąg p, a w chwili $u > t+1$ z B odjeżdża pociąg q, który był w B już w chwili t. Załóżmy dodatkowo, że w chwilach $[t+1, u-1]$ z B nie odjeżdżają żadne inne pociągi. Wówczas możemy dokonać następującej zmiany planu: niech pociąg q odjeżdża z B w chwili $t+1$ (nie będzie on kolidował z pociągami odjeżdżającymi wcześniej ze względu na przejazd minutę wcześniej pociągu p), zaś wszystkie odjazdy z B zaplanowane między $t+1$ a $u-1$ oraz wszystkie odjazdy z A zaplanowane między $t+s$ a $u-s$ opóźniamy o minutę (nie będą one kolidowały z pociągami odjeżdżającymi później, gdyż poprzednio q odjeżdżał z B w chwili u).

Nowy plan ma czas zakończenia taki sam albo lepszy niż plan wyjściowy. Wykonując serię takich zamian, możemy uzyskać plan optymalny, który spełnia założenia

lematu. Podobnie jak w dowodzie faktu 2 polecamy dociekliwemu Czytelnikowi sprawdzenie, że zawsze wystarczy skończona liczba zamian. ■



Rys. 2: Plan przemieszczania się pociągów podobny do tego z rysunku 1, ale wykorzystujący obserwację z lematu 1.

Możemy rozważać teraz tylko takie rozwiązania, w których spójne bloki pociągów kolejno odjeżdżają do B i od razu wracają do A. Pozwala to napisać prostszy schemat programowania dynamicznego. Niech dp_i oznacza najwcześniejszą minutę, po której i-ty pociąg (i wszystkie wcześniejsze) może się znaleźć ponownie w A. Wtedy $dp_0 = 0$, zaś wynikiem poszukiwanym w zadaniu jest dp_n.

Jak wyznaczyć dp_i? Przypuśćmy, że w optymalnym rozwiązaniu pociąg i pojechał w bloku $j, \ldots, i$. Wtedy

$$dp_i = \max(dp_{j-1} + (i-j), a_i) + (i-j) + 2 \cdot s. \tag{3}$$

Interpretacja tego wzoru jest taka sama jak przy wzorach (1), (2). Aby obliczyć dp_i, wystarczy wybrać minimum po wszystkich j, co daje algorytm działający w czasie $O(n^2)$. Jest on szybszy od poprzedniego i pozwalał zdobyć 50% punktów. Można go znaleźć w plikach `zals1.cpp` i `zals4.pas`.

Rozwiązanie liniowe – wzorcowe

Rozwiązanie optymalne wykorzystuje to samo podejście co poprzedni algorytm, jednak oblicza dp_i zdecydowanie sprytniej.

Fakt 3. *Dla każdego i zachodzi* $dp_{i+1} \geqslant dp_i + 2$.

Dowód: Rozważmy optymalny plan dla $i+1$ pociągów. Zgodnie z faktem 1 możemy założyć, że pociąg o numerze $i+1$ jako ostatni wyjeżdża z A i jako ostatni wraca do A. Usuńmy te dwa przejazdy, a wszystkie pociągi wracające razem z nim wyślijmy z B o minutę wcześniej. Skonstruowany plan sprowadza pierwsze i pociągów do A w czasie nie większym niż $dp_{i+1} - 2$. ■

Oznaczmy

$$\begin{aligned} f(i,j) &:= dp_{j-1} + (i-j) + (i-j) + 2 \cdot s, \\ g(i,j) &:= a_i + (i-j) + 2 \cdot s, \\ m(i,j) &:= \max\left(f(i,j),\, g(i,j)\right). \end{aligned}$$

Przy takich oznaczeniach

$$dp_i = \min_{1 \leqslant j \leqslant i} m(i, j).$$

Fakt 4. *Dla ustalonego i funkcja g jest ściśle malejąca, a funkcja f jest niemalejąca.*

Dowód: Mamy $g(i, j+1) = g(i, j) - 1$, co uzasadnia pierwszą część tezy. Aby udowodnić drugą część, wystarczy spojrzeć na różnicę $f(i, j+1) - f(i, j) = dp_j - dp_{j-1} - 2 \geqslant 0$. Ostatnia nierówność wynika z faktu 3. ■

Niech k_i będzie największym takim $j \in \{1, \ldots, i\}$, że $f(i, j) \leqslant g(i, j)$ (jeśli takie j nie istnieje, to przyjmujemy $k_i = 1$). Z faktu 4 wynika, że dla wszystkich $j < k_i$ zachodzi $f(i, j) < g(i, j)$.

Fakt 5. *Tak dobrane k_i spełnia $m(i, k_i) = \min_{1 \leqslant j \leqslant i} m(i, j)$.*

Dowód: Dla $j > k_i$ mamy

$$m(i, j) = f(i, j) \geqslant f(i, k_i + 1) \geqslant g(i, k_i + 1) + 1 = g(i, k_i) = m(i, k_i).$$

Natomiast dla $j < k_i$ zachodzi

$$m(i, j) = g(i, j) > g(i, k_i) = m(i, k_i),$$

co dowodzi tezy. ■

Wiemy zatem, że $dp_i = m(i, k_i)$. Korzystając z faktu 4, można znajdować k_i przy użyciu wyszukiwania binarnego. My jednak pójdziemy dalej i zaproponujemy jeszcze efektywniejszy sposób wyznaczania k_i.

Fakt 6. *Ciąg (k_i) jest niemalejący.*

Dowód: Zauważmy, że

$$\begin{aligned} g(i, j) - f(i, j) = a_i - dp_{j-1} - (i - j) \leqslant \\ \leqslant a_{i+1} - dp_{j-1} - (i + 1 - j) = g(i + 1, j) - f(i + 1, j), \end{aligned}$$

ponieważ $a_{i+1} > a_i$. W takim razie nierówność $f(i, j) \leqslant g(i, j)$ implikuje $f(i + 1, j) \leqslant g(i + 1, j)$. Zgodnie z definicją k_i oznacza to, że $k_{i+1} \geqslant k_i$. ■

Skoro $k_{i+1} \geqslant k_i$, to zwiększając i, wystarczy sprawdzić, czy k_i się nie zwiększa. Obserwacja ta pozwala znajdować k_i w zamortyzowanym czasie stałym, dzięki czemu można zaimplementować cały algorytm w złożoności liniowej. Rozwiązanie wzorcowe zostało zaimplementowane w plikach `zal.cpp`, `zal1.pas` i `zal2.cpp`.

Rozwiązanie alternatywne

Okazuje się, że zadanie można rozwiązać także innym sposobem, korzystając z całkiem odmiennej obserwacji. Podobnie jak poprzednio przekształcamy najpierw ciąg (t_i) na (a_i). Następnie chcielibyśmy umieć odpowiadać szybko na pytanie, czy wszystkie pociągi mogą przejechać w czasie T minut. Okazuje się, że opłaca się wysłać ostatni pociąg do B dokładnie w chwili a_n, a tuż przed nim jak najwięcej innych pociągów. Wówczas razem z ostatnim do A zdąży wrócić $k := T - a_n - 2 \cdot s + 1$ pociągów (wliczając jego samego).

Lemat 2. Pociągi mogą przejechać w czasie T wtedy i tylko wtedy, gdy $k > 0$ oraz pierwszych $\max(0, n - k)$ pociągów może przejechać w czasie $\max(0, a_n - k + 1)$.

Dowód: ($\Leftarrow$) Najpierw wykażemy, że jeśli oba warunki po prawej są spełnione, to istnieje plan przejazdu, w którym ostatni pociąg powraca nie później niż w minucie T. Jeśli $k \geqslant n$, to wszystkie pociągi odjeżdżają z A w jednym bloku w minutach $a_n - n + 1, \ldots, a_n$ (jest to możliwe, ponieważ ciąg (a_i) jest rosnący), ostatni z nich przybywa w minucie $a_n + s$ i wtedy zaczynają wracać do A. Ostatni z nich wróci w minucie $a_n + s + n - 1 + s \leqslant a_n + k + 2 \cdot s - 1 = T$.

Jeśli zaś $k < n$, to zgodnie z założeniem pierwsze $n - k$ pociągów powraca do A w czasie nie późniejszym niż $a_n - k + 1$. Wysyłamy ostatni blok k pociągów w minutach $a_n - k + 1, \ldots, a_n$. Podobnie jak poprzednio, ostatni z nich przybywa do B w minucie $a_n + s$. Jeśli pociągi zaczną od razu odjeżdżać do A, to ostatni przybędzie w chwili $a_n + k + 2 \cdot s - 1 = T$.

($\Rightarrow$) Przypuśćmy teraz, że istnieje plan przejazdu o czasie zakończenia $\leqslant T$. Wtedy oczywiście $k > 0$, bo inaczej nawet ostatni pociąg nie zdążyłby wrócić do A. Jeśli $k > n$, to teza w oczywisty sposób zachodzi; odtąd zakładamy, że $k \leqslant n$. Niech m oznacza liczbę pociągów, które wracają w ostatnim bloku do A (tzn. $(m + 1)$-szy przejazd od końca następuje z A do B). Zauważmy, że te pociągi muszą wracać razem z ostatnim, a ten znajdzie się w B najwcześniej w minucie $a_n + s$. W takim razie ostatni pociąg powróci do A najwcześniej w chwili $(a_n + s) + (m + s - 1)$. Jako że $T = a_n + k + 2 \cdot s - 1$, to musi zachodzić $m \leqslant k$.

Przypomnijmy, że zgodnie z faktem 1 możemy zakładać, że pociągi odjeżdżają z obu stacji w tej samej kolejności, w której przybyły do A. Postarajmy się przekształcić nasz plan dla n pociągów w plan dla $n - k$ pociągów, usuwając je po kolei od ostatniego. Za każdym razem kiedy usuwamy z planu jeden pociąg p, czas zakończenia zmniejsza się o co najmniej 2, bowiem ostatni przyjazd do B następuje o minutę wcześniej, zaś wszystkie powroty po odjeździe p z A można przyspieszyć o jedną minutę. Jeżeli żaden pociąg nie kursuje po przybyciu p do A, to usunięcie p pozwala zakończyć załadunek aż o $2 \cdot s$ minut szybciej. Skoro $m \leqslant k$, to w trakcie usuwania k ostatnich pociągów, przynajmniej dla jednego z nich przydarzy się opisana wyżej sytuacja. Zatem otrzymamy plan dla $n - k$ pociągów o czasie zakończenia $\leqslant T - 2 \cdot (k - 1) - 2 \cdot s = a_n - k + 1$, co należało udowodnić. ∎

Stosując wielokrotnie lemat 2, możemy w czasie liniowym rozstrzygnąć, czy istnieje plan o czasie zakończenia nieprzekraczającym T. Możemy skorzystać z wyszukiwania binarnego do wyznaczenia najmniejszego T o tej własności. Wiemy, że T nie może

być mniejsze niż $a_n + 2 \cdot s$, natomiast $T = a_n + 2 \cdot s + n - 1$ na pewno wystarczy. Otrzymujemy zatem rozwiązanie o złożoności czasowej $O(n \log n)$ – nieco gorszej od wzorcowej, ale w rzeczywistości nie da się go odróżnić od rozwiązania wzorcowego i zdobywa 100% punktów. Zostało ono zaimplementowane w pliku `zal3.cpp`.

Rozwiązania niepoprawne

Pierwszym błędem, który mogli popełnić zawodnicy, jest niezauważenie, że wynik może nie mieścić się w 32-bitowej zmiennej całkowitej. Testy wykrywające to niedopatrzenie były warte w sumie 25% punktów.

Drugim błędem jest zignorowanie możliwości, że pociągi mogą pojawiać się w A w tym samym momencie. Można było stracić w ten sposób aż 58% punktów.

Rozwiązania zachłanne, który wysyłały pociągi w trasę, gdy tylko nie jechał żaden inny pociąg z naprzeciwka, nie uzyskiwały żadnych punktów.

XXVI Międzynarodowa Olimpiada Informatyczna,

Tajpej, Tajwan 2014

Krzysztof Diks, Jakub Radoszewski
Tłumaczenie

IOI 2014, dzień pierwszy, 15.07.2014

Gra

Jian-Jia jest chłopcem, który uwielbia grać w gry. Zapytany o coś, nigdy nie odpowiada bezpośrednio, lecz zamiast tego wymyśla grę, w wyniku której jego rozmówca może poznać odpowiedź na swoje pytanie. Pewnego dnia Jian-Jia spotkał swoją przyjaciółkę Mei-Yu i opowiedział jej o sieci połączeń lotniczych na Tajwanie. Na Tajwanie jest n miast ponumerowanych liczbami $0, \ldots, n-1$. Między niektórymi parami miast istnieją bezpośrednie połączenia lotnicze. Każde takie połączenie jest dwukierunkowe.

Mei-Yu była ciekawa, czy za pomocą samolotów można przemieścić się pomiędzy każdymi dwoma miastami na Tajwanie (bezpośrednio lub pośrednio). Jian-Jia nie odpowiedział na pytanie przyjaciółki, ale zaproponował jej pewną grę. Mei-Yu może w niej zadawać Jian-Jia pytania postaci „Czy istnieje **bezpośrednie** *połączenie lotnicze pomiędzy miastami x oraz y?”. Mei-Yu ma zadać łącznie $r = n(n-1)/2$ pytań, pytając o każde połączenie dokładnie raz. Jian-Jia odpowiada na każde pytanie od razu, nie czekając na kolejne pytania przyjaciółki. Mei-Yu wygrywa, jeśli po zadaniu i pytań, dla pewnego $i < r$, może stwierdzić, czy sieć połączeń lotniczych jest spójna, tzn. czy da się podróżować samolotami pomiędzy każdymi dwoma miastami (bezpośrednio lub pośrednio). Jeśli do stwierdzenia, czy sieć jest spójna czy nie, potrzebuje ona r pytań, wówczas wygrywa Jian-Jia.*

Żeby gra była ciekawsza dla Jian-Jia, jego przyjaciółka zgodziła się, aby gra nie odnosiła się do rzeczywistej sieci lotniczej na Tajwanie. Zamiast tego Jian-Jia może tworzyć strukturę wyimaginowanej sieci na bieżąco w trakcie gry, biorąc pod uwagę swoje wcześniejsze odpowiedzi na pytania Mei-Yu. Twoim zadaniem jest pomóc Jian-Jia zwyciężyć, podpowiadając mu odpowiedzi na kolejne pytania.

Przykłady

Reguły gry zostaną objaśnione na trzech przykładach. W każdym z przykładów liczba miast to $n = 4$, a liczba rund pytań i odpowiedzi to $r = 6$.

W pierwszym przykładzie (tabela poniżej) Jian-Jia **przegrywa**, *ponieważ po rundzie 4 Mei-Yu wie na pewno, że można przelecieć samolotami pomiędzy każdymi dwoma miastami, niezależnie od odpowiedzi Jian-Jia na pytania 5 i 6.*

runda	*pytanie*	*odpowiedź*
1	*0, 1*	*tak*
2	*3, 0*	*tak*
3	*1, 2*	*nie*
4	*0, 2*	*tak*
5	*3, 1*	*nie*
6	*2, 3*	*nie*

W kolejnym przykładzie Mei-Yu może wykazać już po 3 rundzie, że **nie można** *przelecieć samolotami pomiędzy miastami 0 i 1, niezależnie od tego, co Jian-Jia odpowie na pytania 4, 5 i 6. Jian-Jia znowu przegrywa.*

runda	*pytanie*	*odpowiedź*
1	*0, 3*	*nie*
2	*2, 0*	*nie*
3	*0, 1*	*nie*
4	*1, 2*	*tak*
5	*1, 3*	*tak*
6	*2, 3*	*tak*

W ostatnim przykładzie Mei-Yu nie może stwierdzić, czy pomiędzy każdymi dwoma miastami da się przelecieć samolotami, czy nie, aż do momentu, gdy zada wszystkie 6 pytań. Jian-Jia **wygrywa** *grę. Jest tak dlatego, że jeśli Jian-Jia odpowie na ostatnie pytanie „tak" (patrz tabela poniżej), to podróż lotnicza pomiędzy każdą parą miast jest możliwa, natomiast jeśli Jian-Jia odpowie na ostatnie pytanie „nie", wówczas istnieje para miast, pomiędzy którymi podróż lotnicza nie jest możliwa.*

runda	*pytanie*	*odpowiedź*
1	*0, 3*	*nie*
2	*1, 0*	*tak*
3	*0, 2*	*nie*
4	*3, 1*	*tak*
5	*1, 2*	*nie*
6	*2, 3*	*tak*

Zadanie

Napisz program, który pomoże Jian-Jia zwyciężyć w grze. Zauważ, że ani Mei-Yu, ani Jian-Jia nie znają strategii przeciwnika. Mei-Yu może pytać o połączenia pomiędzy parami miast w dowolnej kolejności, a Jian-Jia musi odpowiadać na każde pytanie od razu, nie znając kolejnych pytań. Twoje zadanie polega na zaimplementowaniu dwóch następujących funkcji.

- `initialize(n)` – *Funkcja* `initialize` *zostanie wywołana jako pierwsza. Parametr n oznacza liczbę miast.*
- `hasEdge(u, v)` – *Funkcja* `hasEdge` *będzie wywoływana $r = n(n-1)/2$ razy. Te wywołania reprezentują pytania Mei-Yu i będą wykonywane w kolejności ich zadawania. Na każde z nich należy odpowiedzieć, czy istnieje bezpośrednie połączenie lotnicze pomiędzy miastami u i v. Jeśli takie bezpośrednie połączenie istnieje, wówczas wynikiem wywołania funkcji powinno być 1, a w przeciwnym przypadku powinno to być 0.*

Podzadania

Każde podzadanie składa się z pewnej liczby rozgrywek. Twój program otrzyma punkty za dane podzadanie, tylko jeśli w imieniu Jian-Jia wygra wszystkie rozgrywki.

podzadanie	*liczba punktów*	n
1	*15*	$n = 4$
2	*27*	$4 \leqslant n \leqslant 80$
3	*58*	$4 \leqslant n \leqslant 1500$

Implementacja

Powinieneś zgłosić dokładnie jeden plik o nazwie `game.c`*,* `game.cpp` *lub* `game.pas`*. W tym pliku powinna znaleźć się implementacja funkcji opisanych powyżej, o następujących sygnaturach.*

Programy w C/C++

```
void initialize(int n);
int hasEdge(int u, int v);
```

Programy w Pascalu

```
procedure initialize(n: longint);
function hasEdge(u, v: longint): longint;
```

Przykładowy program sprawdzający

Przykładowy program sprawdzający wczytuje dane w następującym formacie:

- *wiersz 1:* `n`
- *kolejne r wierszy: każdy wiersz zawiera dwie liczby całkowite* `u` *i* `v`*, które opisują pytanie odnoszące się do miast u i v.*

Ograniczenia

Limit czasu: *1 s*

Dostępna pamięć: *256 MB*

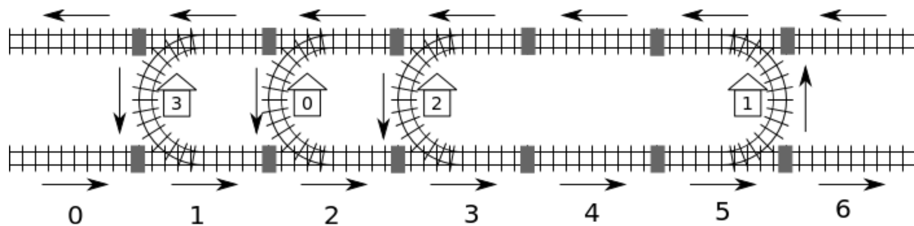
Krzysztof Diks, Jakub Radoszewski
Tłumaczenie

IOI 2014, dzień pierwszy, 15.07.2014

Kolej

Przez Tajwan przebiega długa linia kolejowa łącząca zachodnie i wschodnie wybrzeże wyspy. Linia ta składa się z m bloków. Kolejne bloki są ponumerowane liczbami $0, \ldots, m-1$, począwszy od zachodniego końca linii. W każdym bloku znajduje się jednokierunkowy tor biegnący na zachód położony na północy bloku oraz jednokierunkowy tor biegnący na wschód położony na południu bloku. Między torami bloku może znajdować się stacja kolejowa.

