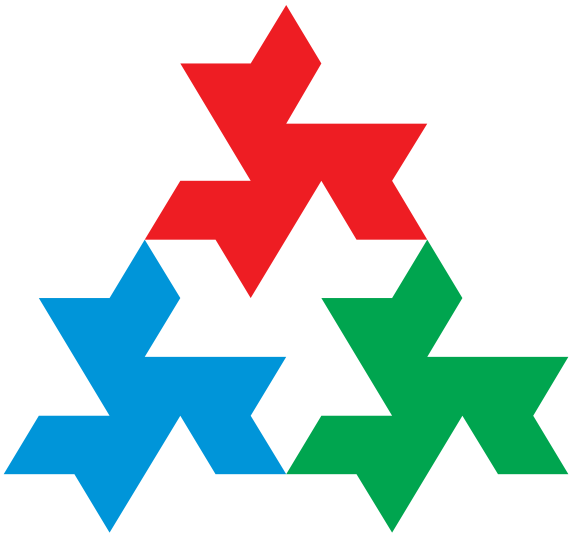
MINISTERSTWO EDUKACJI NARODOWEJ
FUNDACJA ROZWOJU INFORMATYKI
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ



XVIII OLIMPIADA INFORMATYCZNA
2010/2011

Olimpiada Informatyczna jest organizowana przy współudziale

WARSZAWA, 2011

MINISTERSTWO EDUKACJI NARODOWEJ
FUNDACJA ROZWOJU INFORMATYKI
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

XVIII OLIMPIADA INFORMATYCZNA
2010/2011

WARSZAWA, 2011

Autorzy tekstów:

prof. dr hab. Krzysztof Diks
mgr Tomasz Idziaszek
mgr Marian M. Kędzierski
Paweł Mechliński
Piotr Niedźwiedź
Jakub Pachocki
dr Jakub Pawlewicz
mgr Michał Pilipczuk
prof. dr hab. Wojciech Rytter
mgr Bartosz Szreder
Jacek Tomasiewicz
dr Tomasz Waleń
dr Jakub Wojtaszczyk

Autorzy programów:

Mateusz Baranowski
Dawid Dąbrowski
Bartosz Górski
Adam Karczmarz
Tomasz Kociumaka
Alan Kutniewski
Jacek Migdał
Błażej Osiński
Jakub Pachocki
dr Paweł Parys
mgr Marcin Pilipczuk
mgr Juliusz Sompolski
Zbigniew Wojna
dr Jakub Wojtaszczyk
mgr Filip Wolski
Bartłomiej Wołowiec

Opracowanie i redakcja:

Tomasz Kociumaka
dr Marcin Kubica
mgr Jakub Radoszewski

Tłumaczenie treści zadań:

dr Lech Duraj
dr Grzegorz Herman
prof. dr hab. Paweł Idziak
mgr Jakub Łącki
Mirosław Michalski

Skład:

Tomasz Kociumaka
mgr Jakub Radoszewski

Pozycja dotowana przez Ministerstwo Edukacji Narodowej.

Druk książki został sfinansowany przez 

ISBN 978–83–930856–7–5

Spis treści

Krzysztof Diks

Wstęp

Drogi Czytelniku!

Przekazujemy do rąk uczestników i sympatyków Olimpiady Informatycznej kolejną „niebieską książeczkę", zawierającą sprawozdanie i zadania z XVIII Olimpiady Informatycznej. Zadania olimpijskie nie są łatwe, ale dlatego ich rozwiązywanie daje nie tylko olbrzymią satysfakcję, lecz także umożliwia zdobycie wiedzy i umiejętności, które okazują się potem przydatne w życiu zawodowym. O tym, że takie podejście przynosi owoce, świadczą wyniki naszych olimpijczyków na zawodach międzynarodowych, znakomite osiągnięcia podczas studiów oraz zainteresowanie pracodawców z najwyższej, informatycznej półki.

Podczas XXIII Międzynarodowej Olimpiady Informatycznej, która odbyła się w roku 2011 w Tajlandii, Polacy zdobyli dwa złote medale (Jan Kanty Milczek i Piotr Bejda), medal srebrny (Łukasz Jocz) oraz medal brązowy (Krzysztof Leszczyński).

Znakomitymi wynikami zakończyła się XVIII Olimpiada Krajów Europy Środkowej, która odbyła się w Gdyni. Sądząc po wynikach, Polacy nie okazali się gościnni. Pierwsze dwa miejsca i złote medale zdobyli Krzysztof Pszeniczny i Jan Kanty Milczek. Złoty medal wywalczył także Piotr Bejda. Ponadto Wiktor Kuropatwa i Krzysztof Leszczyński zdobyli medale srebrne, a Łukasz Jocz, Marcin Smulewicz i Mateusz Kopeć medale brązowe.

Sukcesy nie ominęły naszych reprezentantów podczas XVII Bałtyckiej Olimpiady Informatycznej. Pierwsze miejsce zajął Mateusz Gołębiewski (złoty medal), drugie — Bartosz Tarnawski (złoty medal), czwarte — Mateusz Kopeć (złoty medal). Pozostali nasi reprezentanci, Piotr Bejda, Wiktor Kuropatwa i Krzysztof Pszeniczny, zdobyli odpowiednio dwa medale srebrne i jeden brązowy.

Mam nadzieję, że ta książeczka przyczyni się do tego, że kolejne pokolenie podąży śladami finalistów XVIII Olimpiady Informatycznej. Życzę wszystkim satysfakcji ze studiowania tych materiałów.

Na koniec chciałbym gorąco podziękować wszystkim zaangażowanym w organizację Olimpiady, często nieznanym z imienia i nazwiska, a bez których organizacja zawodów nie byłaby możliwa. Słowa podziękowania należą się też wypróbowanym instytucjonalnym przyjaciołom Olimpiady, którzy wspierają nas finansowo i organizacyjnie: Ministerstwu Edukacji Narodowej, firmie Asseco Poland SA, Ogólnopolskiej Fundacji Edukacji Komputerowej oraz Fundacji Rozwoju Informatyki.

Krzysztof Diks

Sprawozdanie z przebiegu XVIII Olimpiady Informatycznej w roku szkolnym 2010/2011

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 XVIII 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.

5 października 2010 roku rozesłano do 3432 szkół i zespołów szkół młodzieżowych ponadgimnazjalnych plakaty informujące o rozpoczęciu XVIII Olimpiady oraz promujące sukcesy młodych polskich informatyków. Zawody I stopnia rozpoczęły się 18 października 2010 roku. Ostatecznym terminem nadsyłania prac konkursowych był 15 listopada 2010 roku.

Zawody II i III stopnia były dwudniowymi sesjami stacjonarnymi, poprzedzonymi jednodniowymi sesjami próbnymi. Zawody II stopnia odbyły się w ośmiu okręgach: Białymstoku, Gliwicach, Krakowie, Rzeszowie, Sopocie, Toruniu, Warszawie i Wrocławiu w dniach 8–10 lutego 2011 roku, natomiast zawody III stopnia odbyły się w siedzibie firm Combidata Poland SA w Sopocie, w dniach 5–9 kwietnia 2011 roku.

Uroczystość zakończenia XVIII Olimpiady Informatycznej odbyła się 9 kwietnia 2011 roku w siedzibie firm Combidata Poland SA i Asseco Poland SA w Gdyni przy ul. Podolskiej 21.

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:
dr Marcin Kubica (Uniwersytet Warszawski)

kierownik Jury:
mgr 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 Barbara Klunder (Uniwersytet Mikołaja Kopernika w Toruniu)
mgr Anna Beata Kwiatkowska (Gimnazjum i Liceum Akademickie w Toruniu)
prof. dr hab. Krzysztof Loryś (Uniwersytet Wrocławski)
prof. dr hab. Jan Madey (Uniwersytet Warszawski)
prof. dr hab. Wojciech Rytter (Uniwersytet Warszawski)
dr hab. Krzysztof Stencel, prof. UW (Uniwersytet Warszawski)
prof. dr hab. Maciej Sysło (Uniwersytet Wrocławski)
dr Maciej Ślusarek (Uniwersytet Jagielloński)
mgr Krzysztof J. Święcicki
dr Tomasz Waleń (Uniwersytet Warszawski)
dr hab. inż. Stanisław Waligórski, prof. UW (Uniwersytet Warszawski)

sekretarz Komitetu Głównego:
Monika Kozłowska-Zając (OEIiZK)

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

Komitet Główny odbył 4 posiedzenia.

Komitety okręgowe

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

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

Komitet Okręgowy we Wrocławiu
przewodniczący:
prof. dr hab. Krzysztof Loryś (Uniwersytet Wrocławski)
zastępca przewodniczącego:
dr Helena Krupicka (Uniwersytet Wrocławski)
sekretarz:
inż. Maria Woźniak (Uniwersytet Wrocławski)
członkowie:
dr hab. Tomasz Jurdziński (Uniwersytet Wrocławski)
dr Przemysława Kanarek (Uniwersytet Wrocławski)
dr Witold Karczewski (Uniwersytet Wrocławski)

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

Komitet Okręgowy w Toruniu
przewodniczący:
prof. dr hab. Adam Ochmański (Uniwersytet Mikołaja Kopernika w Toruniu)
zastępca przewodniczącego:
dr Barbara Klunder (Uniwersytet Mikołaja Kopernika w Toruniu)
sekretarz:
mgr inż. Rafał Kluszczyński (Uniwersytet Mikołaja Kopernika w Toruniu)
członkowie:
mgr Anna Beata Kwiatkowska (Gimnazjum i Liceum Akademickie w Toruniu)
mgr Robert Mroczkowski (Uniwersytet Mikołaja Kopernika w Toruniu)
mgr Radosław Rudnicki (Uniwersytet Mikołaja Kopernika w Toruniu)

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

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 Dorosik (Politechnika Śląska w Gliwicach)
mgr inż. Dariusz Myszor (Politechnika Śląska w Gliwicach)

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

Komitet Okręgowy w Krakowie
przewodniczący:
prof. dr hab. Paweł Idziak (Uniwersytet Jagielloński)

zastępca przewodniczącego:
dr Maciej Ślusarek (Uniwersytet Jagielloński)
sekretarz:
mgr Monika Gillert (Uniwersytet Jagielloński)
członkowie:
mgr Henryk Białek (emerytowany pracownik Małopolskiego Kuratorium Oświaty)
dr Iwona Cieślik (Uniwersytet Jagielloński)
mgr Grzegorz Gutowski (Uniwersytet Jagielloński)
Marek Wróbel (student Uniwersytetu Jagiellońskiego)

Siedzibą Komitetu Okręgowego w Krakowie jest Katedra Algorytmiki Uniwersytetu Jagiellońskiego, ul. Łojasiewicza 6.

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

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

Komitet Okręgowy w Poznaniu
przewodniczący:
dr hab. inż. Robert Wrembel, prof. PP (Politechnika Poznańska)
zastępca przewodniczącego:
mgr inż. Szymon Wąsik (Politechnika Poznańska)
sekretarz:
inż. Michał Połetek (Politechnika Poznańska)
członkowie:
inż. Mariola Galas (Politechnika Poznańska)
mgr inż. Piotr Gawron (Politechnika Poznańska)
dr Maciej Machowiak (Politechnika Poznańska)
dr Jacek Marciniak (Uniwersytet Adama Mickiewicza w Poznaniu)
dr inż. Maciej Miłostan (Politechnika Poznańska)

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

Pomorski Komitet Okręgowy
przewodniczący:
prof. dr hab. inż. Marek Kubale (Politechnika Gdańska)
zastępca przewodniczącego:
dr hab. Andrzej Szepietowski, prof. UG (Uniwersytet Gdański)
sekretarz:
dr inż. Krzysztof Ocetkiewicz (Politechnika Gdańska)
członkowie:
dr inż. Dariusz Dereniowski (Politechnika Gdańska)
dr inż. Adrian Kosowski (Politechnika Gdańska)
dr inż. Michał Małafiejski (Politechnika Gdańska)
mgr inż. Ryszard Szubartowski (III Liceum Ogólnokształcące im. Marynarki Wojennej RP w Gdyni)
dr Paweł Żyliński (Uniwersytet Gdański)

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

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:

Maciej Andrejczuk
Mateusz Baranowski
Arkadiusz Betkier
Dawid Dąbrowski
Konrad Gołuchowski
Bartosz Górski
mgr Tomasz Idziaszek
Adam Karczmarz
Tomasz Kociumaka
Alan Kutniewski
Marek Marczykowski
Mirosław Michalski
Jacek Migdał
Piotr Niedźwiedź
dr Paweł Parys
mgr Juliusz Sompolski
Zbigniew Wojna
dr Jakub Wojtaszczyk
Bartłomiej Wołowiec

ZAWODY I STOPNIA

W zawodach I stopnia XVIII Olimpiady Informatycznej wzięło udział 932 zawodników. Decyzją Komitetu Głównego zdyskwalifikowano 15 zawodników. Powodem dys-

kwalifikacji była niesamodzielność rozwiązań zadań konkursowych. Sklasyfikowano 917 zawodników.

Decyzją Komitetu Głównego do zawodów zostało dopuszczonych 46 uczniów gimnazjów. Pochodzili oni z następujących szkół:

Gimnazjum nr 24 z Oddziałami Dwujęzycznymi (ZSO nr 1)	Gdynia	8 uczniów
Gimnazjum nr 50 (ZSO nr 6)	Bydgoszcz	5
Gimnazjum z Oddziałami Dwujęzycznymi nr 42	Warszawa	3
Gimnazjum Akademickie (ZS UMK Gimnazjum i Liceum Akademickie)	Toruń	2
Gimnazjum nr 13 im. Stanisława Staszica (ZS nr 82)	Warszawa	2
Gimnazjum nr 58 z Oddziałami Dwujęzycznymi im. Władysława IV (ZS nr 15)	Warszawa	2
Społecznie Gimnazjum nr 2 STO	Białystok	1 uczeń
Społecznie Gimnazjum nr 8 STO	Białystok	1
Gimnazjum nr 9 im. Powstańców Wielkopolskich	Bydgoszcz	1
Gimnazjum Jezuitów im. św. Stanisława Kostki	Gdynia	1
Gimnazjum FILOMATA	Gliwice	1
Gimnazjum nr 1 im. św. Jadwigi Królowej	Jasło	1
Gimnazjum nr 2 im. Adama Mickiewicza	Kraków	1
Gimnazjum nr 16 im. Króla Stefana Batorego	Kraków	1
Gimnazjum nr 24 (ZS nr 7)	Lublin	1
Gimnazjum nr 8 im. Tadeusza Kościuszki	Łódź	1
Gimnazjum nr 4 im. Marii Skłodowskiej-Curie (ZSO nr 2)	Olsztyn	1
Gimnazjum nr 13 z Oddziałami Dwujęzycznymi (ZS nr 3)	Płock	1
Gimnazjum nr 3 im. Jana Kochanowskiego	Radom	1
Gimnazjum im. Krzysztofa Kamila Baczyńskiego	Sędziszów Małopolski	1
Społeczne Gimnazjum STO	Siedlce	1
Gimnazjum nr 16 (ZSO nr 7)	Szczecin	1
Gimnazjum nr 10 (ZSO nr 1)	Tarnów	1
Gimnazjum nr 77 im. Ignacego Domeyki (ZS nr 51)	Warszawa	1
Gimnazjum nr 123 z Oddziałami Dwujęzycznymi i Oddziałami Integracyjnymi im. Jana Pawła II	Warszawa	1
Społeczne Gimnazjum „Dwójka” nr 45 (Zespół Szkół Społecznych STO)	Warszawa	1
Społeczne Gimnazjum nr 333 STO	Warszawa	1
„Żagle” Gimnazjum Stowarzyszenia STERNIK	Warszawa	1
Gimnazjum nr 49 z Oddziałami Dwujęzycznymi (ZS nr 14)	Wrocław	1
Gimnazjum im. Mikołaja Kopernika	Zalasewo	1

Kolejność województw pod względem liczby uczestników była następująca:

mazowieckie	147 zawodników	wielkopolskie	47
małopolskie	146	lubelskie	33
dolnośląskie	95	zachodniopomorskie	31
śląskie	91	łódzkie	26
pomorskie	72	świętokrzyskie	17
podlaskie	68	warmińsko-mazurskie	17
kujawsko-pomorskie	62	lubuskie	8
podkarpackie	49	opolskie	8

W zawodach I stopnia najliczniej reprezentowane były szkoły:

V LO im. Augusta Witkowskiego	Kraków	61 uczniów
XIV LO im. Stanisława Staszica (ZS nr 82)	Warszawa	61
XIV LO im. Polonii Belgijskiej (ZS nr 14)	Wrocław	41
I LO im. Adama Mickiewicza	Białystok	35
III LO im. Marynarki Wojennej RP (ZSO nr 1)	Gdynia	32
VI LO im. Jana i Jędrzeja Śniadeckich (ZSO nr 6)	Bydgoszcz	20
XIII Liceum Ogólnokształcące (ZSO nr 7)	Szczecin	20
I LO im. Tadeusza Kościuszki	Legnica	19
Liceum Akademickie (ZS UMK Gimnazjum i Liceum Akademickie)	Toruń	15
V Liceum Ogólnokształcące	Bielsko-Biała	14
VIII LO im. Adama Mickiewicza	Poznań	14
III LO im. Adama Mickiewicza	Wrocław	14
II LO im. Króla Jana III Sobieskiego	Kraków	13
V LO im. Stefana Żeromskiego	Gdańsk	11
VIII LO im. Marii Skłodowskiej-Curie	Katowice	10
I LO im. Bartłomieja Nowodworskiego	Kraków	9
VI LO im. Jana Kochanowskiego (ZSO nr 6)	Radom	9
Gimnazjum nr 24 z Oddziałami Dwujęzycznymi (ZSO nr 1)	Gdynia	8
I LO im. Mikołaja Kopernika	Krosno	8
III LO im. Adama Mickiewicza	Tarnów	7
Technikum nr 3 im. Rotmistrza Witolda Pileckiego (Zespół Szkół Technicznych)	Wodzisław Śląski	6
X LO im. Stefanii Sempołowskiej	Wrocław	6
Gimnazjum nr 50 (ZSO nr 6)	Bydgoszcz	5
VI LO im. Wacława Sierpińskiego	Gdynia	5
II LO im. Czesława Miłosza	Jaworzno	5
II LO im. Mikołaja Kopernika	Kędzierzyn-Koźle	5
II LO im. Hetmana Jana Zamoyskiego	Lublin	5
III LO im. św. Jana Kantego (ZSO nr 3)	Poznań	5
I LO im. Marii Konopnickiej	Suwałki	5
II LO im. Hetmana Jana Tarnowskiego (ZSO nr 2)	Tarnów	5
VIII LO im. Króla Władysława IV (ZS nr 15)	Warszawa	5
I LO im. Edwarda Dembowskiego	Zielona Góra	5

VI LO im. Króla Zygmunta Augusta	Białystok	4 uczniów
Technikum nr 1 (Zespół Szkół Techniczno-Informatycznych)	Gliwice	4
I LO im. Króla Stanisława Leszczyńskiego	Jasło	4
I LO im. Marii Skłodowskiej-Curie	Maków Maz.	4
I LO im. Jana Długosza (ZSO nr 1)	Nowy Sącz	4
II LO im. Cypriana Kamila Norwida	Ostrołęka	4
LO im. Mikołaja Kopernika	Ostrów Maz.	4
Technikum Komunikacji (Zespół Szkół Komunikacji im. Hipolita Cegielskiego)	Poznań	4
I LO im. ks. Stanisława Konarskiego	Rzeszów	4
I LO im. Jana Ignacego Kraszewskiego	Biała Podlaska	3
LO im. Stanisława Wyspiańskiego (ZSO)	Biecz	3
I LO im. Mikołaja Kopernika	Bielsko-Biała	3
Technikum nr 3 (Zespół Szkół Elektronicznych, Elektrycznych i Mechanicznych)	Bielsko-Biała	3
I LO im. Stanisława Staszica	Chrzanów	3
IX LO	Gdańsk	3
I LO im. Edwarda Dembowskiego (ZSO nr 10)	Gliwice	3
II LO im. Króla Jana III Sobieskiego	Grudziądz	3
I LO im. Stefana Żeromskiego (ZSO nr 1)	Jelenia Góra	3
I LO z Oddziałami Dwujęzycznymi im. Mikołaja Kopernika (ZSO nr 1)	Katowice	3
IV LO im. Hanki Sawickiej (Zespół Szkół Ponadpodstawowych nr 2)	Kielce	3
LO im. Kazimierza Wielkiego	Koło	3
III LO im. Jana Kochanowskiego	Kraków	3
VIII LO im. Stanisława Wyspiańskiego	Kraków	3
I LO im. Tadeusza Kościuszki (ZSO)	Łomża	3
Publiczne LO Politechniki Łódzkiej	Łódź	3
Technikum (Michalicki Zespół Szkół Ponadgimnazjalnych im. ks. Bronisława Markiewicza)	Miejsce Piastowe	3
II LO im. Mikołaja Kopernika	Mielec	3
IV LO im. Marii Skłodowskiej-Curie (ZSO nr 2)	Olsztyn	3
Technikum nr 6 (Zespół Szkół Elektronicznych)	Rzeszów	3
I LO im. Bolesława Prusa	Siedlce	3
I LO im. Kazimierza Jagiellończyka	Sieradz	3
II LO z Oddziałami Dwujęzycznymi im. Adama Mickiewicza (ZSO nr 2)	Słupsk	3
IV LO im. Bolesława Prusa	Szczecin	3
Gimnazjum z Oddziałami Dwujęzycznymi nr 42	Warszawa	3
IX LO im. Klementyny Hoffmanowej	Warszawa	3
XXVIII LO im. Jana Kochanowskiego	Warszawa	3

Najliczniej reprezentowane były miasta:

Kraków	100 uczestników	Jasło	6
Warszawa	99	Łomża	6
Wrocław	64	Rybnik	6
Białystok	48	Suwałki	6
Gdynia	46	Wodzisław Śląski	6
Bydgoszcz	30	Biała Podlaska	5
Poznań	28	Jaworzno	5
Szczecin	26	Kędzierzyn-Koźle	5
Toruń	23	Ostrołęka	5
Legnica	20	Słupsk	5
Bielsko-Biała	19	Maków Mazowiecki	4
Gdańsk	17	Mielec	4
Tarnów	17	Ostrów Mazowiecka	4
Katowice	15	Siedlce	4
Rzeszów	15	Biecz	3
Radom	14	Chorzów	3
Kielce	13	Chrzanów	3
Gliwice	10	Grudziądz	3
Lublin	10	Jelenia Góra	3
Łódź	10	Koło	3
Krosno	9	Miejsce Piastowe	3
Nowy Sącz	9	Nowy Targ	3
Olsztyn	7	Płock	3
Zielona Góra	7	Sieradz	3
Chełm	6	Tarnowskie Góry	3
Częstochowa	6		

Zawodnicy uczęszczali do następujących klas:

do klasy I gimnazjum	3
do klasy II gimnazjum	6
do klasy III gimnazjum	37
do klasy I szkoły ponadgimnazjalnej	146
do klasy II szkoły ponadgimnazjalnej	305
do klasy III szkoły ponadgimnazjalnej	381
do klasy IV szkoły ponadgimnazjalnej	39

W zawodach I stopnia zawodnicy mieli do rozwiązania pięć zadań:

- „Konspiracja” autorstwa Jakuba Wojtaszczyka
- „Lizak” autorstwa Jakuba Pachockiego
- „Piorunochron” autorstwa Piotra Niedźwiedzia
- „Przekładanka” autorstwa Krzysztofa Diksa i Wojciecha Ryttera
- „Wykres” autorstwa Jakuba Pawlewicza

każde oceniane maksymalnie po 100 punktów.

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

- **KON** — Konspiracja

	KON	
	liczba zawodników	czyli
100 pkt.	118	12,97%
75–99 pkt.	54	5,89%
50–74 pkt.	17	1,85%
1–49 pkt.	64	6,98%
0 pkt.	123	13,41%
brak rozwiązania	541	59,00%

- **LIZ** — Lizak

	LIZ	
	liczba zawodników	czyli
100 pkt.	281	30,64%
75–99 pkt.	33	3,60%
50–74 pkt.	12	1,31%
1–49 pkt.	400	43,62%
0 pkt.	146	15,92%
brak rozwiązania	45	4,91%

- **PIO** — Piorunochron

	PIO	
	liczba zawodników	czyli
100 pkt.	55	6,0%
75–99 pkt.	20	2,18%
50–74 pkt.	119	12,98%
1–49 pkt.	507	55,29%
0 pkt.	61	6,65%
brak rozwiązania	155	16,90%

- **PRZ** — Przekładanka

	PRZ	
	liczba zawodników	czyli
100 pkt.	139	15,16%
75–99 pkt.	88	9,60%
50–74 pkt.	23	2,51%
1–49 pkt.	83	9,05%
0 pkt.	133	14,50%
brak rozwiązania	451	49,17%

- **WYK** — Wykres

	WYK	
	liczba zawodników	czyli
100 pkt.	4	0,44%
75–99 pkt.	9	0,98%
50–74 pkt.	8	0,87%
1–49 pkt.	15	1,64%
0 pkt.	32	3,49%
brak rozwiązania	849	92,58%

W sumie za wszystkie 5 zadań konkursowych:

SUMA	liczba zawodników	czyli
500 pkt.	2	0,22%
375–499 pkt.	36	3,93%
250–374 pkt.	136	14,83%
125–249 pkt.	200	21,81%
1–124 pkt.	453	49,40%
0 pkt.	90	9,81%

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

ZAWODY II STOPNIA

Do zawodów II stopnia, które odbyły się w dniach 8–10 lutego 2011 roku w siedmiu stałych okręgach i w Białymstoku zakwalifikowano 403 zawodników, którzy osiągnęli w zawodach I stopnia wynik nie mniejszy niż 100 pkt.

Zawodnicy uczęszczali do szkół w następujących województwach:

Białystok	40 uczestników	Sopot	50
Gliwice	36	Toruń	32
Kraków	87	Warszawa	80
Rzeszów	21	Wrocław	57

Siedmiu zawodników nie stawiło się na zawody, w zawodach wzięło udział 396 zawodników. Zawodnicy uczęszczali do szkół w następujących województwach:

małopolskie	79 uczestników	wielkopolskie	12
mazowieckie	71	zachodniopomorskie	10
dolnośląskie	40	łódzkie	9
podlaskie	40	lubelskie	7
śląskie	33	warmińsko-mazurskie	7
kujawsko-pomorskie	31	lubuskie	5
pomorskie	31	świętokrzyskie	5
podkarpackie	15	opolskie	1

W zawodach II stopnia najliczniej reprezentowane były szkoły:

V LO im. Augusta Witkowskiego	Kraków	51 uczniów
XIV LO im. Stanisława Staszica (ZS nr 82)	Warszawa	45
I LO im. Adama Mickiewicza	Białystok	29
III LO im. Marynarki Wojennej RP (ZSO nr 1)	Gdynia	22
XIV LO im. Polonii Belgijskiej (ZS nr 14)	Wrocław	22
VI LO im. Jana i Jędrzeja Śniadeckich (ZSO nr 6)	Bydgoszcz	16
XIII Liceum Ogólnokształcące (ZSO nr 7)	Szczecin	10

Liceum Akademickie (ZS UMK Gimnazjum i Liceum Akademickie)	Toruń	10
II LO im. Króla Jana III Sobieskiego	Kraków	7
III LO im. Adama Mickiewicza	Wrocław	7
V Liceum Ogólnokształcące	Bielsko-Biała	6
VIII LO im. Marii Skłodowskiej-Curie	Katowice	6
VI LO im. Jana Kochanowskiego (ZSO nr 6)	Radom	6
III LO im. Adama Mickiewicza	Tarnów	6
I LO im. Tadeusza Kościuszki	Legnica	5
Gimnazjum nr 24 z Oddziałami Dwujęzycznymi (ZSO nr 1)	Gdynia	4
II LO im. Czesława Miłosza	Jaworzno	4
VIII LO im. Adama Mickiewicza	Poznań	4
I LO im. Marii Konopnickiej	Suwałki	4
VIII LO im. Króla Władysława IV (ZS nr 15)	Warszawa	4
I LO im. Edwarda Dembowskiego	Zielona Góra	4
I LO im. Stefana Żeromskiego (ZSO nr 1)	Jelenia Góra	3
I LO im. Mikołaja Kopernika	Krosno	3
IV LO im. Marii Skłodowskiej-Curie (ZSO nr 2)	Olsztyn	3
III LO im. św. Jana Kantego (ZSO nr 3)	Poznań	3
II LO im. Hetmana Jana Tarnowskiego (ZSO nr 2)	Tarnów	3

Najliczniej reprezentowane były miasta:

Kraków	63 uczestników	Olsztyn	6
Warszawa	58	Rzeszów	6
Białystok	34	Kielce	5
Wrocław	32	Legnica	5
Gdynia	27	Jaworzno	4
Bydgoszcz	20	Łódź	4
Tarnów	11	Suwałki	4
Toruń	11	Biała Podlaska	3
Szczecin	10	Gdańsk	3
Katowice	9	Gliwice	3
Poznań	8	Jasło	3
Radom	7	Jelenia Góra	3
Bielsko-Biała	6	Krosno	3

Zawodnicy uczęszczali do następujących klas:

do klasy II gimnazjum	2
do klasy III gimnazjum	17
do klasy I szkoły ponadgimnazjalnej	60
do klasy II szkoły ponadgimnazjalnej	138
do klasy III szkoły ponadgimnazjalnej	172
do klasy IV szkoły ponadgimnazjalnej	7

W dniu 11 lutego 2011 roku odbyła się sesja próbna, na której zawodnicy rozwiązywali nieliczące się do ogólnej klasyfikacji zadanie „Sejf" autorstwa Mariana M. Kędzierskiego. W dniach konkursowych (12–13 lutego) zawodnicy rozwiązywali następujące zadania:

- w pierwszym dniu zawodów:
  - „Różnica" autorstwa Jacka Tomasiewicza
  - „Śmieci" autorstwa Michała Pilipczuka
- w drugim dniu zawodów:
  - „Rotacje drzew" autorstwa Tomasza Walenia
  - „Temperatura" autorstwa Jacka Tomasiewicza

każde oceniane maksymalnie po 100 punktów.