Możliwe są trzy typy bloków. Blok typu **C** *zawiera stację kolejową, na którą wjeżdża się z toru północnego i którą opuszcza się torem południowym. Blok typu* **D** *zawiera stację kolejową, na którą wjeżdża się z toru południowego i którą opuszcza się torem północnym. Blok* **pusty** *nie zawiera stacji kolejowej. Dla przykładu, na poniższym rysunku bloki 0, 4 i 6 są puste, bloki 1, 2 i 3 są typu C, a blok 5 jest typu D. Tory kolejnych bloków są połączone za pomocą* **łączników***; są one przedstawione na rysunku jako zaciemnione prostokąciki.*



Linia kolejowa zawiera n stacji ponumerowanych od 0 do $n-1$. Należy założyć, że za pomocą torów kolejowych **można dotrzeć z każdej stacji do każdej innej stacji***. Przykładowo, aby dotrzeć ze stacji 0 do stacji 2, należy zacząć w bloku 2, następnie przejechać przez bloki 3 i 4, używając torów południowych, później przejechać przez blok 5, mijając stację 1, następnie przejechać przez blok 4, używając toru północnego, by na końcu dojechać do stacji 2 w bloku 3.*

Jako że dwie stacje może łączyć więcej niż jedna trasa, odległość między stacjami definiujemy jako **minimalną liczbę łączników** *na ścieżce między tymi stacjami. Dla przykładu, najkrótsza ścieżka ze stacji 0 do stacji 2 biegnie blokami 2-3-4-5-4-3 i przechodzi przez 5 łączników, więc odległość między tymi dwiema stacjami jest równa 5.*

Struktura systemu kolejowego była przechowywana z użyciem specjalnego oprogramowania. Niestety usterka zasilania spowodowała uszkodzenie danych programu, przez co informacje o położeniach stacji i typach bloków zostały utracone. W pamięci pozostała tylko informacja o numerze bloku, w którym znajduje się stacja 0, oraz informacja, że jest to blok typu C. Szczęśliwie, program wciąż pozwala odpowiadać na zapytania o odległość między dowolnymi dwiema stacjami. Przykładowo, w programie możemy zapytać: „Jaka jest odległość od stacji 0 do stacji 2" i uzyskać odpowiedź: 5.

Zadanie

Napisz funkcję `findLocation(n, first, location, stype)`*, która określi dla każdej stacji numer oraz typ bloku, w którym ta stacja się znajduje. Parametry tej funkcji to:*

- `n`*: liczba stacji*
- `first`*: numer bloku, w którym znajduje się stacja 0*
- `location`*: tablica rozmiaru n;* `location[i]` *powinno na końcu działania funkcji zawierać numer bloku, w którym znajduje się stacja i.*
- `stype`*: tablica rozmiaru n;* `stype[i]` *powinno na końcu działania funkcji wskazywać typ bloku, w którym znajduje się stacja i: 1 dla typu C, a 2 dla typu D.*

Przy ustalaniu położeń stacji i typów bloków Twoja funkcja może używać funkcji `getDistance`*:*

- `getDistance(i, j)` *daje w wyniku odległość od stacji i do stacji j.* `getDistance(i, i)` *daje wynik 0.* `getDistance(i, j)` *daje wynik -1, jeśli i lub j znajduje się poza zakresem* $0 \leqslant i, j \leqslant n-1$*.*

Podzadania

We wszystkich podzadaniach liczba bloków m nie przekracza 1 000 000. W niektórych podzadaniach liczba wywołań funkcji `getDistance` *jest ograniczona. Ograniczenie to zależy od konkretnego podzadania. Jeśli Twój program przekroczy to ograniczenie, zostanie oceniony jako błędny.*

podzadanie	liczba punktów	n	max liczba wywołań getDistance	dodatkowe ograniczenia
1	8	$1 \leqslant n \leqslant 100$	brak ograniczeń	Wszystkie stacje poza stacją 0 znajdują się w blokach typu D.
2	22	$1 \leqslant n \leqslant 100$	brak ograniczeń	Wszystkie stacje położone na wschód od stacji 0 znajdują się w blokach typu D, a wszystkie stacje położone na zachód od stacji 0 znajdują się w blokach typu C.
3	26	$1 \leqslant n \leqslant 5\,000$	$n(n-1)/2$	brak
4	44	$1 \leqslant n \leqslant 5\,000$	$3(n-1)$	brak

Implementacja

Powinieneś zgłosić dokładnie jeden plik o nazwie `rail.c`*,* `rail.cpp` *lub* `rail.pas`*. W pliku powinna znaleźć się implementacja funkcji* `findLocation` *opisanej powyżej, o następującej sygnaturze. W przypadku programu w C/C++ powinieneś także załączyć plik nagłówkowy* `rail.h`*.*

Programy w C/C++

```
void findLocation(int n, int first, int location[], int stype[]);
```

Programy w Pascalu

```
procedure findLocation(n, first : longint; var location,
stype : array of longint);
```

Sygnatury funkcji `getDistance` *są następujące.*

Programy w C/C++

```
int getDistance(int i, int j);
```

Programy w Pascalu

```
function getDistance(i, j: longint): longint;
```

Przykładowy program sprawdzający

Przykładowy program sprawdzający wczytuje dane w następującym formacie:

- *wiersz 1: numer podzadania*
- *wiersz 2:* `n`
- *wiersz* $3 + i$ *(*$0 \leqslant i \leqslant n - 1$*):* `stype[i]` *(1 dla typu C i 2 dla typu D),* `location[i]`.

Przykładowy program sprawdzający wypisze komunikat „`Correct`*", jeśli* `location[0]`*, ...,* `location[n-1]` *oraz* `stype[0]`*, ...,* `stype[n-1]` *na końcu wywołania Twojej funkcji* `findLocation` *będą identyczne z tymi podanymi na wejściu, a w przeciwnym przypadku wypisze komunikat „*`Incorrect`*".*

Ograniczenia

Limit czasu: *3 s*

Dostępna pamięć: *256 MB*

Krzysztof Diks, Jakub Radoszewski
Tłumaczenie

IOI 2014, dzień pierwszy, 15.07.2014

Ściana

Jian-Jia buduje ścianę złożoną z jednakowych cegieł. Ściana zawiera n kolumn cegieł, ponumerowanych od 0 do $n-1$, patrząc od lewej do prawej. Kolumny mogą mieć różne wysokości – wysokością kolumny nazywamy liczbę cegieł w tej kolumnie.

Jian-Jia buduje ścianę w następujący sposób. Początkowo w żadnej kolumnie nie ma żadnych cegieł. Następnie Jian-Jia wykonuje k faz, z których każda polega na **dodawaniu** *lub* **usuwaniu** *cegieł. Proces budowania kończy się, gdy wszystkie k faz zostaje ukończonych. W każdej fazie Jian-Jia ma wybrany spójny ciąg kolumn oraz wysokość h i wykonuje następujące czynności:*

- *W fazie* **dodawania** *w każdej kolumnie o wysokości mniejszej niż h w zadanym ciągu kolumn Jian-Jia umieszcza dodatkowo tyle cegieł, by miała ona wysokość dokładnie h. Kolumny o wysokości nie mniejszej niż h pozostawia bez zmian.*
- *W fazie* **usuwania** *z każdej kolumny o wysokości większej niż h w zadanym ciągu kolumn Jian-Jia usuwa tyle cegieł, by miała ona wysokość dokładnie h. Kolumny o wysokości co najwyżej h pozostawia bez zmian.*

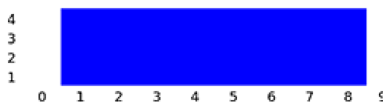
Twoim zadaniem jest wyznaczyć ostateczny kształt ściany.

Przykład

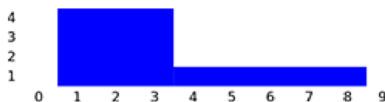
Załóżmy, że ściana zawiera 10 kolumn cegieł, a nasz bohater wykonuje 6 faz. Wszystkie zakresy wymienione w poniższej tabeli zawierają końce. Rysunki poglądowe ściany po wykonaniu kolejnych faz są umieszczone poniżej.

faza	*typ fazy*	*zakres*	*wysokość*
0	*dodawanie*	*kolumny od 1 do 8*	*4*
1	*usuwanie*	*kolumny od 4 do 9*	*1*
2	*usuwanie*	*kolumny od 3 do 6*	*5*
3	*dodawanie*	*kolumny od 0 do 5*	*3*
4	*dodawanie*	*kolumna 2*	*5*
5	*usuwanie*	*kolumny od 6 do 7*	*0*

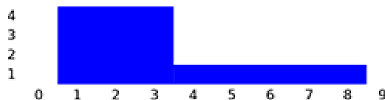
Ponieważ początkowo kolumny są puste, to po fazie 0 każda z kolumn od 1 do 8 zawiera po 4 cegły. Kolumny 0 i 9 pozostają puste.



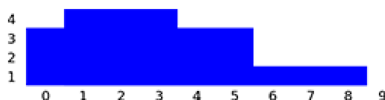
W fazie 1, z kolumn od 4 do 8 zostają usunięte cegły tak, że każda z tych kolumn na końcu fazy zawiera po 1 cegle, natomiast kolumna 9 nadal jest pusta. Kolumny od 0 do 3, które znajdują się poza przedziałem usuwania, pozostają nietknięte.



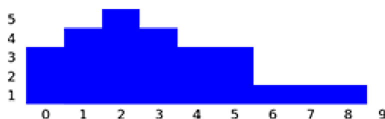
W fazie 2 nic się nie zmienia, ponieważ każda z kolumn od 3 do 6 zawiera nie więcej niż 5 cegieł.



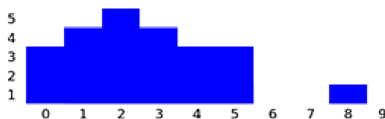
Po fazie 3 liczba cegieł w każdej z kolumn 0, 4 i 5 wzrasta do 3.



Po fazie 4 kolumna 2 zawiera 5 cegieł.



W fazie 5 z kolumn 6 i 7 zostaną usunięte wszystkie cegły.



Zadanie

Mając dany opis k faz, wyznacz liczbę cegieł w poszczególnych kolumnach po wykonaniu wszystkich faz. Napisz funkcję `buildWall(n, k, op, left, right, height, finalHeight)`*:*

- `n`*: liczba kolumn*
- `k`*: liczba faz*
- `op`*: tablica rozmiaru k;* `op[i]` *określa typ fazy i: 1 dla dodawania i 2 dla usuwania, dla* $0 \leqslant i \leqslant k-1$.
- `left` *oraz* `right`*: tablice rozmiaru k; ciąg kolumn rozważany w fazie i rozpoczyna się kolumną* `left[i]` *i kończy się kolumną* `right[i]` *(i zawiera oba końce:* `left[i]` *oraz* `right[i]`*), dla* $0 \leqslant i \leqslant k-1$. *Zawsze będzie zachodzić* `left[i]` $\leqslant$ `right[i]`.
- `height`*: tablica rozmiaru k;* `height[i]` *określa parametr wysokości w fazie i, dla* $0 \leqslant i \leqslant k-1$.
- `finalHeight`*: tablica rozmiaru n; obliczona przez Twój program ostateczna wysokość kolumny i powinna znaleźć się w polu* `finalHeight[i]`*, dla* $0 \leqslant i \leqslant n-1$.

Podzadania

We wszystkich podzadaniach parametry wysokości w każdej fazie są nieujemnymi liczbami całkowitymi nie większymi niż 100 000.

podzadanie	liczba punktów	n	k	dodatkowe ograniczenia
1	8	$1 \leqslant n \leqslant 10\ 000$	$1 \leqslant k \leqslant 5000$	brak
2	24	$1 \leqslant n \leqslant 100\ 000$	$1 \leqslant k \leqslant 500\ 000$	wszystkie fazy dodawania występują przed wszystkimi fazami usuwania
3	29	$1 \leqslant n \leqslant 100\ 000$	$1 \leqslant k \leqslant 500\ 000$	brak
4	39	$1 \leqslant n \leqslant 2\ 000\ 000$	$1 \leqslant k \leqslant 500\ 000$	brak

Implementacja

Powinieneś zgłosić dokładnie jeden plik o nazwie `wall.c`*,* `wall.cpp` *lub* `wall.pas`*. W pliku powinna znaleźć się implementacja funkcji podanej powyżej o następującej sygnaturze. W przypadku programu w C/C++ powinieneś także załączyć plik nagłówkowy* `wall.h`*.*

Programy w C/C++

```
void buildWall(int n, int k, int op[], int left[], int right[],
int height[], int finalHeight[]);
```

Programy w Pascalu

```
procedure buildWall(n, k : longint; op, left, right, height :
array of longint; var finalHeight : array of longint);
```

Przykładowy program sprawdzający

Przykładowy program sprawdzający wczytuje dane w następującym formacie:

- *wiersz 1:* `n, k`
- *wiersz* $2 + i$ $(0 \leqslant i \leqslant k - 1)$*:* `op[i]`, `left[i]`, `right[i]`, `height[i]`.

Ograniczenia

Limit czasu: *3 s*

Dostępna pamięć: *256 MB*

Krzysztof Diks, Jakub Radoszewski
Tłumaczenie

IOI 2014, dzień drugi, 17.07.2014

Gondole

Gondole Mao-Kong należą do najpopularniejszych atrakcji Tajpeju. Ten system gondoli składa się z zamkniętej w pierścień szyny, pojedynczej stacji oraz n gondoli, ponumerowanych kolejno od 1 do n, które poruszają się w kółko po szynie w ustalonym kierunku. Po tym, jak gondola i przejedzie przez stację, kolejną gondolą na stacji jest gondola $i+1$, jeśli $i < n$, oraz gondola 1, jeśli $i = n$.

Gondole mogą się jednak psuć. Szczęśliwie dostępna jest nieograniczona liczba zapasowych gondoli, ponumerowanych kolejno liczbami $n+1$, $n+2$, itd. Gdy gondola zepsuje się, zostaje ona zastąpiona (dokładnie w tym samym miejscu szyny) przez pierwszą dostępną gondolę zapasową, tj. gondolę zapasową o najmniejszym dostępnym numerze. Przykładowo, jeśli w systemie jest pięć gondoli i gondola 1 zepsuje się, zostanie ona zastąpiona gondolą 6.

Uwielbiasz spędzać czas, stojąc na stacji i oglądając kolejno przejeżdżające przez nią gondole. **Ciągiem gondolowym** *nazywamy ciąg n numerów gondoli, które przejeżdżają kolejno przez stację. Mogło się zdarzyć, że przed Twoim przyjazdem niektóre gondole zepsuły się (i zostały zastąpione przez gondole zapasowe), jednak żadna gondola nie psuje się, podczas gdy przebywasz na stacji i obserwujesz gondole.*

Zauważ, że w zależności od tego, która gondola jako pierwsza wjeżdża na stację, ta sama kolejność gondoli na szynie może dać różne ciągi gondolowe. Dla przykładu, jeśli żadna gondola jeszcze się nie zepsuła, zarówno (2, 3, 4, 5, 1), jak i (4, 5, 1, 2, 3) są możliwymi ciągami gondolowymi, natomiast (4, 3, 2, 5, 1) nie może być takim ciągiem (ponieważ gondole występują w złej kolejności).

Jeśli teraz zepsuje się gondola 1, jako ciąg gondolowy możemy zaobserwować m.in. ciąg (4, 5, 6, 2, 3). Jeśli jako kolejna zepsuje się gondola 4, zostanie ona zastąpiona przez gondolę 7, co może prowadzić do ciągu gondolowego (6, 2, 3, 7, 5). Jeśli dalej zepsuje się gondola 7, zastąpi ją gondola 8, a końcowym ciągiem gondolowym może być np. (3, 8, 5, 6, 2).

zepsuta gondola	*nowa gondola*	*możliwy ciąg gondolowy*
1	6	(4, 5, 6, 2, 3)
4	7	(6, 2, 3, 7, 5)
7	8	(3, 8, 5, 6, 2)

Ciągiem zastąpień *nazywamy ciąg numerów kolejnych gondol, które uległy zepsuciu. Ciągiem zastąpień dla powyższego przykładu będzie więc (1, 4, 7). Powiemy, że ciąg zastąpień r* **prowadzi do** *ciągu gondolowego g, jeśli po tym, jak zepsują się kolejno gondole wymienione w ciągu r, możemy zaobserwować ciąg gondolowy g.*

Sprawdzanie poprawności ciągu gondolowego

W trzech pierwszych podzadaniach Twój program powinien sprawdzić, czy dany ciąg może być ciągiem gondolowym. W poniższej tabeli znajdują się przykłady ciągów gondolowych oraz takich, które nie mogą być ciągami gondolowymi. Napisz funkcję `valid(n, inputSeq)`*:*

- `n`*: długość ciągu*

- `inputSeq`*: tablica rozmiaru n;* `inputSeq[i]` *to i-ty element ciągu, dla $0 \leqslant i \leqslant n-1$.*
- *Wynikiem funkcji powinno być 1, jeśli dany ciąg jest ciągiem gondolowym, a 0 w przeciwnym przypadku.*

Podzadania 1, 2, 3

podzadanie	*liczba punktów*	n	`inputSeq`
1	5	$n \leqslant 100$	*każda liczba od 1 do n występuje dokładnie raz*
2	5	$n \leqslant 100\,000$	$1 \leqslant$ `inputSeq[i]` $\leqslant n$
3	10	$n \leqslant 100\,000$	$1 \leqslant$ `inputSeq[i]` $\leqslant 250\,000$

Przykłady

podzadanie	`inputSeq`	*wynik funkcji*	*uwagi*
1	(1, 2, 3, 4, 5, 6, 7)	1	
1	(3, 4, 5, 6, 1, 2)	1	
1	(1, 5, 3, 4, 2, 7, 6)	0	*1 nie może wystąpić tuż przed 5*
1	(4, 3, 2, 1)	0	*4 nie może wystąpić tuż przed 3*
2	(1, 2, 3, 4, 5, 6, 5)	0	*dwie gondole o numerze 5*
3	(2, 3, 4, 9, 6, 7, 1)	1	*ciąg zastąpień to (5, 8)*
3	(10, 4, 3, 11, 12)	0	*4 nie może wystąpić tuż przed 3*

Wyznaczanie ciągu zastąpień

W trzech kolejnych podzadaniach Twój program powinien skonstruować przykładowy ciąg zastąpień, który prowadzi do zadanego ciągu gondolowego. Jeśli istnieje wiele możliwych ciągów zastąpień, możesz wybrać dowolny z nich. Napisz funkcję `replacement(n, gondolaSeq, replacementSeq)`*:*

- `n`*: długość ciągu gondolowego*
- `gondolaSeq`*: tablica rozmiaru n; jest zagwarantowane, że* `gondolaSeq` *to ciąg gondolowy, a* `gondolaSeq[i]` *to i-ty element tego ciągu, dla $0 \leqslant i \leqslant n-1$.*
- *Wynikiem funkcji powinno być l, to jest długość ciągu zastąpień.*
- `replacementSeq`*: tablica wystarczająco duża na to, by przechować cały ciąg zastąpień; funkcja powinna umieścić i-ty element ciągu zastąpień w polu* `replacementSeq[i]`*, dla $0 \leqslant i \leqslant l-1$.*

Podzadania 4, 5, 6

podzadanie	*liczba punktów*	n	`gondolaSeq`
4	5	$n \leqslant 100$	$1 \leqslant$ `gondolaSeq[i]` $\leqslant n+1$
5	10	$n \leqslant 1000$	$1 \leqslant$ `gondolaSeq[i]` $\leqslant 5000$
6	20	$n \leqslant 100\,000$	$1 \leqslant$ `gondolaSeq[i]` $\leqslant 250\,000$

Przykłady

podzadanie	`gondolaSeq`	*wynik funkcji*	`replacementSeq`
4	*(3, 1, 4)*	*1*	*(2)*
4	*(5, 1, 2, 3, 4)*	*0*	*( )*
5	*(2, 3, 4, 9, 6, 7, 1)*	*2*	*(5, 8)*

Zliczanie ciągów zastąpień

W czterech ostatnich podzadaniach Twój program powinien wyznaczyć liczbę różnych ciągów zastąpień, które prowadzą do podanego ciągu (który może być lub nie być ciągiem gondolowym), modulo 1 000 000 009. Napisz funkcję `countReplacement(n, inputSeq)`*:*

- `n`*: długość ciągu*
- `inputSeq`*: tablica rozmiaru n;* `inputSeq[i]` *to i-ty element ciągu, dla* $0 \leqslant i \leqslant n-1$*.*
- *Jeśli dany ciąg jest ciągiem gondolowym, funkcja powinna wyznaczyć liczbę ciągów zastąpień, które prowadzą do tego ciągu gondolowego (która może być bardzo duża), i podać w wyniku tę liczbę modulo 1 000 000 009. Jeśli dany ciąg nie jest ciągiem gondolowym, wynikiem funkcji powinno być 0. Jeśli dany ciąg jest ciągiem gondolowym odpowiadającym sytuacji, że żadna gondola nie zepsuła się, wynikiem funkcji powinno być 1.*

Podzadania 7, 8, 9, 10

podzadanie	*liczba punktów*	*n*	`inputSeq`
7	*5*	$4 \leqslant n \leqslant 50$	$1 \leqslant$ `inputSeq[i]` $\leqslant n+3$
8	*15*	$4 \leqslant n \leqslant 50$	$1 \leqslant$ `inputSeq[i]` $\leqslant 100$ *i co najmniej* $n-3$ *spośród gondoli* $1,\ldots,n$ *nie zepsuło się.*
9	*15*	$n \leqslant 100\,000$	$1 \leqslant$ `inputSeq[i]` $\leqslant 250\,000$
10	*10*	$n \leqslant 100\,000$	$1 \leqslant$ `inputSeq[i]` $\leqslant 1\,000\,000\,000$

Przykłady

podzadanie	`inputSeq`	*wynik funkcji*	*ciąg zastąpień*
7	*(1, 2, 7, 6)*	*2*	*(3, 4, 5) lub (4, 5, 3)*
8	*(2, 3, 4, 12, 6, 7, 1)*	*1*	*(5, 8, 9, 10, 11)*
9	*(4, 7, 4, 7)*	*0*	`inputSeq` *nie jest ciągiem gondolowym*
10	*(3, 4)*	*2*	*(1, 2) lub (2, 1)*

Implementacja

Powinieneś zgłosić dokładnie jeden plik o nazwie `gondola.c`*,* `gondola.cpp` *lub* `gondola.pas`*. W tym pliku powinna znaleźć się implementacja wszystkich trzech funkcji opisanych powyżej*

(nawet jeśli planujesz rozwiązać tylko niektóre podzadania), o następujących sygnaturach. W przypadku programu w C/C++ powinieneś także załączyć plik nagłówkowy `gondola.h`*.*

Programy w C/C++

```
int valid(int n, int inputSeq[]);
int replacement(int n, int gondolaSeq[], int replacementSeq[]);
int countReplacement(int n, int inputSeq[]);
```

Programy w Pascalu

```
function valid(n: longint; inputSeq: array of longint): integer;
function replacement(n: longint; gondolaSeq: array of longint;
var replacementSeq: array of longint): longint;
function countReplacement(n: longint; inputSeq: array of longint):
longint;
```

Przykładowy program sprawdzający

Przykładowy program sprawdzający wczytuje dane w następującym formacie:

- *wiersz 1:* `T`*, numer podzadania, które Twój program ma rozwiązać ($1 \leqslant T \leqslant 10$)*
- *wiersz 2:* `n`*, długość ciągu wejściowego*
- *wiersz 3: Jeśli* `T` *jest równe 4, 5 lub 6, wiersz ten zawiera* `gondolaSeq[0]`*, ...,* `gondolaSeq[n-1]`*. W przeciwnym razie wiersz ten zawiera* `inputSeq[0]`*, ...,* `inputSeq[n-1]`*.*

Ograniczenia

Limit czasu: *1 s*

Dostępna pamięć: *256 MB*

Krzysztof Diks, Jakub Radoszewski
Tłumaczenie

IOI 2014, dzień drugi, 17.07.2014

Przyjaciel

Budujemy sieć społecznościową złożoną z n osób ponumerowanych liczbami $0, \ldots, n-1$. Gdy do sieci dodawane są kolejne osoby, pewne pary osób w sieci stają się przyjaciółmi. Jeśli osoba x staje się przyjacielem osoby y, to osoba y również staje się przyjacielem osoby x.

Osoby są dodawane do sieci w n fazach, ponumerowanych od 0 do $n-1$. Osoba i jest dodawana do sieci w fazie i. W fazie 0 w sieci jest umieszczana tylko osoba 0. W każdej z następnych $n-1$ faz nowa osoba jest zapraszana do sieci przez **gospodarza***, którym może być dowolna osoba znajdująca się już w sieci. Osoba będąca gospodarzem w fazie i (dla $0 < i < n$) może dodać do sieci osobę i zgodnie z jednym z trzech następujących protokołów:*

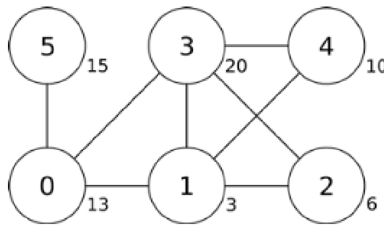
- **IAmYourFriend** – *czyni osobę i przyjacielem gospodarza (i nikogo więcej).*
- **MyFriendsAreYourFriends** – *czyni osobę i przyjacielem* **każdej** *osoby będącej aktualnym przyjacielem gospodarza. Zwróć uwagę, że w tym protokole osoba i* **nie** *zostaje przyjacielem gospodarza.*
- **WeAreYourFriends** – *czyni osobę i przyjacielem gospodarza oraz* **wszystkich** *osób będących jego aktualnymi przyjaciółmi.*

Po zbudowaniu sieci chcielibyśmy wybrać **próbkę** *osób z sieci do badania jej własności. Ponieważ przyjaciele mają zazwyczaj podobne zainteresowania, w próbce nie może znaleźć się żadna para przyjaciół. Z każdą osobą związana jest jej* **adekwatność** *dla badań, wyrażona dodatnią liczbą całkowitą. Naszym celem jest znalezienie próbki o maksymalnej sumarycznej adekwatności.*

Przykład

faza	*gospodarz*	*protokół*	*dodane pary przyjaciół*
1	0	*IAmYourFriend*	(1, 0)
2	0	*MyFriendsAreYourFriends*	(2, 1)
3	1	*WeAreYourFriends*	(3, 1), (3, 0), (3, 2)
4	2	*MyFriendsAreYourFriends*	(4, 1), (4, 3)
5	0	*IAmYourFriend*	(5, 0)

Początkowo w sieci jest tylko osoba 0. Gospodarz fazy 1 (osoba 0) zaprasza nową osobę 1 przy użyciu protokołu IAmYourFriend, co oznacza, że osoby 0 i 1 stają się przyjaciółmi. Gospodarz fazy 2 (ponownie osoba 0) zaprasza osobę 2 za pomocą MyFriendsAreYourFriends, w wyniku czego osoba 1 (jedyny przyjaciel gospodarza) staje się jedynym przyjacielem osoby 2. Gospodarz fazy 3 (osoba 1) dodaje do sieci osobę 3, wykonując WeAreYourFriends, czyniąc osobę 3 przyjacielem osoby 1 (gospodarza) oraz osób 0 i 2 (przyjaciół gospodarza). W tabeli powyżej zaprezentowano też fazy 4 i 5. Końcowa sieć jest przedstawiona na rysunku poniżej, na którym liczby w kółkach to etykiety osób, natomiast liczby obok kółek to ich adekwatności. Próbka składająca się z osób 3 i 5 ma sumaryczną adekwatność równą $20 + 15 = 35$, która jest największą możliwą sumaryczną adekwatnością próbki.



Zadanie

Mając dane opisy faz oraz adekwatności poszczególnych osób, znajdź próbkę o maksymalnej sumarycznej adekwatności. Twoje zadanie polega na napisaniu funkcji `findSample(n, confidence, host, protocol)`*:*

- `n`*: liczba osób*
- `confidence`*: tablica rozmiaru n;* `confidence[i]` *podaje adekwatność osoby i.*
- `host`*: tablica rozmiaru n;* `host[i]` *podaje gospodarza w fazie i.*
- `protocol`*: tablica rozmiaru n;* `protocol[i]` *podaje kod protokołu używanego w fazie i* $(0 < i < n)$*: 0 oznacza IAmYourFriend, 1 oznacza MyFriendsAreYourFriends, natomiast 2 oznacza WeAreYourFriends.*
- *Ponieważ faza 0 nie ma gospodarza,* `host[0]` *oraz* `protocol[0]` *nie są określone i nie powinny być wykorzystywane w Twoim programie.*
- *Wynikiem funkcji powinna być największa możliwa sumaryczna adekwatność próbki.*

Podzadania

W niektórych podzadaniach wykorzystuje się tylko część z protokołów, zgodnie z tabelą poniżej.

podzadanie	liczba punktów	n	adekwatność ($a \geqslant 1$)	używane protokoły
1	11	$2 \leqslant n \leqslant 10$	$a \leqslant 1\,000\,000$	wszystkie trzy protokoły
2	8	$2 \leqslant n \leqslant 1000$	$a \leqslant 1\,000\,000$	MyFriendsAreYourFriends
3	8	$2 \leqslant n \leqslant 1000$	$a \leqslant 1\,000\,000$	WeAreYourFriends
4	19	$2 \leqslant n \leqslant 1000$	$a \leqslant 1\,000\,000$	IAmYourFriend
5	23	$2 \leqslant n \leqslant 1000$	wszystkie adekwatności są równe 1	MyFriendsAreYourFriends i IAmYourFriend
6	31	$2 \leqslant n \leqslant 100\,000$	$a \leqslant 10\,000$	wszystkie trzy protokoły

Implementacja

Powinieneś zgłosić dokładnie jeden plik o nazwie `friend.c`, `friend.cpp` *lub* `friend.pas`*. W pliku powinna znaleźć się implementacja funkcji opisanej powyżej, o następującej sygnaturze. W przypadku programu w C/C++ powinieneś także załączyć plik nagłówkowy* `friend.h`*.*

Programy w C/C++

```
int findSample(int n, int confidence[], int host[], int protocol[]);
```

Programy w Pascalu

```
function findSample(n: longint, confidence: array of longint,
host: array of longint; protocol: array of longint): longint;
```

Przykładowy program sprawdzający

Przykładowy program sprawdzający wczytuje dane w następującym formacie:

- *wiersz 1:* `n`
- *wiersz 2:* `confidence[0]`, ..., `confidence[n-1]`
- *wiersz 3:* `host[1]`, `protocol[1]`, `host[2]`, `protocol[2]`, ..., `host[n-1]`, `protocol[n-1]`.

Przykładowy program sprawdzający wypisze na wyjście wynik funkcji `findSample`.

Ograniczenia

Limit czasu: *1 s*

Dostępna pamięć: *16 MB*

Krzysztof Diks, Jakub Radoszewski
Tłumaczenie

IOI 2014, dzień drugi, 17.07.2014

Wakacje

Jian-Jia planuje spędzić swoje następne wakacje na Tajwanie. Podczas wakacji będzie podróżował od miasta do miasta, starając się zwiedzić jak najwięcej atrakcji w różnych miastach.

Na Tajwanie znajduje się n miast. Są one usytuowane wzdłuż jednej autostrady. Miasta są ponumerowane kolejno od 0 do $n-1$. Sąsiednimi miastami dla miasta i, gdzie $0 < i < n-1$, są miasta $i-1$ oraz $i+1$. Jedynym sąsiednim miastem dla miasta 0 jest miasto 1, a jedynym sąsiednim miastem dla miasta $n-1$ jest miasto $n-2$.

W każdym mieście znajduje się pewna liczba atrakcji. Jian-Jia ma do dyspozycji d dni wakacji i chce w tym czasie zobaczyć możliwie najwięcej atrakcji. Już zdecydował, od którego miasta rozpocznie zwiedzanie Tajwanu. Każdego dnia podczas swoich wakacji Jian-Jia może albo przemieścić się do sąsiedniego miasta, albo zobaczyć wszystkie atrakcje w mieście, w którym się aktualnie znajduje. Jednego dnia nie może jednocześnie zwiedzać miasta i przemieszczać się między miastami. Jian-Jia **nigdy nie ogląda atrakcji w tym samym mieście więcej niż raz**, *nawet jeśli znajdzie się tam wiele razy. Pomóż Jian-Jia zaplanować wakacje tak, żeby zobaczył możliwie najwięcej różnych atrakcji.*

Przykład

Załóżmy, że Jian-Jia ma 7 dni wakacji, do zwiedzenia jest 5 miast (wymienionych w tabeli poniżej), a zwiedzanie rozpoczyna się od miasta o numerze 2. Pierwszego dnia Jian-Jia odwiedza 20 atrakcji w mieście 2. Drugiego dnia przemieszcza się z miasta 2 do miasta 3, a trzeciego dnia zwiedza w tym mieście 30 atrakcji. Następne trzy dni spędza, podróżując z miasta 3 do miasta 0. Siódmego dnia zwiedza 10 atrakcji w mieście 0. Łączna liczba atrakcji odwiedzonych przez Jian-Jia wynosi $20 + 30 + 10 = 60$, co jest największą możliwą liczbą atrakcji, jakie Jian-Jia może odwiedzić w ciągu 7 dni, rozpoczynając zwiedzanie od miasta 2.

miasto	*liczba atrakcji*
0	*10*
1	*2*
2	*20*
3	*30*
4	*1*

dzień	*akcja*
1	*zwiedza atrakcje w mieście 2*
2	*przemieszcza się z miasta 2 do miasta 3*
3	*zwiedza atrakcje w mieście 3*
4	*przemieszcza się z miasta 3 do miasta 2*
5	*przemieszcza się z miasta 2 do miasta 1*
6	*przemieszcza się z miasta 1 do miasta 0*
7	*zwiedza atrakcje w mieście 0*

Zadanie

Napisz funkcję `findMaxAttraction(n, start, d, attraction)`*, która obliczy maksymalną liczbę atrakcji możliwą do odwiedzenia przez Jian-Jia.*

- `n`*: liczba miast*
- `start`*: indeks miasta, z którego rozpoczyna się zwiedzanie*
- `d`*: liczba dni*
- `attraction`*: tablica rozmiaru n;* `attraction[i]` *jest liczbą atrakcji w mieście i, dla* $0 \leqslant i \leqslant n-1$.
- *Wynikiem funkcji powinna być maksymalna liczba atrakcji możliwych do odwiedzenia przez Jian-Jia.*

Podzadania

We wszystkich podzadaniach zachodzi $0 \leqslant d \leqslant 2n + \lfloor n/2 \rfloor$. *W każdym mieście znajduje się nieujemna liczba atrakcji.*

podzadanie	liczba punktów	n	max liczba atrakcji w jednym mieście	miasto startowe
1	7	$2 \leqslant n \leqslant 20$	1 000 000 000	dowolne
2	23	$2 \leqslant n \leqslant 100\,000$	100	miasto 0
3	17	$2 \leqslant n \leqslant 3000$	1 000 000 000	dowolne
4	53	$2 \leqslant n \leqslant 100\,000$	1 000 000 000	dowolne

Implementacja

Powinieneś zgłosić dokładnie jeden plik o nazwie `holiday.c`, `holiday.cpp` *lub* `holiday.pas`. *W pliku powinna znaleźć się implementacja funkcji opisanej powyżej, o następującej sygnaturze. W przypadku programu w C/C++ powinieneś także załączyć plik nagłówkowy* `holiday.h`.

Zauważ, że wynik może być duży, a do przechowywania wyniku funkcji `findMaxAttraction` *jest używany 64-bitowy typ całkowity.*

Programy w C/C++

```
long long int findMaxAttraction(int n, int start, int d,
int attraction[]);
```

Programy w Pascalu

```
function findMaxAttraction(n, start, d : longint;
attraction : array of longint): int64;
```

Przykładowy program sprawdzający

Przykładowy program sprawdzający wczytuje dane w następującym formacie:

- *wiersz 1:* `n`, `start`, `d`
- *wiersz 2:* `attraction[0]`, ..., `attraction[n-1]`.

Przykładowy program sprawdzający wypisze na wyjście wynik funkcji `findMaxAttraction`.

Ograniczenia

Limit czasu: *5 s*

Dostępna pamięć: *64 MB*

XX Bałtycka Olimpiada Informatyczna,

Połąga, Litwa 2014

Dawid Dąbrowski, Jakub Radoszewski
Tłumaczenie

BOI 2014, dzień pierwszy, 27.04.2014

Ciąg

Adrian napisał na tablicy ciąg K kolejnych, dodatnich liczb całkowitych, zaczynając od liczby N. Pod jego nieobecność Bartek zmazał w każdej liczbie wszystkie cyfry oprócz jednej. W ten sposób utworzył ciąg K cyfr.

Zadanie

Mając dany ciąg Bartka, znajdź najmniejszą wartość N, od której mógł zaczynać się ciąg Adriana.

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę całkowitą K – długość ciągu na tablicy. Drugi wiersz zawiera K liczb całkowitych $B_1, B_2, \ldots, B_K$ ($0 \leqslant B_i \leqslant 9$) – ciąg Bartka, podany w kolejności, w jakiej jest zapisany na tablicy.

Wyjście

W jedynym wierszu wyjścia powinna znaleźć się najmniejsza liczba N, od której mógł zaczynać się ciąg Adriana.

Przykład

Dla danych wejściowych:
```
6
7 8 9 5 1 2
```
poprawnym wynikiem jest:
```
47
```

$N = 47$ odpowiada ciągowi Adriana 47 48 49 50 51 52, z którego mógł powstać podany ciąg Bartka. Jako że nie mogło tak być dla żadnego mniejszego N, poprawną odpowiedzią jest właśnie 47.

Ocenianie

Podzadanie 1 (9 punktów): $1 \leqslant K \leqslant 1\,000$, *poprawna odpowiedź nie przekracza 1 000*

Podzadanie 2 (33 punkty): $1 \leqslant K \leqslant 1\,000$

Podzadanie 3 (25 punktów): $1 \leqslant K \leqslant 100\,000$, *wszystkie elementy w ciągu Bartka są równe*

Podzadanie 4 (33 punkty): $1 \leqslant K \leqslant 100\,000$

Ograniczenia

Limit czasu: *1 s*

Dostępna pamięć: *256 MB*

Dawid Dąbrowski, Jakub Radoszewski
Tłumaczenie

BOI 2014, dzień pierwszy, 27.04.2014

Policjant i złodziej

Przestępczość w Bajtogrodzie jest na porządku dziennym. Szczególnie często mają miejsce kradzieże. Jedną z przyczyn tego stanu rzeczy może być fakt, iż w pogoń za złodziejem rusza zazwyczaj tylko jeden policjant znajdujący się akurat w terenie. Pościg w staroświeckim stylu odbywa się wąskimi uliczkami łączącymi skrzyżowania Bajtogrodu, a dzięki dobrej znajomości miasta złodziejowi nierzadko udaje się umknąć policjantowi.