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

- **SEJ — próbne** — Sejf

	SEJ — próbne	
	liczba zawodników	czyli
100 pkt.	2	0,51%
75–99 pkt.	1	0,25%
50–74 pkt.	12	3,03%
1–49 pkt.	22	5,56%
0 pkt.	252	63,64%
brak rozwiązania	107	27,02%

- **ROZ** — Różnica

	ROZ	
	liczba zawodników	czyli
100 pkt.	48	12,12%
75–99 pkt.	22	5,56%
50–74 pkt.	41	10,35%
1–49 pkt.	233	58,84%
0 pkt.	36	9,09%
brak rozwiązania	16	4,04%

- **SMI** — Śmieci

	SMI	
	liczba zawodników	czyli
100 pkt.	23	5,81%
75–99 pkt.	8	2,02%
50–74 pkt.	29	7,32%
1–49 pkt.	117	29,55%
0 pkt.	89	22,47%
brak rozwiązania	130	32,83%

- **ROT** — Rotacje drzew

	ROT	
	liczba zawodników	czyli
100 pkt.	7	1,77%
75–99 pkt.	3	0,76%
50–74 pkt.	7	1,77%
1–49 pkt.	160	40,40%
0 pkt.	92	23,23%
brak rozwiązania	127	32,07%

- **TEM** — Temperatura

	TEM	
	liczba zawodników	czyli
100 pkt.	60	15,15%
75–99 pkt.	15	3,79%
50–74 pkt.	61	15,40%
1–49 pkt.	133	33,59%
0 pkt.	116	29,29%
brak rozwiązania	11	2,78%

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

SUMA	liczba zawodników	czyli
400 pkt.	1	0,25%
300–399 pkt.	14	3,54%
200–299 pkt.	41	10,35%
100–199 pkt.	107	27,02%
1–99 pkt.	219	55,30%
0 pkt.	14	3,54%

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

ZAWODY III STOPNIA

Zawody III stopnia odbyły się w ośrodku firmy Combidata Poland w Sopocie w dniach od 5 do 9 kwietnia 2011 roku. Do zawodów III stopnia zakwalifikowano 80 najlepszych uczestników zawodów II stopnia, którzy uzyskali wynik nie mniejszy niż 175 pkt.

Zawodnicy uczęszczali do szkół w następujących województwach:

mazowieckie	17 uczestników	zachodniopomorskie	5
małopolskie	15	warmińsko-mazurskie	3
dolnośląskie	8	łódzkie	2
kujawsko-pomorskie	7	podkarpackie	2
podlaskie	7	lubuskie	1
pomorskie	7	wielkopolskie	1
śląskie	5		

Niżej wymienione szkoły miały w finale więcej niż jednego zawodnika:

V LO im. Augusta Witkowskiego	Kraków	15 uczniów
XIV LO im. Stanisława Staszica (ZS nr 82)	Warszawa	10
III LO im. Marynarki Wojennej RP (ZSO nr 1)	Gdynia	6
XIII Liceum Ogólnokształcące (ZSO nr 7)	Szczecin	5
I LO im. Adama Mickiewicza	Białystok	4
VI LO im. Jana i Jędrzeja Śniadeckich (ZSO nr 6)	Bydgoszcz	4

XIV LO im. Polonii Belgijskiej (ZS nr 14)	Wrocław	4
VI LO im. Jana Kochanowskiego (ZSO nr 6)	Radom	3
III LO im. Adama Mickiewicza	Wrocław	3
IV LO im. Marii Skłodowskiej-Curie (ZSO nr 2)	Olsztyn	2
LO (ZS nr 1 im. Jana Pawła II)	Przysucha	2
I LO im. Marii Konopnickiej	Suwałki	2
Liceum Akademickie (ZS UMK Gimnazjum i Liceum Akademickie)	Toruń	2

Zawodnicy uczęszczali do następujących klas:

do klasy II gimnazjum	1 uczeń
do klasy III gimnazjum	4 uczniów
do klasy I szkoły ponadgimnazjalnej	15
do klasy II szkoły ponadgimnazjalnej	25
do klasy III szkoły ponadgimnazjalnej	35

5 kwietnia odbyła się sesja próbna, na której zawodnicy rozwiązywali nieliczące się do ogólnej klasyfikacji zadanie „Dynamit" autorstwa Jacka Tomasiewicza. W dniach konkursowych (6–7 kwietnia) zawodnicy rozwiązywali następujące zadania:

- w pierwszym dniu zawodów:
  - „Impreza" autorstwa Jakuba Wojtaszczyka
  - „Inspekcja" autorstwa Wojciecha Ryttera i Bartosza Szredera
  - „Okresowość" autorstwa Wojciecha Ryttera
- w drugim dniu zawodów:
  - „Konkurs programistyczny" autorstwa Tomasz Idziaszka
  - „Meteory" autorstwa Pawła Mechlińskiego i Jakuba Pachockiego
  - „Patyczki" autorstwa Michała Pilipczuka

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:

- **DYN — próbne** — Dynamit

	DYN — próbne	
	liczba zawodników	czyli
100 pkt.	26	32,50%
75–99 pkt.	6	7,50%
50–74 pkt.	2	2,50%
1–49 pkt.	28	35,00%
0 pkt.	10	12,50%
brak rozwiązania	8	10,0%

- **IMP** — Impreza

	IMP	
	liczba zawodników	czyli
100 pkt.	14	17,50%
75–99 pkt.	1	1,25%
50–74 pkt.	1	1,25%
1–49 pkt.	29	36,25%
0 pkt.	20	25,00%
brak rozwiązania	15	18,75%

- **INS** — Inspekcja

	INS	
	liczba zawodników	czyli
100 pkt.	20	25,00%
75–99 pkt.	2	2,50%
50–74 pkt.	6	7,50%
1–49 pkt.	8	10,00%
0 pkt.	35	43,75%
brak rozwiązania	9	11,25%

- **OKR** — Okresowość

	OKR	
	liczba zawodników	czyli
100 pkt.	1	1,25%
75–99 pkt.	0	0,00%
50–74 pkt.	2	2,50%
1–49 pkt.	44	55,00%
0 pkt.	8	10,00%
brak rozwiązania	25	31,25%

- **MET** — Meteory

	MET	
	liczba zawodników	czyli
100 pkt.	2	2,50%
75–99 pkt.	1	1,25%
50–74 pkt.	16	20,00%
1–49 pkt.	51	63,75%
0 pkt.	4	5,00%
brak rozwiązania	6	7,50%

- **PAT** — Patyczki

	PAT	
	liczba zawodników	czyli
100 pkt.	37	46,25%
75–99 pkt.	5	6,25%
50–74 pkt.	3	3,75%
1–49 pkt.	34	43,50%
0 pkt.	0	0,00%
brak rozwiązania	1	1,25%

- **PRO** — Konkurs programistyczny

	PRO	
	liczba zawodników	czyli
100 pkt.	5	6,25%
75–99 pkt.	3	3,75%
50–74 pkt.	4	5,00%
1–49 pkt.	27	33,75%
0 pkt.	19	23,75%
brak rozwiązania	22	27,50%

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

SUMA	liczba zawodników	czyli
450–600 pkt.	2	2,50%
300–449 pkt.	10	12,50%
150–299 pkt.	35	43,75%
1–149 pkt.	33	41,25%
0 pkt.	0	0,00%

W dniu 9 kwietnia 2011 roku, w siedzibie firm Asseco Poland SA i Combidata Poland SA w Gdyni, ogłoszono wyniki finału XVIII Olimpiady Informatycznej 2010/2011 i rozdano nagrody ufundowane przez: Asseco Poland SA, Ministerstwo Edukacji Narodowej, Olimpiadę Informatyczną, Wydawnictwa Naukowe PWN i Wydawnictwo „Delta".

Poniżej zestawiono listę wszystkich laureatów i wyróżnionych finalistów:

(1) **Jan Kanty Milczek**, 3 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP w Gdyni, 570 pkt., laureat I miejsca

(2) **Krzysztof Leszczyński**, 3 klasa, I Liceum Ogólnokształcące im. Marii Konopnickiej w Suwałkach, 466 pkt., laureat I miejsca

(3) **Piotr Bejda**, 2 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie, 409 pkt., laureat I miejsca

(4) **Łukasz Jocz**, 3 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza w Białymstoku, 394 pkt., laureat I miejsca

(5) **Mateusz Kopeć**, 2 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza w Białymstoku, 371 pkt., laureat II miejsca

(6) **Mateusz Gołębiewski**, 2 klasa, XIV Liceum Ogólnokształcące im. Polonii Belgijskiej we Wrocławiu, 370 pkt., laureat II miejsca

(7) **Krzysztof Pszeniczny**, 1 klasa, Liceum im. Jana Pawła II Sióstr Prezentek w Rzeszowie, 370 pkt., laureat II miejsca

(8) **Wiktor Kuropatwa**, 2 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie, 350 pkt., laureat II miejsca

(9) **Bartosz Tarnawski**, 2 klasa, Katolickie Liceum Ogólnokształcące im. bł. ks. Emila Szramka w Katowicach, 320 pkt., laureat II miejsca

(10) **Krzysztof Kiewicz**, 2 klasa, VII Liceum Ogólnokształcące im. Króla Władysława IV w Warszawie, 309 pkt., laureat II miejsca

(11) **Marcin Smulewicz**, 2 klasa, Liceum Ogólnokształcące im. Bolesława Prusa w Skierniewicach, 309 pkt., laureat II miejsca

(12) **Michał Zając**, 2 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie, 309 pkt., laureat II miejsca

(13) **Karol Farbiś**, 1 klasa, VI Liceum Ogólnokształcące im. Jana Kochanowskiego w Radomiu, 289 pkt., laureat III miejsca

(14) **Michał Łowicki**, 2 klasa, III Liceum Ogólnokształcące im. Adama Mickiewicza we Wrocławiu, 287 pkt., laureat III miejsca

(15) **Radosław Serafin**, 1 klasa, III Liceum Ogólnokształcące im. Adama Mickiewicza we Wrocławiu, 287 pkt., laureat III miejsca

(16) **Mateusz Jurczyk**, 3 klasa, VIII Liceum Ogólnokształcące im. Marii Skłodowskiej-Curie w Katowicach, 279 pkt., laureat III miejsca

(17) **Konrad Sikorski**, 1 klasa, VI Liceum Ogólnokształcące im. Jana Kochanowskiego w Radomiu, 279 pkt., laureat III miejsca

(18) **Adam Czapliński**, 1 klasa, IV Liceum Ogólnokształcące im. Marii Skłodowskiej-Curie w Olsztynie, 277 pkt., laureat III miejsca

(19) **Sebastian Jaszczur**, 3 klasa gimnazjum, Gimnazjum nr 50 w Bydgoszczy, 273 pkt., laureat III miejsca

(20) **Maciej Matraszek**, 3 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica w Warszawie, 270 pkt., laureat III miejsca

(21) **Michał Piekarz**, 2 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie, 257 pkt., laureat III miejsca

(22) **Maciej Borsz**, 3 klasa, VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich w Bydgoszczy, 250 pkt., laureat III miejsca

(23) **Krzysztof Lis**, 3 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica w Warszawie, 250 pkt., laureat III miejsca

(24) **Piotr Żurkowski**, 3 klasa, VIII Liceum Ogólnokształcące im. Adama Mickiewicza w Poznaniu, 250 pkt., laureat III miejsca

(25) **Jakub Sygnowski**, 3 klasa, Liceum Ogólnokształcące w Zespole Szkół nr 1 im. Jana Pawła II w Przysusze, 249 pkt., laureat III miejsca

(26) **Kamil Rychlewicz**, 3 klasa gimnazjum, Gimnazjum nr 8 im. Tadeusza Kościuszki w Łodzi, 248 pkt., laureat III miejsca

(27) **Wojciech Kozaczewski**, 2 klasa, III Liceum Ogólnokształcące im. Adama Mickiewicza we Wrocławiu, 236 pkt., finalista z wyróżnieniem

(28) **Kamil Łukasz**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie, 229 pkt., finalista z wyróżnieniem

(29) **Aleksander Kramarz**, 2 klasa, VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich w Bydgoszczy, 228 pkt., finalista z wyróżnieniem

(30) **Wojciech Nadara**, 2 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica w Warszawie, 226 pkt., finalista z wyróżnieniem

(31) **Paweł Nowak**, 1 klasa, XIII Liceum Ogólnokształcące w Szczecinie, 220 pkt., finalista z wyróżnieniem

(32) **Bartłomiej Dudek**, 2 klasa, XIV Liceum Ogólnokształcące im. Polonii Belgijskiej we Wrocławiu, 212 pkt., finalista z wyróżnieniem

(33) **Paweł Kubiak**, 3 klasa, I Liceum Ogólnokształcące z Oddziałami Dwujęzycznymi im. Mikołaja Kopernika w Katowicach, 212 pkt., finalista z wyróżnieniem

(34) **Karol Pokorski**, 3 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP w Gdyni, 209 pkt., finalista z wyróżnieniem

(35) **Michał Dyrek**, 2 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie, 203 pkt., finalista z wyróżnieniem

Lista pozostałych finalistów w kolejności alfabetycznej:

- **Stanisław Barzowski**, 3 klasa gimnazjum, Gimnazjum nr 24 z Oddziałami Dwujęzycznymi w Gdyni
- **Rafał Bielenia**, 3 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza w Białymstoku
- **Robert Błaszkiewicz**, 2 klasa, XIII Liceum Ogólnokształcące w Szczecinie
- **Franciszek Boehlke**, 3 klasa, XIII Liceum Ogólnokształcące w Szczecinie
- **Piotr Chabierski**, 1 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP w Gdyni
- **Jakub Czarnowicz**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie
- **Michał Darkowski**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie
- **Wojciech Donderowicz**, 3 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica w Warszawie
- **Bartłomiej Gajewski**, 3 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica w Warszawie
- **Michał Glapa**, 3 klasa gimnazjum, Gimnazjum „Filomata" w Gliwicach
- **Łukasz Gładczuk**, 3 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica w Warszawie
- **Katarzyna Jabłonowska**, 3 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica w Warszawie
- **Piotr Jagiełło**, 3 klasa, XIV Liceum Ogólnokształcące im. Polonii Belgijskiej we Wrocławiu
- **Wojciech Janczewski**, 1 klasa, I Liceum Ogólnokształcące im. Tadeusza Kościuszki w Legnicy
- **Maciej Arkadiusz Kisiel**, 3 klasa, IV Liceum Ogólnokształcące im. Marii Skłodowskiej-Curie w Olsztynie
- **Krzysztof Kleiner**, 2 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie
- **Jakub Kołodziej**, 2 klasa, I Liceum Ogólnokształcące im. Wojciecha Kętrzyńskiego w Giżycku
- **Michał Kosnowski**, 3 klasa, Liceum Akademickie w Zespole Szkół Uniwersytetu Mikołaja Kopernika w Toruniu
- **Michał Kowalczyk**, 1 klasa, VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich w Bydgoszczy
- **Michał Kowalik**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie
- **Filip Kowalski**, 2 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP w Gdyni
- **Piotr Kozakowski**, 1 klasa, I Liceum Ogólnokształcące im. Edwarda Dembowskiego w Zielonej Górze

- **Przemysław Jakub Kozłowski**, 2 klasa gimnazjum, Społeczne Gimnazjum nr 8 Społecznego Towarzystwa Oświatowego w Białymstoku
- **Krzysztof Kulig**, 2 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie
- **Tom Macieszczak**, 2 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica w Warszawie
- **Błażej Magnowski**, 1 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP w Gdyni
- **Miłosz Makowski**, 2 klasa, Liceum Akademickie w Zespole Szkół Uniwersytetu Mikołaja Kopernika w Toruniu
- **Adam Malczewski**, 3 klasa, XIII Liceum Ogólnokształcące w Szczecinie
- **Bartosz Marcinkowski**, 3 klasa, XIII Liceum Ogólnokształcące w Szczecinie
- **Wojciech Marczenko**, 3 klasa, XXVII Liceum Ogólnokształcące im. Tadeusza Czackiego w Warszawie
- **Damian Orlef**, 3 klasa, III Liceum Ogólnokształcące w Zabrzu
- **Piotr Pakosz**, 3 klasa, Liceum Ogólnokształcące w Zespole Szkół nr 1 im. Jana Pawła II w Przysusze
- **Szymon Policht**, 2 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie
- **Stanisław Purgał**, 1 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica w Warszawie
- **Marcin Regdos**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie
- **Damian Repke**, 3 klasa, VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich w Bydgoszczy
- **Marek Rusinowski**, 1 klasa, II Liceum Ogólnokształcące m. Mikołaja Kopernika w Mielcu
- **Wojciech Sidor**, 3 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP w Gdyni
- **Dariusz Sosnowski**, 2 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza w Białymstoku
- **Szymon Stankiewicz**, 1 klasa, VI Liceum Ogólnokształcące im. Jana Kochanowskiego w Radomiu
- **Maciej Szeptuch**, 3 klasa, XIV Liceum Ogólnokształcące im. Polonii Belgijskiej we Wrocławiu
- **Mateusz Twaróg**, 2 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie
- **Łukasz Wałejko**, 3 klasa, I Liceum Ogólnokształcące im. Marii Konopnickiej w Suwałkach
- **Arkadiusz Wróbel**, 1 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica w Warszawie
- **Konrad Zemek**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie

Komitet Główny Olimpiady Informatycznej przyznał następujące nagrody rzeczowe:

(1) puchar przechodni ufundowany przez Olimpiadę Informatyczną przyznano zwycięzcy XVIII Olimpiady, Janowi Kantemu Milczkowi,
(2) puchar ufundowany przez Olimpiadę Informatyczną przyznano zwycięzcy XVIII Olimpiady, Janowi Kantemu Milczkowi,
(3) złote, srebrne i brązowe medale ufundowane przez MEN przyznano odpowiednio laureatom I, II i III miejsca,
(4) IPAD-y (3 szt.) ufundowane przez Asseco Poland SA i MEN przyznano laureatom I miejsca,
(5) Tablety Samsung Galaxy (9 szt.) ufundowane przez Asseco Poland SA i MEN przyznano jednemu laureatowi I miejsca i laureatom II miejsca,
(6) książki ufundowane przez PWN przyznano wszystkim laureatom i finalistom,
(7) roczną prenumeratę miesięcznika „Delta" przyznano wszystkim laureatom i wyróżnionym finalistom.

Komitet Główny powołał następujące reprezentacje.

- Na **Międzynarodową Olimpiadę Informatyczną IOI'2011**, która odbędzie się w Tajlandii, w terminie 22–29 lipca 2011 roku oraz **Olimpiadę Informatyczną Krajów Europy Środkowej CEOI'2011**, która odbędzie się w Gdyni, w terminie 7–12 lipca 2011 roku:

  (1) Jan Kanty Milczek
  (2) Krzysztof Leszczyński
  (3) Piotr Bejda
  (4) Łukasz Jocz

  rezerwowi:

  (5) Mateusz Kopeć
  (6) Krzysztof Pszeniczny
  (7) Mateusz Gołębiewski

  II drużyna na CEOI:

  (5) Mateusz Kopeć
  (6) Krzysztof Pszeniczny
  (7) Mateusz Gołębiewski
  (8) Wiktor Kuropatwa

  rezerwowi:

  (9) Bartosz Tarnawski
  (10) Krzysztof Kiewicz
  (11) Marcin Smulewicz
  (12) Michał Zając

- Na **Bałtycką Olimpiadę Informatyczną BOI'2011**, która odbędzie się w Danii w terminie 29 kwietnia – 3 maja 2011 roku pojadą zawodnicy, którzy nie uczęszczają do klas maturalnych, w kolejności rankingowej:

(1) Piotr Bejda
(2) Mateusz Kopeć
(3) Krzysztof Pszeniczny
(4) Mateusz Gołębiewski
(5) Wiktor Kuropatwa
(6) Bartosz Tarnawski

rezerwowi:

(7) Krzysztof Kiewicz
(8) Marcin Smulewicz
(9) Michał Zając

Polscy reprezentanci uzyskali następujące wyniki:

- Mateusz Gołębiewski – złoty medal (1. miejsce)
- Bartosz Tarnawski – złoty medal (2. miejsce)
- Mateusz Kopeć – złoty medal (4. miejsce)
- Piotr Bejda – srebrny medal
- Wiktor Kuropatwa – srebrny medal
- Krzysztof Pszeniczny – brązowy medal

- W **obozie czesko-polsko-słowackim**, który odbędzie się na Słowacji, wezmą udział zawodnicy, którzy będą reprezentować Polskę na Międzynarodowej Olimpiadzie Informatycznej, wraz z zawodnikami rezerwowymi.
- W **Obozie Naukowo-Treningowym im. Antoniego Kreczmara** wezmą udział reprezentanci na Międzynarodową Olimpiadę Informatyczną wraz z rezerwowymi oraz laureaci i finaliści Olimpiady, którzy nie uczęszczają w tym roku szkolnym do programowo najwyższej klasy szkoły ponadgimnazjalnej.

Sekretariat wystawił łącznie 26 zaświadczeń o uzyskaniu tytułu laureata, 9 zaświadczeń o uzyskaniu tytułu wyróżnionego finalisty oraz 45 zaświadczeń o uzyskaniu tytułu finalisty XVIII Olimpiady Informatycznej.

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

- Marcin Andrychowicz (student Uniwersytetu Warszawskiego)
  - Jakub Sygnowski – laureat III miejsca
  - Piotr Pakosz – finalista
- Jarosław Bartos (Publiczne Gimnazjum, Przysucha)
  - Konrad Sikorski – laureat III miejsca
- Michał Bejda (student Uniwersytetu Jagiellońskiego, Kraków)
  - Piotr Bejda – laureat I miejsca
  - Wiktor Kuropatwa – laureat II miejsca
  - Krzysztof Kleiner – finalista
  - Szymon Policht – finalista

- Jarosław Błasiok (student Uniwersytetu Warszawskiego)
  - Mateusz Jurczyk – laureat III miejsca
- Iwona Bujnowska (I Liceum Ogólnokształcące im. Adama Mickiewicza, Białystok)
  - Mateusz Kopeć – laureat II miejsca
  - Przemysław Jakub Kozłowski – finalista
  - Dariusz Sosnowski – finalista
- Ireneusz Bujnowski (I Liceum Ogólnokształcące im. Adama Mickiewicza, Białystok)
  - Łukasz Jocz – laureat I miejsca
  - Mateusz Kopeć – laureat II miejsca
  - Rafał Bielenia – finalista
  - Przemysław Jakub Kozłowski – finalista
  - Dariusz Sosnowski – finalista
- Magda Burakowska (Zespół Szkół Ogólnokształcących nr 2, Olsztyn)
  - Adam Czapliński – laureat III miejsca
  - Maciej Kisiel – finalista
- Marek Cygan (doktorant Uniwersytetu Warszawskiego)
  - Maciej Borsz – laureat III miejsca
  - Sebastian Jaszczur – laureat III miejsca
  - Aleksander Kramarz – finalista z wyróżnieniem
  - Michał Kowalczyk – finalista
  - Damian Repke – finalista
- Czesław Drozdowski (XIII Liceum Ogólnokształcące, Szczecin)
  - Paweł Nowak – finalista z wyróżnieniem
  - Robert Błaszkiewicz – finalista
  - Franciszek Boehlke – finalista
  - Adam Malczewski – finalista
  - Bartosz Marcinkowski – finalista
- Marcin Dublański (student Uniwersytetu Wrocławskiego)
  - Bartłomiej Dudek – finalista z wyróżnieniem
- Lech Duraj (Uniwersytet Jagielloński, Kraków)
  - Piotr Bejda – laureat I miejsca
  - Wiktor Kuropatwa – laureat II miejsca
  - Michał Zając – laureat II miejsca
  - Michał Piekarz – laureat III miejsca
  - Michał Dyrek – finalista z wyróżnieniem
  - Krzysztof Kleiner – finalista
  - Krzysztof Kulig – finalista
  - Szymon Policht – finalista
  - Mateusz Twaróg – finalista

- Andrzej Dyrek (V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków)
  - Piotr Bejda – laureat I miejsca
  - Wiktor Kuropatwa – laureat II miejsca
  - Michał Zając – laureat II miejsca
  - Michał Piekarz – laureat III miejsca
  - Michał Dyrek – finalista z wyróżnieniem
  - Kamil Łukasz – finalista z wyróżnieniem
  - Jakub Czarnowicz – finalista
  - Michał Darkowski – finalista
  - Krzysztof Kleiner – finalista
  - Michał Kowalik – finalista
  - Krzysztof Kulig – finalista
  - Szymon Policht – finalista
  - Marcin Regdos – finalista
  - Mateusz Twaróg – finalista
  - Konrad Zemek – finalista
- Marek Gałaszewski (I Liceum Ogólnokształcące im. Marii Konopnickiej, Suwałki)
  - Krzysztof Leszczyński – laureat I miejsca
  - Łukasz Wałejko – finalista
- Witold Jarnicki (Google Kraków)
  - Krzysztof Kulig – finalista
- Evelyn Jelec (I Liceum Ogólnokształcące im. Wojciecha Kętrzyńskiego, Giżycko)
  - Jakub Kołodziej – finalista
- Łukasz Kalinowski (student Uniwersytetu im. Adama Mickiewicza, Poznań)
  - Wojciech Nadara – finalista z wyróżnieniem
- Sławomir Krzywicki (I Liceum Ogólnokształcące im. Edwarda Dembowskiego, Zielona Góra)
  - Piotr Kozakowski – finalista
- Bożena Kubiak (I Liceum Ogólnokształcące im. Mikołaja Kopernika, Katowice)
  - Paweł Kubiak – finalista z wyróżnieniem
- Anna Beata Kwiatkowska (Zespół Szkół UMK Gimnazjum i Liceum Akademickie, Toruń)
  - Miłosz Makowski – finalista
- Romualda Laskowska (I Liceum Ogólnokształcące im. Tadeusza Kościuszki, Legnica)
  - Wojciech Janczewski – finalista
- Paweł Lipski (absolwent Liceum FILOMATA, Gliwice)
  - Michał Glapa – finalista
- Mirosław Mortka (VI Liceum Ogólnokształcące im. Jana Kochanowskiego, Radom)

  - Karol Farbiś – laureat III miejsca
  - Konrad Sikorski – laureat III miejsca
  - Szymon Stankiewicz – finalista
- Zofia Olędzka (Zespół Szkół Ogólnokształcących nr 6, Bydgoszcz)
  - Sebastian Jaszczur – laureat III miejsca
- Grzegorz Owsiany (Wyższa Szkoła Informatyki i Zarządzania, Rzeszów)
  - Krzysztof Pszeniczny – laureat II miejsca
  - Marek Rusinowski – finalista
- Andrzej Pezarski (Uniwersytet Jagielloński, Kraków)
  - Jakub Czarnowicz – finalista
  - Michał Kowalik – finalista
  - Marcin Regdos – finalista
  - Konrad Zemek – finalista
- Małgorzata Piekarska (Zespół Szkół Ogólnokształcących nr 6, Bydgoszcz)
  - Maciej Borsz – laureat III miejsca
  - Aleksander Kramarz – finalista z wyróżnieniem
  - Michał Kowalczyk – finalista
  - Damian Repke – finalista
- Anna Pokorska (Skórcz)
  - Karol Pokorski – finalista z wyróżnieniem
- Adam Polak (student Uniwersytetu Jagiellońskiego, Kraków)
  - Piotr Bejda – laureat I miejsca
  - Wiktor Kuropatwa – laureat II miejsca
  - Krzysztof Kleiner – finalista
  - Krzysztof Kulig – finalista
  - Szymon Policht – finalista
  - Mateusz Twaróg – finalista
- Damian Rusak (student (Uniwersytetu Wrocławskiego)
  - Mateusz Gołębiewski – laureat II miejsca
  - Piotr Jagiełło – finalista
  - Maciej Szeptuch – finalista
- Antoni Salamon (Katolickie Liceum Ogólnokształcące, Katowice)
  - Bartosz Tarnawski – laureat II miejsca
- Agnieszka Samulska (VIII Liceum Ogólnokształcące i 58 Gimnazjum im. Króla Władysława IV, Warszawa)
  - Krzysztof Kiewicz – laureat II miejsca
- Piotr Sielski (Uniwersytet Łódzki)
  - Kamil Rychlewicz – laureat III miejsca
- Piotr Suwara (student Uniwersytetu Warszawskiego)
  - Wojciech Nadara – finalista z wyróżnieniem
- Bartosz Szreder (student Uniwersytetu Warszawskiego)
  - Wojciech Nadara – finalista z wyróżnieniem
  - Wojciech Donderowicz – finalista

- Ryszard Szubartowski (III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia)
  - Jan Kanty Milczek – laureat I miejsca
  - Stanisław Barzowski – finalista
  - Piotr Chabierski – finalista
  - Błażej Magnowski – finalista
  - Wojciech Sidor – finalista
- Michał Śliwiński (III Liceum Ogólnokształcące im. Adama Mickiewicza, Wrocław)
  - Radosław Serafin – laureat III miejsca
  - Wojciech Kozaczewski – finalista z wyróżnieniem
- Joanna Śmigielska (XIV Liceum Ogólnokształcące im. Stanisława Staszica, Warszawa)
  - Krzysztof Lis – laureat III miejsca
  - Maciej Matraszek – laureat III miejsca
  - Bartłomiej Gajewski – finalista
  - Katarzyna Jabłonowska – finalistka
  - Tom Macieszczak – finalista
  - Stanisław Purgał – finalista
  - Arkadiusz Wróbel – finalista
- Tomasz Żurkowski (student Politechniki Poznańskiej)
  - Piotr Żurkowski – laureat III miejsca

Zgodnie z decyzją Komitetu Głównego z dnia 8 kwietnia 2011 roku, opiekunowie naukowi laureatów i finalistów, będący nauczycielami szkół, otrzymają nagrody pieniężne.

Podobnie jak w ubiegłych latach w przygotowaniu jest publikacja zawierająca pełną informację o XVIII 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, 20 czerwca 2011 roku

Regulamin Olimpiady Informatycznej

§1 WSTĘP

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

§2 CELE OLIMPIADY I SPOSOBY ICH OSIĄGANIA

(1) Stworzenie motywacji dla zainteresowania nauczycieli i uczniów informatyką.

(2) Rozszerzanie współdziałania nauczycieli akademickich z nauczycielami szkół w kształceniu młodzieży uzdolnionej.

(3) Stymulowanie aktywności poznawczej młodzieży informatycznie uzdolnionej.

(4) Kształtowanie umiejętności samodzielnego zdobywania i rozszerzania wiedzy informatycznej.

(5) Stwarzanie młodzieży możliwości szlachetnego współzawodnictwa w rozwijaniu swoich uzdolnień, a nauczycielom — warunków twórczej pracy z młodzieżą.

(6) Wyłanianie reprezentacji Rzeczypospolitej Polskiej na Międzynarodową Olimpiadę Informatyczną i inne międzynarodowe zawody informatyczne.