Komenda Stołeczna Policji w Bajtogrodzie (KSPB) organizuje zgrupowanie poświęcone zmniejszeniu skali przestępczości w mieście. Jednym z pomysłów jest wprowadzenie automatycznego systemu planowania tras pościgu za złodziejami. W tym celu KSPB zdobyło już dokładny plan miasta. Teraz poproszono Cię, abyś przygotował program, który korzystając z tych danych, umożliwi efektywne planowanie pościgu.

Pościg policjanta za złodziejem modelujemy następująco:

1. *Policjant wybiera skrzyżowanie, na którym rozpoczyna swój patrol.*
2. *Następnie złodziej wybiera skrzyżowanie, przy którym dokona włamania (wie on, na którym skrzyżowaniu znajduje się policjant). Od tego momentu zakładamy, że policjant i złodziej znają wzajemnie swoje położenia.*
3. *W swoim ruchu policjant przemieszcza się na sąsiednie skrzyżowanie (tzn. skrzyżowanie połączone bezpośrednio uliczką ze skrzyżowaniem, na którym jest obecnie) lub decyduje się czekać (tzn. nie przemieszcza się).*
4. *W swoim ruchu złodziej przemieszcza się na sąsiednie skrzyżowanie. Zauważ, że w przeciwieństwie do policjanta, złodziej nigdy nie czeka w swoim ruchu – na złodzieju czapka gore.*
5. *Policjant i złodziej wykonują ruchy na przemian (począwszy od policjanta), aż do momentu, gdy:*
   (a) *wcześniejsza sytuacja powtórzy się (przez sytuację rozumiemy pozycje obu graczy oraz to, do którego gracza należy najbliższy ruch). Oznacza to, że złodziej może unikać spotkania z policjantem w nieskończoność, więc przyjmujemy, że złodziej uciekł policjantowi; albo*
   (b) *policjant i złodziej spotkają się na tym samym skrzyżowaniu po ruchu któregoś z nich. Wówczas policjant łapie złodzieja.*

Zadanie

Napisz program, który mając dany plan miasta, stwierdzi, czy policjant może złapać złodzieja, a jeśli tak, przeprowadzi pościg w imieniu policjanta.

Twój program powinien założyć, że złodziej porusza się w sposób optymalny.

Implementacja

Powinieneś zaimplementować dwie funkcje:

- `start(N, A)` *o następujących parametrach:*
  - N – *liczba skrzyżowań (skrzyżowania są ponumerowane od 0 do* $N-1$*)*
  - A – *dwuwymiarowa tablica opisująca uliczki; dla* $0 \leqslant i, j \leqslant N-1$,

$$A[i,j] = \begin{cases} \texttt{false} & \textit{jeśli skrzyżowania } i \textit{ oraz } j \textit{ nie są połączone uliczką} \\ \texttt{true} & \textit{jeśli skrzyżowania } i \textit{ oraz } j \textit{ są połączone uliczką.} \end{cases}$$

  Wszystkie uliczki są dwukierunkowe (tzn. $A[i,j] = A[j,i]$ *dla wszystkich* i *oraz* j*) i każda uliczka łączy dwa różne skrzyżowania (tzn.* $A[i,i]$ *będzie równe* `false` *dla wszystkich* i*). Możesz ponadto założyć, że za pomocą systemu uliczek można przedostać się z dowolnego skrzyżowania na dowolne inne skrzyżowanie.*

  Jeśli w tak opisanym mieście policjant może złapać złodzieja, wynikiem funkcji `start` *powinien być numer skrzyżowania, na którym policjant powinien rozpocząć swój patrol. W przeciwnym razie wynikiem funkcji powinno być* -1.

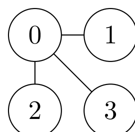
- `nextMove(R)` *przyjmującą jako parametr liczbę* R *oznaczającą numer skrzyżowania, przy którym znajduje się złodziej, i zwracającą numer skrzyżowania, przy którym policjant znajdzie się po wykonaniu swojego ruchu.*

Funkcja `start` *zostanie wywołana dokładnie raz, przed wszystkimi wywołaniami funkcji* `nextMove`*. Jeśli wynikiem funkcji* `start` *będzie* -1*, funkcja* `nextMove` *nie będzie wywoływana. W przeciwnym razie, funkcja* `nextMove` *będzie wywoływana w kółko aż do końca pościgu. Program zakończy się, gdy spełniony zostanie jeden z poniższych warunków:*

- *funkcja* `nextMove` *zwróci niepoprawny ruch;*
- *wcześniejsza sytuacja powtórzy się;*
- *złodziej zostanie złapany.*

Przykład

Przyjrzyjmy się przykładowi opisanemu przez poniższy rysunek. W tym przykładzie każde skrzyżowanie jest dobrą pozycją początkową dla policjanta. Jeśli policjant rozpocznie patrol przy skrzyżowaniu numer 0, w swoim pierwszym ruchu może czekać – wówczas złodziej sam na niego wpadnie. Jeśli zaś policjant rozpocznie patrol przy jakimkolwiek innym skrzyżowaniu, może poczekać, aż złodziej znajdzie się przy skrzyżowaniu numer 0, i wówczas przejść na to skrzyżowanie.



Oto jedno z możliwych wykonań programu dla tego przykładu:

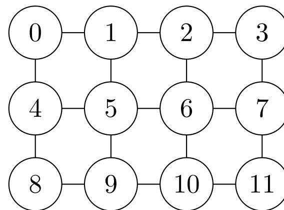
Wywołanie funkcji	**Wynik**
`start(4, [[0, 1, 1, 1], [1, 0, 0, 0], [1, 0, 0, 0], [1, 0, 0, 0]])`	3
`nextMove(1)`	3
`nextMove(0)`	0

Uwaga: w powyższym wywołaniu funkcji `start` *liczba* `0` *oznacza* `false`*, a liczba* `1` *oznacza* `true`*.*

Ocenianie

Podzadanie 1 (16 punktów): *$2 \leqslant N \leqslant 500$. Każda para skrzyżowań jest połączona za pomocą dokładnie jednej ścieżki złożonej z uliczek.*

Podzadanie 2 (14 punktów): *$2 \leqslant N \leqslant 500$. Sieć skrzyżowań i uliczek tworzy kratkę. Kratka składa się z co najmniej dwóch wierszy i co najmniej dwóch kolumn, a numeracja skrzyżowań odpowiada schematowi przedstawionemu na rysunku poniżej.*



Podzadanie 3 (30 punktów): *$2 \leqslant N \leqslant 100$*

Podzadanie 4 (40 punktów): *$2 \leqslant N \leqslant 500$*

Twoje rozwiązanie musi spełniać dwa wymagania:

1. *poprawnie stwierdzać, czy policjant może złapać złodzieja;*
2. *jeśli to jest możliwe, skutecznie łapać złodzieja, wykonując ruchy w imieniu policjanta.*

W przypadku podzadań 1 i 2, Twoje rozwiązanie musi spełnić oba te wymagania, aby uzyskać jakiekolwiek punkty. Natomiast w podzadaniach 3 i 4, rozwiązania, które spełniają tylko pierwsze wymaganie, uzyskają 30% punktów za odpowiednie podzadanie. Jeśli w swoim rozwiązaniu chciałbyś uzyskać jedynie tę częściową punktację, możesz zakończyć swój program, wykonując niepoprawny ruch (np. zwrócić -1 w funkcji `nextMove`*).*

Pamiętaj, że standardowe wymagania (zmieszczenie się w limitach czasu i pamięci oraz brak błędów wykonania) i tak muszą być spełnione przez Twój program, aby miał szansę uzyskać jakieś punkty.

Ograniczenia

Limit czasu: *1,5 s*

Dostępna pamięć: *256 MB*

Eksperymenty

Przykładowy program oceniający znajdujący się na Twoim komputerze wczytuje dane ze standardowego wejścia. W pierwszym wierszu wejścia powinna znaleźć się liczba całkowita N – liczba skrzyżowań. W kolejnych N wierszach powinna znaleźć się macierz sąsiedztwa A. Każdy z tych wierszy powinien zawierać N liczb, z których każda to 0 albo 1. Macierz musi być symetryczna, a na jej głównej przekątnej muszą być same zera.

Kolejny wiersz powinien zawierać liczbę 1, jeśli policjant może złapać złodzieja, a 0 w przeciwnym przypadku.

Jeśli policjant może złapać złodzieja, w kolejnych N wierszach powinna zostać opisana strategia złodzieja. Każdy z tych wierszy powinien zawierać $N + 1$ liczb całkowitych z zakresu od 0 do $N - 1$. Liczba znajdująca się w wierszu r i kolumnie c, dla $c < N$, odpowiada sytuacji, w której ruch należy do złodzieja, policjant znajduje się przy skrzyżowaniu numer r, a złodziej przy skrzyżowaniu numer c. Liczba ta powinna oznaczać numer skrzyżowania, na które w tej sytuacji przemieszcza się złodziej. Liczby znajdujące się na głównej przekątnej nie są istotne, ponieważ odpowiadają one sytuacji, w której policjant i złodziej znajdują się na tym samym skrzyżowaniu. Ostatnia liczba w wierszu numer r opisuje numer skrzyżowania, przy którym złodziej dokonuje włamania, jeśli policjant rozpoczął patrol przy skrzyżowaniu numer r.

Poniżej znajduje się przykładowe wejście do przykładowego programu oceniającego opisujące trzy skrzyżowania połączone każde z każdym:

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

Natomiast poniższe wejście odpowiada przykładowi podanemu w treści zadania:

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

Dawid Dąbrowski, Jakub Radoszewski
Tłumaczenie

BOI 2014, dzień pierwszy, 27.04.2014

Troje przyjaciół

Troje przyjaciół gra w następującą grę. Pierwszy z nich układa pewien napis S. Następnie drugi tworzy napis T złożony z dwóch identycznych kopii napisu S. Na koniec, trzeci z przyjaciół dokłada jedną literę na początku, na końcu, bądź w środku napisu T, tworząc napis U.

Zadanie

Masz dany końcowy napis U. Zrekonstruuj początkowy napis S.

Wejście

W pierwszym wierszu wejścia znajduje się długość N końcowego napisu U. W drugim wierszu znajduje się napis U, składający się z wielkich liter alfabetu angielskiego (A, B, C, ..., Z).

Wyjście

Twój program powinien wypisać początkowy napis S. Są jednak dwa wyjątki:

1. *Jeśli nie jest możliwe, by podany napis końcowy U powstał w wyniku opisanej zabawy, program powinien wypisać* `NOT POSSIBLE` *(co oznacza: niemożliwe).*
2. *Jeśli jest więcej niż jeden początkowy napis S, z którego mógł powstać końcowy napis U, program powinien wypisać* `NOT UNIQUE` *(co oznacza: niejednoznaczny).*

Przykłady

Dla danych wejściowych:
```
7
ABXCABC
```
poprawnym wynikiem jest:
```
ABC
```

Dla danych wejściowych:
```
6
ABCDEF
```
poprawnym wynikiem jest:
```
NOT POSSIBLE
```

Dla danych wejściowych:
```
9
ABABABABA
```
poprawnym wynikiem jest:
```
NOT UNIQUE
```

Ocenianie

Podzadanie 1 (35 punktów): $2 \leqslant N \leqslant 2\,001$

Podzadanie 2 (65 punktów): $2 \leqslant N \leqslant 2\,000\,001$

Ograniczenia

Limit czasu: *0,5 s*

Dostępna pamięć: *256 MB*

Dawid Dąbrowski, Jakub Radoszewski
Tłumaczenie

BOI 2014, dzień drugi, 28.04.2014

Demarkacja

Przez długie lata Bajtocja była wyspą, na której w pokoju żyli poddani króla Bajtazara I. Jednak po jego nagłej śmierci dwaj królewscy synowie – bliźniacy Bitoni i Bajtoni – nie mogli dojść do porozumienia, który z nich powinien objąć tron. Postanowili więc podzielić wyspę na dwie prowincje, którymi będą rządzić niezależnie.

Na prostokątnej mapie Bajtocja ma kształt wielokąta o N wierzchołkach, którego każdy bok jest równoległy do jednego z boków mapy, a każde dwa kolejne boki są prostopadłe. Żadne dwa boki nie dotykają się ani nie przecinają, oprócz kolejnych boków, które mają wspólny koniec. Współrzędne wierzchołków wielokąta są liczbami całkowitymi.

Bitoni i Bajtoni chcą podzielić ten wielokąt na dwie figury przystające za pomocą jednego odcinka równoległego do któregoś z boków mapy i zawartego w wielokącie. (Mówimy, że dwie figury są przystające, jeśli jedną z nich można przekształcić w drugą za pomocą symetrii, obrotów oraz przesunięć.) Współrzędne końców odcinka dzielącego muszą być liczbami całkowitymi.

Królewscy synowie poprosili Cię, abyś stwierdził, czy taki podział jest w ogóle możliwy.

Zadanie

Mając dany kształt wyspy, sprawdź, czy można ją podzielić za pomocą poziomego lub pionowego odcinka na dwa przystające wielokąty. Jeśli taki podział istnieje, znajdź dowolny odcinek, który go powoduje.

Wejście

W pierwszym wierszu wejścia znajduje się jedna liczba całkowita N – liczba wierzchołków wielokąta. W i-tym z kolejnych N wierszy znajduje się para liczb całkowitych X_i, Y_i ($0 \leqslant X_i, Y_i \leqslant 10^9$), oznaczających współrzędne i-tego wierzchołka wielokąta.

Wierzchołki są podane w kolejności ich występowania na obwodzie wielokąta, tak więc odcinki $(X_1, Y_1) - (X_2, Y_2)$, $(X_2, Y_2) - (X_3, Y_3)$, $\ldots$, $(X_{N-1}, Y_{N-1}) - (X_N, Y_N)$ oraz $(X_N, Y_N) - (X_1, Y_1)$ są kolejnymi bokami wielokąta. Ponadto, każde dwa kolejne boki są prostopadłe.

Wyjście

Twój program powinien wypisać jeden wiersz. Jeśli jest możliwy podział wyspy na dwa przystające wielokąty za pomocą odcinka poziomego lub pionowego o końcach w punktach o współrzędnych całkowitych, wypisz cztery liczby całkowite x_1, y_1, x_2, y_2 pooddzielane pojedynczymi odstępami oznaczające końce (x_1, y_1) i (x_2, y_2) tego odcinka. Musi zachodzić $x_1 = x_2$ lub $y_1 = y_2$. Odcinek musi w całości zawierać się wewnątrz danego wielokąta i tylko jego końce powinny dotykać brzegu wielokąta.

Jeśli żądany podział nie jest możliwy, wypisz jedno słowo `NO`*.*

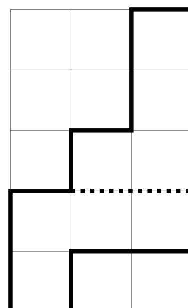
Przykłady

Dla danych wejściowych:

```
10
0 0
1 0
1 1
3 1
3 5
2 5
2 3
1 3
1 2
0 2
```

poprawnym wynikiem jest:

`1 2 3 2`



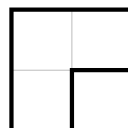
Poprawnym rozwiązaniem jest także `3 2 1 2`.

Dla danych wejściowych:

```
6
0 0
1 0
1 1
2 1
2 2
0 2
```

poprawnym wynikiem jest:

`NO`



W tym przypadku nie da się podzielić wyspy na dwa przystające wielokąty.

Ocenianie

Podzadanie 1 (12 punktów): $4 \leqslant N \leqslant 100\,000$. *Każda prosta, która dzieli wielokąt na części, dzieli go na dokładnie dwie części.*

Podzadanie 2 (15 punktów): $4 \leqslant N \leqslant 200$

Podzadanie 3 (23 punkty): $4 \leqslant N \leqslant 2\,000$

Podzadanie 4 (50 punktów): $4 \leqslant N \leqslant 100\,000$

Ograniczenia

Limit czasu: *0,5 s*

Dostępna pamięć: *256 MB*

Dawid Dąbrowski, Jakub Radoszewski
Tłumaczenie

BOI 2014, dzień drugi, 28.04.2014

Portale

W labiryncie znajduje się ciasto, a Ty masz na nie ogromną ochotę. Dysponujesz mapą labiryntu, która jest prostokątną planszą złożoną z R wierszy oraz C kolumn. Każda z komórek mapy zawiera jeden z następujących znaków:

- `#` *(hasz) oznaczający ścianę,*
- `.` *(kropka) oznaczający wolne pole,*
- `S` *(wielka litera s) oznaczający wolne pole, w którym się aktualnie znajdujesz,*
- `C` *(wielka litera c) oznaczający wolne pole, w którym znajduje się ciasto.*

Możesz poruszać się jedynie po wolnych polach planszy. Możesz przejść między dwoma wolnymi polami, jeśli na mapie sąsiadują ze sobą bokiem. Dodatkowo, prostokątny obszar labiryntu opisany na mapie jest otoczony z zewnątrz przez ściany.

Aby szybciej dobrać się do ciasta, zaopatrzyłeś się w urządzenie do tworzenia portali firmy Przysłona Nauka™. W dowolnym momencie może ono wystrzelić portal w jednym z czterech kierunków: góra, dół, lewo, prawo. Gdy portal jest wystrzelony w pewnym kierunku, będzie się w nim poruszał, dopóki nie natrafi na ścianę. Gdy to nastąpi, portal pojawi się na ścianie, w którą trafił – po tej stronie ściany, w którą uderzył.

W dowolnym momencie mogą istnieć co najwyżej dwa portale. Jeśli zdecydujesz się użyć urządzenia, gdy w labiryncie są już rozmieszczone dwa portale, jeden z nich (wybrany przez Ciebie) zostanie zniszczony. Wystrzelenie portalu w kierunku strony ściany, na której znajduje się już inny portal, zastąpi go (a zatem po każdej ze stron ściany może znajdować się co najwyżej jeden portal). Zauważ, że na jednej ścianie mogą znajdować się dwa portale, ale na różnych jej stronach.

Jeśli w labiryncie są umieszczone dwa portale, możesz użyć ich do teleportacji. Stojąc przy jednym z portali, możesz pójść w jego kierunku, w wyniku czego znajdziesz się na wolnym polu sąsiadującym z drugim portalem. Zajmuje to tyle samo czasu, co przejście między sąsiednimi polami.

Możesz założyć, że wystrzelenie portalu nie zajmuje czasu, a przemieszczenie się między sąsiednimi polami labiryntu oraz teleportacja przy użyciu portali zajmuje jednostkę czasu.

Zadanie

Mając daną mapę labiryntu wraz z Twoją pozycją startową oraz pozycją ciasta, wyznacz najkrótszy czas, w jakim możesz dobrać się do smakołyku.

Wejście

W pierwszym wierszu wejścia znajdują się dwie liczby całkowite: liczba wierszy mapy R oraz liczba kolumn mapy C. Kolejne R wierszy opisuje mapę. Każdy z nich zawiera C znaków: `#`, `.`, `S` *i/lub* `C` *(znaczenie znaków wyjaśnione jest na górze).*

Każdy ze znaków `S` *oraz* `C` *pojawi się na wejściu dokładnie raz.*

Wyjście

Wyjście powinno zawierać jedną liczbę całkowitą – najkrótszy czas, w jakim możesz dostać się do ciasta z Twojej pozycji startowej. Możesz założyć, że droga do ciasta zawsze istnieje.

Przykład

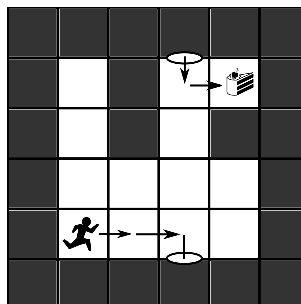
Dla danych wejściowych:

```
4 4
.#.C
.#.#
....
S...
```

poprawnym wynikiem jest:

```
4
```

Jeden z najkrótszych sposobów dojścia do ciasta to: 1) zrób krok w prawo, 2) zrób krok w prawo, wystrzel jeden portal do góry i jeden w dół, 3) przejdź przez dolny portal, 4) zrób krok w prawo i zjedz ciasto.



Ocenianie

Podzadanie 1 (11 punktów): $1 \leqslant R \leqslant 10, 1 \leqslant C \leqslant 10$

Podzadanie 2 (20 punktów): $1 \leqslant R \leqslant 50, 1 \leqslant C \leqslant 50$

Podzadanie 3 (20 punktów): $1 \leqslant R \leqslant 200, 1 \leqslant C \leqslant 200$. *Każde wolne pole ma co najmniej jedną sąsiadującą z nim ścianę.*

Podzadanie 4 (19 punktów): $1 \leqslant R \leqslant 200, 1 \leqslant C \leqslant 200$

Podzadanie 5 (30 punktów): $1 \leqslant R \leqslant 1\,000, 1 \leqslant C \leqslant 1\,000$

Ograniczenia

Limit czasu: *1 s*

Dostępna pamięć: *256 MB*

Dawid Dąbrowski, Jakub Radoszewski
Tłumaczenie

BOI 2014, dzień drugi, 28.04.2014

Sędziwi listonosze

Jest rok 2036. W Europie średni wiek mieszkańców drastycznie wzrósł. Aby utrzymać seniorów w dobrej kondycji, Europejskie Ministerstwo ds. Większości (już teraz seniorzy stanowią większość!) wymyśliło dla nich zajęcie – dostarczanie zwykłych przesyłek listownych, których adresatami są, swoją drogą, zazwyczaj właśnie seniorzy. Pomysł ma zostać wcielony w życie na całym obszarze Starego Kontynentu.

W Ministerstwie trwają prace nad projektem seniorskiego systemu pocztowego. Europę podzielono na pewną liczbę okręgów pocztowych. Każdy z nich pokryty jest siecią ulic złożoną z ulic i skrzyżowań. Wszystkie ulice są dwukierunkowe. W każdym okręgu pocztowym jest dostatecznie wielu seniorów, których można zatrudnić jako listonoszy. Każdego ranka każdy z listonoszy otrzymuje worek z listami, które ma dostarczyć na trasie złożonej z pewnej liczby ulic. Trasy muszą być odpowiednie dla seniorów, co Ministerstwo wyraziło w następujących dyrektywach:

- *Trasa musi zaczynać się i kończyć przy tym samym skrzyżowaniu.*
- *Trasa nie może przechodzić przez to samo skrzyżowanie ani tą samą ulicą więcej niż raz. (Żeby seniorom łatwo było ją zapamiętać.)*
- *Żadne dwie trasy nie mogą mieć wspólnych ulic; innymi słowy, każda ulica jest obsługiwana przez dokładnie jednego listonosza. (Chcemy uniknąć konkurencji między listonoszami–seniorami.)*

Tak więc wszystkie trasy łącznie muszą pokrywać całą sieć ulic, a każda ulica musi należeć do dokładnie jednej trasy.

Zadanie

Ministerstwo poprosiło Cię o przygotowanie oprogramowania, które na podstawie schematu sieci ulic w danym okręgu pocztowym wyznaczy zbiór tras odpowiednich dla seniorów, które pokryją wszystkie ulice w tym okręgu.

Wejście

Na wejściu znajduje się opis sieci ulic.

W pierwszym wierszu wejścia znajdują się dwie liczby całkowite N oraz M. N oznacza liczbę skrzyżowań, a M oznacza liczbę ulic. Skrzyżowania są ponumerowane od 1 do N.

Każdy z kolejnych M wierszy zawiera dwie liczby całkowite u oraz v ($1 \leqslant u, v \leqslant N$, $u \neq v$) reprezentujące ulicę łączącą skrzyżowania u oraz v.

Dane wejściowe spełniają następujące warunki:

1. *Każde dwa skrzyżowania są połączone co najwyżej jedną ulicą.*

2. *Każde dwa skrzyżowania są połączone ścieżką złożoną z ulic.*
3. *Istnieje rozwiązanie, czyli zbiór tras odpowiednich dla seniorów, które pokrywają wszystkie ulice w sieci.*

Wyjście

Każdy wiersz wyjścia powinien opisywać jedną trasę. Opisem trasy jest lista numerów skrzyżowań znajdujących się na trasie. Skrzyżowania należy wypisać w kolejności występowania na trasie, przy czym skrzyżowanie początkowe (a zarazem końcowe) należy wypisać na początku (i tylko raz).

Jeśli istnieje wiele poprawnych rozwiązań, Twój program powinien wypisać dowolne jedno z nich.

Przykład

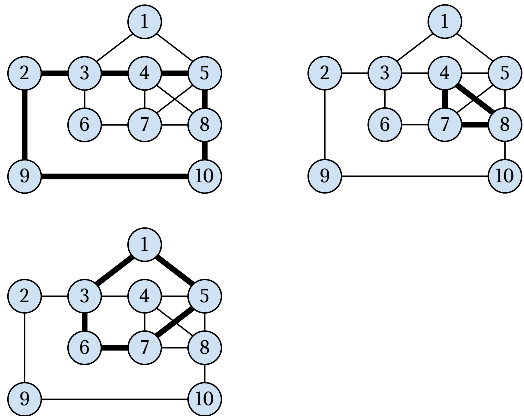
Dla danych wejściowych:

```
10 15
1 3
5 1
2 3
9 2
3 4
6 3
4 5
7 4
4 8
5 7
8 5
6 7
7 8
8 10
10 9
```

poprawnym wynikiem jest:

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

Na rysunku przedstawiono sieć ulic i trzy trasy odpowiednie dla seniorów, którymi można pokryć wszystkie ulice.



W tym przykładzie jest wiele możliwych rozwiązań, a wśród nich takie, które składają się tylko z dwóch tras.

Ocenianie

Podzadanie 1 (38 punktów): $3 \leqslant N \leqslant 2\,000$, $3 \leqslant M \leqslant 100\,000$

Podzadanie 2 (17 punktów): $3 \leqslant N \leqslant 100\,000$, $3 \leqslant M \leqslant 100\,000$

Podzadanie 3 (45 punktów): $3 \leqslant N \leqslant 500\,000$, $3 \leqslant M \leqslant 500\,000$

Ograniczenia

Limit czasu: *0,5 s*

Dostępna pamięć: *256 MB*

XX Olimpiada Informatyczna Krajów Europy Środkowej

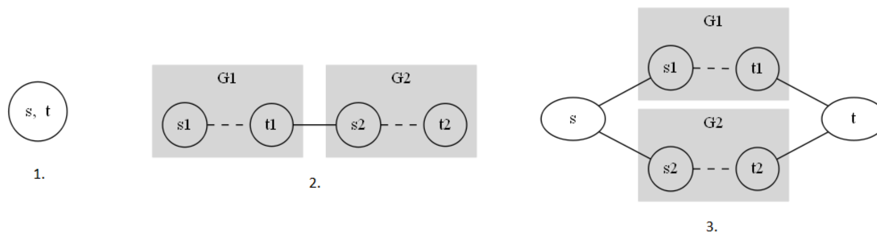
Primošten, Chorwacja 2013

Jakub Łącki, Jakub Pawlewicz
Tłumaczenie

Paszor

Na wybrzeżu Adriatyku znajduje się mnóstwo fantastycznych plaż. Niestety do wielu z nich nie da się dojechać samochodem. Ponieważ wiele osób nad morze przyjeżdża samochodem, na jednej z łąk nieopodal wybrzeża zostanie wybudowany parking. Architekt, który projektuje parking, pracował wcześniej jako inżynier, dlatego układ parkingu przypomina graf szeregowo-równoległy, często wykorzystywany przy projektowaniu obwodów elektrycznych.

Parking składa się z miejsc parkingowych połączonych dwukierunkowymi ulicami. Każda ulica łączy dwa różne miejsca parkingowe; każda para miejsc może być połączona co najwyżej jedną ulicą. Na każdym miejscu parkingowym może znajdować się **co najwyżej jeden samochód**. *Jeśli dane miejsce parkingowe jest zajęte, żaden samochód nie może przez nie przejechać.*



Rys. 1: Zasady budowania paszorów

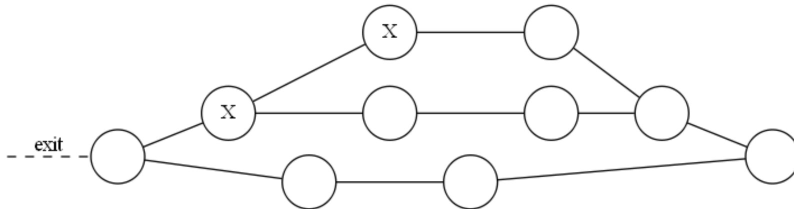
Parking szeregowo-równoległy (w skrócie **paszor***) jest to parking zawierający dwa wyróżnione miejsca parkingowe zwane* **źródłem** *i* **ujściem***, powstały w wyniku ciągu operacji szeregowego i równoległego złączania. Każdy paszor można zadać za pomocą ciągu znaków – ciąg ten opisuje strukturę parkingu i pozycje zaparkowanych samochodów. Poprawny paszor i jego opis są określone rekurencyjnie (ilustracje do każdego z poniższych punktów można znaleźć na rysunku 1):*

1. *Parking składający się z jednego miejsca parkingowego (bez żadnych ulic) to poprawny paszor. Miejsce to jest zarówno źródłem, jak i ujściem. Opis takiego paszoru to jedna litera* `o`*, jeśli to miejsce jest wolne, lub* `x`*, w przypadku gdy miejsce jest zajęte.*

2. *Jeśli G_1 i G_2 to dwa paszory, ich szeregowe złączenie również jest paszorem. Szeregowym złączeniem G_1 i G_2 nazywamy parking G powstały przez wstawienie ulicy między ujściem G_1 i źródłem G_2. Źródłem G jest źródło G_1, zaś ujściem G staje się ujście G_2. Jeśli E_1 i E_2 to opisy paszorów G_1 i G_2, opisem G jest* `S`E_1E_2`#`*. Innymi słowy, opis składa się kolejno z wielkiej litery* `S`*, opisów paszorów G_1 i G_2 oraz znaku* `#`*.*

3. *Jeśli G_1 i G_2 to dwa paszory, ich równoległe złączenie również jest paszorem. Równoległym złączeniem G_1 i G_2 nazywamy parking G powstały przez dodanie do G_1 i G_2 dwóch nowych miejsc parkingowych s i t oraz dwóch ulic łączących s ze źródłami G_1 i G_2 oraz dwóch kolejnych ulic łączących t z ujściami G_1 i G_2. Źródłem nowo powstałego paszoru*

G jest parking s, zaś ujściem – parking t. Niech E_1 i E_2 to opisy paszorów G_1 i G_2, zaś E_s i E_t to opisy, odpowiednio, źródła s i ujścia t (te opisy to zawsze jedna mała litera: `o`, *jeśli odpowiadające miejsce jest puste, lub* `x`, *gdy jest ono zajęte). Wówczas opisem G jest* $\texttt{P}E_s\texttt{|}E_1E_2\texttt{|}E_t\texttt{\#}$. *Innymi słowy, opis składa się kolejno z: wielkiej litery* `P`, *opisu miejsca parkingowego, znaku* `|`, *opisu złączanych paszorów, znaku* `|`, *opisu miejsca parkingowego i znaku* `#`.



Rys. 2: Paszor opisany w pierwszym przykładowym wejściu

Na przykład, opisem paszoru z powyższego rysunku jest `Po|Px|Sxo#Soo#|o#Soo#|o#`. *Zauważ, że liczba małych liter w opisie paszoru jest równa liczbie miejsc parkingowych. Co więcej, istnieje jednoznaczna odpowiedniość między miejscami parkingowymi a małymi literami z opisu.*

Z parkingu można wyjechać **dokładnie jednym wyjazdem**, *bezpośrednio połączonym ze źródłem paszoru. Powiemy, że samochód* **nie jest zablokowany**, *jeśli może* **wyjechać z paszoru**, *to znaczy dotrzeć do źródła, poruszając się jedynie po* **ulicach i niezajętych miejscach parkingowych**. *W powyższym przykładzie żaden z samochodów nie jest zablokowany, jednak jeśli dostawilibyśmy samochód w ujściu (na rysunku najbliżej prawego brzegu), jeden z samochodów zostałby zablokowany. Dozwolone jest parkowanie samochodów w źródle, jednak spowoduje to zablokowanie wszystkich samochodów na paszorze.*

Zadanie

Dany jest opis paszoru, w którym niektóre miejsca mogą być zajęte przez zaparkowane samochody. Wiemy, że żaden z samochodów nie jest zablokowany. Obsługa parkingu chciałaby tak ustawiać przyjeżdżające samochody, by **żaden samochód nie został zablokowany**. *Napisz program, który wyznaczy* **największą łączną liczbę samochodów**, *które mogą zostać zaparkowane na podanym paszorze (włączając w to już zaparkowane), tak by żaden samochód* **nie był zablokowany**. *Nie jest dopuszczalne przesuwanie uprzednio zaparkowanych samochodów. Ponadto, Twój program powinien wyznaczyć pewne ustawienie obliczonej liczby samochodów na paszorze.*

Wejście

Pierwszy wiersz wejścia zawiera niepusty ciąg składający się z co najwyżej 100 000 znaków. Ciąg ten jest opisem paszoru i składa się z wielkich liter `P` *i* `S`, *małych liter* `o` *i* `x` *oraz znaków* `#` *(kod ASCII 35) i* `|` *(kod ASCII 124). Opis jest zgodny z podanymi wcześniej zasadami. Żaden z samochodów zaparkowanych na podanym paszorze nie będzie zablokowany.*

Wyjście

Wyjście powinno składać się z dwóch wierszy. Pierwszy wiersz powinien zawierać jedną liczbę całkowitą M oznaczającą maksymalną liczbę samochodów, które można zaparkować na podanym paszorze.

W drugim wierszu należy wypisać ciąg znaków, który opisuje wyznaczone rozwiązanie. Ciąg powinien zawierać dokładnie M znaków `x` *i powstawać z ciągu podanego na wejściu przez zamianę pewnych znaków* `o` *na* `x`*.*

Może istnieć wiele optymalnych rozwiązań. Twój program może wypisać dowolne z nich.

Ocenianie

- *Jeśli wyjście jest niepoprawne lub niepełne, jednak maksymalna liczba samochodów, jakie da się zaparkować (tj. liczba w pierwszym wierszu wyjścia), jest poprawna, rozwiązanie otrzyma 80% punktów za test.*
- *W testach wartych łącznie 30 punktów łączna liczba miejsc parkingowych nie przekracza 20.*
- *W kolejnych testach, wartych łącznie 40 punktów, wszystkie miejsca podanego paszoru są wolne, tj. wejście nie zawiera liter* `x`*.*

Przykłady

Dla danych wejściowych:

```
Po|Px|Sxo#Soo#|o#Soo#|o#
```

poprawnym wynikiem jest:

```
3
Po|Px|Sxo#Sox#|o#Soo#|o#
```

Natomiast dla danych wejściowych:

```
Po|SPo|oo|o#Px|oo|o##Po|Sxo#Po|ox|o#|o#|o#
```

poprawnym wynikiem jest:

```
7
Po|SPo|xx|o#Px|ox|o##Po|Sxx#Po|ox|o#|o#|o#
```

Dodatkowe narzędzie

Program `splot_tool` *(do pobrania na stronie zawodów) może posłużyć do wizualizowania plików wejściowych i wyjściowych. Program ten przyjmuje jeden argument wiersza poleceń – poprawny plik wejściowy lub wyjściowy i generuje obraz w formacie* `svg`*, przedstawiający paszor opisany w podanym pliku. Wygenerowany obraz można obejrzeć w przeglądarce internetowej. Przykładowe użycie:*

```
$ ./splot_tool splot.dummy.out.1
splot.dummy.out.1 parsed (10 parking spaces).
splot.dummy.out.1.svg created.
$ chromium splot.dummy.out.1.svg
```

Narzędzie sprawdza także, czy opis paszoru ma poprawny format i wypisuje informacje o ewentualnych błędach. **Nie przeprowadza** *ono jednak jakichkolwiek innych sprawdzeń. W szczególności nawet dla błędnych rozwiązań może zakończyć się pomyślnie. Program działa jedynie dla paszorów, których opis ma długość co najwyżej 200 znaków.*

Ograniczenia

Limit czasu: *2 s*

Dostępna pamięć: *256 MB*

Jakub Łącki, Jakub Pawlewicz
Tłumaczenie

CEOI 2013, dzień pierwszy, 15.10.2013

Skarb

Wskutek niedawnego trzęsienia ziemi z Morza Śródziemnego wyłoniła się nowa wyspa! Pierwsi podróżnicy przybyli na wyspę znaleźli na niej pewne niezwykłe urządzenie. Mimo że nie znaleziono instrukcji obsługi, zespół archeologów i hakerów komputerowych dał radę odtworzyć sposób jego działania i ustalił, że udziela ono informacji o wyspie i ukrytych na niej tajemniczych skarbach. Urządzeniu nadano nazwę „wyrocznia”.

Wyspę traktujemy jako kwadrat o rozmiarach $N \times N$, który składa się z kwadratowych pól rozmiaru 1×1 ustawionych w N wierszy i N kolumn. Zarówno wiersze, jak i kolumny są ponumerowane od 1 do N. Na niektórych polach ukryte są skarby. Wyrocznia dostarcza informacje o położeniu skarbów na wyspie: dla podanego wycinka wyspy w kształcie prostokąta, mówi ona, na ilu polach w tym prostokącie znajdują się skarby.

Choć wyrocznia odpowiada na pytania dotyczące prostokątów o dowolnych rozmiarach, im dokładniejszej informacji żądamy (tj. im mniejszy jest prostokąt), tym więcej energii potrzebuje wyrocznia do wygenerowaniu wyniku. Konkretnie, jeżeli w skład prostokąta wchodzi S pól, to wyrocznia zużywa dokładnie $1 + N^2 - S$ jednostek energii.

Zadanie

Napisz program, który komunikując się z wyrocznią, wyznaczy **położenia wszystkich pól ze skarbami**. *Jednocześnie program powinien zadbać o to, by wyrocznia nie zużyła za dużo energii – im mniej energii zużyje, tym lepiej. Nie jest wymagane, aby ilość zużytej energii była najmniejsza możliwa. Szczegóły podane są w punkcie „Ocenianie”.*

Komunikacja

To zadanie jest interaktywne. Twój program zadaje pytania wyroczni, używając standardowego wyjścia i otrzymuje odpowiedzi na standardowym wejściu.