(7) Cele Olimpiady są osiągane poprzez:

- organizację olimpiady przedmiotowej z informatyki dla uczniów szkół ponadgimnazjalnych;
- organizowanie corocznych obozów naukowych dla wyróżniających się uczestników olimpiad;
- organizowanie warsztatów treningowych dla nauczycieli zainteresowanych przygotowywaniem uczniów do udziału w olimpiadach;

- przygotowywanie i publikowanie materiałów edukacyjnych dla uczniów zainteresowanych udziałem w olimpiadach i ich nauczycieli.

§3 ORGANIZACJA OLIMPIADY

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

(2) Olimpiada jest trójstopniowa.

(3) W Olimpiadzie mogą brać indywidualnie udział uczniowie wszystkich typów szkół ponadgimnazjalnych i szkół średnich dla młodzieży, dających możliwość uzyskania matury.

(4) W Olimpiadzie mogą również uczestniczyć — za zgodą Komitetu Głównego — uczniowie szkół podstawowych, gimnazjów, zasadniczych szkół zawodowych i szkół zasadniczych.

(5) Zawody I stopnia mają charakter otwarty i polegają na samodzielnym rozwiązywaniu przez uczestnika zadań ustalonych dla tych zawodów oraz przekazaniu rozwiązań w podanym terminie; miejsce i sposób przekazania określone są w „Zasadach organizacji zawodów", zwanych dalej Zasadami.

(6) Zawody II stopnia są organizowane przez komitety okręgowe Olimpiady lub instytucje upoważnione przez Komitet Główny.

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

(8) Rozwiązaniem zadania zawodów I, II i III stopnia są, zgodnie z treścią zadania, dane lub program. Program powinien być napisany w języku programowania i środowisku wybranym z listy języków i środowisk ustalanej przez Komitet Główny i ogłaszanej w Zasadach.

(9) Rozwiązania są oceniane automatycznie. Jeśli rozwiązaniem zadania jest program, to jest on uruchamiany na testach z przygotowanego zestawu.
Podstawą oceny jest zgodność sprawdzanego programu z podaną w treści zadania specyfikacją, poprawność wygenerowanego przez program wyniku, czas działania tego programu oraz ilość wymaganej przez program pamięci. Jeśli rozwiązaniem zadania jest plik z danymi, wówczas ocenia się poprawność danych. Za każde zadanie zawodnik może zdobyć maksymalnie 100 punktów, gdzie 100 jest sumą maksymalnych liczb punktów za poszczególne testy (lub dane z wynikami) dla tego zadania. Oceną rozwiązań zawodnika jest suma punktów za poszczególne zadania. Oceny rozwiązań zawodników są podstawą utworzenia listy rankingowej zawodników po zawodach każdego stopnia.

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

(11) Rozwiązania zespołowe, niesamodzielne, niezgodne z „Zasadami 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.

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

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

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

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

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

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

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

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

(16) Komitet Główny kwalifikuje do zawodów II i III stopnia odpowiednią liczbę uczestników, których rozwiązania zadań stopnia niższego zostaną ocenione najwyżej. Zawodnicy zakwalifikowani do zawodów III stopnia otrzymują tytuł finalisty Olimpiady Informatycznej.

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

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

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

§4 KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ

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

(2) Komitet pracuje w dotychczasowym składzie, powołanym w roku 2008/2009.

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

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

(5) Komitet powołuje i rozwiązuje komitety okręgowe Olimpiady.

(6) Komitet:

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

   (b) udziela wyjaśnień w sprawach dotyczących Olimpiady;

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

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

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

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

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

(9) Do organizacji zawodów II stopnia w miejscowościach, których nie obejmuje żaden komitet okręgowy, Komitet powołuje komisję zawodów co najmniej 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 kierownika organizacyjnego Olimpiady.

(12) Komitet przyjmuje plan finansowy Olimpiady na pierwszym posiedzeniu Komitetu w nowym roku szkolnym.

(13) Komitet przyjmuje sprawozdanie finansowe z przebiegu Olimpiady i przedkłada je Organizatorowi.

(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.

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

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

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

(18) Przewodniczący:

  (a) czuwa nad całokształtem prac Komitetu;

  (b) zwołuje posiedzenia Komitetu;

  (c) przewodniczy tym posiedzeniom;

  (d) reprezentuje Komitet na zewnątrz;

  (e) czuwa nad prawidłowością wydatków związanych z organizacją i przeprowadzeniem Olimpiady oraz zgodnością działalności Komitetu z przepisami.

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

  (a) zadania Olimpiady;

  (b) rozwiązania zadań Olimpiady przez okres 2 lat;

  (c) rejestr wydanych zaświadczeń i dyplomów laureatów;

  (d) listy laureatów i ich nauczycieli;

  (e) dokumentację statystyczną i finansową.

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

§5 KOMITETY OKRĘGOWE

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

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

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

§6 PRZEBIEG OLIMPIADY

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

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

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

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

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

§7 UPRAWNIENIA I NAGRODY

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

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

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

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

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

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

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

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

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

§8 FINANSOWANIE OLIMPIADY

(1) Komitet Główny finansuje działania Olimpiady zgodnie z umową podpisaną przez Ministerstwo Edukacji Narodowej i Fundację Rozwoju Informatyki. Komitet Główny będzie także zabiegał o pozyskanie dotacji z innych organizacji wspierających.

§9 PRZEPISY KOŃCOWE

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

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

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

Zasady organizacji zawodów XVIII Olimpiady Informatycznej w roku szkolnym 2010/2011

§1 WSTĘP

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

§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 350 uczestników, których rozwiązania zadań I stopnia zostaną ocenione najwyżej; do zawodów III stopnia — 70 uczestników, których rozwiązania zadań II stopnia zostaną ocenione najwyżej. Komitet Główny może zmienić podane liczby zakwalifikowanych uczestników co najwyżej o 30%.

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

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

(10) Terminarz zawodów:

- zawody I stopnia — 18 października–15 listopada 2010 r.
  ogłoszenie wyników:
  - w witrynie Olimpiady — 10 grudnia 2010 r.,
  - rozesłanie pocztą wyników oraz materiałów Olimpiady i Asseco — 15 grudnia 2010 r.
- zawody II stopnia — 08–10 lutego 2011 r.
  ogłoszenie wyników w witrynie Olimpiady — 18 lutego 2011 r.
- zawody III stopnia — 05–09 kwietnia 2011 r.

§3 ROZWIĄZANIA ZADAŃ

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

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

(3) W przypadku uznania przez Komitet Główny pracy za niesamodzielną lub zespołową zawodnicy mogą zostać zdyskwalifikowani.

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

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

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

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

(7) Skompilowane programy będą uruchamiane w 32-bitowym systemie Linux.

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

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

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

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

§4 ZAWODY I STOPNIA

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

- poprzez System Internetowy Olimpiady, zwany dalej SIO, o adresie: `http://sio.mimuw.edu.pl`, do 15 listopada 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 15 listopada 2010 r. (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 (dyskietkę lub CD-ROM) w standardzie dla komputerów PC 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. 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 29.11.2010 r. poprzez SIO każdy zawodnik będzie mógł zapoznać się ze wstępną oceną swojej pracy.

(10) Do 03.12.2010 r. (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 03.12.2010 r. nie będą rozpatrywane.

§5 ZAWODY II I III STOPNIA

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

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

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

(4) Komisje Regulaminowe powołane przez komitety okręgowe lub Komitet Główny czuwają nad prawidłowością przebiegu zawodów i pilnują przestrzegania Regulaminu Olimpiady i Zasad Organizacji Zawodów.

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

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

(7) W przypadku stwierdzenia awarii sprzętu w czasie zawodów, termin zakończenia pracy uczestnika zostaje odpowiednio przedłużony. Awarie sprzętu należy zgłaszać dyżurującym członkom Komisji Regulaminowej.

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

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

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

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

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

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

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

(15) Rozwiązanie każdego zadania można zgłosić w SIO co najwyżej 10 razy. Spośród tych zgłoszeń ocenianie jest jedynie najpóźniejsze.

(16) Jeżeli awaria systemu SIO bądź połączenia sieciowego uniemożliwiła zawodnikom wysyłanie rozwiązań przed końcem zawodów, wówczas ci zawodnicy mają możliwość pozostawienia swoich rozwiązań na komputerze w katalogu wskazanym przez organizatorów. Zawodnicy, którzy chcą skorzystać z tej możliwości, niezwłocznie po zakończeniu sesji a przed opuszczeniem sali zawodów powinni wręczyć pisemne oświadczenie dyżurującemu w tej sali członkowi Komisji Regulaminowej. Oświadczenie to musi zawierać imię i nazwisko zawodnika, numer stanowiska oraz informację o zadaniach, których rozwiązania powinny zostać pobrane z komputera.. Złożenie takiego oświadczenia powoduje, że rozwiązania wskazanych zadań złożone wcześniej w SIO nie będą rozpatrywane.

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

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

§6 UPRAWNIENIA I NAGRODY

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

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

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

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

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

(6) Komitet Główny ustala skład reprezentacji Polski na XXIII Międzynarodową Olimpiadę Informatyczną w 2011 roku na podstawie wyników Olimpiady oraz regulaminu tej Olimpiady Międzynarodowej.

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

(8) Wyznaczeni przez Komitet Główny reprezentanci Polski na olimpiady międzynarodowe zostaną zaproszeni do nieodpłatnego udziału w XII Obozie Naukowo-Treningowym im. Antoniego Kreczmara, który odbędzie się w czasie wakacji 2011 r. Do nieodpłatnego udziału w Obozie Komitet Główny może zaprosić także innych finalistów, którzy nie są w ostatniej programowo klasie swojej szkoły, w zależności od uzyskanych wyników.

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

§7 PRZEPISY KOŃCOWE

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

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

(3) Uczniowie zakwalifikowani do udziału w zawodach II i III stopnia są zwolnieni z zajęć szkolnych na czas niezbędny do udziału w zawodach; mają także zagwarantowane na czas tych zawodów bezpłatne zakwaterowanie, wyżywienie i zwrot kosztów przejazdu.

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

Zawody I stopnia

opracowania zadań

Jakub Wojtaszczyk
Treść zadania, Opracowanie

Bartosz Górski
Program

Dostępna pamięć: 128 MB.

OI, Etap I, 18.10–15.11.2010

Konspiracja

Wroga Bitocja napadła zdradziecko na Bajtocję i okupuje znaczną jej część. Król Bajtocji, Bajtazar, chce zorganizować na okupowanych terenach ruch oporu. Bajtazar rozpoczął od wytypowania osób, które utworzą strukturę organizacyjną ruchu oporu. Należy je podzielić na dwie grupy: **konspiratorów**, *którzy będą prowadzić bezpośrednią działalność na terenie okupowanym, oraz* **grupę wsparcia**, *która będzie działać z terytorium wolnej Bajtocji.*

Pojawił się jednak pewien problem — podział taki musi spełniać następujące warunki:

- *Każde dwie osoby z grupy wsparcia powinny się znać — zagwarantuje to, że będą stanowiły zwartą i zgraną grupę.*
- *Konspiratorzy nie powinni się znać nawzajem.*
- *Żadna z grup nie może być pusta, tzn. musi być przynajmniej jeden konspirator i jedna osoba w grupie wsparcia.*

Bajtazar zastanawia się, ile jest różnych sposobów podziału wytypowanych osób na dwie grupy zgodnie z powyższymi warunkami, a przede wszystkim, czy taki podział jest w ogóle możliwy. Jako że sam nie całkiem umie sobie z tym poradzić, poprosił Cię o pomoc.

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita n ($2 \leqslant n \leqslant 5\,000$), oznaczająca liczbę osób zaangażowanych w organizację ruchu oporu. Osoby te są ponumerowane od 1 do n. W kolejnych n wierszach opisane jest, które osoby się znają. W i-tym z tych wierszy znajduje się opis znajomych osoby nr i, złożony z liczb całkowitych pooddzielanych pojedynczymi odstępami. Pierwsza z tych liczb, k_i ($0 \leqslant k_i \leqslant n-1$), oznacza liczbę znajomych osoby nr i. Dalej w wierszu znajduje się k_i liczb całkowitych $a_{i,1}, a_{i,2}, \ldots, a_{i,k_i}$. Liczby $a_{i,j}$ spełniają $1 \leqslant a_{i,j} \leqslant n$, $a_{i,j} \neq i$ oraz są podane w kolejności rosnącej. Możesz założyć, że jeśli w ciągu liczb a_i występuje liczba x, to także w ciągu liczb a_x występuje liczba i.

Wyjście

W pierwszym i jedynym wierszu standardowego wyjścia Twój program powinien wypisać jedną liczbę całkowitą: liczbę sposobów, na które osoby mające utworzyć ruch oporu mogą zostać podzielone na grupę wsparcia i grupę działającą na terenach okupowanych. Jeżeli nie istnieje żaden podział wytypowanych osób na dwie grupy spełniający podane warunki, wówczas poprawnym wynikiem jest 0.

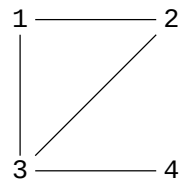
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
3
```



Wyjaśnienie do przykładu: *Dla tej grupy spiskowców są możliwe trzy podziały na grupy. Grupę konspiratorów mogą stanowić uczestnicy o numerach 1 i 4, o numerach 2 i 4 lub sam uczestnik o numerze 4.*

Rozwiązanie

Wprowadzenie

W zadaniu mamy dany graf nieskierowany o n wierzchołkach i m krawędziach. Każdy wierzchołek odpowiada jednemu z uczestników ruchu oporu. Pomiędzy dwoma wierzchołkami jest krawędź, jeśli ci uczestnicy znają się wzajemnie. Naszym celem jest obliczenie, na ile sposobów można podzielić zbiór wierzchołków na dwa zbiory tak, aby spełnione były następujące warunki:

- każde dwa wierzchołki pierwszego zbioru są połączone krawędzią (taki zbiór wierzchołków nazywa się *kliką*)
- pomiędzy żadnymi dwoma wierzchołkami drugiego zbioru nie ma krawędzi (taki zbiór wierzchołków nazywa się *zbiorem niezależnym*)
- żaden z dwóch zbiorów nie jest pusty.

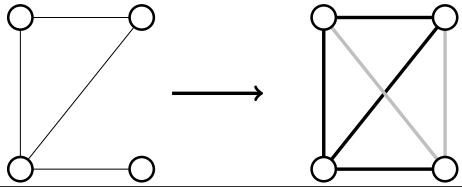
Czyli chcemy obliczyć liczbę sposobów podziału naszego grafu na dwa niepuste podgrafy, tak aby jednym z nich była klika, a drugim zbiór niezależny.

Łatwo spostrzec, że w tym zadaniu występuje symetria pomiędzy klikami a zbiorami niezależnymi. Ze względu na tę symetrię sprowadzimy nasz problem szukania podziałów do problemu szukania kolorowań wierzchołków. To uprości język opisu rozwiązania.

Na początku dopełnimy zatem nasz graf do grafu pełnego i pokolorujemy każdą z krawędzi na czarno lub szaro. Krawędzie, które znajdowały się w grafie przed dopełnieniem, pokolorujemy na czarno, a pozostałe na szaro, patrz rys. 1.

Zacznijmy od następującego spostrzeżenia.

Obserwacja 1. Jeśli jakieś dwa wierzchołki są w oryginalnym grafie połączone krawędzią, to przynajmniej jeden z nich należy do kliki. Jeśli nie są połączone krawędzią, to przynajmniej jeden z nich należy do zbioru niezależnego.

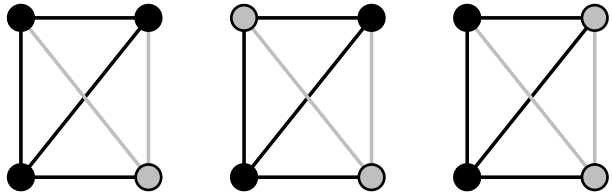


Rys. 1: Sposób pokolorowania krawędzi dla przykładowego grafu z treści zadania.

Ta obserwacja orzeka, że w zmodyfikowanym grafie chcemy znaleźć liczbę wszystkich kolorowań wierzchołków na czarno lub szaro, które spełniają poniższe warunki:

1. istnieje para wierzchołków o różnych kolorach
2. każda czarna krawędź ma co najmniej jeden z końców pokolorowany na czarno
3. każda szara krawędź ma co najmniej jeden z końców pokolorowany na szaro.

Czyli tak naprawdę chcemy, aby czarne wierzchołki tworzyły czarną klikę (żeby każda krawędź między dwoma czarnymi wierzchołkami była czarna), a szare szarą klikę, patrz też rys. 2.



Rys. 2: Poprawne kolorowania grafu z rys. 1.

W dalszych rozważaniach łatwiej będzie zapomnieć o pierwszym z powyższych warunków. Zauważmy, że wystarczy umieć zliczyć wszystkie kolorowania spełniające dwa ostatnie warunki. Zaiste — jeśli umiemy to zrobić, to możemy następnie sprawdzić, czy któreś z kolorowań „wszystko na czarno" i „wszystko na szaro" spełnia warunki 2 i 3 (nie mogą obydwa jednocześnie spełniać, jako że $n \geqslant 2$, a zatem w pokolorowanym grafie znajduje się przynajmniej jedna krawędź i musi istnieć przynajmniej jeden wierzchołek tego samego koloru co ta krawędź), a jeśli tak, to odpowiednio zmniejszyć wynik uzyskany z pominięciem pierwszego warunku. Sprawdzenie tych dwóch specjalnych kolorowań możemy wykonać w czasie $O(n^2)$, a zatem wystarczająco szybko. W dalszym ciągu będziemy zatem poszukiwać kolorowań spełniających warunki 2 i 3. Takie kolorowania nazwiemy *dobrymi*.

Rozwiązanie wzorcowe – analiza stopni

Zaczniemy od zaprezentowania efektownego rozwiązania tego zadania, które wymaga wyłącznie znajomości stopni wierzchołków w grafie, bez znajomości konkretnych kra-

wędzi. Aby na końcu uzyskać bardzo prosty algorytm, zbadamy dokładnie strukturę dobrych kolorowań.

Obserwacja 2. Załóżmy, że mamy pewne dobre kolorowanie, w którym wierzchołek v jest czarny, a wierzchołek w — szary. Wtedy liczba czarnych krawędzi wychodzących z v jest nie mniejsza niż liczba czarnych krawędzi wychodzących z w.

Dowód: Załóżmy, że w rozwiązaniu jest k czarnych wierzchołków. Wtedy z v wychodzi przynajmniej $k-1$ czarnych krawędzi (do każdego z pozostałych czarnych wierzchołków), zaś z w — co najwyżej k czarnych krawędzi, jako że szare wierzchołki muszą być połączone z w szarą krawędzią. Zatem aby z w wychodziło więcej czarnych krawędzi niż z v, musi być tak, że z w wychodzi dokładnie k czarnych krawędzi, zaś z v — dokładnie $k-1$. Ale skoro z v wychodzi $k-1$ czarnych krawędzi, to krawędź vw jest szara, a zatem z w nie może wychodzić k czarnych krawędzi, sprzeczność. ■

W dalszym ciągu liczbę czarnych krawędzi wychodzących z danego wierzchołka nazwiemy *czarnym stopniem* i oznaczać będziemy przez $cs(v)$.

Obserwacja 3. Niech dane będzie dowolne kolorowanie wierzchołków grafu na czarno i szaro, w którym jest k czarnych wierzchołków. To kolorowanie jest dobre wtedy i tylko wtedy, gdy

$$\sum_{v \text{ czarny}} cs(v) \;=\; k(k-1) + \sum_{v \text{ szary}} cs(v).$$

Dowód: Wpierw załóżmy, że mamy kolorowanie spełniające podaną wyżej tożsamość. Niech CC oznacza liczbę czarnych krawędzi łączących dwa czarne wierzchołki, SS liczbę czarnych krawędzi łączących dwa szare wierzchołki, zaś CS — liczbę czarnych krawędzi łączących dwa wierzchołki różnych kolorów. Wtedy lewa strona tożsamości jest równa $2CC + CS$, zaś suma znajdująca się po prawej stronie tożsamości to $2SS + CS$. Wstawiając to do tożsamości, otrzymujemy $2CC = k(k-1) + 2SS$.

Ale jeśli czarnych wierzchołków jest k, to krawędzi między nimi jest dokładnie $k(k-1)/2$, a zatem $CC \leqslant k(k-1)/2$. Mamy więc

$$k(k-1) \;\geqslant\; 2CC \;=\; k(k-1) + 2SS,$$

skąd $SS = 0$, zaś $CC = k(k-1)/2$. Zatem faktycznie wszystkie krawędzie łączące czarne wierzchołki są czarne, zaś wszystkie krawędzie łączące szare wierzchołki są szare.

Teraz rozważmy dowolne dobre kolorowanie. Dla takiego kolorowania mamy $CC = k(k-1)/2$ oraz $SS = 0$. To kolorowanie spełnia zatem tożsamość z obserwacji. ■

Teraz jesteśmy gotowi do sformułowania rozwiązania. Zaczynamy od wyznaczenia dla każdego wierzchołka liczby $cs(v)$ — te liczby to po prostu liczby k_i podane na wejściu. W tym rozwiązaniu nie będziemy musieli w ogóle zapamiętywać liczb a_{ij} (choć oczywiście chcemy je wczytać, by dojść do kolejnej liczby k_i). Porządkujemy liczby $cs(v)$ w kolejności nierosnącej. Następnie dla kolejnych $k = 1, 2, \ldots, n$ obliczamy sumę pierwszych k spośród liczb $cs(v)$ i sprawdzamy, czy jest ona o dokładnie

$k(k-1)$ większa od sumy pozostałych liczb cs. Tu uwaga implementacyjna — jako że $n \leqslant 5\,000$, to nasze rozwiązanie będzie wystarczająco szybkie, nawet jeśli będziemy dla każdej liczby k obliczali te sumy od zera, ale oczywiście bardziej eleganckie jest obliczanie sum dla k poprzez dodanie do pierwszej sumy wartości $cs(v_k)$ i odjęcie tej wartości od drugiej sumy.

Jeżeli okaże się, że zachodzi tożsamość z Obserwacji 3, to znaczy, że znaleźliśmy rozwiązanie. Jeśli istnieje jakieś rozwiązanie, to na mocy Obserwacji 2 znajdziemy je — wystarczy przeglądać wierzchołki w kolejności nierosnących stopni cs. Trzeba jeszcze zwrócić uwagę na to, że może zdarzyć się sytuacja, w której więcej niż jedna liczba ma tę samą wartość cs — wówczas może być więcej rozwiązań. Jako że na mocy Obserwacji 3 same liczby cs mówią nam, czy jakiś zbiór wierzchołków jest rozwiązaniem, czy nie, to jeżeli znajdziemy rozwiązanie, w którym spośród q liczb o tej samej wartości cs wzięliśmy do rozwiązania l, to wybór dowolnych innych l też da nam rozwiązanie — zatem za jednym razem znaleźliśmy $\binom{q}{l}$ rozwiązań.

Ostatecznie, gdy w wyniku dodawania kolejnych liczb w którymś momencie będzie zachodzić tożsamość z Obserwacji 3, to sprawdzamy, jak wiele wierzchołków ma tę samą wartość cs co właśnie dodany wierzchołek (oznaczamy tę liczbę przez q) oraz jak wiele z nich jest w rozwiązaniu (tę liczbę oznaczamy przez l), i do wyniku dodajemy $\binom{q}{l}$.

To daje nam poprawne rozwiązanie bez jakiejkolwiek dalszej analizy. Wnikliwy Czytelnik dostrzeże (i samodzielnie udowodni albo wywnioskuje to z rozważań przedstawionych przy omówieniu rozwiązań alternatywnych), że l może być równe tylko 0, 1, $q-1$ lub q, a zatem otrzymane wyniki nie będą zbyt duże, dzięki czemu nie musimy wcześniej obliczać liczb $\binom{q}{l}$. Dowód tego faktu pozostawiamy Czytelnikowi.

Rozwiązanie to zostało zaimplementowane w pliku `kon11.cpp`. Działa w złożoności czasowej i pamięciowej $O(n+m)$, jako że czarne stopnie wierzchołków możemy uporządkować nierosnąco za pomocą sortowania kubełkowego.

Rozwiązanie alternatywne – trójkąty

Opis tego rozwiązania zacznijmy od dwóch prostych obserwacji.

Obserwacja 4. Niech A, B i C będą trzema wierzchołkami i niech krawędzie AB i BC będą czarne, a krawędź AC szara. Wtedy w każdym dobrym kolorowaniu wierzchołek B jest czarny.

Dowód: Jeśli wierzchołek B byłby szary, to A i C musiałyby być czarne (ponieważ każda czarna krawędź musi mieć co najmniej jeden koniec czarny). Wtedy szara krawędź AC miałaby oba końce czarne, co nie jest możliwe. ■

Zauważmy zatem, że w każdym trójkącie, którego krawędzie *nie są* jednego koloru, możemy jednoznacznie wyznaczyć kolor jednego z wierzchołków. Taki trójkąt ma zawsze dwie krawędzie tego samego koloru — wspólny wierzchołek tych dwóch krawędzi ma jednoznacznie wyznaczony kolor.

Obserwacja 5. Jeśli mamy n niepokolorowanych wierzchołków, które tworzą jednokolorową klikę (każda krawędź między tymi wierzchołkami jest tego samego koloru), to możemy je dobrze pokolorować na $n+1$ sposobów.

Dowód: Załóżmy, że wierzchołki tworzą czarną klikę (dowód w przypadku szarej jest analogiczny). Możemy wtedy:

- pokolorować wszystkie wierzchołki na czarno lub
- wybrać jeden wierzchołek, pokolorować go na szaro, zaś resztę pokolorować na czarno.

Nie możemy pokolorować dwóch lub więcej wierzchołków na szaro, bo między każdą parą wierzchołków jest czarną krawędź, a ona nie może mieć dwóch końców szarych. Mamy zatem dokładnie $n + 1$ rozwiązań. ■

Rozwiązanie powolne – $O(n^4)$

Jeśli znajdziemy jakiś różnokolorowy trójkąt, to możemy odpowiedni jego wierzchołek pokolorować na odpowiedni kolor (dokładniej, możemy wyznaczyć jeden z wierzchołków i kolor, który ten wierzchołek będzie miał w *każdym* dobrym kolorowaniu). Następnie możemy sprawdzić, czy pokolorowanie tego wierzchołka nie wymusza pokolorowania jego sąsiadów. Jeśli jakiś wierzchołek pokolorujemy na czarno (odpowiednio na szaro), możemy wszystkich jego sąsiadów, z którymi jest on połączony szarą (odpowiednio czarną) krawędzią, pokolorować na szaro (odpowiednio czarno). Dla każdego pokolorowanego sąsiada możemy też sprawdzić rekurencyjnie, czy jego pokolorowanie wymusza kolejne pokolorowania. Wszystkie pokolorowania, które wykonamy w ten sposób, są zgodne z dowolnym dobrym kolorowaniem grafu. W szczególności, gdybyśmy w wyniku takiego pokolorowania otrzymali dwa wierzchołki tego samego koloru połączone krawędzią przeciwnego koloru, to wiemy, że nie istnieje żadne dobre kolorowanie tego grafu. Po sprawdzeniu wszystkich wymuszeń ograniczamy nasz graf G do podgrafu G' zawierającego tylko niepokolorowane wierzchołki i rozwiązujemy rekurencyjnie nasz problem z mniejszym n. Zauważmy, że zachodzi następujący fakt.

Obserwacja 6. Każde dobre kolorowanie grafu G' (skonstruowanego w opisany powyżej sposób) tworzy wraz z wyznaczonym wcześniej pokolorowaniem pozostałych wierzchołków dobre kolorowanie grafu G.

Dowód: Załóżmy przeciwnie. To znaczy, że istnieje pewna krawędź uv w G, dla której u i v obydwa mają inny kolor niż krawędź uv. Nie może być tak, że u i v oba są w G', bo G' jest z założenia dobrze pokolorowane. Nie może być też tak, że u i v oba są poza G', bo już na etapie wymuszeń stwierdzilibyśmy, że nie istnieją dobre kolorowania. Załóżmy zatem, że $u \in G'$, $v \notin G'$. Ale wtedy krawędź łącząca u z v jest innego koloru niż v — zatem kolor u zostałby ustalony na etapie rekurencyjnych sprawdzeń. ■

Załóżmy zatem, że po przejrzeniu wszystkich różnokolorowych trójkątów i wykonaniu wszystkich wymuszeń rekurencyjnych pozostało nam jeszcze k niepokolorowanych wierzchołków. Skoro nie ma żadnego różnokolorowego trójkąta o wszystkich wierzchołkach wśród tych k, to tworzą one jednokolorową klikę (czarną lub szarą). Taką klikę możemy pokolorować na $k + 1$ sposobów.

Na początek możemy zatem rozważyć następujące naiwne rozwiązanie:

```
1:  function koloruj(int A, char K)
2:  begin
3:    kolor[A] := K;
4:    for B := 1 to n do
5:      if kolor(A, B) ≠ kolor[A] then begin
6:        if kolor[B] = '?' then return false;
7:        if kolor[B] = '?' then
8:          if not koloruj(B, kolor(A, B)) then return false;
9:      end
10:   return true;
11: end
12:
13: program Konspiracja
14: begin
15:   for A := 1 to n do kolor[A] := '?';
16:   zmieniany := true;
17:   while zmieniany do begin
18:     zmieniany := false;
19:     foreach A, B, C ∈ [1, n] do
20:       if kolor[A] = kolor[B] = kolor[C] = '?' then
21:         if (kolor(A, B) ≠ kolor(A, C)) and
22:               (kolor(A, B) ≠ kolor(B, C)) then begin
23:           zmieniany := true;
24:           if not koloruj(C, kolor(A, C)) then return 0;
25:         end
26:   end
27:   wynik := 1;
28:   for A := 1 to n do
29:     if kolor[A] = '?' then wynik := wynik + 1;
30:   return wynik;
31: end
```

Za każdym obrotem pętli **while** liczba niepokolorowanych wierzchołków (tych o kolorze '?') maleje przynajmniej o 1, zatem tych obrotów jest $O(n)$. Wyszukiwanie trójkątów zajmuje czas $O(n^3)$. Zanalizujmy jeszcze rekurencyjne kolorowanie (czyli funkcję *koloruj*). Otóż funkcja *koloruj* jest wywoływana tylko dla wierzchołka bez koloru i go koloruje — zatem w trakcie całego programu jest co najwyżej n wywołań tej funkcji. Pojedyncze wywołanie tej funkcji, nie licząc zejścia rekurencyjnego, jest realizowane w czasie $O(n)$ (przejrzenie wszystkich sąsiadów), a zatem łączny czas potrzebny na wszystkie wywołania funkcji *koloruj* w całym algorytmie to $O(n^2)$. Ten fakt warto zapamiętać, z funkcji *koloruj* będziemy jeszcze korzystać w następnych rozwiązaniach, a teraz zobaczyliśmy, że same wywołania tej funkcji zajmują wystarczająco mało czasu, byśmy mogli je wykonać w rozwiązaniu wzorcowym.

Zatem łączny koszt czasowy tego algorytmu to $O(n^4)$, a pamięciowy — $O(n^2)$. Nieco gorszą jego implementację można znaleźć w plikach `kons1.cpp` i `kons2.pas`. Na zawodach za takie rozwiązanie można było uzyskać do 30 punktów.

Rozwiązanie powolne – $O(n^3)$

Można zauważyć, że w poprzednim rozwiązaniu często zdarza się, że wielokrotnie przeglądamy jeden trójkąt. A zamiast tego możemy przeglądać wszystkie trójkąty po kolei, nie wracając już do tych raz przejrzanych. Wszystkich trójkątów jest $O(n^3)$, a wymuszanie amortyzuje się do czasu $O(n^2)$.

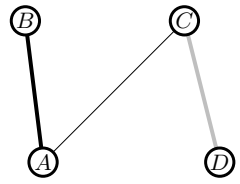
Rozwiązanie to zostało zaimplementowane w plikach `kons3.cpp` i `kons4.pas`. Działa w złożoności czasowej $O(n^3)$ i pamięciowej $O(n^2)$. Uzyskuje 50 punktów.

Rozwiązanie efektywne – $O(n^2)$

Postaramy się poprawić poprzednie rozwiązanie. Chcemy szybciej wyszukiwać różnokolorowe trójkąty, które wymuszą nam kolorowanie jeszcze nie pokolorowanych wierzchołków.

Obserwacja 7. Jeśli mamy dwie różnokolorowe krawędzie AB i CD, to z nich możemy od razu otrzymać różnokolorowy trójkąt.

Dowód: Jeśli te krawędzie mają wspólny wierzchołek, to już mamy ten trójkąt (bo krawędź między dwoma różnymi wierzchołkami będzie miała inny kolor niż krawędź AB lub CD).



Rys. 3: Ilustracja drugiej części dowodu Obserwacji 7.

Teraz załóżmy, że te krawędzie nie mają wspólnego wierzchołka. Prowadzimy krawędź AC. Jeśli AC ma taki sam kolor jak AB, to krawędzie AC i CD tworzą różnokolorowy trójkąt. Jeśli AC ma taki sam kolor jak CD, to krawędzie AC i AB tworzą różnokolorowy trójkąt. ■

Nasz ulepszony algorytm działa następująco. Tworzymy dwie listy krawędzi; na jednej liście będziemy trzymać czarne krawędzie, a na drugiej szare. Początkowo każdą krawędź wkładamy na odpowiednią listę. Krawędź będziemy zdejmować z listy dopiero wtedy, kiedy przynajmniej jeden z jej końców będzie już pokolorowany.

Załóżmy, że w pewnym momencie wykonywania algorytmu obie listy są niepuste. Weźmy dwie różnokolorowe krawędzie, będące pierwszymi elementami list. Wówczas, korzystając z Obserwacji 7, możemy w czasie stałym wygenerować różnokolorowy trójkąt. Każdy wierzchołek tego trójkąta jest końcem jednej z naszych krawędzi. Na mocy Obserwacji 4 możemy teraz dla jednego z wierzchołków trójkąta wywołać funkcję *koloruj*.

Za każdym razem, kiedy wywołujemy funkcję *koloruj* i przeglądamy sąsiadów wierzchołka, próbujemy usunąć z odpowiedniej listy krawędzie łączące ten wierzchołek

z jego sąsiadami. (Uwaga implementacyjna: łatwiej jest, zamiast usuwać krawędź z listy w momencie kolorowania jednego z jej końców, sprawdzać dopiero przy pobieraniu z listy, czy dana krawędź nie powinna była zostać usunięta). Jeśli w trakcie kolorowania otrzymamy sprzeczność, to wypisujemy zero i kończymy działanie programu.

Powiedzmy zatem, że któraś z list, po pewnej liczbie kroków, stanie się pusta (kroków jest co najwyżej n, bo co najwyżej tyle razy możemy wywołać funkcję *koloruj*). To oznacza, że niepokolorowane wierzchołki tworzą jednokolorową klikę, a zatem, na mocy Obserwacji 5, mamy $k+1$ rozwiązań, przy czym k to liczba niepokolorowanych wierzchołków.

Tworzenie i przeglądanie list zajmuje czas $O(n^2)$ (tyle jest krawędzi), wszystkie wywołania funkcji *koloruj* zajmują łącznie również czas $O(n^2)$.

Rozwiązanie to zostało zaimplementowane w `kon.cpp`, `kon1.pas` i `kon2.cpp`. Działa w złożoności czasowej $O(n^2)$ i pamięciowej $O(n^2)$. Uzyskuje 100 punktów.

Rozwiązanie alternatywne – 2-SAT z poprawianiem

W tym rozwiązaniu zauważamy, że jeśli mamy dobre kolorowanie, to inne dobre kolorowania nie mogą się od niego za bardzo różnić.

Załóżmy, że w pewnym dobrym kolorowaniu K wierzchołki u i v są czarne. Ale wtedy, ponieważ K jest dobre, mamy, że krawędź uv jest czarna — a zatem w dowolnym dobrym kolorowaniu co najwyżej jeden z wierzchołków u, v jest szary. Stąd, spośród wierzchołków, które są pokolorowane na czarno w K, w dowolnym innym dobrym pokolorowaniu co najwyżej jeden jest szary (i analogicznie spośród szarych w K co najwyżej jeden jest czarny). To oznacza, że dowolne inne dobre kolorowanie mogę dostać z K przez jedną z trzech następujących akcji:

- przemalowanie jednego czarnego wierzchołka na szaro
- przemalowanie jednego szarego wierzchołka na czarno
- zamiana jednego czarnego wierzchołka z jednym szarym wierzchołkiem (czyli przemalowanie czarnego wierzchołka na szaro, a szarego na czarno).

W tym rozwiązaniu znajdziemy jakieś dobre kolorowanie i obliczymy, ile innych pokolorowań da się uzyskać z tego kolorowania za pomocą wyżej wymienionych akcji.

Problem 2-SAT

Chcemy zacząć od znalezienia jakiegokolwiek rozwiązania. W tym celu sprowadzimy nasz problem do znanego problemu 2-SAT. W problemie 2-SAT mamy dane N zmiennych logicznych (przyjmujących wartości **true** i **false**). Mamy też dany ciąg M *klauzul*, czyli formuł logicznych postaci $a \vee b$, przy czym każda z liter a, b oznacza albo pojedynczą zmienną, albo jej negację. W problemie 2-SAT pytamy, czy istnieje takie przypisanie zmiennym wartości, że każda z M klauzul jest spełniona (tzn. prawdziwa), a jeśli tak, to prosimy o podanie takiego wartościowania. Problem 2-SAT można rozwiązać w czasie $O(N + M)$, taki algorytm można znaleźć np. w opisie rozwiązania

zadania *Śluzy* z Bałtyckiej Olimpiady Informatycznej 2008 [1], zapisany w niejawny sposób w opracowaniu zadania *Spokojna komisja* z VIII Olimpiady Informatycznej [8] albo na stronie `http://was.zaa.mimuw.edu.pl/?q=node/39`.

Jak ten problem ma się do naszego zadania? Otóż stowarzyszymy z każdym wierzchołkiem v zmienną x_v, która przyjmuje wartość **true**, jeśli x_v jest pokolorowany na czarno, a **false**, jeśli na szaro. Z każdą krawędzią czarną uv kojarzymy klauzulę $x_u \vee x_v$ (bo jeden z jej końców musi być czarny), a z każdą szarą krawędzią kojarzymy $\neg x_u \vee \neg x_v$ (bo któryś koniec musi być szary). I teraz, faktycznie, rozwiązanie problemu 2-SAT dla tego zestawu zmiennych i klauzul jest równoważne dobremu kolorowaniu.

Algorytm 2-SAT będzie tu działał w czasie $O(n^2)$, gdyż w naszym przypadku $N = n$, $M = n(n-1)/2$.

Rozwiązanie powolne – $O(n^4)$

Przy pomocy algorytmu 2-SAT generujemy jakiekolwiek rozwiązanie. Następnie wykonujemy każdą z $O(n^2)$ możliwych akcji i sprawdzamy, czy uzyskane rozwiązanie jest poprawne. Mamy $O(n)$ możliwych akcji przemalowania jednego wierzchołka na inny kolor i $O(n^2)$ par wierzchołków do zamiany kolorów. Zliczamy otrzymane rozwiązania (łącznie z pierwotnym, wygenerowanym przez algorytm 2-SAT) i wypisujemy odpowiedź.

Rozwiązanie to zostało zaimplementowane w plikach `kons5.cpp` i `kons6.pas`. Działa w złożoności czasowej $O(n^4)$ i pamięciowej $O(n^2)$. Na zawodach uzyskiwało 30 punktów.

Rozwiązanie powolne – $O(n^3)$

Widać, że najbardziej czasochłonne jest sprawdzanie akcji typu „zamiana wierzchołków". Zauważmy, że wierzchołek czarny można zamienić z szarym tylko wtedy, gdy jest to jego jedyny szary sąsiad, z którym jest połączony czarną krawędzią, ewentualnie gdy w ogóle nie ma żadnych takich szarych sąsiadów połączonych czarną krawędzią. Faktycznie, jeśli byłoby więcej takich sąsiadów, to po zamianie tej pary dwa szare wierzchołki byłyby połączone czarną krawędzią, co dałoby sprzeczność. Podobnie, jeśli mamy dokładnie jednego takiego sąsiada i próbowalibyśmy dokonać zamiany z jakimś innym szarym wierzchołkiem.

Dzięki temu spostrzeżeniu liczba par do sprawdzenia w porównaniu do poprzedniego rozwiązania zmalała nam z $O(n^2)$ do $O(n)$. Mamy dalej $O(n)$ możliwych akcji przemalowania jednego wierzchołka na inny kolor i tylko $O(n)$ par wierzchołków do zamiany kolorów. Poprawność rozwiązania weryfikujemy w czasie $O(n^2)$.

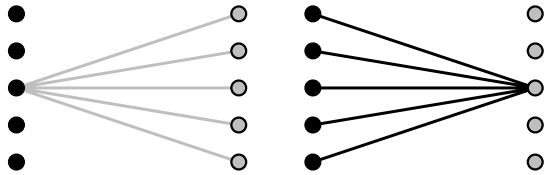
Rozwiązanie to zostało zaimplementowane w plikach `kons7.cpp` i `kons8.pas`. Działa w złożoności czasowej $O(n^3)$ i pamięciowej $O(n^2)$. Uzyskuje 50 punktów.

Rozwiązanie efektywne – $O(n^2)$

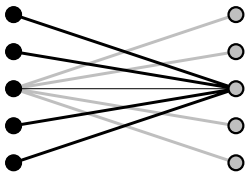
Wiemy, że może się zdarzyć, że jest $\Theta(n)$ rozwiązań (np. gdy graf wejściowy jest czarną kliką). Wobec tego będziemy musieli wykonać $\Omega(n)$ sprawdzeń poprawności

[1] Książeczka BOI 2008 jest dostępna pod adresem `http://b08.oi.edu.pl/downloads/booklet.pdf` (jedynie w wersji anglojęzycznej).

(czyli $\Omega(n)$ akcji). Zatem, by osiągnąć satysfakcjonującą nas efektywność algorytmu, musimy szybciej niż w czasie $O(n^2)$ sprawdzać poprawność rozwiązania. W tym celu zrezygnujemy z naiwnego sprawdzania poprawności.



Aby dało się przemalować wierzchołek czarny na szaro, musi on być połączony z każdym szarym wierzchołkiem szarą krawędzią. Jeśli byłby połączony z jakimś szarym wierzchołkiem czarną krawędzią, to po jego przemalowaniu ta krawędź prowadziłaby do sprzeczności. Analogiczne spostrzeżenie można poczynić przy przemalowywaniu szarego wierzchołka na czarno, patrz rysunek powyżej. W obydwu przypadkach otrzymany warunek na poprawność przemalowania jest zarazem konieczny i dostateczny.



Łącząc te dwa spostrzeżenia, można zauważyć, że czarny wierzchołek u możemy zamienić z szarym wierzchołkiem v wtedy i tylko wtedy, gdy u jest połączony z każdym szarym wierzchołkiem prócz v szarą krawędzią, a v jest połączony z każdym czarnym wierzchołkiem prócz u czarną krawędzią. Kolor krawędzi między zamienianymi wierzchołkami może być dowolny, patrz też rysunek powyżej.

Załóżmy, że nasze startowe rozwiązanie ma k czarnych wierzchołków i $n - k$ szarych. Wtedy, korzystając z powyższych spostrzeżeń, wnioskujemy, że:

- czarny wierzchołek u możemy przemalować na szaro, jeśli ma dokładnie $k - 1$ incydentnych czarnych krawędzi;
- analogicznie szary wierzchołek możemy przemalować na czarno, jeśli ma dokładnie k incydentnych czarnych krawędzi;
- wierzchołki u (czarny) i v (szary) możemy zamienić kolorami, jeśli uv jest czarna, zaś u i v mają po k czarnych sąsiadów, lub uv jest szara i u i v mają po $k - 1$ czarnych sąsiadów.

Jeśli na wstępie zliczymy szare krawędzie incydentne z każdym z wierzchołków (czas $O(n^2)$), to następnie poprawność każdej akcji możemy sprawdzić w czasie $O(1)$.

To rozwiązanie zostało zaimplementowane w plikach `kon5.cpp` i `kon6.pas`. Działa w złożoności czasowej $O(n^2)$ i pamięciowej $O(n^2)$. Uzyskuje 100 punktów.

Rozwiązanie alternatywne – konstrukcja rozwiązań

Tym razem podejdziemy do tego zadania jeszcze inaczej. Można, mianowicie, zauważyć, że wszystkich dobrych kolorowań będzie co najwyżej $n+1$ (wynika to z analizy dowolnego z poprzednich dwóch rozwiązań). Postaramy się skonstruować wszystkie rozwiązania od początku w sposób iteracyjny. Zaczniemy od $n = 1$ — wtedy są 2 rozwiązania. Teraz będziemy dodawać kolejne wierzchołki i poszerzać na bieżąco wszystkie rozwiązania.

Rozwiązanie powolne – $O(n^3)$

Dla każdego $k = 1, 2, \ldots, n$ utrzymujemy listę wszystkich poprawnych kolorowań dla pierwszych k wierzchołków. Tych kolorowań może być maksymalnie $k+1$. Teraz pokażemy, jak dodać jeden wierzchołek i uzyskać kolorowania dla $k+1$ wierzchołków.

Dodajemy nowy wierzchołek, nadając mu najpierw kolor czarny, a potem szary, i sprawdzamy, które z rozwiązań dla k są niesprzeczne po rozszerzeniu ich do $k+1$. Sprawdzenie wykonujemy w czasie $O(k)$, bo wystarczy sprawdzić, czy krawędzie wychodzące z nowo dodanego wierzchołka nie powodują sprzeczności.

Potencjalnie z każdego rozwiązania dla k wierzchołków można uzyskać dwa nowe rozwiązania, a zatem mamy $O(k)$ rozwiązań do sprawdzenia. W ten sposób wszystkie kolorowania dla $k+1$ wierzchołków konstruujemy w czasie $O(k^2)$. Zwróćmy uwagę, że w tym podejściu w istotny sposób korzystamy z tego, że *a priori* wiemy, iż rozważanych rozwiązań będzie w każdym kroku co najwyżej $O(k)$, co wcale z analizy tego algorytmu nie wynika.

Rozwiązanie to zostało zaimplementowane w plikach `kons9.cpp` i `kons10.pas`. Działa w złożoności czasowej $O(n^3)$ i pamięciowej $O(n^2)$. Na zawodach uzyskiwało 50 punktów.

Rozwiązanie efektywne – $O(n^2)$

Aby powyższe rozwiązanie usprawnić, należy porządnie zanalizować, jaką postać mają wszystkie poprawne kolorowania. Przyglądając się np. pierwszym dwóm rozwiązaniom, widzimy, że istnieje zbiór wierzchołków, które mają ten sam kolor w każdym dobrym kolorowaniu, oraz jakaś reszta. Wszystkie krawędzie w obrębie reszty mają ten sam kolor, powiedzmy czarny. Jeśli wierzchołków w reszcie jest k, to mamy $k+1$ rozwiązań — jedno, które wszystkie k wierzchołków koloruje na czarno, i k, które dokładnie jeden z k wierzchołków kolorują na szaro.

Podobnie jak w poprzednim rozwiązaniu, będziemy konstruowali iteracyjnie kolorowania dla kolejnych k. Jednak tym razem nie będziemy trzymać wszystkich kolorowań, lecz będziemy pamiętać, które wierzchołki mają już wymuszone kolory i jakie, a które tworzą jednokolorową klikę.

Dodając nowy wierzchołek v, rozpoczynamy od sprawdzenia, czy krawędzie z v do już pokolorowanych wierzchołków nie wymuszają kolorowania v (czyli czy czarne krawędzie nie idą do szarych wierzchołków lub czy szare krawędzie nie idą do czarnych). Jeśli okaże się, że nowy wierzchołek ma wymuszony kolor, to sprawdzamy, czy nie wymusza on kolorów wierzchołków z kliki (jeśli wymusza, to kolorujemy je, usuwamy z kliki i przerzucamy je do zbioru wierzchołków pokolorowanych).

Załóżmy dla uproszczenia, że w klice mamy same czarne krawędzie. Wtedy jeżeli jakikolwiek wierzchołek kliki wymusimy na szaro, to wszystkie pozostałe wierzchołki kliki automatycznie wymuszają się na czarno, natomiast wymuszenie jakiegokolwiek wierzchołka kliki na czarno nie wymusza niczego na pozostałych wierzchołkach kliki. W szczególności, gdyby v miał wymusić dwa wierzchołki w klice na szaro, to możemy od razu zwrócić zero. Dzięki tym spostrzeżeniom, przy wymuszaniu nie musimy przeglądać krawędzi należących do kliki.

Jeżeli nowo dodany wierzchołek v miał wymuszony kolor, to po wymuszeniach w klice możemy przejść do dodawania następnego wierzchołka. Tu uwaga — może się zdarzyć, że nasza „klika", którą pamiętamy, stanie się w wyniku wymuszeń pusta albo jednoelementowa, kiedy to nie ma jednoznacznie określonego koloru. Dla algorytmu jest obojętne, jaki kolor w takiej sytuacji uznamy, że ma klika.

Jeśli pokolorowane wierzchołki nie wymuszają żadnego koloru nowemu wierzchołkowi, to teraz patrzymy na krawędzie z niego do kliki. Trzeba rozważyć kilka przypadków:

- jeśli wszystkie krawędzie do kliki są tego samego koloru co klika, to po prostu dodajemy ten wierzchołek do kliki;
- jeśli jest dokładnie jedna krawędź innego koloru, to wszystkie wierzchołki (poza tym, do którego idzie ta jedna krawędź) kolorujemy na kolor, jakiego była klika, a nową kliką staje się ta jedna krawędź innego koloru. Tu korzystamy ze spostrzeżenia o kolorowaniu trójkątów różnokolorowych;
- jeśli są co najmniej dwie krawędzie innego koloru, to dostajemy różnokolorowy trójkąt, który wymusza kolor nowo dodanego wierzchołka, i przechodzimy do przypadku, w którym dodawany wierzchołek ma wymuszony kolor. Tu, ponownie, korzystamy ze spostrzeżenia o trójkątach różnokolorowych.

Dodanie nowego wierzchołka zajmuje czas $O(k)$. Rozwiązanie to zostało zaimplementowane w plikach `kon7.cpp`, `kon8.cpp` i `kon9.pas`. Działa w złożoności czasowej i pamięciowej $O(n^2)$. Uzyskuje 100 punktów.

Testy

Zostało przygotowanych 10 zestawów testów po 3 testy w zestawie (w niektórych 5). Testy a to testy losowe, w których wynikiem zawsze jest 0. Testy b to testy losowe, w których wynik jest mały i rozwiązania mają prawie tyle samo czarnych i szarych wierzchołków. Testy c to testy losowe, w których wynik jest duży.

Nazwa	n
kon1a.in	10
kon1b.in	8
kon1c.in	2
kon1d.in	8

Nazwa	n
kon1e.in	2
kon2a.in	27
kon2b.in	29
kon2c.in	30

Nazwa	n
kon3a.in	90
kon3b.in	78
kon3c.in	84
kon4a.in	123

Nazwa	n
kon4b.in	196
kon4c.in	139
kon5a.in	195
kon5b.in	254
kon5c.in	187
kon6a.in	1 300
kon6b.in	1 534
kon6c.in	1 832

Nazwa	n
kon7a.in	1 500
kon7b.in	1 953
kon7c.in	2 343
kon8a.in	2 500
kon8b.in	2 693
kon8c.in	2 783
kon9a.in	3 900

Nazwa	n
kon9b.in	3 984
kon9c.in	4 032
kon10a.in	5 000
kon10b.in	4 963
kon10c.in	4 984
kon10d.in	5 000
kon10e.in	5 000

Jakub Pachocki
Treść zadania, Opracowanie

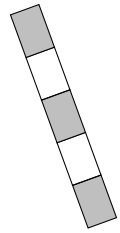
Mateusz Baranowski
Program

Dostępna pamięć: 64 MB.

OI, Etap I, 18.10–15.11.2010

Lizak

Bajtazar prowadzi w Bajtogrodzie sklep ze słodyczami. Wśród okolicznych dzieci najpopularniejszymi słodyczami są lizaki waniliowo-truskawkowe. Składają się one z wielu segmentów jednakowej długości, z których każdy ma jeden smak — waniliowy lub truskawkowy. Cena lizaka jest równa sumie wartości jego segmentów; segment waniliowy kosztuje jednego bajtalara, a truskawkowy dwa bajtalary.



Rys. 1: Przykładowy lizak o pięciu segmentach, trzech truskawkowych i dwóch waniliowych, ułożonych na przemian. Cena tego lizaka wynosi 8 bajtalarów.

Obecnie Bajtazarowi został na składzie tylko jeden (za to być może bardzo długi) lizak. Bajtazar zdaje sobie sprawę, że być może nikt nie będzie chciał go kupić w całości, dlatego dopuszcza możliwość łamania go na granicach segmentów w celu uzyskania lizaka o mniejszej długości. Fragment lizaka przeznaczony ostatecznie do sprzedaży musi pozostać niepołamany.

Doświadczenie pokazuje, że klienci najczęściej chcą kupić lizaka za całe swoje kieszonkowe. Bajtazar zastanawia się, dla wielu możliwych wartości k, jak przełamać posiadany lizak tak, aby otrzymać lizak o cenie równej dokładnie k bajtalarów. Ponieważ zadanie nie jest wcale proste, poprosił Cię o pomoc.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie liczby całkowite n oraz m ($1 \leqslant n, m \leqslant 1\,000\,000$) oddzielone pojedynczym odstępem. Oznaczają one odpowiednio liczbę segmentów ostatniego pozostałego w sklepie lizaka oraz liczbę rozpatrywanych wartości k. Segmenty lizaka są ponumerowane kolejno od 1 do n. W drugim wierszu znajduje się n-literowy opis lizaka, złożony z liter `T` *i* `W`*, przy czym* `T` *oznacza segment truskawkowy, zaś* `W` *— waniliowy; i-ta z tych liter opisuje smak i-tego segmentu. W kolejnych m wierszach znajdują się kolejne wartości k do rozpatrzenia ($1 \leqslant k \leqslant 2\,000\,000$), po jednej w wierszu.*

Wyjście

Twój program powinien wypisać na standardowe wyjście dokładnie m wierszy zawierających wyniki dla kolejnych wartości k, po jednym wyniku w wierszu. Jeśli dla danej wartości k nie da

się wyłamać z lizaka spójnego fragmentu o wartości równej k bajtalarów, należy wypisać słowo `NIE`*. W przeciwnym przypadku należy wypisać dwie liczby l oraz r (*$1 \leqslant l \leqslant r \leqslant n$*) oddzielone pojedynczym odstępem, takie że fragment lizaka złożony z segmentów o numerach od l do r włącznie ma wartość dokładnie k bajtalarów. Jeśli istnieje wiele możliwych odpowiedzi, Twój program może podać dowolną z nich.*

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`5 3`	`1 3`
`TWTWT`	`2 2`
`5`	`NIE`
`1`	
`7`	

Wyjaśnienie do przykładu: *Przykład opisuje lizak z rys. 1. Segmenty o numerach od 1 do 3 tworzą lizak postaci* `TWT`*, wart 5 bajtalarów. Segment numer 2 ma smak waniliowy i kosztuje 1 bajtalara. Z tego lizaka nie da się w żaden sposób uzyskać lizaka wartego 7 bajtalarów.*

Rozwiązanie

Wprowadzenie

Występujący w zadaniu lizak możemy wyobrazić sobie jako n-elementowy ciąg złożony z jedynek (segmenty waniliowe) i dwójek (segmenty truskawkowe). Musimy umieć odpowiedzieć na m zapytań postaci: czy jakiś spójny (tzn. jednokawałkowy) fragment lizaka ma sumę dokładnie k. Zauważmy, że wartości parametrów n oraz m w zadaniu mogą być dosyć duże (górne ograniczenie: 1 000 000).

Rozwiązanie o koszcie czasowym $O(n^2 \cdot m)$

W najprostszym rozwiązaniu na każde zapytanie odpowiadamy niezależnie. Odpowiedź na pojedyncze zapytanie (z parametrem k) wymaga wówczas przejrzenia wszystkich $\frac{n \cdot (n+1)}{2}$ możliwych do wyłamania lizaków, obliczenia ceny każdego z nich i sprawdzenia, czy jest równa k.

Złożoność czasowa takiego rozwiązania wynosi $O(n^3 \cdot m)$. Przy bardziej przemyślanej implementacji można uzyskać czas działania $O(n^2 \cdot m)$, jeśli nie będziemy wyznaczać ceny każdego lizaka od nowa, ale skorzystamy z wcześniej obliczonych wartości. Przykładowo, możemy najpierw rozważyć wszystkie fragmenty lizaka zaczynające się od pierwszego segmentu (łącznie w czasie $O(n)$), następnie fragmenty rozpoczynające się od drugiego segmentu itd.

Rozwiązanie to uzyskiwało na zawodach około 14 punktów. Jego implementacja znajduje się w plikach `lizs0.c` i `lizs1.pas`.

Rozwiązanie o koszcie czasowym $O(n^2 + m)$

Powyższe rozwiązanie można usprawnić, przeglądając na wstępie wszystkie możliwe do wyłamania lizaki i zapamiętując, dla każdej możliwej ceny k z zakresu od 1 do $2n$, jeden z przedziałów reprezentujących fragment lizaka o koszcie k, oczywiście jeśli taki przedział istnieje. Umożliwia to późniejsze odpowiadanie na zapytania w czasie stałym.

Rozwiązanie to zostało zaimplementowane w plikach `lizs2.c` i `lizs3.pas`. Na zawodach uzyskiwało około 29 punktów.

Rozwiązanie wzorcowe

Rozwiązanie wzorcowe opiera się na następującej obserwacji.

Fakt 1. *Mając dany fragment lizaka $[l, r]$ o koszcie $k \geqslant 3$, możemy w czasie stałym wyznaczyć pewien fragment o koszcie $k - 2$.*

Dowód: Jeżeli pierwszy lub ostatni segment naszego fragmentu jest truskawkowy (ma koszt równy 2), to wystarczy go odłamać. Jeżeli oba końce są waniliowe, odłamujemy obydwa. ■

Bezpośrednio z powyższego faktu otrzymujemy następujący wniosek:

Wniosek 1. Jeżeli znamy fragment lizaka o maksymalnym koszcie parzystym i fragment o maksymalnym koszcie nieparzystym, jesteśmy w stanie w czasie $O(n)$ wyznaczyć fragmenty o wszystkich możliwych wartościach.

Zauważmy, że jednym z dwóch fragmentów wymienionych we Wniosku 1 jest zawsze cały lizak. Drugim natomiast jest najdłuższy fragment zawierający o jeden segment waniliowy mniej niż cały lizak. Taki fragment można znaleźć w czasie liniowym, wyszukując pozycje skrajnych segmentów waniliowych w lizaku, l i r, i wybierając dłuższy z fragmentów $[l + 1, n]$ oraz $[1, r - 1]$. Dodajmy dla jasności, że jeśli lizak nie zawiera segmentów waniliowych, to podane przedziały nie istnieją (wszystkie fragmenty lizaka mają parzyste ceny).

W ten sposób przed wczytaniem zapytań zapamiętujemy wszystkie możliwe odpowiedzi (podobnie jak w drugim rozwiązaniu nieoptymalnym), a potem odpowiadamy na zapytania w czasie stałym.

Poniżej ten algorytm zapisany w pseudokodzie. Opisy poszczególnych segmentów lizaka przechowujemy w nim w tablicy $smak[1..n]$, natomiast do zapamiętywania wyników wstępnych obliczeń wykorzystujemy tablicę $przedział[1..2n]$, której wszystkie elementy są początkowo ustawione na **nil**.