- *Twój program powinien najpierw wczytać liczbę całkowitą N ($2 \leqslant N \leqslant 100$), oznaczającą rozmiar wyspy.*
- *Aby zadać pytanie wyroczni, program powinien wypisać wiersz składający się z czterech liczb całkowitych R_1, C_1, R_2 i C_2 pooddzielanych pojedynczymi odstępami, gdzie $1 \leqslant R_1 \leqslant R_2 \leqslant N$ i $1 \leqslant C_1 \leqslant C_2 \leqslant N$. W przypadku, gdy wymienione zależności nie będą spełnione bądź wiersz będzie niezgodny ze specyfikacją, program otrzyma zero punktów za test.*
- *Odpowiedź wyroczni pobiera się, wyczytując wiersz zawierający jedną liczbę całkowitą – liczbę pól ze skarbem w podanym prostokącie, tj. liczbę takich pól (R, C), że $R_1 \leqslant R \leqslant R_2$, $C_1 \leqslant C \leqslant C_2$ i na polu w wierszu R i kolumnie C jest skarb.*
- *Kiedy tylko Twój program zakończy zadawać pytania, powinien on wypisać wiersz zawierający słowo* `END`*. Następnie powinien wypisać N wierszy, po jednym dla każdego*

wiersza wyspy. Każdy z tych wierszy powinien składać się z N znaków `0` *(zero) lub* `1` *(jeden). C-ty znak w R-tym wierszu odpowiada polu w C-tej kolumnie i R-tym wierszu wyspy. Znak* `1` *oznacza, że na odpowiednim polu znajduje się skarb, zaś* `0` *– że pole to jest puste. Wykonanie Twojego programu zostanie automatycznie przerwane zaraz po wypisaniu rozwiązania.*

- *Do poprawnej komunikacji niezbędne jest opróżnianie bufora standardowego wyjścia przez Twój program po każdym pytaniu i po wypisaniu rozwiązania. Można zobaczyć sobie w przykładowych programach, jak to się robi.*

Możesz założyć, że wyrocznia poprawnie odpowiada na pytania oraz że rozmieszczenie pól ze skarbami jest ustalone zawczasu, przed rozpoczęciem komunikacji. Innymi słowy sprawdzaczka nie jest złośliwa – odpowiedzi wyroczni nie zależą od pytań poprzednio zadanych przez Twój program.

Ocenianie

Każdy test jest warty 10 punktów. Jeżeli wyjście Twojego programu jest niepoprawne, przyznane zostaje zero punktów. W przeciwnym razie liczba otrzymanych punktów zależy od łącznej ilości energii zużytej przez wyrocznię. Oznaczmy tę wartość przez K. Wówczas:

- *Jeśli $K \leqslant \frac{7}{16}N^4 + N^2$, otrzymasz 10 punktów.*
- *W przeciwnym razie, jeśli $K \leqslant \frac{7}{16}N^4 + 2N^3$, otrzymasz 8 punków.*
- *W przeciwnym razie, jeśli $K \leqslant \frac{3}{4}N^4$, otrzymasz 4 punkty.*
- *W przeciwnym razie, jeśli $K \leqslant N^4$, otrzymasz 1 punkt.*
- *W przeciwnym razie otrzymasz 0 punktów.*

Ponadto, w testach wartych łącznie co najmniej 40% wszystkich punktów N nie przekracza 20. Rozwiązanie zostanie uznane za poprawne nawet jeśli Twój program je zgadnie bez zadawania żadnych pytań.

Przykład

W poniższym przykładzie w lewej kolumnie w osobnych wierszach podane są pytania skierowane do wyroczni. Druga kolumna zawiera rozmiar wyspy (w pierwszym wierszu) oraz odpowiedzi wyroczni.

wyjście	**wejście**
	2
1 1 1 1	0
1 2 1 2	1
2 1 2 2	2
END	
01	
11	

Testowanie

Aby przetestować swój program należy utworzyć plik z opisem wyspy. W pierwszym wierszu tego pliku powinna się znajdować liczba całkowita N. Po niej powinno nastąpić N wierszy opisujących położenia skarbów, w takim samym formacie jak opis wyspy przy podawaniu wyniku. Na przykład dla testu z przykładu w poprzednim punkcie opis wygląda następująco:

```
2
01
11
```

Następnie należy użyć programu `treasure_test`*, który można pobrać ze strony zawodów. Program uruchamia się przy pomocy polecenia:*

```
./treasure_test ./moje_rozwiazanie plik_wejsciowy
```

Program ten wypisuje jedynie informacje o poprawności odpowiedzi. Plik `treasure.log` *zawiera dodatkowo informacje o pytaniach, jakie zadał Twój program, i otrzymanych odpowiedziach.*

Ograniczenia

Limit czasu: *1 s*

Dostępna pamięć: *256 MB*

Jakub Łącki, Jakub Pawlewicz
Tłumaczenie

CEOI 2013, dzień pierwszy, 15.10.2013

Tramwaj

W nowym tramwaju kursującym po Zagrzebiu siedzenia ustawione są w N rzędów po dwa siedzenia (innymi słowy w N rzędów i dwie kolumny). Rzędy numerujemy liczbami od 1 do N, a kolumny liczbami 1 i 2. Odległość między siedzeniem w wierszu R_A i kolumnie C_A oraz siedzeniem w wierszu R_B i kolumnie C_B to odległość Euklidesowa między ich środkami (zakładamy, że siedzenia to kwadraty 1×1, ustawione w prostokąt o wymiarach $N \times 2$), czyli po prostu $\sqrt{(R_A - R_B)^2 + (C_A - C_B)^2}$.

Pasażerowie cenią sobie spokój, dlatego gdy wsiadają do tramwaju, wybierają miejsce, które znajduje się jak najdalej od innych pasażerów. Mianowicie, każdy pasażer wybiera wolne miejsce, którego **odległość od najbliższego zajętego miejsca jest jak największa**. *Jeśli istnieje wiele takich miejsc, preferowane jest miejsce w rzędzie o niższym numerze. Jeśli to nadal nie przynosi rozstrzygnięcia, pasażer siada na miejscu o mniejszym numerze kolumny (spośród najdalszych miejsc o jak najniższym numerze rzędu). Po wybraniu miejsca, pasażer siada i nie rusza się aż do opuszczenia tramwaju. Pierwszy pasażer zajmuje miejsce w pierwszym rzędzie i pierwszej kolumnie.*

Zadanie

Napisz program, który dla podanego ciągu wydarzeń (wydarzenie to pojawienie się pasażera lub wyjście pewnego pasażera z tramwaju) wyznaczy, na którym z miejsc siedział każdy z pasażerów. Początkowo tramwaj jest pusty.

Wejście zawiera M zdarzeń podanych w kolejności chronologicznej. Zdarzenia numerujemy kolejno od 1 do M. Każde zdarzenie jest jednego z dwóch typów. Zdarzenie typu `E` *oznacza, że pewien pasażer wsiadł do tramwaju. Z kolei zdarzenie typu* `L` *oznacza, że któryś z pasażerów opuścił tramwaj. W tym przypadku dana jest także liczba P, która oznacza, że wysiadł pasażer, którego pojawienie się w tramwaju było wydarzeniem numer P.*

Możesz założyć, że każdy pasażer wsiadający do tramwaju będzie mógł znaleźć wolne miejsce.

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby całkowite N i M ($1 \leqslant N \leqslant 150\,000$, $1 \leqslant M \leqslant 30\,000$) oznaczające liczbę rzędów w tramwaju oraz liczbę zdarzeń. Kolejne M wierszy opisuje poszczególne wydarzenia. W K-tym wierszu znajduje się opis K-tego wydarzenia. Jest to albo pojedynczy znak `E`*, albo znak* `L`*, po którym następuje pojedynczy odstęp i liczba P_K ($1 \leqslant P_K < K$). W tym drugim przypadku wydarzenie numer P_K jest wydarzeniem typu* `E`. *Każdy pasażer wysiada z tramwaju co najwyżej jednokrotnie.*

Wyjście

Twój program powinien wypisać po jednym wierszu dla każdego wydarzenia typu `E`*. Kolejno wypisywane wiersze powinny odpowiadać kolejnym wydarzeniom typu* `E`*. W każdym z wierszy powinny znaleźć się dwie liczby całkowite oddzielone pojedynczym odstępem – numer rzędu i kolumny, które określają położenie siedzenia wybranego przez odpowiedniego pasażera.*

Ocenianie

- *W testach wartych łącznie 25 punktów zachodzi* $N \leqslant 150$ *oraz* $M \leqslant 150$.
- *W testach wartych łącznie 45 punktów zachodzi* $N \leqslant 1500$ *oraz* $M \leqslant 1500$.
- *W testach wartych łącznie 65 punktów zachodzi* $N \leqslant 150\,000$ *oraz* $M \leqslant 1500$.

Przykłady

Dla danych wejściowych:

```
3 7
E
E
E
L 2
E
L 1
E
```

poprawnym wynikiem jest:

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

Dla danych wejściowych:

```
13 9
E
E
E
E
E
E
E
E
E
```

poprawnym wynikiem jest:

```
1 1
13 2
7 1
4 2
10 1
2 2
3 1
5 1
6 2
```

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`10 9`	`1 1`
`E`	`10 2`
`E`	`5 2`
`E`	`7 1`
`E`	`4 2`
`L 3`	`2 2`
`E`	`4 1`
`E`	
`L 6`	
`E`	

Ograniczenia

Limit czasu: *1 s*

Dostępna pamięć: *256 MB*

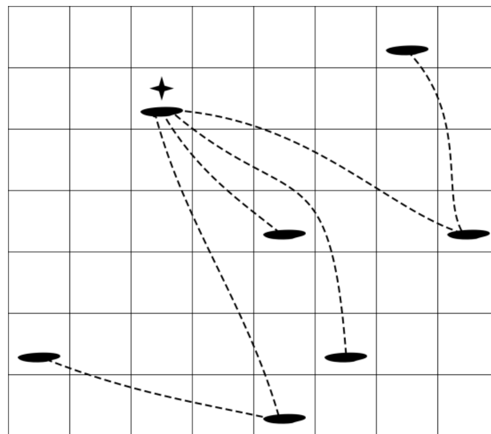
Jakub Łącki, Jakub Pawlewicz
Tłumaczenie

CEOI 2013, dzień drugi, 17.10.2013

Adriatyk

W latach dziewięćdziesiątych dwudziestego wieku Chorwacja reklamowała się jako „kraj tysiąca wysp”. Wprawdzie to hasło reklamowe nie jest do końca poprawne (na terytorium Chorwacji znajduje się ponad 1000 wysp), jednak bez wątpienia zwiedzanie wysp i żeglowanie pomiędzy nimi przyciągało wielu turystów.

Na potrzeby tego zadania przyjmijmy, że Adriatyk to kwadrat o wymiarach 2500×2500 *składający się z pól – kwadratów o wymiarach* 1×1. *Pola ustawione są w 2500 rzędów i tyleż samo kolumn. Wiersze numerujemy kolejno od 1 do 2500 od góry do dołu (z północy na południe). Podobnie, kolumny numerujemy kolejno od 1 do 2500 od lewej do prawej (z zachodu na wschód). Na mapie znajduje się* N *wysp, które numerujemy od 1 do* N. *Każda z wysp zajmuje jedno pole. Położenie wyspy numer* K *zadane jest przy pomocy numeru wiersza* R_K *i kolumny* C_K. *Na jednym polu może znajdować się co najwyżej jedna wyspa.*



Rys. 1: Mapa odpowiadająca pierwszemu przykładowi poniżej

Ze względu na kierunki wiatrów i prądów morskich, między wyspami A *i* B *można bezpośrednio przepłynąć tylko wtedy, gdy* B *znajduje się w kierunku północno–zachodnim od* A *(tj. gdy* $R_A < R_B$ *i* $C_A < C_B$*) lub południowo–wschodnim od* A *(*$R_A > R_B$ *i* $C_A > C_B$*). Wyspy* A *i* B *mogą leżeć daleko od siebie, a na drodze znajdować się mogą inne wyspy, jednak nie wpływa to na możliwość przepłynięcia między wyspami. Jeśli bezpośredni rejs z wyspy* A *na wyspę* B *nie jest możliwy, nadal może się dać przemieścić między nimi za pomocą większej liczby rejsów. Odległością żeglarską między* A *i* B *nazywamy najmniejszą liczbę rejsów potrzebnych do dostania się z* A *do* B.

W przykładzie z powyższego rysunku z wyspy w drugim wierszu i trzeciej kolumnie możemy bezpośrednio przepłynąć do czterech innych wysp. Odległość żeglarska do pozostałych dwóch wysp to dwa.

Zadanie

Organizatorzy nadchodzącej konferencji żeglarskiej zastanawiają się, na której wyspie ją zlokalizować. Aby ocenić konkretną wyspę, chcieliby oni wiedzieć, jaka jest **minimalna łączna liczba rejsów***, które byłyby potrzebne, gdyby do rozważanej wyspy miała przypłynąć jedna żaglówka* **z każdej z pozostałych wysp***. Innymi słowy, chcieliby oni poznać sumę odległości żeglarskich między rozważaną wyspą a wszystkimi innymi. Napisz program, który dla podanych położeń N wysp, dla każdej wyspy K wyznaczy sumę odległości żeglarskich od wszystkich wysp do wyspy K.*

Możesz założyć, że między każdą parą wysp da się przepłynąć (bezpośrednio lub pośrednio).

Wejście

Pierwszy wiersz wejścia zawiera liczbę całkowitą N ($3 \leqslant N \leqslant 250\,000$) oznaczającą liczbę wysp. Każdy z kolejnych N wierszy opisuje położenie jednej wyspy. Opis to para liczb całkowitych z przedziału od 1 do 2500 (włącznie). Pierwsza z nich określa numer wiersza, a druga – numer kolumny, w której znajduje się wyspa.

Wyjście

Wypisz N wierszy; i–ty z tych wierszy powinien zawierać sumę odległości żeglarskich i–tej wyspy podanej w wejściu od wszystkich innych wysp.

Ocenianie

- *W testach wartych łącznie 25 punktów N jest nie większe niż 100.*
- *W testach wartych łącznie 50 punktów N jest nie większe niż 1500.*
- *W testach wartych łącznie 60 punktów N jest nie większe niż 5000.*
- *W testach wartych łącznie 80 punktów N jest nie większe niż 25 000.*

Przykłady

Dla danych wejściowych:	*poprawnym wynikiem jest:*
7	16
1 7	11
7 5	12
4 5	11
4 8	12
6 6	16
6 1	8
2 3	

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
3
4
4
3
```

Ograniczenia

Limit czasu: *2 s*

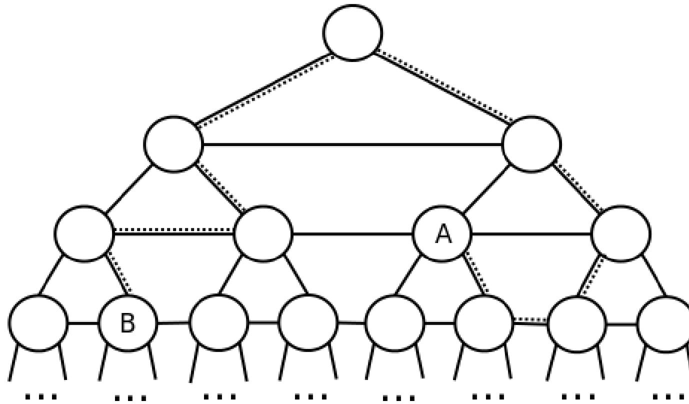
Dostępna pamięć: *256 MB*

Jakub Łącki, Jakub Pawlewicz
Tłumaczenie

CEOI 2013, dzień drugi, 17.10.2013

Plansza

Mirosław i Sławomir mają nową grę planszową. Plansza do tej gry przypomina nieskończone pełne drzewo binarne. Składa się ona z wierzchołków oraz łączących je dwukierunkowych dróg. Korzeń drzewa znajduje się na górze – mówimy, że jest on na **zerowym poziomie***. Każdy wierzchołek ma dwoje dzieci:* **lewe dziecko** *i* **prawe dziecko***, które są umieszczone odpowiednio w dół i na lewo oraz w dół i na prawo od swojego rodzica. Poziom dziecka jest o jeden większy od poziomu rodzica. Oprócz dróg łączących rodzica z dziećmi, na planszy znajdują się również drogi łączące wierzchołki w poziomie. Mianowicie, każdy wierzchołek (poza skrajnie prawymi wierzchołkami każdego poziomu) jest połączony drogą z wierzchołkiem znajdującym się bezpośrednio na prawo od niego (na tym samym poziomie).*



Rys. 1: Drugi z przykładów podanych poniżej

Każda **ścieżka** *w tej grze składa się z ciągu kroków polegających na przejściu po drodze łączącej dwa wierzchołki. Każdy krok jest oznaczony pojedynczą literą:*

- *znak* `1` *oznacza przejście z wierzchołka do jego lewego dziecka,*
- *znak* `2` *oznacza przejście z wierzchołka do jego prawego dziecka,*
- *znak* `U` *oznacza przejście z wierzchołka do jego rodzica,*
- *znak* `L` *oznacza przejście z wierzchołka do najbliższego wierzchołka na lewo na tym samym poziomie,*
- *znak* `R` *oznacza przejście z wierzchołka do najbliższego wierzchołka na prawo na tym samym poziomie.*

Na przykład, jeśli zaczniemy w korzeniu drzewa i wykonamy kroki opisane ciągiem `221LU`*, to znajdziemy się w wierzchołku oznaczonym literą A na rysunku powyżej.*

Zadanie

Napisz program, który dla podanych dwóch wierzchołków na planszy wyznaczy najmniejszą liczbę kroków, jakie należy wykonać, aby przejść z jednego wierzchołka do drugiego. Wierzchołki te zadane są przy pomocy ścieżek, jakimi można do nich dojść z korzenia. Jeśli te dwie ścieżki prowadzą do tego samego wierzchołka, szukana liczba kroków wynosi zero.

Wejście

Pierwszy wiersz wejścia zawiera ciąg składający się z co najwyżej 100 000 znaków opisujący ścieżkę z korzenia do pierwszego wierzchołka.

Drugi wiersz wejścia zawiera ciąg składający się z co najwyżej 100 000 znaków opisujący ścieżkę z korzenia do drugiego wierzchołka.

Podane ścieżki są poprawne, tj. każdy podany krok daje się wykonać.

Wyjście

Jedyny wiersz wyjścia powinien zawierać jedną liczbę całkowitą oznaczającą najmniejszą liczbę kroków potrzebnych do przejścia z pierwszego wierzchołka do drugiego.

Ocenianie

Niech D będzie najmniejszą taką liczbą całkowitą, że obie ścieżki przechodzą wyłącznie przez wierzchołki o poziomach nieprzekraczających D.

- *W testach wartych łącznie 20 punktów D wynosi co najwyżej 10.*
- *W testach wartych łącznie 40 punktów D wynosi co najwyżej 50.*
- *W testach wartych łącznie 70 punktów D wynosi co najwyżej 1000.*

Przykłady

Dla danych wejściowych:
```
111RRRRRRR
222
```
poprawnym wynikiem jest:
```
0
```

Dla danych wejściowych:
```
221LU
12L2
```
poprawnym wynikiem jest:
```
3
```

Dla danych wejściowych:
```
11111
222222
```
poprawnym wynikiem jest:
```
10
```

Ograniczenia

Limit czasu: *0,2 s*

Dostępna pamięć: *256 MB*

Jakub Łącki, Jakub Pawlewicz
Tłumaczenie

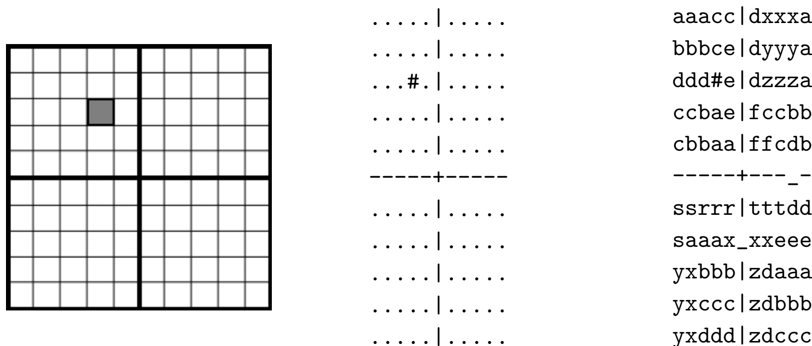
CEOI 2013, dzień drugi, 17.10.2013

Podlewanie

Sara jest zapalonym rolnikiem. Swoją pasję realizuje na dużej prostokątnej działce, którą uprawia. Działka ta składa się z kwadratowych **poletek** *ustawionych w $5 \cdot R$ rzędów i $5 \cdot C$ kolumn.*

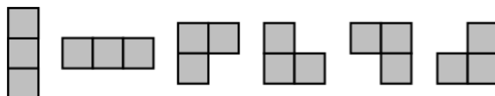
Co pięć rzędów działka przedzielona jest poziomą (tj. przebiegającą równolegle do rzędów poletek) miedzą. Podobnie, co pięć kolumn przez działkę przebiega pionowa (równoległa do kolumn) miedza. W efekcie miedze dzielą działkę na $R{\cdot}C$ obszarów, które składają się z 25 poletek. Obszary te będziemy nazywać **łanami***.*

Sara zmaga się z dwoma poważnymi problemami: ptakami i suszami. Aby odstraszyć plonożerne ptaki, na niektórych łanach umieściła strachy na wróble. Jeśli w obrębie łanu stoi strach na wróble, zajmuje on całe poletko. Na jednym łanie stoi **co najwyżej jeden strach na wróble***.*



Rys. 1: Przykładowy wygląd działki Sary, jego opis w formacie tekstowym oraz pewne poprawne rozstawienie zraszaczy.

Podczas wielomiesięcznych susz Sara używa zraszaczy, by nawodnić uprawiane rośliny. Każdy ze zraszaczy składa się z trzech dysz: jednej głównej dyszy i dwóch dysz bocznych. Zraszacz zajmuje **dokładnie trzy poletka** *i podlewa każde z nich. Dysze boczne znajdują się na dokładnie* **dwóch poletkach sąsiednich** *(na rysunku: w górę, dół, lewo lub prawo) z główną dyszą. Zatem każdy zraszacz może mieć jedno z poniższych ustawień:*



Sara chciałaby tak rozstawić zraszacze na działce, by na każdym poletku niezajętym przez strach na wróble znajdowała się **dokładnie jedna** *dysza zraszacza. Dysze nie mogą znajdować się poza działką Sary ani na poletku zajętym przez strach na wróble.*

Trzy poletka nawadniane przez każdy zraszacz nie muszą należeć do tego samego łanu: mogą również leżeć w sąsiednich łanach. W takim przypadku Sara musi **przekopać miedzę w miejscu pomiędzy sąsiednimi poletkami nawadnianymi przez jeden zraszacz***. Kopanie to męczące zajęcie, dlatego Sara wolałaby nie wykonywać zbyt wielu przekopów.*

Zadanie

Dla podanego opisu działki Sary, wyznacz poprawne rozstawienie zraszaczy. Ocena Twojego rozwiązania będzie zależeć od liczby przekopów, jakie należy wykonać – szczegóły znajdują się w punkcie Ocenianie.

To jest zadanie z otwartym wejściem. Dostępnych jest 10 plików z danymi wejściowymi, a Twoim zadaniem jest wygenerowanie plików zawierających poprawne odpowiedzi. Pliki wejściowe możesz pobrać ze strony zawodów.

Możesz założyć, że dla każdego pliku wejściowego istnieje poprawne rozwiązanie. Jeśli istnieje wiele poprawnych odpowiedzi, możesz zgłosić dowolną z nich.

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby całkowite R i C ($1 \leqslant R, C \leqslant 100$) określające wymiary działki Sary.

W każdym z następnych $6 \cdot R - 1$ wierszy znajduje się ciąg $6 \cdot C - 1$ znaków. Ciągi te opisują działkę Sary i miedze. Wprawdzie opis każdej miedzy składa się z osobnych znaków, jednak w rzeczywistości miedze są nieskończenie cienkie.

Każde poletko jest opisane jednym znakiem. Znak '`.`' (kropka) oznacza puste poletko, zaś znak '`#`' (ASCII 35) to strach na wróble. Pionowe miedze oznaczone są znakami '`|`' (ASCII 124), zaś poziome – znakami '`-`' (minus). Znak '`+`' (plus) oznacza przecięcie miedzy.

Wyjście

Wyjście powinno zawierać tekstowy opis działki z poprawnym rozstawieniem zraszaczy, w takim samym formacie, jak w pliku wejściowym. Każdy przekop miedzy należy oznaczyć znakiem '`_`' (podkreślenie). Wszystkie kropki w pliku wejściowym (puste poletka) należy zamienić na litery od '`a`' do '`z`', tak aby zachowane były poniższe warunki:

1. *Trzy poletka nawadniane przez ten sam zraszacz powinny być oznaczone tą samą literą (nawet jeśli poletka te należą do różnych łanów).*
2. *Jeśli dwa sąsiednie poletka na* **tym samym** *łanie podlewają inne zraszacze, należy oznaczyć je różnymi literami.*
3. *Jeśli dwa sąsiednie poletka w* **różnych łanach** *są podlewane przez* **różne zraszacze***, a między tymi łanami znajduje się przekop, należy oznaczyć je różnymi literami.*
4. *Sąsiednie zraszacze z różnych łanów mogą być oznaczone tymi samymi literami, jeśli tylko spełnione są powyższe warunki.*

Ocenianie

Za każdy test można otrzymać 10 punktów. Jeśli wyznaczone ustawienie zraszaczy nie jest poprawne, otrzymasz 0 punktów. W przeciwnym razie:

- *Jeśli liczba przekopów nie przekracza $R \cdot C$, otrzymasz 10 punktów.*
- *Jeśli zaś przekopów będzie więcej niż $R \cdot C$, otrzymasz 5 punktów.*

W 4 testach (z 10) na każdym łanie znajduje się strach na wróble.

Przykład

Dla danych wejściowych:

```
2 2
.....|.....
.....|.....
...#.|.....
.....|.....
.....|.....
-----+-----
.....|.....
.....|.....
.....|.....
.....|.....
.....|.....
```

poprawnym wynikiem jest:

```
aaacc|dxxxa
bbbce|dyyya
ddd#e|dzzza
ccbae|fccbb
cbbaa|ffcdb
-----+---_-
ssrrr|tttdd
saaax_xxeee
yxbbb|zdaaa
yxccc|zdbbb
yxddd|zdccc
```

XXI Olimpiada Informatyczna Krajów Europy Środkowej

Jena, Niemcy 2014

Jakub Pawlewicz, Anna Zych
Tłumaczenie

CEOI 2014, dzień pierwszy, 20.06.2014

Karnawał

Piotr ma N przyjaciół, ponumerowanych od 1 do N. Każdy z nich zakupił dokładnie jeden kostium, który zamierza zakładać na tegoroczne imprezy karnawałowe. W tym sezonie dostępnych jest C różnych rodzajów kostiumów oznaczonych numerami od 1 do C. Niektórzy przyjaciele Piotra mogli zakupić ten sam rodzaj kostiumu.

Piotr chciałby dowiedzieć się, którzy przyjaciele zakupili takie same kostiumy. W tym celu organizuje on imprezy, na które zaprasza niektórych przyjaciół. Piotr jest świadom, że z rana po imprezie nie będzie w stanie odtworzyć, kto był w jakim kostiumie, ale za to będzie pamiętał, ile różnych rodzajów kostiumów widział podczas imprezy. Teraz zastanawia się, czy może tak podobierać gości na każdą z imprez, aby na końcu móc określić, którzy przyjaciele mają takie same kostiumy. Pomóż Piotrowi.

Interakcja

Twój program powinien komunikować się ze sprawdzaczką przez standardowe wejście i wyjście.

Na początku program powinien wczytać jeden wiersz zawierający liczbę całkowitą N – liczbę przyjaciół.

Dalsza komunikacja ze sprawdzaczką powinna wyglądać następująco. Aby opisać imprezę, należy wypisać pojedynczy wiersz, który zaczyna się od liczby k – liczby zaproszonych przyjaciół ($1 \leqslant k \leqslant N$), po której następuje lista numerów tych przyjaciół. Należy pamiętać o opróżnianiu bufora wyjścia! (np. poprzez użycie `fflush(stdout);` *lub* `cout << endl;`*) Następnie program powinien wczytać odpowiedź sprawdzaczki: pojedynczy wiersz zawierający liczbę różnych rodzajów kostiumów, które przyjaciele mieli na tej imprezie.*

Gdy program określi, którzy przyjaciele mają takie same kostiumy, powinien wypisać jeden wiersz zawierający odpowiedź. Na początku wiersza powinna być liczba 0. Następnie powinno wystąpić N liczb całkowitych oddzielonych pojedynczymi odstępami $c_1, \ldots, c_N$ ($1 \leqslant c_i \leqslant C$ dla wszystkich i). Liczba c_i określa rodzaj kostiumu zakupionego przez przyjaciela o numerze i. Numeracja rodzajów kostiumów nie ma znaczenia; ważne jest tylko, aby takie same kostiumy miały te same numery, a różne kostiumy miały różne numery. Wszystkie numery powinny być pomiędzy 1 a C.

Zaraz po tym należy zakończyć program (poprzez `return 0;`*).*

Ocenianie

$2 \leqslant N \leqslant 150$

Jeśli program potrzebuje P imprez, aby określić podział kostiumów dla testu, wtedy otrzyma:

- *0 punktów, jeśli $11\,500 < P$,*
- *20% punktów, jeśli $3500 < P \leqslant 11\,500$, oraz*
- *wszystkie możliwe punkty, jeśli $P \leqslant 3500$.*

Przykłady

sprawdzaczka	*5*
program	*5 1 2 3 4 5*
sprawdzaczka	*3*
program	*2 2 5*
sprawdzaczka	*1*
program	*2 1 2*
sprawdzaczka	*2*
program	*1 4*
sprawdzaczka	*1*
program	*0 2 1 2 3 1*

W pierwszym przykładzie występuje 5 osób z kostiumami o numerach 1 2 1 3 2. W przykładowej komunikacji wiersze zaczynające się od słowa **sprawdzaczka** *opisują dane wczytywane przez program zawodnika, natomiast wiersze zaczynające się od słowa* **program** *opisują wyjście programu zawodnika. Imprezy opisane w pierwszym przykładzie nie wystarczają, aby z całą pewnością określić podział kostiumów. Tutaj program szczęśliwie zgadł go poprawnie.*

sprawdzaczka	*3*
program	*3 1 2 3*
sprawdzaczka	*2*
program	*2 1 3*
sprawdzaczka	*1*
program	*0 1 2 1*

W drugim przykładzie występują 3 osoby z kostiumami o numerach 1 2 1. Tutaj wystarczyły dwie imprezy, aby określić podział kostiumów.

Ograniczenia

Limit czasu: *1 s*

Dostępna pamięć: *256 MB*

Jakub Pawlewicz, Anna Zych
Tłumaczenie

CEOI 2014, dzień pierwszy, 20.06.2014

Las Fangorn

Pewnego razu zaprawiony w boju krasnolud Gimli, przemierzając ogromne połacie lasu Fangorn, rozejrzał się dookoła i zaniepokoił. Drzewa w lesie wyglądały jakoś podejrzanie. Gimli, chcąc wydostać się ze złowrogiego lasu, postanowił dotrzeć do jednego z obozów na jego granicy. Aby czuć się bezpiecznie na drodze, Gimli musi w każdym momencie wędrówki mieć na oku wszystkie drzewa w lesie. Na szczęście krasnoludy mają doskonały wzrok i widzą nieskończenie daleko w każdym kierunku.

Las Fangorn jest prostokątem F o bokach równoległych do osi układu współrzędnych. Lewy dolny róg ma współrzędne $(0, 0)$, a prawy górny róg ma współrzędne (w, h). Drzewa rosną idealnie pionowo i są na tyle cienkie, że w widoku z góry mogą być reprezentowane jako punkty. Wszystkie drzewa w lesie leżą wewnątrz prostokąta F (żadne drzewo nie znajduje się na brzegu F). Z kolei wszystkie obozy leżą na brzegu F. Gimli widzi drzewo wtedy i tylko wtedy, gdy żadne inne drzewo nie leży dokładnie na odcinku łączącym pozycje Gimliego i drzewa.

Z każdego obozu Gimli widzi wszystkie drzewa. Na początku Gimli znajduje się w lesie (w jego wnętrzu, nie na brzegu), a jego pozycja nie pokrywa się z pozycją żadnego drzewa (krasnoludy nie chodzą po drzewach). Las jest bardzo stary i żadne trzy drzewa nie leżą na jednej prostej.

Gimli może bezpiecznie dotrzeć do obozu c, jeśli istnieje ścieżka γ prowadząca z jego początkowej pozycji do pozycji obozu c, taka że z dowolnego punktu na γ widać wszystkie drzewa. Pomóż Gimliemu i napisz program, który znajdzie wszystkie obozy, do których Gimli może bezpiecznie dotrzeć.

Wejście

W pierwszym wierszu wejścia znajdują się dwie liczby całkowite dodatnie w i h, które wyznaczają prostokąt F. Drugi wiersz zawiera dwie liczby całkowite dodatnie x_G i y_G – współrzędne początkowej pozycji Gimliego.

Trzeci wiersz zawiera liczbę obozów C. Kolejne C wierszy opisuje obozy: i-ty z tych wierszy zawiera dwie liczby całkowite x_i oraz y_i będące współrzędnymi obozu i.

Następny wiersz podaje liczbę drzew N. Kolejne N wierszy opisuje drzewa; każdy z nich zawiera dwie liczby całkowite x oraz y będące współrzędnymi drzewa. Żadne dwa z tych wierszy nie są identyczne.

Wyjście

Pierwszy wiersz wyjścia powinien podawać liczbę m obozów, do których Gimli może bezpiecznie dotrzeć. Jeśli $m \neq 0$, kolejny wiersz powinien podawać numery tych obozów w porządku rosnącym.

Ocenianie

We wszystkich testach zachodzi $3 \leqslant N \leqslant 2000$, $1 \leqslant C \leqslant 10\,000$ *oraz* $0 < w, h \leqslant 10^9$.

Podzadanie 1 (30 punktów): $N \leqslant 70$, $C \leqslant 5$

Podzadanie 2 (20 punktów): *Drzewa są wierzchołkami wielokąta wypukłego oraz* $N \leqslant 200$, $C \leqslant 5$.

Podzadanie 3 (30 punktów): $C \leqslant 5$

Podzadanie 4 (20 punktów): *brak dodatkowych ograniczeń*

Przykłady

Dla danych wejściowych:

```
9 4
1 2
3
7 4
1 4
0 2
4
1 1
6 1
3 3
8 3
```

poprawnym wynikiem jest:

```
2
1 3
```

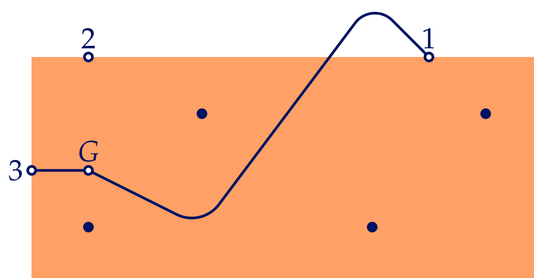
Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
0
```

Oto ilustracja do pierwszego przykładu razem z bezpiecznymi ścieżkami z pozycji początkowej Gimliego do wszystkich bezpiecznie osiągalnych obozów:



Zauważ, że Gimli na swojej ścieżce może opuścić las. Niemniej jednak, nawet będąc poza obszarem lasu, musi cały czas widzieć wszystkie drzewa.

Ograniczenia

Limit czasu: *3 s*

Dostępna pamięć: *64 MB*

Jakub Pawlewicz, Anna Zych
Tłumaczenie

CEOI 2014, dzień pierwszy, 20.06.2014

Pytanie

A i B zamierzają włamać się do ściśle tajnego laboratorium. Niestety drzwi są wyposażone w system zabezpieczający. System ten w chwili wejścia do laboratorium zadaje pytanie (które dla prostoty zakodowane jest poprzez liczbę całkowitą q, $1 \leqslant q \leqslant N$), na które należy odpowiedzieć „yes" lub „no". Jeśli odpowiedź jest poprawna, system otwiera drzwi, w przeciwnym razie włącza się alarm. Zarówno A jak i B wiedzą, że pytanie q zadawane przez system to zawsze x lub y ($x \neq y$), przy czym poprawna odpowiedź na x to „yes", a na y to „no".

Jednakże, planując napad, A i B nie mogli przypomnieć sobie wartości x i y. Toteż postanowili, że B uda się do wejścia i spróbuje odpowiedzieć na pytanie systemu, podczas gdy A przyczai się w pobliżu, aby zapewnić potencjalną drogę ucieczki.

Lecz wtem, jak tylko B podszedł do drzwi i usłyszał pytanie, A przypomniał sobie zarówno x i y, jak i poprawne odpowiedzi. Niestety, odległość dzieląca go od B nie pozwala mu przekazać B dokładnych instrukcji. Może jedynie wykrzyknąć w kierunku B jedną dodatnią liczbę całkowitą h. Tak więc A spróbuje zakodować w h wystarczającą ilość informacji, jakiej B potrzebuje do udzielenia poprawnej odpowiedzi.

Pomóż A i B zaplanować zawczasu system komunikacji, który pomoże im w takiej sytuacji. Napisz program, który może zagrać rolę zarówno A jak i B. Twój program powinien:

1) *dla zadanych wartości N, x, y pomóc A i powiedzieć, jaką liczbę h krzyknąć do B, oraz*

2) *dla zadanych wartości N, q, h pomóc B i podać mu odpowiedź na pytanie systemu.*

Twój program będzie testowany w następujący sposób. Wpierw, program zostanie poproszony o pomoc dla A i wygenerowanie liczb h dla pewnej liczby testów (jeden test odpowiada jednemu pytaniu). Następnie, program zostanie poproszony o pomoc dla B – będzie miał wówczas do dyspozycji liczby h wygenerowane przed chwilą dla zadanych testów. Tak więc program zostanie wywołany dokładnie dwa razy dla każdego zestawu testów.

Wejście

Pierwszy wiersz wejścia zawiera liczbę całkowitą 1 (program gra rolę A) lub 2 (program gra rolę B). Drugi wiersz wejścia zawiera dwie liczby całkowite N i T (N jest takie samo dla wszystkich T testów z wejścia). Reszta wejścia opisuje T testów, po jednym w wierszu.

Jeśli w pierwszym wierszu wejścia jest liczba 1, to i-ty z kolejnych T wierszy zawiera dwie liczby całkowite x oraz y ($1 \leqslant x, y \leqslant N$, $x \neq y$).

Jeśli w pierwszym wierszu wejścia jest liczba 2, to i-ty z kolejnych T wierszy zawiera dwie liczby całkowite q i h ($1 \leqslant q \leqslant N$, $h \geqslant 1$); h jest liczbą, którą program wygenerował dla i-tego testu w pierwszym przebiegu.

Wyjście

Jeśli program pomaga A, to wyjście powinno składać się z T wierszy, gdzie i-ty wiersz zawiera liczbę całkowitą h ($h \geqslant 1$), którą A powinien krzyknąć do B w przypadku testu i.