```
1: procedure Spamiętaj(l, r, k)
2: begin
3:   przedział[k] := [l, r];
4:   if k ⩾ 3 then begin
5:     if smak[l] = T then Spamiętaj(l + 1, r, k − 2)
```

```
6:      else if smak[r] = 'T' then Spamiętaj(l, r − 1, k − 2)
7:      else Spamiętaj(l + 1, r − 1, k − 2);
8:    end
9:  end
10:
11: begin
12:   Wczytaj(n, m, smak);
13:   cena := 0;
14:   for i := 1 to n do
15:     if smak[i] = 'W' then cena := cena + 1
16:     else cena := cena + 2;
17:   Spamiętaj(1, n, cena);
18:   l := −1;
19:   r := −1;
20:   for i := 1 to n do
21:     if smak[i] = 'W' then begin
22:       if l = −1 then l := i;
23:       r := i;
24:     end
25:   if l ≠ −1 and r < n − l + 1 then
26:     Spamiętaj(l + 1, n, cena − 2 · l + 1)
27:   else if r ≠ −1 then
28:     Spamiętaj(1, r − 1, cena − 2 · (n − r) − 1);
29:   for i := 1 to m do begin
30:     Wczytaj(k);
31:     if (k > 2 · n) or (przedział[k] = nil) then Wypisz(„NIE")
32:     else Wypisz(przedział[k]);
33:   end
34: end
```

Złożoność czasowa rozwiązania wzorcowego to $O(n + m)$. Jego implementację można znaleźć w plikach `liz.c` i `liz0.pas`.

Co dalej?

Kluczem do rozwiązania zadania okazał się fakt, że wszystkie segmenty mają koszty równe 1 lub 2. Pozostawiamy Czytelnikowi poszukiwanie algorytmu działającego w czasie $O(n)$, w przypadku gdy lizaki Bajtazara mogą mieć również segmenty wiśniowe, kosztujące 3 bajtalary. Ciekawostką niech będzie fakt, że gdyby koszty poszczególnych segmentów mogły być jeszcze większe (ale ograniczone z góry przez stałą), zadanie można by rozwiązać w złożoności czasowej $O(n \log n)$, stosując jednakże dość zaawansowaną technikę zwaną szybką transformatą Fouriera — opis tego algorytmu można znaleźć np. w książce [22].

Testy

Zadanie było sprawdzane na 12 zestawach danych testowych, z których każdy zawierał trzy pojedyncze testy.

Nazwa	n	m	Opis
liz1a.in	1	4	minimalny test
liz1b.in	7	3	prosty test poprawnościowy
liz1c.in	10	4	mały test poprawnościowy
liz2a.in	25	8	mały test poprawnościowy
liz2b.in	50	40	mały test losowy
liz2c.in	70	300	mały test losowy, dużo wartości
liz3a.in	200	30	mały test, mało wartości, mało segmentów waniliowych
liz3b.in	1 000	200	mały test, dużo segmentów waniliowych
liz3c.in	2 000	20 000	mały test, dużo wartości, mało segmentów waniliowych
liz4a.in	5 000	10 000	średni test
liz4b.in	6 500	10 000	średni test, mało segmentów waniliowych
liz4c.in	8 000	12 000	średni test, dużo segmentów waniliowych
liz5a.in	20 000	20 000	średni test, mało segmentów waniliowych
liz5b.in	20 000	20 000	średni test, dużo segmentów waniliowych
liz5c.in	20 000	100 000	średni test, segmenty waniliowe daleko od końców lizaka
liz6a.in	50 000	80 000	średni test, mało segmentów waniliowych
liz6b.in	50 000	400 000	średni test, dużo segmentów waniliowych, dużo wartości
liz6c.in	50 000	200 000	średni test, segmenty waniliowe daleko od końców lizaka
liz7a.in	70 000	70 000	średni test
liz7b.in	70 000	100 000	średni test, dużo segmentów waniliowych
liz7c.in	80 000	200 000	średni test, segmenty waniliowe daleko od końców lizaka
liz8a.in	100 000	300 000	duży test, mało segmentów waniliowych
liz8b.in	100 000	300 000	duży test, dużo segmentów waniliowych
liz8c.in	100 000	400 000	duży test, segmenty waniliowe bardzo daleko od końców lizaka

Nazwa	n	m	Opis
liz9a.in	300 000	600 000	duży test, mało segmentów waniliowych, segmenty waniliowe daleko od końców lizaka
liz9b.in	300 000	600 000	duży test, segmenty waniliowe daleko od końców lizaka
liz9c.in	400 000	700 000	duży test, segmenty waniliowe bardzo daleko od końców lizaka
liz10a.in	500 000	500 000	duży test
liz10b.in	500 000	700 000	duży test, dużo segmentów waniliowych
liz10c.in	600 000	800 000	duży test, segmenty waniliowe bardzo daleko od końców lizaka
liz11a.in	900 000	1 000 000	duży test, mało segmentów waniliowych
liz11b.in	900 000	1 000 000	duży test, dużo segmentów waniliowych
liz11c.in	900 000	1 000 000	duży test, segmenty waniliowe bardzo daleko od końców lizaka
liz12a.in	1 000 000	1 000 000	maksymalny test, losowy
liz12b.in	1 000 000	1 000 000	maksymalny test, losowy
liz12c.in	1 000 000	1 000 000	maksymalny test, trzy segmenty waniliowe umieszczone prawie na środku lizaka

Piotr Niedźwiedź — Treść zadania, Opracowanie
Jakub Pachocki, Juliusz Sompolski — Program

Dostępna pamięć: 128 MB. *OI, Etap I, 18.10–15.11.2010*

Piorunochron

Postępujące zmiany klimatu zmusiły władze Bajtogrodu do wybudowania dużego piorunochronu, który chroniłby wszystkie budynki w mieście. Wszystkie budynki stoją w rzędzie przy jednej prostej ulicy i są ponumerowane kolejno od 1 do n.

Zarówno wysokości budynków, jak i wysokość piorunochronu wyrażają się nieujemnymi liczbami całkowitymi. Bajtogród dysponuje funduszami na wybudowanie tylko jednego piorunochronu. Co więcej, im wyższy ma być piorunochron, tym będzie droższy.

Aby piorunochron o wysokości p umieszczony na dachu budynku i (o wysokości h_i) mógł skutecznie chronić wszystkie budynki, dla każdego innego budynku j (o wysokości h_j) musi zachodzić następująca nierówność:

$$h_j \leqslant h_i + p - \sqrt{|i-j|}.$$

Tutaj $|i-j|$ oznacza wartość bezwzględną różnicy liczb i oraz j.

Bajtazar, burmistrz Bajtogrodu, poprosił Cię o pomoc. Napisz program, który dla każdego budynku i ($i = 1, \ldots, n$) obliczy, jaka jest minimalna wysokość piorunochronu, który umieszczony na budynku i będzie chronił wszystkie budynki.

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita n ($1 \leqslant n \leqslant 500\,000$) oznaczająca liczbę budynków w Bajtogrodzie. W każdym z kolejnych n wierszy znajduje się jedna liczba całkowita h_i ($0 \leqslant h_i \leqslant 1\,000\,000\,000$), oznaczająca wysokość i-tego budynku.

Wyjście

Twój program powinien wypisać na standardowe wyjście n wierszy. W i-tym wierszu powinna znaleźć się nieujemna liczba całkowita p_i, oznaczająca minimalną wysokość piorunochronu na i-tym budynku.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
6	2
5	3
3	5
2	3
4	5
2	4
4	

Rozwiązanie

Wprowadzenie

Uproszczenie

W zadaniach, w których badany obiekt jest symetryczny, często można rozpatrzyć osobno jego lewą i prawą część, po czym na tej podstawie obliczyć wynik dla całości. W naszym przypadku tak właśnie jest. Wystarczy, że zaprojektujemy algorytm obliczający wynik dla prawostronnych piorunochronów — wówczas dla budynku i szukamy takiej minimalnej wysokości p_i, żeby dla każdego $j \geqslant i$ zachodziło:

$$h_j \leqslant h_i + p_i - \sqrt{|i-j|}.$$

Analogicznie rozważamy wysokość l_i lewostronnego piorunochronu, tj. dla $j \leqslant i$. Ostateczna wysokość i-tego piorunochronu to $\max(l_i, p_i)$.

Od tej pory skupimy się tylko na obliczeniu wartości p_i; wartości l_i obliczymy za pomocą tego samego algorytmu zapisanego dla odwróconego ciągu wysokości budynków.

Wzór na p_i

Zgodnie z treścią zadania, aby piorunochron na dachu budynku i chronił budynek j (dla $j \geqslant i$), musi zachodzić nierówność:

$$h_j \leqslant h_i + p_i - \sqrt{j-i}.$$

Jest ona równoważna następującej:

$$p_i \geqslant h_j + \sqrt{j-i} - h_i.$$

Wprowadźmy oznaczenie $piorunochron(i,j) = h_j + \sqrt{j-i} - h_i$. Skoro szukamy możliwie najmniejszego całkowitego p_i, to musi być ono równe:

$$p_i = \max_{j \geqslant i} \{\lceil piorunochron(i,j) \rceil\}. \tag{1}$$

W powyższym zapisie wyrażenie $\lceil x \rceil$ oznacza najmniejszą liczbę naturalną nie mniejszą niż x („sufit z x").

Obliczanie pierwiastków

Niezależnie od tego, jaką metodą będziemy próbowali rozwiązać to zadanie, nie unikniemy konieczności obliczania pierwiastków liczb całkowitych (lub tych pierwiastków zaokrąglonych w górę, zależnie od rozwiązania). Języki programowania wykorzystywane na zawodach (C/C++, Pascal) mają wbudowaną funkcję `sqrt`, która pozwala obliczać pierwiastki z liczb zmiennopozycyjnych. Warto jednak pamiętać o tym, że funkcja ta jest dosyć powolna (zachęcamy Czytelników do poeksperymentowania, ile wywołań tej funkcji można wykonać w ciągu, powiedzmy, sekundy). Dlatego dobrym

pomysłem w każdym, właściwie, rozwiązaniu było spamiętanie na początku pierwiastków wszystkich liczb całkowitych od 0 do $n-1$ (lub sufitów z nich), a potem korzystanie już tylko z takich stablicowanych wartości. W dalszej części opisu pojawiają się określenia typu „wersja zoptymalizowana rozwiązania" — oznaczają one, między innymi, zastosowanie podanego tu usprawnienia.

Rozwiązanie siłowe

To rozwiązanie opiera się na bezpośrednim zastosowaniu wzoru (1). Złożoność czasowa to $O(n^2)$, a stosowne implementacje można znaleźć w plikach `pios1.cpp`, `pios2.pas`, `pios3.cpp` i `pios4.pas`. Można było za nie uzyskać, w zależności od optymalizacji programu (w tym użycia pewnych sprytnych heurystyk, szybko wykluczających niektóre budynki z rozważań), od 27 do nawet 45 punktów.

Rozwiązania $O(n\sqrt{n})$

Zauważmy, że wyrażenie $\lceil\sqrt{j-i}\rceil$ przyjmuje wartości od 0 do $\lceil\sqrt{n}\rceil$. Dla danego budynku i, budynki o numerach $j \geqslant i$ można pogrupować w przedziały, w których zachodzi $\lceil\sqrt{j-i}\rceil = k$, a dokładniej:

$$L_{i,k} \stackrel{\text{def}}{=} i + (k-1)^2 + 1 \quad \leqslant \quad j \quad \leqslant \quad i + k^2 \stackrel{\text{def}}{=} R_{i,k}.$$

Możemy w takim razie zapisać wzór (1) w nowej postaci:

$$p_i = \max_{k \in 0..\lceil\sqrt{n}\rceil} \left\{ k - h_i + \max_{L_{i,k} \leqslant j \leqslant R_{i,k}} h_j \right\}.$$

Zakładając, że dla danego przedziału potrafilibyśmy w miarę szybko znaleźć największą wartość, która w nim występuje, to wartość p_i moglibyśmy obliczyć, badając jedynie $O(\sqrt{n})$ takich przedziałów. Znajdowanie maksimum na przedziale jest już klasycznym problemem, który można rozwiązać na wiele sposobów. Przedstawimy w skrócie dwa z nich.

Drzewo przedziałowe

Drzewo przedziałowe to struktura danych, która zajmuje $O(n)$ pamięci, można ją skonstruować w czasie $O(n)$ i umożliwia ona wyszukiwanie maksimum na przedziale w złożoności czasowej $O(\log n)$. Więcej na temat drzew przedziałowych można przeczytać np. w opracowaniu zadania *Tetris 3D* z XIII Olimpiady Informatycznej [13] albo na stronie `http://was.zaa.mimuw.edu.pl/?q=node/8`.

Złożoność czasowa rozwiązania używającego drzew przedziałowych to $O(n\sqrt{n}\log n)$, a pamięciowa to $O(n)$. Implementacje znajdują się w plikach `pios5.cpp` i `pios6.pas`.

Struktura danych podobna do słownika podsłów bazowych[1]

Jest to dwuwymiarowa tablica postaci:

$$t[k][i] = \max_{i \leqslant j < i+2^k} h_j \qquad \text{dla } k = 0, 1, \ldots, \lfloor \log n \rfloor,\ i = 1, 2, \ldots, n - 2^k + 1.$$

Tablica ta ma rozmiar $O(n \log n)$ i konstruuje się ją w tej samej złożoności czasowej. Najpierw wyznaczamy $t[0][i] = h_i$. Następnie dla kolejnych k mamy:

$$t[k][i] = \max(t[k-1][i],\ t[k-1][i + 2^{k-1}]).$$

Po wypełnieniu wszystkich komórek takiej tablicy maksima na przedziałach możemy już wyznaczać w czasie stałym:

$$\max_{l \leqslant j \leqslant r} h_j \quad = \quad \max\left(t[k_{r-l+1}][l],\ t[k_{r-l+1}][r - 2^{k_{r-l+1}} + 1]\right),$$

przy czym $k_d \stackrel{\text{def}}{=} \lfloor \log_2 d \rfloor$. Wszystkie wartości k_d możemy obliczyć w złożoności czasowej $O(n)$, a następnie spamiętać w tablicy. Używając tej struktury danych do wyznaczania maksimów na przedziale, uzyskamy algorytm o łącznej złożoności czasowej $O(n \log n + n\sqrt{n}) = O(n\sqrt{n})$ i pamięciowej $O(n \log n)$. Takie rozwiązanie zostało zaimplementowane w plikach `piob3.cpp` i `piob4.pas`.

Za rozwiązania z opisywanych grup można było uzyskać od 27 do ok. 50 punktów.

Alternatywne rozwiązanie $O(n\sqrt{n})$

Opis tego rozwiązania rozpocznijmy od dwóch prostych spostrzeżeń.

Obserwacja 1. Niech M oznacza numer najwyższego budynku. Wtedy dla każdego budynku j, dla którego $h_j < h_M - \lceil \sqrt{n} \rceil$, oraz każdego budynku i zachodzi nierówność:

$$\begin{aligned} \lceil \mathit{piorunochron}(i,j) \rceil &= h_j + \left\lceil \sqrt{|j-i|} \right\rceil - h_i < \\ &< h_M + \left\lceil \sqrt{|M-i|} \right\rceil - h_i = \lceil \mathit{piorunochron}(i,M) \rceil . \end{aligned}$$

Stąd, przy wyznaczaniu wysokości piorunochronu możemy ograniczyć się do rozpatrywania budynków j o wysokościach $h_M - \lceil \sqrt{n} \rceil \leqslant h_j \leqslant h_M$.

Obserwacja 2. Rozważmy trzy budynki i, a, b, takie że $i \leqslant a < b$ oraz $h_a \leqslant h_b$. Wówczas:

$$\begin{aligned} \lceil \mathit{piorunochron}(i,a) \rceil &= h_a + \left\lceil \sqrt{a-i} \right\rceil - h_i \\ &\leqslant h_b + \left\lceil \sqrt{b-i} \right\rceil - h_i = \lceil \mathit{piorunochron}(i,b) \rceil . \end{aligned}$$

[1] Słownik podsłów bazowych to tekstowa struktura danych służąca do przypisywania podsłowom wyjściowego tekstu jednoznacznych identyfikatorów. Więcej o tej strukturze można przeczytać w książce [20] (algorytm Karpa-Millera-Rosenberga, KMR) lub w opisie rozwiązania zadania *Powtórzenia* z VII Olimpiady Informatycznej [7].

Innymi słowy, wyznaczając maksimum we wzorze (1), możemy nie rozpatrywać budynku a, jeśli położony na prawo od niego budynek b jest co najmniej tak samo wysoki jak a.

Wykorzystując podane obserwacje, możemy dojść do efektywnego, a zarazem dosyć prostego rozwiązania. Przechodzimy budynki od prawej do lewej, zapamiętując dla każdej wysokości od $h_M - \lceil\sqrt{n}\rceil$ do h_M największy numer budynku o tej wysokości. Są to tak zwane *prawostronne maksima*. Aby wyznaczyć wysokości prawostronnych piorunochronów, dla każdego budynku i sprawdzamy $O(\sqrt{n})$ kandydatów na maksimum i wybieramy największego z nich. Być może nie znajdziemy żadnego kandydata, np. dla budynku o numerze n — wtedy możemy uznać, że wysokość piorunochronu to 0. Zauważmy, że nasz algorytm może nie obliczyć prawdziwego p_i, gdyż obliczana przez nas wartość to tak naprawdę:

$$p'_i \stackrel{\text{def}}{=} \max_{j \geqslant i,\ h_M - \lceil\sqrt{n}\rceil \leqslant h_j} \{\lceil \mathit{piorunochron}(i,j) \rceil\}\,. \tag{2}$$

Jednak, na mocy Obserwacji 1, to nam wystarcza. Innymi słowy, jeśli $p'_i < p_i$, to ostateczna wysokość piorunochronu będzie poprawna po uwzględnieniu lewostronnego piorunochronu. Sumaryczna złożoność czasowa powyższego algorytmu to $O(n\sqrt{n})$, a pamięciowa to $O(n)$.

Implementacje tego rozwiązania można znaleźć w plikach `pios7.cpp`, `pios8.pas`, `pios9.cpp` i `pios10.pas`. Na zawodach takie rozwiązania, w zależności od jakości implementacji, uzyskiwały od 50 do 75 punktów.

Rozwiązanie wzorcowe

Aby uzyskać rozwiązanie wzorcowe, musimy poczynić jeszcze jedną obserwację, opisującą „podział strefy wpływów" dwóch budynków.

Lemat 1. Jeśli mamy budynki $a < b$ i $h_a > h_b$, to istnieje takie $q_{a,b}$, $0 \leqslant q_{a,b} \leqslant a-1$, że dla $i \in [1,\ q_{a,b}]$:

$$\mathit{piorunochron}(i,a) \geqslant \mathit{piorunochron}(i,b)$$

a dla $i \in [q_{a,b}+1,\ a-1]$:

$$\mathit{piorunochron}(i,a) < \mathit{piorunochron}(i,b).$$

Dowód: Zauważmy, że funkcja

$$f(x) = \sqrt{x+c} - \sqrt{x}$$

dla dowolnej stałej $c > 0$ jest malejąca[2] na przedziale $(0, \infty)$.

[2]Co ciekawe, funkcja zdefiniowana podobnym wzorem $\lceil\sqrt{x+c}\rceil - \lceil\sqrt{x}\rceil$ nie musi już być nierosnąca (dlaczego?). To uzasadnia, dlaczego w sformułowaniu Lematu 1 wyjątkowo porzuciliśmy wszechobecne w tym opracowaniu sufity z *piorunochronów* (czyli sufity z pierwiastków).

Aby się o tym przekonać, zapiszmy ją w nieco inny sposób:

$$f(x) = \sqrt{x+c} - \sqrt{x} = \frac{\left(\sqrt{x+c} - \sqrt{x}\right)\left(\sqrt{x+c} + \sqrt{x}\right)}{\sqrt{x+c} + \sqrt{x}} = \frac{c}{\sqrt{x+c} + \sqrt{x}}$$

Nietrudno zauważyć, że mianownik jest funkcją rosnącą, skąd f jest malejąca.

Wreszcie z faktu, że f jest malejąca, wnosimy, że im większe i, tym mniejsza wartość różnicy $\sqrt{a-i} - \sqrt{b-i}$, co wobec zależności

$$\mathit{piorunochron}(i,a) - \mathit{piorunochron}(i,b) = \sqrt{a-i} - \sqrt{b-i} + h_a - h_b$$

uzasadnia istnienie parametru $q_{a,b}$, dla którego nierówności z lematu są spełnione. ■

Warto zauważyć, że wartości $q_{a,b}$ możemy wyznaczać w czasie $O(\log n)$ za pomocą wyszukiwania binarnego.

Możemy już teraz przejść do opisu algorytmu wzorcowego. Zaczynamy od znalezienia wszystkich prawostronnie maksymalnych budynków o wysokościach nie mniejszych niż $h_M - \lceil\sqrt{n}\rceil$, podobnie jak w poprzednim rozwiązaniu powolnym. Niech $a_1 < a_2 < \ldots < a_d$ będą indeksami kolejnych takich budynków, oczywiście $h_{a_i} > h_{a_{i+1}}$. Przypomnijmy, że na mocy Obserwacji 1 i 2, są to jedyne budynki potrzebne przy obliczaniu wysokości prawostronnych piorunochronów (a dokładniej, przy obliczaniu wartości p'_i ze wzoru (2)).

Dla tak zdefiniowanego ciągu (a_i) chcielibyśmy wyznaczyć punkty podziału ciągu wszystkich budynków na fragmenty, których prawostronny piorunochron jest wyznaczony przez poszczególne a_i. Jako stosowne punkty podziału należałoby przyjąć kolejne wartości q_{a_1,a_2}, q_{a_2,a_3}, ... z Lematu 1. Taki podział jest poprawny, o ile dla każdego j zachodzi $q_{a_j,a_{j+1}} < q_{a_{j+1},a_{j+2}}$. Jeśli jednak w pewnym momencie mamy $q_{a_j,a_{j+1}} \geqslant q_{a_{j+1},a_{j+2}}$, to to po prostu oznacza, że budynek a_{j+1} nie ma wpływu na wysokość prawostronnego piorunochronu żadnego innego budynku, czyli możemy go wyeliminować z ciągu (a_i), zastępując dwa dotychczasowe punkty podziału punktem $q_{a_j,a_{j+2}}$. To postępowanie powtarzamy do momentu, gdy lista punktów podziału jest rosnąca.

W ten sposób uzyskujemy, potencjalnie krótszy, ciąg prawostronnych maksimów $a'_1, \ldots, a'_g$, z odpowiadającym mu ciągiem punktów podziału:

$$s_i = q_{a'_i,a'_{i+1}} \text{ dla } 1 \leqslant i \leqslant g-1, \quad \text{a dodatkowo } s_0 = 0,\ s_g = a'_g.$$

Wówczas dla każdego $m \in [0, g-1]$ oraz $i \in [s_m + 1, s_{m+1}]$ zachodzi:

$$p'_i = \left\lceil \mathit{piorunochron}(i, a'_{m+1}) \right\rceil. \tag{3}$$

To oznacza, że znając ciągi (a'_i) i (s_i), wartości p'_i możemy obliczyć w czasie $O(n)$.

Poniższy pseudokod stanowi ilustrację tego, jak można wyznaczyć ciągi (a'_i) i (s_i) za jednym przejściem przez ciąg (tablicę) h. Elementy ciągów odkładane są na dwa stosy, A oraz S, oba początkowo puste. Zakładamy, że mamy zaimplementowane wyszukiwanie binarne *obliczQ*, które dla dwóch budynków a, b znajduje $q_{a,b}$.

```
1:  A, S := puste stosy;
2:  for i := n downto 1 do begin
3:    if (h[i] ⩾ h[M] − ⌈√n⌉) and (A.empty() or (h[i] > h[A.top()])) then begin
4:      while (not A.empty()) and (S.top() ⩽ obliczQ(i, A.top())) do begin
5:        A.pop(); S.pop();
6:      end
7:      if not A.empty() then begin
8:        A.push(i); S.push(obliczQ(i, A.top()));
9:      end else begin
10:       A.push(i); S.push(i);
11:     end
12:   end
13: end
```

Widzimy, że wyznaczenie naszych podciągów wymaga $O(\sqrt{n})$ wywołań funkcji *obliczQ*, co sumarycznie daje złożoność $O(\sqrt{n}\log n)$. Mając ciągi (a'_i) oraz (s_i), wszystkie wartości p'_i możemy obliczyć w czasie $O(n)$, korzystając ze wzoru (3). W ten sposób uzyskujemy algorytm działający w czasie $O(n + \sqrt{n}\log n) = O(n)$.

Rozwiązanie to można znaleźć w plikach `pio.cpp` i `pio1.pas`. Na zawodach otrzymywało, rzecz jasna, maksymalną punktację.

Testy

Rozwiązania zawodników były sprawdzane na 11 zestawach danych wejściowych. Zestawy o numerach od 6 do 11 składały się z trzech pojedynczych testów, przy czym test *a* w każdej grupie to duży test losowy (testy *9a* i *10a* wymuszają maksymalne możliwe odpowiedzi dla budynku 1), test *b* to duży test losowy o bardzo małej liczbie różnych największych wysokości, a test *c* to test wymuszający na wolniejszych rozwiązaniach wykonanie $\Omega(n\sqrt{n})$ operacji. Poniżej tabela z zestawieniem wszystkich testów.

Nazwa	n	Opis
pio1.in	100	niewielki test poprawnościowy
pio2.in	200	niewielki test poprawnościowy
pio3.in	1 000	niewielki test poprawnościowy
pio4.in	30 000	większy test poprawnościowo-wydajnościowy
pio5.in	50 000	większy test poprawnościowo-wydajnościowy
pio6[abc].in	64 915	grupa dużych testów
pio7[abc].in	98 117	grupa dużych testów
pio8[abc].in	228 423	grupa dużych testów
pio9[abc].in	351 234	grupa dużych testów
pio10[abc].in	500 000	grupa dużych testów
pio11[abc].in	500 000	grupa dużych testów

Krzysztof Diks, Wojciech Rytter
Treść zadania, Opracowanie

Jacek Migdał
Program

Dostępna pamięć: 64 MB.

OI, Etap I, 18.10–15.11.2010

Przekładanka

Bajtazar kupił synkowi Bajtkowi zestaw klocków ponumerowanych od 1 do n i ustawił je w rzędzie w pewnej kolejności. Zadaniem Bajtka jest ustawienie klocków w kolejności numerów tych klocków, od najmniejszego do największego. Jedyne ruchy, jakie może wykonywać Bajtek, to:

- *przełożenie ostatniego klocka na początek (ruch typu* `a`*), oraz*
- *przełożenie trzeciego klocka na początek (ruch typu* `b`*).*

Pomóż Bajtkowi i napisz program, który sprawdzi, czy dany układ klocków da się w ogóle ustawić w żądanej kolejności, a jeżeli tak, to poda, jak to zrobić.

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita n, $1 \leqslant n \leqslant 2\,000$. W drugim wierszu znajduje się n liczb całkowitych z zakresu od 1 do n pooddzielanych pojedynczymi odstępami. Liczby te nie powtarzają się i reprezentują początkowe ustawienie klocków.

Wyjście

Jeśli nie istnieje sekwencja ruchów prowadząca do ustawienia klocków w porządku rosnącym numerów, Twój program powinien wypisać na standardowe wyjście „`NIE DA SIE`*" (bez cudzysłowów).*

W przeciwnym przypadku, w pierwszym wierszu wyjścia powinna znaleźć się jedna liczba całkowita m ($m \leqslant n^2$), oznaczająca liczbę wykonywanych operacji. Przez **operację** *rozumiemy k-krotne wykonanie jednego z ruchów* `a` *lub* `b`*.*

Jeżeli $m > 0$, to w drugim wierszu powinien znaleźć się ciąg m liczb całkowitych z dodanymi pojedynczymi znakami `a` *lub* `b`*. Zapis postaci k*`a` *(dla $0 < k < n$) oznacza k-krotne wykonanie ruchu typu* `a`*. Zapis postaci k*`b` *(dla $0 < k < n$) oznacza k-krotne wykonanie ruchu typu* `b`*.*

Dodatkowo, znaki stowarzyszone z liczbami znajdującymi się w drugim wierszu muszą być ułożone na przemian.

Jeśli istnieje więcej niż jedno rozwiązanie, Twój program może wypisać dowolne z nich.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`4`	`4`
`1 3 2 4`	`3a 2b 2a 2b`

natomiast dla danych:

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

poprawnym wynikiem jest:

```
NIE DA SIE
```

a dla danych:

```
3
1 2 3
```

poprawnym wynikiem jest:

```
0
```

Rozwiązanie

W tym zadaniu tak naprawdę chodzi o posortowanie permutacji liczb $1, 2, \ldots, n$ poprzez wykonanie ciągu operacji dwóch typów: (a) przestawienie ostatniego elementu ciągu na początek, (b) przestawienie trzeciego elementu ciągu na początek.

Czy zawsze jest to możliwe? Żeby odpowiedzieć na to pytanie, potrzebujemy jednej prostej definicji. Parę (a_i, a_j) elementów ciągu $\langle a_1, a_2, \ldots, a_n\rangle$ nazwiemy *inwersją*, jeśli tylko $i < j$ oraz $a_i > a_j$. Dla przykładu, ciąg $\langle 4,\ 5,\ 1,\ 2,\ 3\rangle$ zawiera 6 inwersji. Jeśli ciąg jest uporządkowany rosnąco, liczba jego inwersji wynosi 0.

Zauważmy, że operacja typu (a) zmienia liczbę inwersji w ciągu o nieparzystą liczbę, gdy n jest parzyste, i o parzystą liczbę, gdy n jest nieparzyste. Operacja typu (b) zawsze zmienia liczbę inwersji o parzystą liczbę. Dowody tych prostych obserwacji pozostawiamy Czytelnikowi.

Wynika stąd, że stosując tylko operacje typu (a) i (b), nie da się posortować ciągu długości nieparzystej mającego nieparzystą liczbę inwersji. Przykładowo, nie da się posortować żadnego z ciągów:

$$\langle 1,\ 3,\ 2\rangle,\quad \langle 2,\ 1,\ 3\rangle,\quad \langle 3,\ 2,\ 1\rangle,\quad \langle 2,\ 4,\ 3,\ 5,\ 1\rangle.$$

Pokażemy teraz, że w każdym innym przypadku sortowanie jest możliwe. Ponieważ dla $n \leqslant 3$ rozwiązanie zadania uzyskujemy „na palcach", więc załóżmy, że $n \geqslant 4$.

Najpierw staramy się umieścić $n-2$ najmniejsze elementy na ich końcowych pozycjach. W tym celu będziemy kolejno przesuwać w lewo jedynkę, potem dwójkę, następnie trójkę, itd.

Przesunięcie elementu x o jedną lub dwie pozycje w lewo za pomocą operacji typu (a) i (b) można wykonać następująco:

(OP1) $\langle P,\ u,\ \mathbf{x},\ v,\ S\rangle$ przekształcamy na $\langle P,\ \mathbf{x},\ v,\ u,\ S\rangle$ za pomocą ciągu operacji „$(|S|+3)$`a 2b` $(|P|)$`a`",

(OP2) $\langle P,\ u,\ v,\ \mathbf{x},\ S\rangle$ przekształcamy na $\langle P,\ \mathbf{x},\ u,\ v,\ S\rangle$ za pomocą ciągu operacji „$(|S|+3)$`a b` $(|P|)$`a`",

przy czym P, S to odpowiednio początkowy i końcowy fragment ciągu (prefiks i sufiks), $|P|$ i $|S|$ — długości tych fragmentów, a u, v, x to pojedyncze elementy ciągu.

Jeśli po wykonaniu powyższego otrzymamy ciąg posortowany, to dobrze. W przeciwnym razie pozostał nam do posortowania ciąg $\langle 1, 2, \ldots, n-2, n, n-1\rangle$. Jeśli n jest nieparzyste, to wiemy już, że tego nie da się zrobić — liczba inwersji w wyjściowym ciągu była nieparzysta. A jeśli n jest parzyste, to sortujemy następująco:

(OP3) „`1a 2b` $((n-2)$`a 2b`$)^{(n-4)/2}$ $(n-4)$`a`".

W powyższym zapisie potęgowanie oznacza powtórzenie napisu-podstawy tyle razy, ile wynosi wykładnik. Warto zauważyć, że dla $n = 4$ ostatnia część ma postać `0a`, a zatem, zgodnie z treścią zadania, nie należało jej wypisywać.

Szczególną rolę w naszym algorytmie odgrywają permutacje

$$l_n = \langle 1,\ 2,\ 3,\ \ldots,\ n-2,\ n,\ n-1 \rangle, \quad id_n = \langle 1,\ 2,\ 3,\ \ldots,\ n \rangle.$$

Na przykład

$$l_4 = \langle 1,\ 2,\ 4,\ 3 \rangle, \quad id_4 = \langle 1,\ 2,\ 3,\ 4 \rangle.$$

Używając tych permutacji do uproszczenia zapisu, otrzymujemy następujący pseudokod algorytmu sortowania:

```
1: Algorytm Przekładanka
2: Wczytaj permutację p;
3: for x := 1 to n − 2 do begin
4:    przesuń x w lewo na x-te miejsce, korzystając z operacji OP1 lub OP2
5:       (preferując OP2);
6:    if p = id_n then return;   { koniec }
7:    else if n parzyste then   { wiemy, że p = l_n }
8:       przekształć p na id_n, korzystając z operacji OP3
9:    else wypisz „NIE DA SIE”;
10: end
```

Na koniec pozostawiamy Czytelnikowi sprawdzenie, że powyższy algorytm sortuje dowolną permutację za pomocą co najwyżej n^2 operacji.

Implementacje rozwiązania wzorcowego można znaleźć w plikach `prz.c`, `prz1.pas`, `prz2.cpp` i `prz3.cpp`.

Testy

Rozwiązania zawodników były sprawdzane na 8 grupach testów, z których każda składała się z co najmniej trzech pojedynczych testów.

Nazwa	n	Opis
prz1a.in	1	warunek brzegowy
prz1b.in	4	warunek brzegowy
prz1c.in	15	warunek brzegowy
prz1d.in	2	klocki posortowane
prz1e.in	2	klocki odwrotnie posortowane
prz1f.in	13	nie da się uporządkować
prz2a.in	5	prosty test poprawnościowy
prz2b.in	8	prosty test poprawnościowy
prz2c.in	7	prosty test poprawnościowy, nie da się uporządkować

Nazwa	n	Opis
prz3a.in	100	losowy średni test
prz3b.in	99	losowy średni test, nie da się uporządkować
prz3c.in	98	losowy średni test
prz4a.in	200	losowy średni test
prz4b.in	199	losowy średni test
prz4c.in	198	losowy średni test
prz5a.in	500	losowy średni test
prz5b.in	499	losowy średni test
prz5c.in	498	losowy średni test
prz6a.in	1 600	losowy duży test
prz6b.in	1 599	losowy duży test
prz6c.in	1 598	losowy duży test
prz7a.in	1 800	losowy duży test
prz7b.in	1 799	losowy duży test
prz7c.in	1 798	losowy duży test
prz8a.in	2 000	losowy duży test
prz8b.in	1 999	losowy duży test
prz8c.in	2 000	losowy duży test
prz8d.in	1 989	losowy duży test
prz8e.in	1 999	losowy duży test, nie da się uporządkować

Jakub Pawlewicz — Treść zadania, Opracowanie
Jakub Wojtaszczyk, Filip Wolski — Program

Dostępna pamięć: 64 MB. *OI, Etap I, 18.10–15.11.2010*

Wykres

Wykresem *nazywamy dowolny ciąg punktów na płaszczyźnie. Dany wykres* $(P_1, \ldots, P_n)$ *zamierzamy zastąpić innym wykresem, który będzie miał co najwyżej* m *punktów* ($m \leqslant n$), *ale w taki sposób, aby „przypominał" jak najbardziej oryginalny wykres.*

Nowy wykres tworzymy w ten sposób, że dzielimy ciąg punktów $P_1, \ldots, P_n$ *na* s ($s \leqslant m$) *spójnych podciągów:*

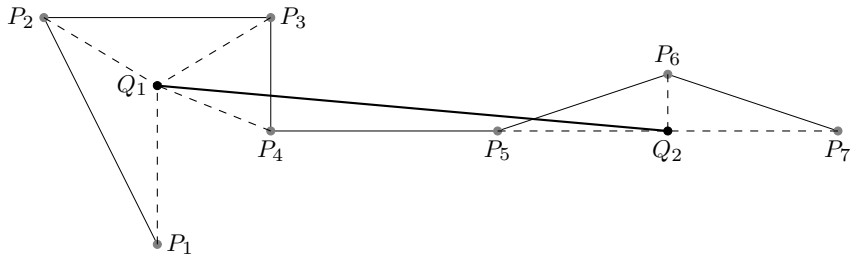
$$(P_{k_0+1}, \ldots, P_{k_1}), \quad (P_{k_1+1}, \ldots, P_{k_2}), \quad \ldots, \quad (P_{k_{s-1}+1}, \ldots, P_{k_s}),$$

przy czym $0 = k_0 < k_1 < k_2 < \ldots < k_s = n$, *a następnie każdy podciąg* $(P_{k_{i-1}+1}, \ldots, P_{k_i})$, *dla* $i = 1, \ldots, s$, *zastępujemy jednym nowym punktem* Q_i. *Mówimy wtedy, że każdy z punktów* $P_{k_{i-1}+1}, \ldots, P_{k_i}$ *został* **ściągnięty** *do punktu* Q_i. *W rezultacie otrzymujemy nowy wykres reprezentowany przez ciąg* $Q_1, \ldots, Q_s$. *Miarą podobieństwa tak utworzonego wykresu do oryginalnego jest maksimum odległości każdego z punktów* $P_1, \ldots, P_n$ *do punktu, do którego został on ściągnięty:*

$$\max_{i=1,\ldots,s} \left(\max_{j=k_{i-1}+1,\ldots,k_i} \left(d\left(P_j, Q_i\right) \right) \right),$$

przy czym $d(P_j, Q_i)$ *jest odległością między* P_j *i* Q_i *i wyraża się standardowym wzorem:*

$$d\left((x_1, y_1), (x_2, y_2)\right) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$



Rys. 1: Przykładowy wykres $(P_1, \ldots, P_7)$ i nowy wykres (Q_1, Q_2), gdzie $(P_1, \ldots, P_4)$ ściągamy do Q_1, natomiast (P_5, P_6, P_7) do Q_2.

Dla danego wykresu składającego się z n *punktów należy znaleźć wykres najbardziej go przypominający, który zawiera co najwyżej* m *punktów, przy czym podział wykresu na spójne podciągi jest dowolny. Ze względu na skończoną precyzję obliczeń, za poprawne będą uznawane wyniki, których podobieństwo do danego wykresu jest co najwyżej o* 0.000001 *większe od optymalnego wyniku.*

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie liczby całkowite n *oraz* m *oddzielone pojedynczym odstępem,* $1 \leqslant m \leqslant n \leqslant 100\,000$. *Każdy z następnych* n *wierszy*

zawiera po dwie liczby całkowite oddzielone pojedynczym odstępem. W $(i+1)$-szym wierszu znajdują się liczby x_i, y_i, $-1\,000\,000 \leqslant x_i, y_i \leqslant 1\,000\,000$, reprezentujące współrzędne (x_i, y_i) punktu P_i.

Wyjście

W pierwszym wierszu standardowego wyjścia należy wypisać jedną liczbę rzeczywistą d, będącą miarą podobieństwa znalezionego wykresu do wykresu oryginalnego. W drugim wierszu należy wypisać jedną liczbę całkowitą s, $1 \leqslant s \leqslant m$. Następnie, w kolejnych s wierszach powinny zostać wypisane współrzędne punktów $Q_1, \ldots, Q_s$, po jednym punkcie w wierszu. W $(i+2)$-gim wierszu powinny znaleźć się liczby rzeczywiste u_i i v_i oddzielone pojedynczym odstępem i określające współrzędne (u_i, v_i) punktu Q_i. Wszystkie liczby rzeczywiste na wyjściu należy wypisać z rozwinięciem do co najwyżej 15 cyfr po kropce dziesiętnej.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`7 2`	`3.00000000`
`2 0`	`2`
`0 4`	`2.00000000 1.76393202`
`4 4`	`11.00000000 1.99998199`
`4 2`	
`8 2`	
`11 3`	
`14 2`	

Rozwiązanie

Treść zadania sugeruje, że należy starać się minimalizować funkcję podobieństwa, mając ograniczoną liczbę punktów przybliżonego wykresu. Znacznie łatwiej natomiast myśleć o tym zadaniu pod kątem minimalizacji liczby punktów przybliżonego wykresu przy określonym maksimum na funkcję podobieństwa. Na tym spostrzeżeniu zasadza się rozwiązanie wzorcowe.

Klasycznym problemem, do którego odwołuje się rozwiązanie tego zadania, jest problem minimalnego koła pokrywającego zadany zbiór punktów na płaszczyźnie (ang. *smallest enclosing disc*). Jak się okazuje, możemy ten problem rozwiązać w *oczekiwanym* czasie liniowym. Taki randomizowany algorytm jest przedstawiony poniżej. Problem ten można nawet rozwiązać w *pesymistycznym* czasie liniowym, jednakże to rozwiązanie jest diametralnie bardziej skomplikowane, jak również ma dużą stałą w złożoności czasowej. Czytelników zainteresowanych takim algorytmem odsyłamy na stronę internetową [37], opis dostępny tylko w języku angielskim.

Rozwiązanie wzorcowe

Rozwiązanie wzorcowe rozwiązuje, w pewnym sensie, problem dualny do tego zadania. Załóżmy, że mamy dane d — maksymalny dystans od punktów wykresu do punktów, do których zostały ściągnięte. Mamy dowolność w wyborze punktów ściągnięcia, a zatem pytanie, czy dany zbiór punktów możemy ściągnąć do jednego punktu, zachowując ograniczenie na d, jest równoważne pytaniu, czy najmniejsze koło pokrywające ten zbiór punktów ma promień co najwyżej d.

Załóżmy dalej, że mamy do dyspozycji funkcję stwierdzającą w czasie $O(k)$, czy dane k punktów leży w pewnym kole o promieniu d. Stosując wyszukiwanie binarne, potrafimy stwierdzić, ile, co najwyżej, może wynosić parametr k_1 (patrz treść zadania). W podobny sposób możemy obliczyć k_2, k_3 itd. — rzecz jasna, każdorazowo opłaca się wybrać największą możliwą wartość parametru. Złożoność takiego rozwiązania niestety nie szacuje się najlepiej. Każde pojedyncze wyszukiwanie binarne miałoby koszt czasowy $O(n \log n)$, co dałoby całkowitą złożoność czasową $O(nm \log n)$.

W zamian możemy zastosować alternatywną metodę wyszukiwania binarnego. Załóżmy, że znamy już maksymalną możliwą wartość k_{i-1} i teraz obliczamy wartość k_i. Niech $t_i = k_i - k_{i-1}$. Najpierw wyznaczamy takie całkowite e, że $2^{e-1} \leqslant t_i < 2^e$. Dopiero potem, już za pomocą „zwykłego" wyszukiwania binarnego, znajdujemy t_i w przedziale $[2^{e-1}, \min(n - k_{i-1}, 2^e - 1)]$. W ten sposób liczba wywołań algorytmu znajdującego minimalne koło pokrywające dany zbiór punktów nie przekroczy $2e$, a największy z tych zbiorów będzie miał nie więcej niż 2^e elementów. Wiemy, że $e = O(\log t_i)$ i $e \cdot 2^e = O(t_i \log t_i)$, a zatem złożoność czasowa całego rozwiązania to

$$\sum_{i=1}^{m} O(t_i \log t_i) \;\leqslant\; \sum_{i=1}^{m} O(t_i \log n) \;=\; O(n \log n).$$

Podsumowując — w złożoności czasowej $O(n \log n)$ potrafimy sprawdzić, ile minimalnie punktów przybliżonego wykresu potrzeba, żeby miara podobieństwa do oryginalnego wykresu wyniosła co najwyżej d. Możemy w takim razie użyć wyszukiwania binarnego, aby znaleźć najmniejsze takie d, dla którego przybliżony wykres ma co najwyżej zadane m punktów. Wymaga to $\lceil \log_2(10^{12}\sqrt{2}) \rceil$ iteracji opisanego algorytmu, jako że początkowy zakres wyszukiwania binarnego rozciąga się od zera do $10^6\sqrt{2}$ (bo koło o promieniu $10^6\sqrt{2}$ i środku w zerze na pewno pokrywa cały wykres), końcowy zakres nie może przekroczyć 10^{-6}, a logarytm dwójkowy ilorazu tych dwóch wartości określa liczbę kroków wyszukiwania binarnego.

Minimalne koło pokrywające zbiór punktów

W tym problemie szukamy najmniejszego koła, które pokrywa zbiór punktów $\{P_1, P_2, \ldots, P_n\}$. Zaprezentujemy klasyczny algorytm, który można znaleźć w książce [23]. Główny pomysł jest taki, aby budować to koło przyrostowo, tzn. kolejno dla $i = 2, \ldots, n$ tworzyć minimalne koło K_i pokrywające pierwsze i punktów $\{P_1, \ldots, P_i\}$.

Spostrzeżenie 1. *Niech $1 < i < n$. Jeśli $P_{i+1} \in K_i$, to $K_{i+1} = K_i$, a w przeciwnym razie P_{i+1} leży na brzegu koła K_{i+1}.*

Spostrzeżenie 1 mówi, że dodanie nowego punktu jest proste, jeżeli leży on wewnątrz dotychczas najmniejszego koła. W przeciwnym razie dostajemy tylko informację, że nowo dodany punkt leży na pewno na brzegu koła, które mamy utworzyć — w tym przypadku będziemy musieli zapewne wykonać więcej pracy. Aby ta sytuacja nie zachodziła zbyt często, wystarczy punkty rozważać w losowej kolejności. Samo spostrzeżenie udowodnimy później, a dotychczasowe rozważania prowadzą do następującego algorytmu. Zakładamy w nim, że punkty $P_1, \ldots, P_n$ są parami różne — jeśli tak nie jest, możemy posortować je po współrzędnych i wyrzucić powtórzenia, wszystko w czasie $O(n \log n)$.

```
1: function MinKoło({P_1, ..., P_n})
2: begin
3:   potasuj zbiór {P_1, ..., P_n}, wybierając losową permutację;
4:   K_2 := najmniejsze koło pokrywające P_1 i P_2;
5:   for i := 2 to n − 1 do
6:     if P_{i+1} ∈ K_i then
7:       K_{i+1} := K_i
8:     else
9:       K_{i+1} := MinKołoZPunktem(P_{i+1}, {P_1, ..., P_i});
10:  return K_n;
11: end
```

Wprowadziliśmy nową funkcję MinKołoZPunktem, którą trzeba zaimplementować. Jak skonstruować najmniejsze koło pokrywające zbiór punktów $\{P_1, \ldots, P_i\}$, wiedząc, że to koło musi mieć na brzegu punkt $Q = P_{i+1}$? Możemy zastosować analogiczne podejście jak poprzednio, a mianowicie, budować to koło przyrostowo. Tym razem kolejno dla $j = 1, \ldots, i$ tworzymy koło K'_j pokrywające pierwsze j punktów $\{P_1, \ldots, P_j\}$ oraz punkt Q, z dodatkowym w warunkiem, że Q musi leżeć na brzegu K'_j. Teraz dodanie kolejnego punktu P_{j+1} polega na sprawdzeniu, czy należy on do bieżącego najmniejszego utworzonego koła — jeśli tak, to koło to nie zmienia się, a w przeciwnym razie musi ono mieć na swym brzegu punkt P_{j+1}. To ostatnie oznacza, że szukane koło ma na brzegu punkty Q i P_{j+1}. Daje nam to taki oto pseudokod funkcji MinKołoZPunktem.

```
1: function MinKołoZPunktem(Q, {P_1, ..., P_i})
2: begin
3:   K'_1 := najmniejsze koło pokrywające P_1 i Q;
4:   for j := 1 to i − 1 do
5:     if P_{j+1} ∈ K'_j then
6:       K'_{j+1} := K'_j
7:     else
8:       K'_{j+1} := MinKołoZ2Punktami(P_{j+1}, Q, {P_1, ..., P_j});
9:   return K'_i;
10: end
```

Teraz z kolei stajemy przed problemem znalezienia najmniejszego koła pokrywającego punkty $\{P_1, \ldots, P_j\}$, które ma na brzegu zadane dwa punkty $Q_1 = P_{j+1}$ i $Q_2 = Q$. Ponownie stosujemy podejście przyrostowe. Jeżeli kolejny punkt P_{k+1} należy do bieżącego koła, to nic nie musimy robić. Jeżeli jest na zewnątrz, to nowo tworzone koło

musi mieć na brzegu P_{k+1}. Musi mieć ono zatem na brzegu punkty: Q_1, Q_2 i P_{k+1}. Koło, dla którego znamy trzy różne punkty z jego brzegu, jest wyznaczone jednoznacznie (jeśli tylko te trzy punkty nie są współliniowe — na szczęście, dzięki konstrukcji całego algorytmu, taka sytuacja nie wystąpi w naszym przypadku). Daje to poniższy pseudokod, który kończy rozwiązanie problemu.

```
1:  function MinKołoZ2Punktami(Q_1, Q_2, {P_1, ..., P_j})
2:  begin
3:    K''_0 := najmniejsze koło pokrywające Q_1 i Q_2;
4:    for k := 0 to j − 1 do
5:      if P_{k+1} ∈ K''_k then
6:        K''_{k+1} := K''_k
7:      else
8:        K''_{k+1} := koło mające na brzegu punkty Q_1, Q_2 i P_{k+1};
9:    return K''_j;
10: end
```

Zanim przejdziemy do analizy czasowej tego algorytmu, wypada udowodnić jego poprawność. Otóż w całym rozumowaniu notorycznie korzystaliśmy z faktu, że jeśli jakiś punkt jest poza najmniejszym kołem, to nowo tworzone koło musi mieć ten punkt na swoim brzegu. Dodatkowo, czasem znaliśmy jakieś punkty, które muszą być na brzegu szukanych kół. Poniższy lemat obejmuje wszystkie tego typu przypadki.

Lemat 2. *Rozważmy sytuację, w której mamy pewien zbiór punktów, nazwijmy go $\mathcal{P}$. Szukamy najmniejszego koła pokrywającego $\mathcal{P}$ z dodatkową informacją o zbiorze punktów, nazwijmy go $\mathcal{Q}$, które muszą należeć do brzegu tego koła (zbiór $\mathcal{Q}$ może być też pusty). Wtedy (i) takie koło jest wyznaczone jednoznacznie. Oznaczmy takie najmniejsze koło przez K.*

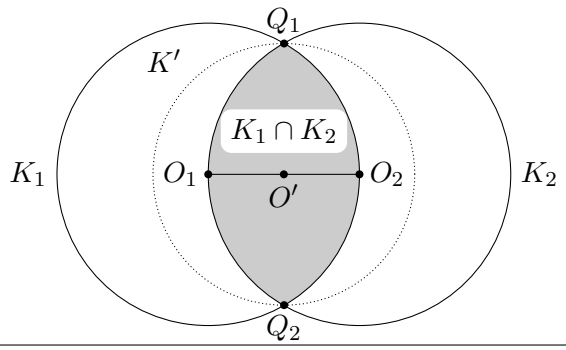
Teraz bierzemy kolejny punkt P. Szukamy najmniejszego koła K' pokrywającego zbiór $\mathcal{P} \cup \{P\}$, takiego że na brzegu tego koła na pewno są punkty ze zbioru $\mathcal{Q}$. Wtedy (ii) jeśli $P \in K$, to $K' = K$, a (iii) jeśli, przeciwnie, $P \notin K$, to K' musi być najmniejszym kołem pokrywającym zbiór $\mathcal{P}$, na którego brzegu muszą znaleźć się punkty ze zbioru $\mathcal{Q} \cup \{P\}$.

Dowód:

(i) Załóżmy przeciwnie, że istnieją dwa koła pokrywające $\mathcal{P}$ mające na brzegu wszystkie punkty z $\mathcal{Q}$. Ponieważ oba są najmniejsze, więc oba mają ten sam promień, tylko różne środki. Niech środki tych kół to O_1 i O_2. Oznaczmy te koła odpowiednio przez K_1 i K_2 (rysunek 1).

Ponieważ zbiór punktów $\mathcal{P}$ zawiera się zarówno w kole K_1, jak i w kole K_2, więc musi zawierać się w ich przecięciu: $\mathcal{P} \subseteq K_1 \cap K_2$. Podobnie jest z brzegiem — zbiór punktów $\mathcal{Q}$ musi należeć do przecięcia brzegów K_1 i K_2. Brzegi kół to okręgi. Okręgi te przecinają się w dwóch punktach, niech będą to Q_1 i Q_2. Stąd, w zbiorze $\mathcal{Q}$ mogą występować co najwyżej dwa punkty: $\mathcal{Q} \subseteq \{Q_1, Q_2\}$.

W związku z powyższym możemy stworzyć inne, mniejsze koło K', które będzie pokrywać $K_1 \cap K_2$ (a tym samym zbiór $\mathcal{P}$) oraz którego brzeg będzie przechodził przez punkty Q_1 i Q_2, co przeczy minimalności kół K_1 i K_2. Wystarczy obrać środek koła O' jako środek odcinka $\overline{O_1O_2}$, tak jak na rysunku 1.

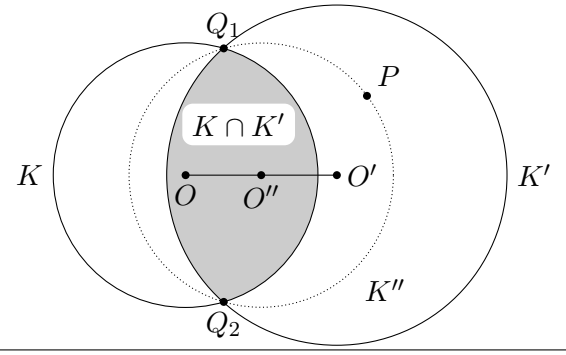


Rys. 1: Założenie o istnieniu dwóch minimalnych kół pokrywających $\mathcal{P}$ i przechodzących przez $\mathcal{Q}$ prowadzi do sprzeczności.

(ii) Szukane koło K' pokrywa również mniejszy zbiór $\mathcal{P}$, więc K' nie może być mniejsze niż K, gdyż K jest najmniejszym możliwym kołem pokrywającym $\mathcal{P}$. Ponieważ $P \in K$, więc K pokrywa zbiór $\mathcal{P} \cup \{P\}$. Z (i) wynika, że nie może istnieć drugie koło o tym samym promieniu, które również pokrywa $\mathcal{P} \cup \{P\}$, zatem $K' = K$.

(iii) Rozumowanie jest podobne jak w punkcie (i). Załóżmy, że szukane koło K' nie ma na brzegu punktu P. Ponieważ oba koła K i K' pokrywają zbiór $\mathcal{P}$, więc $\mathcal{P}$ zawiera się w ich przecięciu: $\mathcal{P} \subseteq K \cap K'$. Ponadto wiemy, że oba koła K i K' mają na brzegu zbiór punktów $\mathcal{Q}$. Oznacza to, że przecięcie brzegów kół K i K' zawiera $\mathcal{Q}$. Brzegi kół to okręgi, a przecięcie dwóch okręgów musi się w naszym przypadku składać z dwóch punktów Q_1 i Q_2. Zatem w skład zbioru $\mathcal{Q}$ mogą wchodzić jedynie te punkty: $\mathcal{Q} \subseteq \{Q_1, Q_2\}$.

Teraz możemy stworzyć inne koło K'', które będzie pokrywać zbiór $K \cap K'$ (a tym samym zbiór $\mathcal{P}$) oraz którego brzeg będzie przechodził przez punkty Q_1, Q_2 i dodatkowo przez punkt P, tak jak na rysunku 2.



Rys. 2: Założenie, że minimalne koło ma punkt P w swoim wnętrzu, prowadzi do utworzenia innego, mniejszego koła z P na brzegu.

Środek O'' takiego koła będzie leżał na odcinku łączącym środki kół K i K', oznaczone odpowiednio przez O i O'. Koło K'' ma promień mniejszy niż koło K', co przeczy minimalności K'.

■

Skoro jesteśmy pewni poprawności naszej metody, możemy przejść do szacowania złożoności czasowej. Zacznijmy od następującego spostrzeżenia, które wynika ze struktury całego algorytmu.

Spostrzeżenie 3. *Najmniejsze koło pokrywające zbiór punktów wyznaczone jest przez co najwyżej trzy punkty z tego zbioru.*

Dowód: Przyjrzyjmy się temu, w których miejscach pseudokodów naszych funkcji tworzymy konkretne koła. Są to: wyznaczanie najmniejszego koła pokrywającego dwa punkty (wiersz 4 funkcji MinKoło, wiersz 3 funkcji MinKołoZPunktem oraz wiersz 3 funkcji MinKołoZ2Punktami) oraz wyznaczanie koła przechodzącego przez trzy punkty (wiersz 8 funkcji MinKołoZ2Punktami). ■

Aby pokryć kołem zbiór punktów $\{P_1, \ldots, P_n\}$, wystarczy zatem znaleźć pewne trzy punkty P_i, P_j i P_k, dla których najmniejsze koło pokrywające jest również najmniejszym kołem pokrywającym cały zbiór. Zauważmy teraz, że sytuacja, w której nowo rozważany punkt P_n jest poza minimalnym kołem pokrywającym zbiór $\{P_1, \ldots, P_{n-1}\}$, może zajść tylko wtedy, gdy jest on jednym z punktów P_i, P_j, P_k.

Teraz możemy przejść do analizy czasowej. W funkcji MinKołoZ2Punktami mamy pętlę wykonującą j iteracji, z których każda zajmuje czas stały. Zatem złożoność czasowa tej funkcji to $O(j)$.

Funkcja MinKołoZPunktem zawiera pętlę wykonującą $i-1$ iteracji po $j = 1, \ldots, i-1$, przy czym każda iteracja wykonuje się w czasie stałym, chyba że punkt P_{j+1} nie należy do najmniejszego koła pokrywającego zbiór $\{Q, P_1, \ldots, P_j\}$. Ta sytuacja ma miejsce tylko wtedy, gdy punkt P_{j+1} jest jednym z trzech wyznaczających najmniejsze koło pokrywające zbiór $\{Q, P_1, \ldots, P_j, P_{j+1}\}$. Ponieważ Q jest jednym z tych trzech punktów, więc P_{j+1} musi być jednym z dwóch pozostałych punktów ze zbioru $\{Q, P_1, \ldots, P_{j+1}\}$, które leżą na obwodzie minimalnego koła pokrywającego ten zbiór punktów. Skoro punkty rozważamy w losowej kolejności, to prawdopodobieństwo ostatniego zdarzenia wynosi $2/(j+1)$ i w tym przypadku wołamy MinKołoZ2Punktami dla j punktów. Podsumowując, oczekiwany czas wykonania funkcji MinKołoZPunktem wynosi

$$\sum_{j=1}^{i-1}\Big(O(1) + \frac{2}{j+1}O(j)\Big) = \sum_{j=1}^{i-1} O(1) = O(i).$$

Podobnie analizujemy naszą główną funkcję MinKoło. Zawiera ona pętlę po $i = 2, \ldots, n-1$, w której każda iteracja wykonuje się w czasie stałym, chyba że punkt P_{i+1} nie należy do najmniejszego koła pokrywającego zbiór $\{P_1, \ldots, P_i\}$, a to jest możliwe tylko wtedy, gdy P_{i+1} jest jednym z trzech punktów wyznaczających najmniejsze koło pokrywające zbiór $\{P_1, \ldots, P_i, P_{i+1}\}$. Prawdopodobieństwo tego ostatniego zdarzenia wynosi $3/(i+1)$ i wtedy trzeba wywołać MinKołoZPunktem dla

i punktów, co zajmuje czas $O(i)$. W rezultacie otrzymujemy oczekiwaną złożoność czasową:

$$\sum_{i=2}^{n-1}\Big(O(1)+\frac{3}{i+1}O(i)\Big)=\sum_{i=1}^{n-1}O(1)=O(n).$$

Udowodniliśmy następujące twierdzenie.

Twierdzenie 4. *Oczekiwany czas wykonania funkcji* MinKoło *wynosi* $O(n)$.

Implementację rozwiązania wzorcowego można znaleźć w pliku `wyk.cpp`.

Rozwiązania wolniejsze

Rozwiązanie tego zadania składa się, *de facto*, z dwóch części — mamy mianowicie algorytm wyszukujący kolejne wartości k_i, wykorzystujący niejako osobną metodę znajdowania najmniejszego koła pokrywającego dany zbiór punktów. Poniżej opisujemy, w jaki sposób można każdą z tych części wykonać nieoptymalnie, a także podajemy, w których spośród rozwiązań nieoptymalnych `wyks[1-8].cpp` ta metoda została wykorzystana.

Kolejne wartości k_i

Łatwo spowolnić rozwiązanie zadania *Wykres*, jeżeli nie użyje się „sprytnej" metody wyszukiwania binarnego kolejnych wartości k_i, lecz zastosuje się zwykłe wyszukiwanie binarne (o takim rozwiązaniu wspominaliśmy już na początku opisu). Tę technikę wykorzystują rozwiązania powolne o numerach 1, 2, 3, 5, 7.

Jeszcze wolniej można szukać kolejnych k_i zwyczajnie przyrostowo, tj. zwiększając k_i o jeden, aż nie będzie możliwe pokrycie kolejnego zbioru punktów kołem o danym, ograniczonym promieniu. Implementacje wykorzystujące tę metodę znajdują się w rozwiązaniach powolnych numer 4, 6, 8.

Koło pokrywające zbiór punktów

Zauważmy, że w rozwiązaniu nie musimy tak naprawdę wyznaczać promieni minimalnych kół — wystarczy nam algorytm sprawdzania, czy istnieje koło o *zadanym* promieniu d, pokrywające dany zbiór punktów.

Ten problem można rozwiązać, na przykład, w czasie kwadratowym — na wszystkie sposoby wybieramy jeden punkt, a następnie, w czasie liniowym, poszukujemy koła o promieniu d z tym właśnie punktem na obwodzie, które pomieści wszystkie pozostałe punkty. Implementacje tej metody można znaleźć w rozwiązaniach powolnych numer 3 i 4.

Dalej, możemy to zaimplementować sześciennie — wybieramy dwa punkty, które jednoznacznie definiują koło o promieniu d (zawierające te punkty na obwodzie), i przeglądamy wszystkie pozostałe punkty, sprawdzając, czy znajdują się wewnątrz tego koła. Implementacje: rozwiązania powolne numer 5 i 6.

W końcu, najwolniejsza metoda działa w złożoności czasowej $O(n^4)$. Tutaj nie potrzebujemy już znać wartości parametru d: wybieramy trzy punkty, które definiują

koło, a następnie sprawdzamy, czy wszystkie pozostałe punkty znajdują się w tym kole. Spośród znalezionych w ten sposób kół wybieramy najmniejsze. Implementacje w rozwiązaniach powolnych numer 7 i 8.

Testy

Zadanie było sprawdzane na 13 zestawach danych testowych, różniących się przede wszystkim sposobem rozmieszczenia punktów na płaszczyźnie: losowe rozmieszczenie (typ 1), kwadratowa spirala (typ 2), zwykła spirala (typ 3), błądzenie losowe (typ 4), punkty gęsto zgrupowane w kilku miejscach płaszczyzny (typ 5), punkty w rogach (typ 6), wszystkie punkty w tym samym miejscu (typ 7). Poniżej tabelka opisująca podstawowe parametry testów oraz ich typy.

Nazwa	n	m	Opis
wyk1a.in	4	4	typ 1
wyk1b.in	4	1	typ 1
wyk2a.in	8	2	typ 1
wyk2b.in	20	10	typ 1
wyk3a.in	10	1	typ 3
wyk3b.in	20	5	typ 2
wyk4a.in	100	5	typ 4
wyk4b.in	100	99	typ 4
wyk5a.in	1 000	500	typ 6
wyk5b.in	1 000	50	typ 7
wyk6a.in	1 000	50	typ 1
wyk6b.in	1 000	500	typ 1
wyk7.in	1 000	20	typ 5
wyk8a.in	1 000	10	typ 3
wyk8b.in	1 000	100	typ 3

Nazwa	n	m	Opis
wyk9a.in	5 000	20	typ 4
wyk9b.in	5 000	100	typ 4
wyk9c.in	5 000	1 000	typ 4
wyk10a.in	50 000	4	typ 5
wyk10b.in	50 000	20	typ 1
wyk11a.in	100 000	100	typ 1
wyk11b.in	100 000	10 000	typ 3
wyk11c.in	100 000	100 000	typ 6
wyk12a.in	100 000	100	typ 3
wyk12b.in	100 000	20	typ 3
wyk12c.in	100 000	200	typ 5
wyk12d.in	100 000	1	typ 5
wyk13a.in	100 000	200	typ 7
wyk13b.in	100 000	80 000	typ 6
wyk13c.in	100 000	1	typ 4

Zawody II stopnia

opracowania zadań

Marian M. Kędzierski
Treść zadania, Opracowanie

Tomasz Kociumaka, Paweł Parys
Program

Dostępna pamięć: 32 MB.

OI, Etap II, dzień próbny, 08.02.2011

Sejf

Bajtazar jest słynnym kasiarzem, który porzucił przestępczy żywot i zajął się testowaniem zabezpieczeń antywłamaniowych. Właśnie dostał do sprawdzenia nowy rodzaj sejfu: kombinatoryczny. Sejf jest otwierany pokrętłem, które kręci się w kółko. Można je ustawić w n różnych pozycjach ponumerowanych od 0 do $n-1$. Ustawienie pokrętła w niektórych pozycjach powoduje otwarcie sejfu, a w innych nie. Przy tym, pozycje otwierające sejf mają taką własność, że jeżeli x i y są takimi pozycjami, to $(x+y) \bmod n$ też powoduje otwarcie sejfu (dotyczy to także przypadku, gdy $x = y$).

Bajtazar sprawdził k różnych pozycji pokrętła: $m_1, m_2, \ldots, m_k$. Pozycje $m_1, m_2, \ldots, m_{k-1}$ nie powodują otwarcia sejfu, dopiero ustawienie pokrętła w pozycji m_k spowodowało jego otwarcie. Bajtazarowi nie chce się sprawdzać wszystkich pozycji pokrętła. Chciałby wiedzieć, na podstawie do tej pory sprawdzonych pozycji, jaka jest maksymalna możliwa liczba pozycji, w których pokrętło otwiera sejf. Pomóż mu i napisz odpowiedni program.

Wejście

Pierwszy wiersz standardowego wejścia zawiera dwie liczby całkowite n oraz k oddzielone pojedynczym odstępem, $1 \leqslant k \leqslant 250\,000$, $k \leqslant n \leqslant 10^{14}$. W drugim wierszu znajduje się k różnych liczb całkowitych, pooddzielanych pojedynczymi odstępami, $m_1, m_2, \ldots, m_k$, $0 \leqslant m_i < n$. Możesz założyć, że dane wejściowe odpowiadają pewnemu sejfowi spełniającemu warunki zadania.

W testach wartych ok. 70% punktów zachodzi $k \leqslant 1000$. W części tych testów, wartych ok. 20% punktów, zachodzą dodatkowo warunki $n \leqslant 10^8$ oraz $k \leqslant 100$.

Wyjście

Twój program powinien wypisać w pierwszym (i jedynym) wierszu standardowego wyjścia jedną liczbę całkowitą: maksymalną liczbę pozycji pokrętła, które mogą powodować otwarcie sejfu.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`42 5` `28 31 10 38 24`	`14`

Rozwiązanie

Analiza problemu

Na początku zastanówmy się, jaka może być postać zbioru pozycji otwierających sejf. Z własności sejfu wynika, że dla dowolnej pozycji otwierającej a, pozycje $2a \bmod n$,

$3a \bmod n, \ldots, na \bmod n = 0$, czyli jej wielokrotności modulo n, również wszystkie są otwierające. Przez x oznaczmy pozycję otwierającą o *najmniejszym dodatnim* numerze. Wypiszmy kolejno jej wielokrotności, tzn. $x, 2x, \ldots, nx$. Wiemy, że ich reszty z dzielenia przez n są pozycjami otwierającymi. Zwróćmy szczególną uwagą na tę wielokrotność $w = i \cdot x$, która jako pierwsza osiąga co najmniej n, tzn. $(i-1) \cdot x < n \leqslant i \cdot x$. Wówczas $w \bmod n = w - n$ jest otwierająca, a przy tym $w \in [n, n + x)$, skąd $w \bmod n \in [0, x)$. Jednakże zdefiniowaliśmy x jako *najmniejszą* dodatnią pozycję otwierającą, skąd wnioskujemy, że $w \bmod n = 0$, a więc $n = i \cdot x$. Oznacza to, że liczba x jest *dzielnikiem* n. Dla podkreślenia tego faktu będziemy ją odtąd oznaczać przez d.

Pamiętamy, że wszystkie wielokrotności tej liczby z przedziału $[0, n)$ (jest ich $\frac{n}{d}$) to pozycje otwierające. Okazuje się, że są to jedyne takie pozycje. Weźmy bowiem dowolną pozycję otwierającą a i podzielmy ją z resztą przez d. Mamy $a = q \cdot d + r$ dla pewnych liczb całkowitych q, r, przy czym $0 \leqslant r < d$ oraz $0 \leqslant q < \frac{n}{d}$. Pozycja $(\frac{n}{d} - q) \cdot d$ jest, jak wiemy, także otwierająca, a więc suma obydwu wspomnianych pozycji modulo n, tzn. $((\frac{n}{d} - q) \cdot d + q \cdot d + r) \bmod n = r$, także. Ponieważ $0 \leqslant r < d$, a d była najmniejszą dodatnią pozycją otwierającą, to wnioskujemy, że $r = 0$, a więc a jest wielokrotnością d.

Tym samym zakończyliśmy dowód kluczowej dla zadania obserwacji. Wykazaliśmy, że każdy możliwy zbiór pozycji otwierających to zbiór wielokrotności pewnego dzielnika liczby n. Łatwo sprawdzić, iż każdy zbiór tej postaci spełnia żądaną kombinatoryczną własność sejfu. Aby jednak taki zbiór pozycji otwierających był zgodny z obserwacjami Bajtazara, muszą być spełnione dodatkowo następujące warunki:

- $m_1, m_2, \ldots, m_{k-1}$ nie należą do zbioru pozycji otwierających, a więc nie są wielokrotnościami powyższej liczby d,
- m_k jest pozycją otwierającą, czyli wielokrotnością d.

Szukamy największego zbioru spełniającego wszystkie powyższe warunki, a dla zadanego d taki zbiór liczy $\frac{n}{d}$ elementów. Innymi słowy, chcemy więc znaleźć najmniejszy dzielnik d spełniający podane wyżej warunki.

Na potrzeby dalszych rozważań oznaczmy liczbę wszystkich dzielników liczby n przez $D(n)$.

Rozwiązanie siłowe

Znaleźliśmy proste sformułowanie naszego problemu w języku teorii liczb: należy znaleźć najmniejszy dzielnik d liczby n, który dzieli również m_k, ale nie dzieli żadnej spośród liczb $m_1, m_2, \ldots, m_{k-1}$.

Możemy więc już podać pierwsze rozwiązanie zadania — sprawdzać wszystkie liczby od 1 do n po kolei i dla każdej z nich weryfikować, czy warunki $d \mid n$, $d \mid m_k$, $d \nmid m_1, \ldots, m_{k-1}$ są spełnione. To daje złożoność $O(n \cdot k)$ lub, jeśli dla $d \nmid n$ nie będziemy sprawdzać kolejnych warunków, $O(n + D(n) \cdot k)$. Takie bardzo proste programy otrzymywały 21 punktów, przykładowe implementacje znajdują się w plikach `sejs3.cpp` i `sejs11.pas`.

Pierwsza optymalizacja

Zastanówmy się, gdzie jest pole do przyspieszenia naszego pierwszego algorytmu. Tracimy dużo czasu, niepotrzebnie przeglądając wszystkie liczby od 1 do n — szukamy przecież dzielnika liczby n. Chcielibyśmy zatem przeglądać tylko dzielniki liczby n.

Generowanie dzielników

Wygenerowanie listy wszystkich dzielników danej liczby jest bardzo proste, jeśli satysfakcjonuje nas złożoność $O(\sqrt{n})$. Wystarczy bowiem przeglądać wszystkie liczby od 1 aż do $\lfloor\sqrt{n}\rfloor$ i, jeśli natrafimy na dzielnik d, wstawić na listę również liczbę $\frac{n}{d}$. Jak widać, taki algorytm w każdym przypadku działa w czasie $\Theta(\sqrt{n})$.

Przedstawimy jeszcze inną metodę, także o złożoności pierwiastkowej, która jednak w większości przypadków jest znacznie szybsza, a przy tym pozwoli lepiej zrozumieć strukturę zbioru dzielników. Pierwszą jej fazą jest znalezienie rozkładu na czynniki pierwsze liczby n. Wykorzystamy algorytm działający w czasie $O(\sqrt{n})$, przedstawiony poniżej.