Jeśli program pomaga B, to wyjście powinno składać się z T wierszy, gdzie i-ty wiersz zawiera słowo `yes` *bądź słowo* `no`*: poprawną odpowiedź na pytanie odpowiadające testowi i.*

Testowanie

W celach testowych możesz użyć dostarczonego skryptu `manager.sh`*.*

W pliku wejściowym do tego skryptu (np. o nazwie `input.txt`*), pierwszy wiersz powinien zawierać liczby N i T. Każdy z kolejnych T wierszy powinien zawierać liczby x, y oraz q opisujące test.*

Aby przetestować swoje rozwiązanie (np. o nazwie pliku wykonywalnego `yoursolution`*), należy użyć polecenia:*

```
./manager.sh ./yoursolution input.txt
```

W wyniku tego polecenia zostaną utworzone pliki `for1.txt` *i* `from1.txt` *(plik wejścia i wyjścia dla części programu pomagającej A) oraz* `for2.txt` *i* `from2.txt` *(to samo dla części pomagającej B). Co więcej, skrypt poinformuje, czy program poprawnie rozwiązał zadane wejście.*

Ograniczenia

We wszystkich plikach wejściowych zachodzi $1 \leqslant N \leqslant 920$ i $1 \leqslant T \leqslant 2\ 000\ 000$.

Punktacja

Punktacja zależy od największej wygenerowanej liczby h we wszystkich testach i plikach wejściowych:

Największe h	**Punkty**
$\geqslant 21$	*0 (widoczne jako zła odpowiedź)*
20	27
19	30
18	33
17	37
16	42
15	50
14	60
13	75
$\leqslant 12$	100

Oczywiście wszystkie odpowiedzi programu w drugim przebiegu dla wszystkich testów muszą być poprawne; w przeciwnym wypadku program dostanie 0 punktów.

Przykład

input.txt	for1.txt	from1.txt	for2.txt	from2.txt
2 01 11	1 5 6 1 2 4 5 1 2 3 5 4 5 5 2	12 2 12 4 2 1	2 5 6 1 12 4 2 2 12 3 4 5 2 2 1	yes yes no yes no no

Zauważ, że wiele innych wyników w pliku `from1.txt` *byłoby poprawnych, natomiast zawartość pliku* `from2.txt`*, jeśli jest poprawna, można wypisać tylko na jeden sposób.*

Ograniczenia

Limit czasu: *3 s*

Dostępna pamięć: *256 MB*

Zużycie czasu i pamięci będzie liczone jako maksimum po obu przebiegach.

Jakub Pawlewicz, Anna Zych
Tłumaczenie

CEOI 2014, dzień drugi, 22.06.2014

007

Szpieg 007 odkryła przebiegły plan jej największego przeciwnika, Doktora Z. de Referowany Wskaźnik-Null (w skrócie, Doktora Nulla): Doktor Null zamierza wyciągnąć wtyczkę jednego z dwóch serwerów CEOI! Doktor Null zaczął wprowadzać swój plan w życie i właśnie podąża w kierunku serwerów. Niestety, oznacza to, że 007 musi w pewnym momencie opuścić przystojnego dżentelmena, z którym akurat je śniadanie, aby ruszyć do akcji.

Zarówno 007 jak i Doktor Null zhakowali satelitarny system śledzenia, w związku z czym w każdej chwili znają wzajemnie swoje pozycje. Obszar obserwowany w tym systemie ma postać **spójnego** *grafu nieskierowanego, a w jego wierzchołkach znajdują się 007, Doktor Null i obydwa serwery.* **Serwery są w sąsiednich wierzchołkach, gdyż znajdują się w tej samej serwerowni.** *Każdy z bohaterów – Doktor Null i 007 – w jednej jednostce czasu może przemieścić się o pojedynczą krawędź. Wyjęcie wtyczki z serwera również zajmuje jedną jednostkę czasu. Oboje wykonują swoje ruchy na przemian, przy czym Doktor Null zaczyna. Każdy z bohaterów może zdecydować się nie wykonywać ruchu, co powoduje natychmiastowe przekazanie kolejki jego przeciwnikowi (bez upływu jednostki czasu).*

007 wygrywa, jeśli złapie ona Doktora Nulla lub nigdy nie dopuści do wyjęcia wtyczki z serwera. 007 łapie Doktora, jak tylko dojdzie do wierzchołka, w którym się on znajduje.

007 chciałaby wiedzieć, ile jeszcze czasu może pozostać na śniadaniu, tak aby być pewną, że wygra niezależnie od tego, co zrobi Doktor Null.

Pomóż jej i napisz program, który wyznaczy największą liczbę jednostek czasu, po upływie których musi ona przerwać śniadanie, aby wciąż jeszcze mogła wygrać z Doktorem. Zauważ, że będąc na śniadaniu, nie może ona złapać Doktora, nawet gdyby znalazł się w tym samym wierzchołku co 007.

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby całkowite N i M – liczbę wierzchołków i liczbę krawędzi w grafie. Wierzchołki są numerowane od 1 do N.

Drugi wiersz zawiera cztery różne liczby całkowite s, d, a, b ($1 \leqslant s, d, a, b \leqslant N$). Oznaczają one następujące wierzchołki w grafie: początkową pozycję 007, początkową pozycję Doktora Nulla oraz pozycje obydwu serwerów.

Następne M wierszy opisuje krawędzie w grafie. Każdy wiersz zawiera dwie liczby całkowite u i v ($1 \leqslant u, v \leqslant N$) – numery wierzchołków połączonych krawędzią.

Wyjście

Wyjście powinno zawierać jedną liczbę całkowitą – największą liczbę jednostek czasu, jaką 007 może pozostać na śniadaniu, tak aby wciąż jeszcze mogła wygrać z Doktorem. W szczególności, jeśli 007 musi wyjść przy jej pierwszym ruchu, należy wypisać 0.

Jeśli 007 nie jest w ogóle w stanie wygrać z Doktorem, należy wypisać -1.

Ograniczenia

Zawsze zachodzi $4 \leqslant N \leqslant 200\,000$ *oraz* $3 \leqslant M \leqslant 600\,000$.

Podzadanie 1 (30 punktów): $N \leqslant 800$ *i* $M \leqslant 1600$

Podzadanie 2 (70 punktów): *brak dodatkowych ograniczeń*

Częściowa punktacja: *Za każdą grupę testów, w której w niektórych testach Twój program wypisze wartość o 1 mniejszą od (nieujemnej) poprawnej odpowiedzi, a poprawną odpowiedź w pozostałych testach, zostanie mu przydzielone 30% punktów przeznaczonych dla tej grupy. Zauważ, że ten wynik częściowy można otrzymać, wypisując* -1*, jeśli 0 było poprawną odpowiedzią.*

Przykłady

Dla danych wejściowych:	*poprawnym wynikiem jest:*

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

Dla danych wejściowych:	*poprawnym wynikiem jest:*

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

Ograniczenia

Limit czasu: *1 s*

Dostępna pamięć: *512 MB*

Jakub Pawlewicz, Anna Zych
Tłumaczenie

Ciasto

Jacek i Agatka nie posiadają się z radości, gdy w domu unosi się zapach świeżo upieczonego ciasta: Jacek uwielbia je jeść, zaś Agatka uwielbia oglądać Jacka spożywającego jej wypieki. Dzisiaj Agatka upiekła makowiec i pokroiła go na N kawałków. Makowiec pieczony był w podłużnej formie: N kawałków leży teraz w szeregu, czekając na Jacka. Kawałki i ich pozycje ponumerowane są od lewej do prawej liczbami od 1 do N: kawałek i leży na pozycji i.

Kolejność, w jakiej Jacek będzie zjadał kolejne kawałki ciasta, zależy od ich smaczności. Dla każdego kawałka i znany jest jego współczynnik smaczności d_i. Jacek zacznie jedzenie od ustalonego kawałka a. W wyniku tego pozycja a stanie się pusta. Następnie, każdy kolejny zjedzony kawałek będzie najmniej smacznym sąsiadem jakiejś pustej pozycji. W ten sposób w każdej chwili puste pozycje będą tworzyć spójny przedział.

Agatka ma jeszcze trochę lukru i zamierza nim poprawić smak niektórych kawałków. Zawsze robi to tak, że osłodzony kawałek staje się jednym z 10 najsmaczniejszych kawałków. Żadne dwa kawałki nigdy nie będą równie smaczne.

Agatka, polewając poszczególne kawałki makowca, chciałaby po każdym kroku móc stwierdzić, ile kawałków zje Jacek przed zjedzeniem danego kawałka, przy założeniu, że nie nastąpiłyby już żadne dalsze zmiany smaczności kawałków.

Pomóż Agatce i napisz program, który obsługuje instrukcje typu „osłodź kawałek” oraz „wyznacz liczbę kawałków, które Jacek zje przed zjedzeniem danego kawałka”.

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby całkowite: liczbę kawałków N oraz liczbę a oznaczającą numer kawałka, który zostanie zjedzony jako pierwszy ($1 \leqslant a \leqslant N$). Drugi wiersz zawiera N parami różnych liczb całkowitych $1 \leqslant d_1, \ldots, d_N \leqslant N$, które są początkowymi współczynnikami smaczności kawałków. Trzeci wiersz zawiera łączną liczbę instrukcji Q. Każdy z kolejnych Q wierszy opisuje pojedynczą instrukcję. Są dwa typy instrukcji:

- `E` i e *(znak „`E`” oraz dwie liczby całkowite $1 \leqslant i \leqslant N$ i $1 \leqslant e \leqslant 10$):*
  taka instrukcja oznacza, że kawałek i staje się e-tym najsmaczniejszym kawałkiem. Liczba kawałków, które przed wykonaniem tej operacji były smaczniejsze od kawałka i, jest równa przynajmniej e.

- `F` b *(znak „`F`” oraz liczba całkowita $1 \leqslant b \leqslant N$):*
  taka instrukcja to zapytanie, ile kawałków Jacek zje przed zjedzeniem kawałka b.

Wyjście

Dla każdej instrukcji typu „F”, w porządku ich występowania na wejściu, należy wypisać pojedynczy wiersz zawierający jedną liczbę: poszukiwaną liczbę kawałków.

Ograniczenia

$N \leqslant 250\,000$, $Q \leqslant 500\,000$

Podzadanie 1 (15 punktów): $N, Q \leqslant 10\,000$

Podzadanie 2 (15 punktów): $N \leqslant 25\,000$, *co najwyżej* 500 *instrukcji typu „F”*

Podzadanie 3 (20 punktów): $Q \leqslant 100\,000$, *co najwyżej* 100 *instrukcji typu „E”*

Podzadanie 4 (50 punktów): *brak dodatkowych ograniczeń*

Przykład

Dla danych wejściowych:

```
5 3
5 1 2 4 3
17
F 1
F 2
F 3
F 4
F 5
E 2 1
F 1
F 2
F 3
F 4
F 5
E 5 2
F 1
F 2
F 3
F 4
F 5
```

poprawnym wynikiem jest:

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

Przed pierwszym polaniem lukrem kawałki zostałyby zjedzone w kolejności 3, 2, 4, 5, 1. Po pierwszym polaniu drugi kawałek jest zbyt smaczny, aby zjeść go tak szybko – kawałki 4 i 5 zostaną zjedzone wcześniej. Natomiast polanie kawałka numer 5 nie ma wpływu na kolejność jedzenia.

Ograniczenia

Limit czasu: *2 s*

Dostępna pamięć: *1024 MB*

Jakub Pawlewicz, Anna Zych
Tłumaczenie

Mur

Od zarania dziejów wyspie Prostokątazji groziły dwa niebezpieczeństwa: powodzie i piraci. Król Prostokątazji postanowił zbudować wielki mur, który ochroniłby wioski znajdujące się na wyspie.

Prostokątazja ma kształt prostokąta. Architekci muru przedstawiają wyspę jako podzieloną na $N \times M$ *kwadratów. Każda wioska leży wewnątrz jednego z kwadratów. Wioska będąca stolicą znajduje się w północno-zachodnim rogu, tzn. wewnątrz lewego górnego kwadratu.*

Mur powinien być zbudowany tak, aby spoza wyspy (czyli z obszaru poza prostokątem) nie było możliwe dotarcie do jakiejkolwiek wioski bez przekraczania muru.

Architekci planują budować mur w następujący sposób. Kolejne segmenty muru będą stawiane wzdłuż boków kwadratów. Pierwszy segment muru zacznie się w lewym górnym rogu prostokąta. Każdy kolejny segment będzie połączony z poprzednim, tzn. musi się z nim stykać w rogu kwadratu (tam gdzie kończy się poprzedni segment). Mur będzie budowany w ten sposób, aż połączy się z powrotem z pierwszym segmentem. Innymi słowy, mur powinien być krzywą zamkniętą składającą się z boków kwadratów.

Z powodu nierównej topografii terenu, z każdym bokiem kwadratu związany jest pewien koszt budowy segmentu muru w tym miejscu. Całkowity koszt budowy muru jest sumą kosztów wybudowania wszystkich segmentów muru. Jeżeli mur przechodzi przez dany segment t *razy, to koszt budowy tego segmentu powinien być policzony* t *razy.*

Król zamierza przeznaczyć możliwie najmniej pieniędzy na budowę muru. Pomóż królowi i napisz program, który dla danych pozycji wiosek na wyspie oraz kosztów budowy segmentów wyznaczy najmniejszy możliwy koszt budowy muru.

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby całkowite N *i* M *– liczbę wierszy i liczbę kolumn prostokąta. Kolejne* N *wierszy opisuje rozmieszczenie wiosek na wyspie. Każdy z tych wierszy składa się z* m *liczby całkowitych 0 lub 1: 0 reprezentuje pusty kwadrat, a 1 oznacza, że w tym kwadracie znajduje się wioska. Pierwsza liczba w pierwszym wierszu jest równa 1.*

Następnie, w kolejnych N *wierszach, z których każdy zawiera* $M + 1$ *dodatnich liczb całkowitych, znajdują się koszty budowy segmentów muru wzdłuż pionowych boków kwadratów.*

Wreszcie w ostatnich $N + 1$ *wierszach, z których każdy zawiera* M *dodatnich liczb całkowitych, znajdują się koszty budowy segmentów muru wzdłuż poziomych boków kwadratów.*

Wyjście

Wyjście powinno zawierać jedną liczbę całkowitą – najmniejszy możliwy koszt budowy muru.

Ograniczenia

Zachodzi $1 \leqslant N, M \leqslant 400$. Dowolny koszt budowy segmentu v spełnia $1 \leqslant v \leqslant 10^9$, jednakże do przechowywania ostatecznego wyniku może być potrzebne użycie 64-bitowej zmiennej całkowitej.

Podzadanie 1 (30 punktów): *liczba wiosek nie przekracza 10 oraz $N, M \leqslant 40$*

Podzadanie 2 (30 punktów): *$N, M \leqslant 40$*

Podzadanie 3 (40 punktów): *brak dodatkowych ograniczeń*

Przykłady

Dla danych wejściowych:

```
3 3
1 0 0
1 0 0
0 0 1
1 4 9 4
1 6 6 6
1 2 2 9
1 1 1
4 4 4
2 4 2
6 6 6
```

poprawnym wynikiem jest:

```
38
```

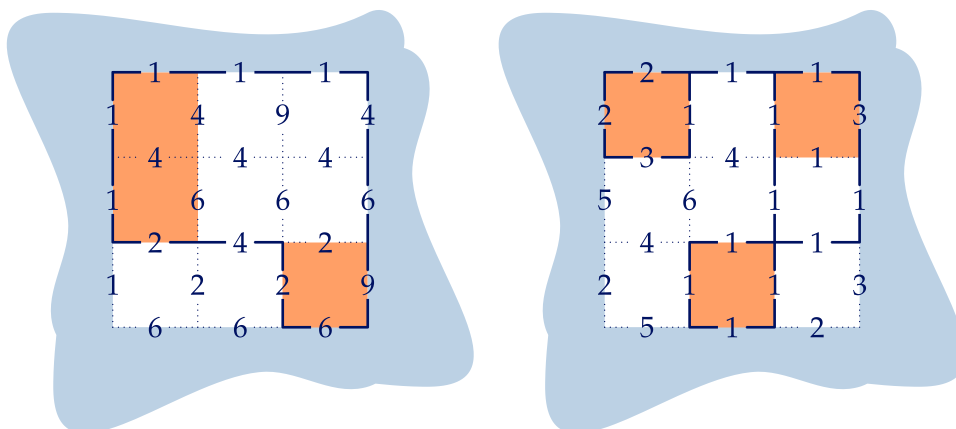
Dla danych wejściowych:

```
3 3
1 0 1
0 0 0
0 1 0
2 1 1 3
5 6 1 1
2 1 1 3
2 1 1
3 4 1
4 1 1
5 1 2
```

poprawnym wynikiem jest:

```
22
```

Sytuacje z przykładów przedstawione są na rysunkach poniżej. Kwadraty zawierające wioski są pokolorowane. Pogrubioną linią oznaczono przykładowy najtańszy mur.



Ograniczenia

Limit czasu: *2 s*

Dostępna pamięć: *1024 MB*

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] *XX Olimpiada Informatyczna 2012/2013.* Warszawa, 2013.

[21] A. V. Aho, J. E. Hopcroft, J. D. Ullman. *Projektowanie i analiza algorytmów komputerowych.* PWN, Warszawa, 1983.

[22] L. Banachowski, A. Kreczmar. *Elementy analizy algorytmów.* WNT, Warszawa, 1982.

[23] L. Banachowski, K. Diks, W. Rytter. *Algorytmy i struktury danych.* WNT, Warszawa, 1996.

[24] J. Bentley. *Perełki oprogramowania.* WNT, Warszawa, 1992.

[25] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Wprowadzenie do algorytmów.* WNT, Warszawa, 2004.

[26] M. de Berg, M. van Kreveld, M. Overmars. *Geometria obliczeniowa. Algorytmy i zastosowania.* WNT, Warszawa, 2007.

[27] R. L. Graham, D. E. Knuth, O. Patashnik. *Matematyka konkretna.* PWN, Warszawa, 1996.

[28] D. Harel. *Algorytmika. Rzecz o istocie informatyki.* WNT, Warszawa, 1992.

[29] D. E. Knuth. *Sztuka programowania.* WNT, Warszawa, 2002.

[30] W. Lipski. *Kombinatoryka dla programistów.* WNT, Warszawa, 2004.

[31] E. M. Reingold, J. Nievergelt, N. Deo. *Algorytmy kombinatoryczne.* WNT, Warszawa, 1985.

[32] K. A. Ross, C. R. B. Wright. *Matematyka dyskretna.* PWN, Warszawa, 1996.

[33] R. Sedgewick. *Algorytmy w C++. Grafy.* RM, 2003.

[34] S. S. Skiena, M. A. Revilla. *Wyzwania programistyczne.* WSiP, Warszawa, 2004.

[35] P. Stańczyk. *Algorytmika praktyczna. Nie tylko dla mistrzów.* PWN, Warszawa, 2009.

[36] M. M. Sysło. *Algorytmy.* WSiP, Warszawa, 1998.

[37] M. M. Sysło. *Piramidy, szyszki i inne konstrukcje algorytmiczne.* WSiP, Warszawa, 1998.

[38] R. J. Wilson. *Wprowadzenie do teorii grafów.* PWN, Warszawa, 1998.

[39] N. Wirth. *Algorytmy + struktury danych = programy.* WNT, Warszawa, 1999.

[40] *W poszukiwaniu wyzwań. Wybór zadań z konkursów programistycznych Uniwersytetu Warszawskiego.* Warszawa, 2012.

[41] Reinhard Diestel. *Graph Theory.* Graduate Texts in Mathematics. Springer, 2010.

[42] Marek Cygan, Fedor Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, Saket Saurabh. *Parameterized Algorithms.* Springer, 2015 (w druku).

Niniejsza publikacja stanowi wyczerpujące źródło informacji o zawodach XXI Olimpiady Informatycznej, przeprowadzonych w roku szkolnym 2013/2014. 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 XXVI Międzynarodowej Olimpiady Informatycznej, XX Bałtyckiej Olimpiady Informatycznej, a także XX i XXI Olimpiady Informatycznej Krajów Europy Środkowej.

XXI 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.

Sponsorzy Olimpiady:

ISBN 978-83-64292-01-9