```
1:  ℓ := 0;
2:  i := 2;
3:  while i · i ⩽ n do begin
4:    if n mod i = 0 then begin
5:      { liczba i jest dzielnikiem pierwszym liczby n }
6:      ℓ := ℓ + 1;
7:      p[ℓ] := i;
8:      α[ℓ] := 0;
9:      while n mod i = 0 do begin
10:       α[ℓ] := α[ℓ] + 1;
11:       n := n/i;
12:     end
13:   end
14:   { n jest liczbą pierwszą lub jedynką }
15:   if n > 1 then begin
16:     p[ℓ] := n;
17:     α[ℓ] := 1;
18:     ℓ := ℓ + 1;
19:   end
20: end
```

Pozostawiamy jako proste ćwiczenie obserwację, że $n = p_1^{\alpha_1} \dots p_\ell^{\alpha_\ell}$, gdzie liczby $p_1 < \dots < p_\ell$ są pierwsze, zaś $\alpha_1, \dots, \alpha_n$ dodatnie. Czas działania algorytmu szacuje się przez $O(\log n + \max(p_{\ell-1}, \sqrt{p_\ell}))$. Owa złożoność nigdy nie przekracza $O(\sqrt{n})$, a zazwyczaj, zwłaszcza gdy liczba n ma dużo dzielników (a dla takich działa wolno reszta rozwiązania zadania), jest znacznie mniejsza.

Mając już rozkład na czynniki pierwsze $n = p_1^{\alpha_1} \dots p_\ell^{\alpha_\ell}$, wystarczy zauważyć, że dzielniki liczby n to dokładnie liczby postaci $p_1^{\beta_1} \dots p_\ell^{\beta_\ell}$, gdzie $\beta_i \in \{0, 1, \dots, \alpha_i\}$. Niech kolejnym ćwiczeniem dla Czytelnika będzie skonstruowanie na tej podstawie

algorytmu działającego w czasie $O(D(n))$. Być może prościej na początek pozwolić na czas rzędu $D(n) \cdot \left(\sum_{i=1}^{\ell} \alpha_i\right)$, czyli pesymistycznie $O(D(n) \cdot \log n)$. Warto przy okazji spostrzec, że $D(n) = (\alpha_1 + 1) \ldots (\alpha_\ell + 1)$, co wkrótce będzie przydatne do szacowania $D(n)$.

Otrzymaliśmy algorytm wyznaczania dzielników, który po sfaktoryzowaniu liczby n działa już w optymalnym czasie $O(D(n))$. Łącznie jest to w naszym przypadku wciąż $O(\sqrt{n})$. Poprawienie tej złożoności jest możliwe, ale leży poza wymaganiami stawianymi uczestnikom Olimpiady[1].

Szacowanie liczby dzielników

Dzięki szybszemu generowaniu zbioru dzielników otrzymaliśmy algorytm działający w czasie $O(\sqrt{n} + k \cdot D(n))$. Aby ocenić, czy takie rozwiązanie wystarczy choćby dla $k \leqslant 1000$, przydałoby się nam jakieś oszacowanie na $D(n)$. Oczywiście $D(n) = O(\sqrt{n})$, jednak przy n rzędu 10^{14} takie ograniczenie nas nie satysfakcjonuje. Intuicyjnie wydaje się jasne, że dzielników powinno być znacznie mniej, ale chcielibyśmy wesprzeć swoje podejrzenia konkretami. Zadanie nie pochodzi z I etapu, więc nie chcemy korzystać przy tym z Internetu czy książek. Możemy jednak napisać program, który da nam jakieś konkretne liczbowe ograniczenie[2].

Sposób I

Nie jesteśmy w stanie przejrzeć wszystkich liczb z zakresu $[1, 10^{14}]$, jednak możemy przejrzeć wszystkie liczby z zakresu $[1, 10^7]$. Oznaczmy przez $D(1,\, s)$ maksymalną liczbę dzielników liczby z przedziału $[1, s]$. Wykorzystamy następujący fakt:

Twierdzenie 1. *Dla każdego $s > 1$ zachodzi $D(1,\, s^2) \leqslant 2 \cdot D(1,\, s)^2$.*

Dowód: Rozważmy dowolną liczbę $t \in [1,\, s^2]$ i jej rozkład na czynniki pierwsze

$$t = p_1^{\alpha_1} p_2^{\alpha_2} \ldots p_\ell^{\alpha_\ell}.$$

Wiemy, że $D(t) = (\alpha_1+1)\cdot(\alpha_2+1)\cdot\ldots\cdot(\alpha_\ell+1)$. Niech $i_1, i_2, \ldots, i_q$ będą pozycjami, dla których $\alpha_i = 1$, ustawionymi w kolejności rosnącej. Przypiszmy $\beta_{i_{2j-1}} = 1$, $\beta_{i_{2j}} = 0$ dla $j = 1, 2, \ldots$ Dla pozostałych i niech $\beta_i = \lfloor \frac{\alpha_i}{2} \rfloor$, gdzie $\lfloor x \rfloor$ oznacza część całkowitą liczby x. Wówczas liczba

$$u = p_1^{\beta_1} p_2^{\beta_2} \ldots p_\ell^{\beta_\ell}$$

należy do przedziału $[1, s]$. Ponadto $D(t) \leqslant 2 \cdot D(u)^2$, ponieważ

$$(\alpha_{i_{2j-1}} + 1) \cdot (\alpha_{i_{2j}} + 1) = 4 = (\beta_{i_{2j-1}} + 1)^2 \cdot (\beta_{i_{2j}} + 1)^2,$$

$$(\alpha_{i_q} + 1) \leqslant 2 \cdot (\beta_{i_q} + 1)^2,$$

[1] Chodzi tu oczywiście o szybsze metody faktoryzacji, jak choćby opisany w książce [22] niedeterministyczny algorytm „rho" Pollarda. Druga faza wymaga tylko lepszego niż $O(\sqrt{n})$ oszacowania na $O(D(n))$.

[2] Teoretyczne oszacowania asymptotyczne są w tym przypadku i tak mało przydatne. Okazuje się bowiem, że $D(n) = O(n^\varepsilon)$ dla dowolnie małego $\varepsilon > 0$, czyli np. $D(n) = O(n^{0{,}0001})$. Jednak, jak się za chwilę przekonamy, jeszcze dla 14-cyfrowych wartości n zdarza się, że $D(n)$ przekracza $n^{0{,}3}$.

natomiast dla pozostałych i

$$(\alpha_i + 1) \leqslant (\beta_i + 1)^2.$$

Stąd już łatwo wynika teza. ■

Jak można obliczyć $D(1,\, 10^7)$? Możemy, posługując się sitem Eratostenesa, wyznaczyć dla każdej liczby jej *najmniejszy dzielnik pierwszy*, co pozwoli szybko generować rozkłady na czynniki pierwsze. Następnie dla każdej liczby z rozważanego przedziału możemy zbudować jej rozkład, obliczając zarazem liczbę jej dzielników. Taki prosty eksperyment daje nam $D(1,\, 10^7) = 448$. Tak więc $D(1,\, 10^{14}) \leqslant 2 \cdot 448^2 = 401\,408$.

Sposób II

Oszacowanie przez 401 408 nadal jest bardzo zgrubne. Okazuje się jednak, że możemy wyznaczyć $D(1,\, 10^{14})$ dokładnie.

Potrzebne nam tutaj będzie pojęcie *liczby antypierwszej*, które może być Czytelnikowi znane np. z jednego z zadań VIII Olimpiady Informatycznej [8]. Liczba antypierwsza to taka, która ma więcej dzielników niż wszystkie liczby od niej mniejsze. Oczywiście, do obliczenia $D(1,\, s)$ wystarcza znalezienie największej liczby antypierwszej nie większej niż s.

Szukanie liczb antypierwszych opiera się na spostrzeżeniach, że ciąg wykładników $\alpha_1, \alpha_2, \ldots, \alpha_\ell$ w jej rozkładzie na czynniki pierwsze musi być ciągiem nierosnącym (przypominamy, że ciąg p_i był rosnący) oraz że w tym rozkładzie muszą występować kolejne najmniejsze liczby pierwsze, tzn. w przedziale (p_i, p_{i+1}) nie może być innej liczby pierwszej. Te uwagi wystarczają już, żeby znaleźć największą liczbę antypierwszą w przedziale $[1, s]$ przez iterację po wszystkich niemalejących ciągach wykładników odpowiadających liczbom mniejszym niż s. Ponieważ takich ciągów jest stosunkowo niewiele, możemy w taki sposób w krótkim czasie przesiać przedział $[1, 10^{14}]$. Wynik jest następujący: $D(1,\, 10^{14}) = 17\,280$; jest on osiągany przez liczbę antypierwszą 97 821 761 637 600.

Podsumowanie

Wiedząc już, ile w najgorszym przypadku może wynosić $D(n)$, możemy stwierdzić, że programy implementujące powyższe algorytmy o złożoności $O(\sqrt{n} + k \cdot D(n))$ powinny dostawać (i dostawały) przynajmniej 70 punktów. Przykładowe implementacje można znaleźć w plikach `sejs1.cpp`, `sejs10.pas` oraz (z drobnymi optymalizacjami) `sejs2.cpp`, `sejs3.cpp` i `sejs5.cpp`.

Rozwiązanie wzorcowe

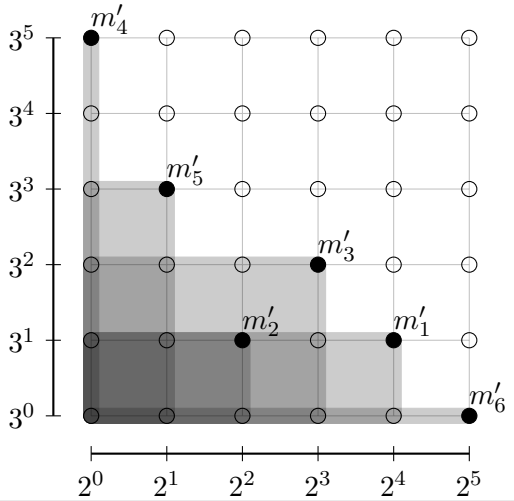
Zanim przejdziemy do kolejnych optymalizacji, dokonamy pewnego uproszczenia, które będzie nas kosztować $O(k \log n)$ (a więc niewiele) czasu, lecz pozwoli rozwiązywać nieco łatwiejszy problem. Otóż, skoro wiemy, że szukane d jest dzielnikiem n, to dla dowolnego i zachodzi równoważność $d \mid m_i \iff d \mid \mathrm{NWD}(m_i, n)$. W szczególności możemy rozważać dzielniki liczby $n' = \mathrm{NWD}(m_k, n)$, a więc zastąpić w sformułowaniu

problemu liczbę n przez n' i pozbyć się warunku związanego z m_k. Ponadto możemy od razu zamienić każdą spośród pozostałych liczb m_i na $m'_i = \mathrm{NWD}(m_i, n')$. W szczególności otrzymujemy algorytm o złożoności $O(\sqrt{n'}+k\cdot(\log n+D(n'))$. Owa złożoność pesymistycznie nie jest jednak lepsza, gdyż m_k może być np. równe 0.

Problemem, jaki nam wciąż pozostaje, jest konieczność wykonywania $O(k)$ kroków algorytmu dla każdego badanego dzielnika liczby n'. Na początek możemy zauważyć, że każde m'_i jest dzielnikiem n', więc wśród nich będzie co najwyżej $D(n') \leqslant 17\,280$ różnych liczb. Tym samym złożoność zmniejsza się do $O(\sqrt{n'}+k\log n+D(n')\cdot\min(k, D(n')))$. Dla nas jest to wciąż zbyt wolno — zajmiemy się konstruowaniem szybszego algorytmu.

Jak więc możemy bardziej zredukować liczbę kroków przeznaczonych na każdy dzielnik? Z pomocą może nam przyjść programowanie dynamiczne, które w przypadku dzielników jest przydatną, lecz nietrywialną metodą. Standardowo programowanie dynamiczne operuje bowiem na tablicach (jedno lub kilkuwymiarowych), rzadziej drzewach czy skierowanych grafach acyklicznych. Widząc zbiór dzielników, niekoniecznie łatwo od razu dostrzec, jak można skonstruować na nim efektywny algorytm dynamiczny.

Zanim podamy rozwiązanie, wprowadźmy pojęcie przydatne do jego opisu. Otóż powiemy, że dzielnik d liczby n' jest *dobry*, jeżeli nie dzieli żadnej spośród liczb $m'_1, \ldots, m'_{k-1}$, w przeciwnym razie będziemy go nazywali *złym*.



Rys. 1: Zbiór dzielników liczby $n' = 6^5 = 7776$ z zaznaczonymi przykładowymi liczbami m'_i dla $i < k$ (pełne kółka). Pozostałe dzielniki są oznaczone pustymi kółkami, złe dzielniki są zamalowane na szaro.

Zachęcamy Czytelnika, aby obejrzawszy rysunek, sam spróbował wymyślić rozwiązanie. Poniżej opisujemy rozwiązanie wzorcowe.

Jak sugeruje rysunek, dwa dzielniki, takie że iloraz jednego przez drugi jest liczbą pierwszą, można traktować jako bliskie sobie, podobnie jak sąsiednie elementy tablicy.

Wówczas złe dzielniki mają regularną strukturę, którą w ogólnym przypadku opisuje następująca obserwacja.

Lemat 1. Niech $d \mid n'$ będzie złym dzielnikiem. Wówczas dla pewnego i zachodzi $d = m'_i$ lub dla pewnej liczby pierwszej p dzielącej n' liczba $d \cdot p$ także jest złym dzielnikiem n'.

Dowód: Jeśli dzielnik d jest zły, to z definicji istnieje takie $i < k$, że $d \mid m'_i$. Jeśli $d = m'_i$, mamy już tezę. W przeciwnym wypadku niech p będzie dowolnym pierwszym dzielnikiem liczby $\frac{m'_i}{d}$ (która oczywiście jest dzielnikiem m'_i, a więc również n'). Wówczas $d \cdot p \mid m'_i$, a więc także jest złym dzielnikiem n'. ■

Wobec tego, jeśli będziemy przeglądać wszystkie dzielniki n' w porządku malejącym, to przy badaniu dzielnika d będziemy już dla wszystkich dzielników postaci $d \cdot p$ wiedzieli, czy są dobre, czy nie. Oczywiście trzeba pamiętać, aby na początku zaznaczyć, że liczby $m'_1, \ldots, m'_{k-1}$ są złe.

Ile zyskujemy w ten sposób? Zamiast $O(\min(k, D(n)))$ kroków algorytmu przeznaczonych na zbadanie jednego dzielnika liczby n' otrzymujemy $O(P(n'))$ kroków, gdzie $P(s)$ to liczba liczb pierwszych dzielących s. Wiemy, że $D(n) \leqslant 17\,280$. Jakie natomiast możemy znaleźć ograniczenie na $P(n)$? Zacznijmy wymnażać kolejne liczby pierwsze, począwszy od 2. Po przemnożeniu pierwszych 13 liczb pierwszych otrzymujemy już wynik większy od 10^{14}. Jeżeli więc $P(s) \geqslant 13$, to nie zwiększając liczby s, możemy zamienić wszystkie wykładniki w jej faktoryzacji na 1, a następnie (potencjalnie) pozmniejszać liczby pierwsze w niej występujące, tak aby były to kolejne liczby pierwsze. Po takich operacjach nadal otrzymamy liczbę większą niż 10^{14}. Wobec tego dla danych z zadania mamy $P(n') \leqslant 12$. Widać, że jest to ponadtysiąckrotnie mniejsze ograniczenie niż poprzednio.

Struktura danych dla dzielników

Pozostaje jeszcze kwestia wyboru struktury danych, w której będziemy przechowywać dzielniki n' wraz z informacją, czy są dobre, złe, czy jeszcze nie zbadane. Poprzednio wystarczała nam zwykła tablica, ponieważ i tak każdy dzielnik rozważaliśmy niezależnie. Teraz potrzebujemy dla danego dzielnika d szybko sprawdzać, czy $d \cdot p$ jest dzielnikiem n, dla kilkunastu różnych liczb pierwszych p. Możemy to robić z użyciem wyszukiwania binarnego w posortowanej tablicy dzielników lub zrównoważonego drzewa poszukiwań binarnych zaimplementowanego za pomocą kontenera `map` z biblioteki STL języka C++. Wtedy każda operacja na tej strukturze będzie kosztowała $O(\log D(n))$. Otrzymamy więc sumaryczną złożoność czasową algorytmu $O(\sqrt{n} + k \log n + D(n)P(n) \log D(n))$, co przy maksymalnych rozmiarach danych daje kilkanaście milionów operacji i wystarczało już do otrzymania 100 punktów.

Istnieje jednak inny, ciekawy pomysł na przechowywanie dzielników n', który nieco jeszcze zmniejsza złożoność asymptotyczną algorytmu. Dosłownie traktuje on analogię między programowaniem dynamicznym na dzielnikach a tym standardowym na tablicy.

Wyobraźmy sobie $P(n')$-wymiarową tablicę *tab*, taką że w komórce $tab[\beta_1][\beta_2]\ldots[\beta_\ell]$ pamiętamy, czy dzielnik

$$p_1^{\beta_1} p_2^{\beta_2} \ldots p_\ell^{\beta_\ell}$$

liczby n' jest dobry, czy zły. Wtedy sprawdzenie dla liczby d liczby $d \cdot p_i$ oznacza po prostu sięgnięcie do sąsiedniej komórki w jednym z wymiarów.

Jak taką tablicę zaimplementować? Stworzenie tablicy o zmiennej liczbie wymiarów może sprawić pewne problemy techniczne (nie radzimy używać 12-wymiarowych tablic czy wektorów), jednak zamiast tworzyć ją jawnie, możemy też symulować ją w tablicy jednowymiarowej *tab1d*. Chcemy zakodować ciągi $\beta_1, \ldots, \beta_\ell$ takie, że $\beta_i \leqslant \alpha_i$ dla każdego i, za pomocą pojedynczych liczb z zakresu $[1, D(n')] = [1, (\alpha_1 + 1) \cdot \ldots \cdot (\alpha_l + 1)]$. Aby to osiągnąć, przypiszmy im numery w kolejności leksykograficznej. Co ciekawe, możemy w ogóle nie konstruować zbioru dzielników, a jedynie wykorzystać następujące odwzorowania:

1. *valueToIndex*(d) zamieniające dzielnik d na indeks odpowiadającego mu pola w tablicy *tab1d*;
2. *indexToValue*(i) dokonujące przeciwnej zamiany;
3. *multiply*(i, j) zwracające liczbę *valueToIndex*$(\mathit{indexToValue}(i) \cdot p_j)$, czyli wartość kolejnego pola w j-tym wymiarze lub informację, że takie pole nie istnieje.

Operacji 1 będziemy potrzebowali $O(k)$ razy (do oznaczenia złych dzielników), operacji 2 — $O(D(n'))$ razy (do sprawdzenia, czy dobry dzielnik jest najmniejszy), zaś operacji 3 — $O(D(n') \cdot P(n'))$ razy. Pierwsze dwie możemy zaimplementować w czasie $O(\log n')$, zaś trzecią w czasie stałym, o ile wcześniej obliczymy pomocniczą tablicę $\mathit{offset}[i] = \Pi_{j=1}^{i}(\alpha_j + 1)$. Dzięki temu otrzymamy algorytm o sumarycznej złożoności

$$O\left(\sqrt{n'} + k \log n + D(n')P(n') + D(n') \log n'\right).$$

Wyniki operacji 2 można także w czasie wygenerować $O(D(n))$ i spamiętać, redukując tym samym złożoność do

$$O\left(\sqrt{n'} + k \log n + D(n')P(n')\right).$$

Szczegóły implementacyjne pozostawiamy jako jeszcze jedno ćwiczenie dla Czytelnika.

Z racji wielu możliwości implementacji poszczególnych faz (dwa sposoby generowania dzielników, kilka struktur danych dla programowania dynamicznego), powstało wiele programów wzorcowych. Znajdują się w plikach `sej.cpp`, `sej[1-8].cpp`, `sej[9-11].pas`. W nagłówku każdego z plików można sprawdzić, których metod w nim użyto.

Heurystyki przyspieszające wolniejsze rozwiązanie

Wróćmy raz jeszcze do rozwiązania, które dla każdego dzielnika liczby n' sprawdzało podzielność wszystkich różnych liczb m'_i przez niego — było to rozwiązanie o złożoności $O(\sqrt{n'} + k \log n + D(n') \cdot \min(k, D(n')))$. Widzimy, że temu rozwiązaniu niewiele brakuje, aby mieścić się w granicach kilku sekund. Okazuje się, że zastosowanie prostej heurystyki dawało w praktyce rozwiązania o czasach działania bliskich rozwiązaniu wzorcowemu, nierzadko nawet niższych. Należy dodać, że podczas zawodów wynik 100

punktów uzyskały tylko takie rozwiązania; wszystkie oparte na pomyśle wzorcowym były, niestety, obarczone drobnymi błędami w implementacji.

Przejdźmy do opisu tejże heurystyki. Oczywiście po znalezieniu takiego m'_i, które jest wielokrotnością badanego dzielnika d liczby n', możemy od razu przerwać badanie d. Przy takim postępowaniu liczyć się zaczyna kolejność, w której sprawdzamy m'_i. Otóż okazuje się, że najbardziej naturalna, rosnąca kolejność wcale nie jest najlepsza, ponieważ małe dzielniki liczb niemal antypierwszych (a dla takich liczb n' algorytm działa powoli) mają mało własnych dzielników, a duże — dużo. Tak więc zamiast rosnącej kolejności liczb m'_i należy rozpatrywać je w kolejności malejącej.

Rozwiązanie takie znajduje się w pliku `sejs9.cpp`. W przypadku wygenerowanych testów to rozwiązanie nieznacznie ustępowało najszybszym spośród rozwiązań wzorcowych, a wyraźnie pokonywało te najwolniejsze[3]. Inne, gorsze rozwiązania oparte na podobnych pomysłach przyspieszenia można znaleźć w plikach `sejs[6-8].cpp`.

Testy

Wśród testów można wyróżnić następujące grupy:

losowe — testy w pełni losowe, oznaczone literką a;

antypierwsze — testy, w których n jest liczbą niemal antypierwszą; zwykle wtedy $D(n)$ jest duże;

antypierwsze szczególne — testy z grupy **antypierwsze**, które dodatkowo mają szczególny warunek na liczbę m_k; testy te są trudne dla powolnych rozwiązań;

specyficzne — testy rozmaitych typów; często n jest w nich liczbą pierwszą (w opisach poniżej $\mathbb{P}$ to zbiór liczb pierwszych, $p_i \in \mathbb{P}$).

Nazwa	n	k	D(n)	Opis
sej1a.in	3 305	10	4	losowy
sej1b.in	840	20	32	antypierwsze
sej1c.in	1	1	1	najmniejszy możliwy test
sej1d.in	1 680	15	40	antypierwsze, $m_k = 0$
sej2a.in	566 345	50	8	losowy
sej2b.in	1 081 080	80	256	antypierwsze
sej2d.in	665 280	53	224	antypierwsze, $m_k = 0$

[3]Wiadomo, że heurystyka ta nieco poprawia złożoność, ale nie tłumaczy to w pełni jej znakomitego działania nawet dla n wyraźnie większych niż 10^{14}. Pozostaje ciekawe pytanie, czy istnieją takie testy, w których spisywałaby się istotnie gorzej niż na tych, gdzie m'_i to wszystkie dzielniki n' mniejsze niż ustalona wartość. Tak właśnie, z bardzo drobnymi losowymi modyfikacjami, wyglądała podczas zawodów większość testów wydajnościowych.

Nazwa	n	k	D(n)	Opis
sej3a.in	88 565 049	100	8	losowy
sej3b.in	73 513 440	100	768	antypierwsze
sej3d.in	73 513 440	89	768	antypierwsze, $m_k = 0$
sej4a.in	7 300 943 302	200	4	losowy
sej4b.in	10 475 665 200	250	2 400	antypierwsze
sej4d.in	6 983 776 800	342	2 304	antypierwsze, $m_k = 0$
sej5a.in	945 494 546 981	500	6	losowy
sej5b.in	963 761 198 400	600	6 720	antypierwsze
sej5c.in	999 999 999 989	2	2	$n \in \mathbb{P}$, $m_k = 0$
sej5d.in	963 761 198 400	656	6 720	antypierwsze, $m_k = 0$
sej6a.in	17 536 146 808 269	1 000	32	losowy
sej6b.in	55 898 149 507 200	1 000	15 360	antypierwsze
sej6c.in	99 999 999 999 973	1	2	$n \in \mathbb{P}$, $m_1 \neq 0$
sej6d.in	55 898 149 507 200	970	15 360	antypierwsze, $m_k = 0$
sej7a.in	5 460 205 985 383	1 000	4	losowy
sej7b.in	65 214 507 758 400	1 000	16 128	antypierwsze
sej7c.in	99 989 999 999 941	1 000	2	$n \in \mathbb{P}$, $m_k = 0$
sej7d.in	32 607 253 879 200	960	13 824	antypierwsze, $m_k = 0$
sej8a.in	59 161 962 579 511	1 000	4	losowy
sej8b.in	93 163 582 512 000	1 000	16 384	antypierwsze
sej8c.in	99 999 999 999 962	1 000	4	$n = 2p_1$, $d = 2$
sej8d.in	55 898 149 507 200	992	15 360	antypierwsze, $m_k = 0$
sej9a.in	68 039 674 035 556	1 000	12	losowy
sej9b.in	97 821 761 637 600	1 000	17 280	antypierwsze
sej9c.in	99 998 289 000 187	1 000	4	$n = p_2 \cdot p_3$
sej9d.in	93 163 582 512 000	978	16 384	antypierwsze, $m_k = 0$
sej10a.in	50 807 342 460 355	1 000	64	losowy
sej10b.in	100 000 000 000 000	1 000	225	antypierwsze
sej10c.in	97 999 999 999 994	1 000	4	$n = 2p_4$, sejf otwierają: 0 i $\frac{n}{2}$
sej10d.in	93 163 582 512 000	1 000	16 384	antypierwsze, $m_k = 0$
sej11a.in	69 909 258 653 683	100 000	4	losowy
sej11b.in	97 821 761 637 600	100 000	17 280	antypierwsze
sej11c.in	97 821 761 637 600	100 000	17 280	antypierwsze, duże m_k

Nazwa	n	k	D(n)	Opis
sej11d.in	97 821 761 637 600	149 000	17 280	antypierwsze, $m_k = 0$
sej11e.in	97 821 761 637 600	100 500	17 280	antypierwsze, $m_k = 0$
sej12a.in	31 451 726 809 867	200 000	32	losowy
sej12b.in	97 821 761 637 600	200 000	17 280	antypierwsze
sej12c.in	48 910 880 818 800	196 054	14 400	antypierwsze, duże m_k
sej12d.in	65 214 507 758 400	156 200	16 128	antypierwsze, $m_k = 0$
sej12e.in	97 821 761 637 600	196 534	17 280	antypierwsze, $m_k = 0$
sej13a.in	16 016 997 334 119	250 000	8	losowy
sej13b.in	97 821 761 637 600	250 000	17 280	antypierwsze
sej13c.in	97 821 761 637 600	241 542	17 280	antypierwsze, duże m_k
sej13d.in	97 821 761 637 600	245 434	17 280	antypierwsze, $m_k = 0$
sej13e.in	97 821 761 637 600	249 542	17 280	antypierwsze, $m_k = 0$
sej14a.in	87 609 341 358 200	250 000	48	losowy
sej14b.in	97 821 761 637 600	250 000	17 280	antypierwsze
sej14c.in	93 163 582 512 000	250 000	16 384	antypierwsze, duże m_k
sej14d.in	97 821 761 637 600	250 000	17 280	antypierwsze, $m_k = 0$
sej14e.in	97 821 761 637 600	249 050	17 280	antypierwsze, $m_k = 0$

Jacek Tomasiewicz — Treść zadania, Opracowanie
Zbigniew Wojna — Program

Dostępna pamięć: 32 MB. *OI, Etap II, dzień pierwszy, 09.02.2011*

Różnica

Mamy dane słowo złożone z n małych liter alfabetu angielskiego `a` – `z`. *Chcielibyśmy wybrać pewien niepusty, spójny (tj. jednokawałkowy) fragment tego słowa, w taki sposób, aby różnica pomiędzy liczbą wystąpień najczęściej i najrzadziej występującej w tym fragmencie litery była jak największa. Zakładamy przy tym, że najrzadziej występująca litera w wynikowym fragmencie słowa musi mieć w tym fragmencie co najmniej jedno wystąpienie. W szczególności, jeżeli fragment składa się tylko z jednego rodzaju liter, to najczęstsza i najrzadsza litera są w nim takie same.*

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą n *(*$1 \leqslant n \leqslant 1\,000\,000$*), oznaczającą długość słowa. Drugi wiersz zawiera słowo składające się z* n *małych liter alfabetu angielskiego.*

W testach wartych przynajmniej 30% punktów zachodzi dodatkowy warunek $n \leqslant 100$*.*

Wyjście

Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą, równą maksymalnej wartości różnicy między liczbą wystąpień najczęściej i najrzadziej występującej litery, jaką możemy znaleźć w pewnym spójnym fragmencie danego słowa.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`10` `aabbaaabab`	`3`

Wyjaśnienie do przykładu: *Fragment słowa, dla którego różnica między liczbą liter* `a` *i* `b` *wynosi 3, to* `aaaba`.

Rozwiązanie

Wprowadzenie

W tym zadaniu problem wyjątkowo nie jest ukryty w ramach historyjki, więc nie ma potrzeby formułowania go ponownie w bardziej formalny sposób. Ustalmy tylko, że przy szacowaniu złożoności rozwiązań przez A będziemy oznaczać rozmiar alfabetu (w przypadku naszego zadania mamy $A = 26$).

Każdy Czytelnik z pewnością zauważy, że skonstruowanie w pełni poprawnego, aczkolwiek siłowego rozwiązania tego zadania nie stanowi większej trudności.

Rozwiązanie siłowe $O(n^3 + n^2 \cdot A)$

Najprostszym rozwiązaniem jest rozpatrzenie wszystkich możliwych fragmentów, a dla każdego z nich znalezienie najczęściej i najrzadziej występującej litery. Oto pseudokod takiego siłowego algorytmu:

```
1:  wynik := 0;
2:  { rozpatrujemy każdy fragment }
3:  for i := 1 to n − 1 do
4:    for j := i + 1 to n do begin
5:      for k := 'a' to 'z' do
6:        licznik[k] := 0;
7:      for k := i to j do
8:        licznik[slowo[k]] := licznik[slowo[k]] + 1;
9:      { znajdujemy maksymalną i minimalną liczbę wystąpień }
10:     maks := 0;   mini := ∞;
11:     for k := 'a' to 'z' do
12:     begin
13:       maks := max(maks, licznik[k]);
14:       if licznik[k] > 0 then
15:         mini := min(mini, licznik[k]);
16:     end
17:     wynik := max(wynik, maks − mini);
18:   end
19: return wynik;
```

Powyższy algorytm działa w czasie $O(n^3+n^2 \cdot A)$, ponieważ wszystkich fragmentów jest $O(n^2)$, a dla każdego z nich znajdujemy w czasie $O(n + A)$ minimalne i maksymalne wystąpienie liter. Za takie rozwiązanie można było uzyskać około 30 punktów. Jego przykładowa implementacja znajduje się w plikach `rozs1.cpp` oraz `rozs4.pas`.

Rozwiązanie wolne $O(n^2 \cdot A)$

Jak widać, większość czasu w poprzednim rozwiązaniu spędzamy, zliczając wystąpienia liter w danym fragmencie. Ten element możemy łatwo poprawić — gdybyśmy wcześniej policzyli dla każdego prefiksu słowa (czyli początkowego jego fragmentu), ile razy każda litera w nim występuje, wystarczyłoby odjąć odpowiednie wartości.

Wobec tego na początku w następujący sposób wypełnimy tablicę *pref*:

```
1: for k := 'a' to 'z' do pref[0][k] := 0;
2: for i := 1 to n do begin
3:   for k := 'a' to 'z' do pref[i][k] := pref[i − 1][k];
4:   pref[i][slowo[i]] := pref[i][slowo[i]] + 1;
5: end
```

Teraz nie musimy przeglądać wszystkich liter każdego fragmentu. Linie 5–8 poprzedniego rozwiązania możemy zastąpić następującym pseudokodem:

```
1: for k := 'a' to 'z' do
2:     licznik[k] := pref[j][k] − pref[i − 1][k];
```

Nowy algorytm działa w czasie $O(n^2 \cdot A)$, gdyż w każdym fragmencie rozważamy każdą literę z alfabetu. Niestety takie rozwiązanie jest wciąż za wolne i dostaje około 40 punktów. Jego implementacja znajduje się w plikach `rozs2.cpp` oraz `rozs5.pas`.

Szybsze rozwiązanie $O(n \cdot A^2)$

Zastanówmy się, jak lepiej rozwiązać zadanie. Złożoność kwadratowa w zależności od n nie jest satysfakcjonująca, więc nie możemy sobie pozwolić na przeglądanie wszystkich fragmentów. Spróbujmy podejść do problemu od innej strony. Rozważmy wszystkie pary różnych liter z założeniem, że docelowo jedna będzie najczęściej występującą, a druga najrzadziej. Teraz dla danej pary, przykładowo (`a`, `b`), będziemy szukać fragmentu, w którym różnica liczb wystąpień `a` i `b` jest możliwie największa. Wymagamy przy tym, żeby w wynikowym fragmencie litera `b` występowała przynajmniej raz.

Gdyby pominąć ostatni warunek, można by prosto sprowadzić problem do klasycznego zadania: znajdowania w ciągu liczb spójnego fragmentu o maksymalnej sumie. Wystarczy zastąpić wystąpienia `a` przez 1, `b` przez -1, a pozostałych liter przez `0`.

Znane są algorytmy wyznaczające maksymalną sumę spójnego fragmentu ciągu w czasie liniowym. Możemy wyliczać dynamicznie aktualną sumę prefiksu ciągu. Aby wyznaczyć podciąg spójny o maksymalnej sumie kończący się na danej pozycji, wybieramy krótszy prefiks o dotychczas najmniejszej sumie. Wynikiem jest różnica między sumami prefiksów: bieżącego i najmniejszego krótszego. Oto pseudokod takiego klasycznego algorytmu:

```
1: wynik, suma, mini := 0;
2: for i := 1 to n do begin
3:     suma := suma + a[i];
4:     mini := min(mini, suma);
5:     wynik := max(wynik, suma − mini);
6: end
7: return wynik;
```

W naszym przypadku musimy jednak pamiętać, że znaleziony fragment musi zawierać przynajmniej jedną -1, a powyższe rozwiązanie tego nie zakłada. Szukając prefiksu o najmniejszej sumie, musimy ograniczyć się tylko do tych, które nie zawierają ostatnio odwiedzonej -1. Zauważmy jeszcze, że każdy interesujący nas prefiks jest albo pusty, albo kończy się liczbą -1. W przeciwnym razie moglibyśmy ten prefiks skrócić, uzyskując prefiks o mniejszej sumie.

Wobec tego, gdy napotkaliśmy już k razy wartość -1, interesują nas jedynie: prefiks pusty oraz prefiksy kończące się w jednej z $k-1$ pierwszych wartości -1. Jeśli jeszcze na żadną -1 nie natrafiliśmy, czyli $k = 0$, to nie możemy rozważać żadnego prefiksu. Poprzedni pseudokod poprawiamy następująco:

```
1: wynik, suma, suma_pop := 0;
2: mini := ∞;
3: for i := 1 to n do begin
```

```
4:    suma := suma + a[i];
5:    if a[i] = −1 then begin
6:       mini := min(mini, suma_pop);
7:       suma_pop := suma;
8:    end
9:    wynik := max(wynik, suma − mini);
10: end
11: return wynik;
```

Tutaj *sum_pop* to suma wartości prefiksu kończącego się ostatnią wczytaną -1; na początku — prefiksu pustego. Wartość *mini* to natomiast minimalna suma prefiksu, który możemy rozważać; na początku nie ma takiego prefiksu.

Otrzymujemy w ten sposób algorytm, który działa w czasie $O(n \cdot A^2)$, gdyż wszystkich par liter z alfabetu jest $O(A^2)$, a znalezienie podciągu o maksymalnej sumie zajmuje nam czas $O(n)$. Rozwiązania tej złożoności otrzymywały co najmniej 60 punktów. Zaprezentowane wyżej zostało zaimplementowane w pliku `rozs9.cpp`.

Rozwiązanie wzorcowe $O(n \cdot A)$

Zastanówmy się, jak poprawić poprzednie rozwiązanie. Zauważmy, że gdy w ciągu rozpatrujemy $a[i] = 0$, to nie dzieje się nic ciekawego. Dlaczego więc nie pominąć wszystkich zer?

Okazuje się, że jest to dobry pomysł. Trzeba oczywiście pamiętać, żeby wycinając zera, nie przeglądać całego ciągu, gdyż wtedy złożoność pozostanie bez zmian. Należy dla każdej litery stworzyć listę zawierającą pozycje jej wystąpień w słowie, a następnie dla danej pary liter scalać listy odpowiadające tym literom.

Oszacujmy czas działania tak poprawionego algorytmu. Generowanie list wystąpień zajmie nam czas $O(n + A)$. Czas działania głównej fazy to suma czasów dla wszystkich par liter z alfabetu. Niech $occ(\ell)$ oznacza liczbę wystąpień litery ℓ w początkowym słowie. Wtedy czas dla pojedynczej pary liter (ℓ_1, ℓ_2) wynosi $O(occ(\ell_1)+occ(\ell_2))$. Ostatecznie sumaryczna złożoność to z dokładnością do czynnika stałego:

$$\sum_{\ell_1}\sum_{\ell_2}\left(occ\left(\ell_1\right)+occ\left(\ell_2\right)\right)=2\sum_{\ell_1}\sum_{\ell_2}occ\left(\ell_2\right)=2A\cdot\sum_{\ell_2}occ\left(\ell_2\right)=O(n\cdot A).$$

Implementacja opisanego rozwiązania wzorcowego znajduje się w pliku `roz8.cpp`. Takie rozwiązanie otrzymuje maksymalną liczbę punktów, podobnie jak inne rozwiązania o tej złożoności.

Alternatywne warianty rozwiązania $O(n \cdot A)$

Przedstawiony wyżej klasyczny algorytm wyznaczania maksymalnej sumy spójnego fragmentu ciągu można zapisać nieco inaczej. Zmienne *suma* i *mini* możemy zastąpić jedną, równą *suma* − *mini*. Wówczas na algorytm pierwotnie skonstruowany jako dynamiczny można spojrzeć jak na rozwiązanie zachłanne.

```
1: wynik, suma := 0;
2: for i := 1 to n do begin
3:    suma := suma + a[i]; { przedłużamy aktualny ciąg }
4:    if suma < 0 then suma := 0; { zaczynamy konstrukcję od nowa }
5:    wynik := max(wynik, suma);
6: end
7: return wynik;
```

Taki algorytm również można w naturalny sposób dopasować do potrzeb naszego zadania, w którym wymagamy wystąpienia choć raz wartości -1. Jest to jednak bardziej skomplikowane. Szczegóły pozostawiamy zainteresowanemu Czytelnikowi jako ćwiczenie. Przykładowa implementacja znajduje się w pliku `roz9.cpp`.

Redukcję złożoności $O(n \cdot A^2)$ o czynnik A można także wykonać zupełnie inaczej. Można równocześnie prowadzić obliczenia dla wszystkich par liter lub dla par o wspólnym pierwszym elemencie. Wówczas używane zmienne zastępujemy odpowiednio dwu- lub jednowymiarowymi tablicami. To podejście daje w praktyce wyraźnie szybsze programy, gdyż pomijamy fazy generowania i scalania list wystąpień. Przykład takiej implementacji można znaleźć w pliku `roz10.cpp`.

Testy

W zestawie były trzy typy testów: małe testy poprawnościowe wygenerowane ręcznie (typ **R**), większe losowe testy wydajnościowe (typ **L**) oraz testy poprawnościowo-wydajnościowe (typ **RR**), w których litery występowały równomiernie.

Nazwa	n	Opis
roz1a.in	10	test typu **R**
roz1b.in	12	test typu **R**
roz1c.in	5	test typu **R**
roz1d.in	3	test typu **R**
roz2a.in	10	test typu **R**
roz2b.in	50	test typu **L**
roz2c.in	17	test typu **RR**
roz3a.in	1	test typu **R**
roz3b.in	100	test typu **L**
roz3c.in	91	test typu **RR**
roz4a.in	2 500	test typu **L**
roz4b.in	10	test typu **R**
roz5a.in	10 000	test typu **L**
roz5b.in	100	test typu **R**

Nazwa	n	Opis
roz6a.in	100 000	test typu **L**
roz6b.in	10	test typu **R**
roz6c.in	19 800	test typu **RR**
roz7a.in	1 000 000	test typu **L**
roz7b.in	5	test typu **R**
roz7c.in	817 824	test typu **RR**
roz8a.in	1 000 000	test typu **L**
roz8b.in	803 088	test typu **RR**
roz8c.in	856 836	test typu **RR**
roz9a.in	1 000 000	test typu **L**
roz9b.in	999 984	test typu **RR**
roz9c.in	999 027	test typu **RR**
roz10a.in	1 000 000	test typu **L**
roz10b.in	1 000 000	test typu **RR**
roz10c.in	999 999	test typu **RR**

Michał Pilipczuk
Treść zadania, Opracowanie

Zbigniew Wojna
Program

Dostępna pamięć: 256 MB.

OI, Etap II, dzień pierwszy, 09.02.2011

Śmieci

Przedsiębiorstwo Oczyszczania Bajtogrodu (POB) podniosło drastycznie ceny za wywóz śmieci. Część mieszkańców zrezygnowała z płacenia za wywóz śmieci i zaczęła wyrzucać je na ulice. W rezultacie wiele ulic Bajtogrodu tonie w śmieciach.

Sieć drogowa Bajtogrodu składa się z n skrzyżowań, z których niektóre połączone są dwukierunkowymi ulicami. Żadne dwie ulice nie łączą tej samej pary skrzyżowań. Niektóre z ulic są zaśmiecone, podczas gdy inne nie.

Burmistrz Bajtogrodu, Bajtazar, zdecydował się na niekonwencjonalną akcję mającą skłonić mieszkańców do płacenia za wywóz śmieci. Postanowił on oczyścić tylko niektóre ulice miasta — te, przy których większość mieszkańców opłaciła wywóz śmieci. Natomiast te ulice, przy których większość mieszkańców nie opłaciła wywozu śmieci, postanowił pozostawić zaśmiecone lub — jeśli to konieczne — zwieźć na nie śmieci z innych ulic! Bajtazar przygotował plan miasta, na którym zaznaczył, które ulice docelowo powinny być czyste, a które zaśmiecone. Niestety, pracownicy POB-u nie są w stanie ogarnąć planu Bajtazara. Są jednak w stanie wykonywać niezbyt skomplikowane zlecenia.

Pojedyncze zlecenie polega na wykonaniu kursu śmieciarką, rozpoczynającego się na dowolnie wybranym skrzyżowaniu, prowadzącego określonymi ulicami i kończącego się na tym samym skrzyżowaniu, na którym zaczął się kurs. Przy tym, każde skrzyżowanie może w jednym kursie zostać odwiedzone co najwyżej raz, z wyjątkiem skrzyżowania, od którego kurs się rozpoczął i na którym się kończy (na którym śmieciarka pojawia się dokładnie dwa razy). Śmieciarka, jadąc zaśmieconą ulicą, sprząta ją, jadąc zaś czystą ulicą, wręcz przeciwnie — zaśmieca ją, wyrzucając śmieci.

Bajtazar zastanawia się, czy może zrealizować swój plan, zlecając ileś kursów śmieciarki. Pomóż mu i napisz program, który wyznaczy zestaw takich kursów lub stwierdzi, że nie jest to możliwe.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie liczby całkowite oddzielone pojedynczym odstępem: n i m ($1 \leqslant n \leqslant 100\,000$, $1 \leqslant m \leqslant 1\,000\,000$), oznaczające odpowiednio liczbę skrzyżowań oraz liczbę ulic w Bajtogrodzie. Skrzyżowania są ponumerowane od 1 do n. W kolejnych m wierszach znajdują się opisy kolejnych ulic, po jednej w wierszu. W każdym z tych wierszy znajdują się po cztery liczby całkowite oddzielone pojedynczymi odstępami: a, b, s i t ($1 \leqslant a < b \leqslant n$, $s, t \in \{0, 1\}$). Taka czwórka oznacza, że skrzyżowania a i b są połączone ulicą, przy czym s oznacza obecny stan zaśmiecenia ulicy (0 oznacza czystą, a 1 zaśmieconą), zaś t stan docelowy według planu Bajtazara.

Możesz założyć, że jeśli istnieje zestaw kursów realizujący plan Bajtazara, to istnieje również taki zestaw, w którym łączna liczba ulic, którymi prowadzą kursy śmieciarki, nie przekracza $5 \cdot m$.

W testach wartych 60% punktów zachodzi dodatkowo ograniczenie $m \leqslant 10\,000$.

Wyjście

Jeżeli za pomocą kursów śmieciarką nie da się zrealizować planu Bajtazara, to pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać słowo „`NIE`*". W przeciwnym razie na wyjściu należy wypisać dowolny zestaw kursów realizujący plan Bajtazara, w którym łączna liczba ulic, którymi prowadzą kursy, nie przekracza $5 \cdot m$. Pierwszy wiersz wyjścia powinien zawierać k: liczbę kursów w zestawie. W kolejnych k wierszach powinny znaleźć się opisy kolejnych kursów, po jednym w wierszu. Wiersz $(i+1)$-szy powinien zaczynać się dodatnią liczbą k_i oznaczającą liczbę ulic, którymi prowadzi i-ty kurs. Po pojedynczym odstępie powinno znaleźć się $k_i + 1$ numerów kolejnych skrzyżowań, przez które prowadzi kurs, pooddzielanych pojedynczymi odstępami.*

Przykład

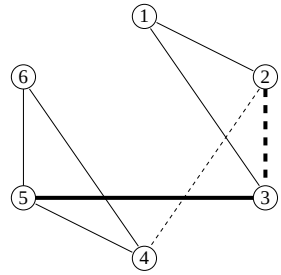
Na rysunkach cienką linią zaznaczono ulice, które są czyste, a grubą te, które są zaśmiecone. Linią przerywaną zaznaczono ulice, które docelowo powinny być czyste, a ciągłą te, które docelowo powinny być zaśmiecone.

Dla danych wejściowych:

```
6 8
1 2 0 1
2 3 1 0
1 3 0 1
2 4 0 0
3 5 1 1
4 5 0 1
5 6 0 1
4 6 0 1
```

jednym z poprawnych wyników jest:

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

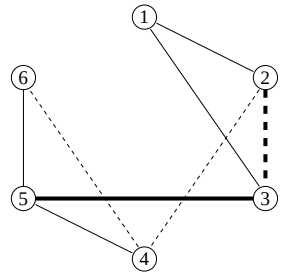


natomiast dla danych wejściowych:

```
6 8
1 2 0 1
2 3 1 0
1 3 0 1
2 4 0 0
3 5 1 1
4 5 0 1
5 6 0 1
4 6 0 0
```

poprawnym wynikiem jest:

```
NIE
```



Rozwiązanie

Analiza problemu

Sformułujmy nasze zadanie w języku teorii grafów. Niech $G = (V, E)$ będzie grafem reprezentującym plan Bajtogrodu: wierzchołki to skrzyżowania, a krawędzie to łączące je ulice. Z treści zadania wiemy, że krawędzie łączą różne wierzchołki i każde dwa wierzchołki łączy co najwyżej jedna krawędź. „Plany zaśmiecenia" możemy reprezentować jako etykietowania krawędzi liczbami 0 (ulica czysta) oraz 1 (ulica zaśmiecona). Kursom śmieciarki odpowiadają natomiast cykle proste, czyli takie cykle w grafie, w których każdy wierzchołek występuje co najwyżej raz. Problem postawiony w zadaniu daje się teraz przetłumaczyć następująco. Mamy dane dwa etykietowania i pytamy, czy da się jedno z nich przekształcić w drugie, używając jedynie operacji zamiany zer na jedynki i jedynek na zera wzdłuż cykli prostych. Jeśli odpowiedź jest twierdząca, chcemy wskazać odpowiedni ciąg operacji. Żądamy przy tym, aby łączna długość cykli nie była zbyt duża.

Na początek, dla wygody zapisu, zmieńmy nieco konwencję. Od teraz ulicę czystą będziemy oznaczać etykietą 1, zaś zaśmieconą — etykietą -1. Ustalmy pewien plan zaśmiecenia, czyli etykietowanie $P : E \to \{-1, 1\}$. Każdemu wierzchołkowi $v \in V$ przypiszmy liczbę $\mu_P(v) = \prod_{vw \in E} P(vw)$, czyli iloczyn etykiet na krawędziach wychodzących z wierzchołka v. Mówiąc prosto, $\mu_P(v)$ jest równe 1, jeśli parzyście wiele krawędzi wychodzących z v jest zaśmieconych, a -1 w przeciwnym przypadku.

Zobaczmy, co się dzieje, gdy wykonujemy operację zamiany wzdłuż pewnego cyklu. Zauważmy, że dla każdego wierzchołka v z cyklu zmienią się dokładnie dwie etykiety wychodzących z niego krawędzi. Oznacza to, że wartość $\mu_P(v)$ zostanie przemnożona przez -1 dwukrotnie, czyli w ogóle się nie zmieni! Jednocześnie, wartości μ_P dla wierzchołków spoza cyklu oczywiście również nie ulegną zmianie. Wynika stąd, że liczby $\mu_P(v)$ dla $v \in V$ nie zmieniają się w ogóle wskutek wykonywanych operacji.

Mając dane (wejściowe) etykietowania P_1 i P_2, możemy obliczyć wartości $\mu_{P_1}(v)$, $\mu_{P_2}(v)$ dla wszystkich wierzchołków $v \in V$. Jeśli któraś para się nie zgodzi, musimy odpowiedzieć „`NIE`". W dalszych rozważaniach przyjmijmy wobec tego, że dla każdego wierzchołka zachodzi $\mu_{P_1}(v) = \mu_{P_2}(v)$. Okazuje się, że jest to warunek wystarczający na istnienie rozwiązania.

Rozważmy zbiór F złożony z tych krawędzi wyjściowego grafu G, na których etykietowania P_1 oraz P_2 są różne. Pokażemy, że w grafie $H = (V, F)$ utworzonym z tych krawędzi stopnie wszystkich wierzchołków są parzyste. Zauważmy, że ponieważ dla dowolnego ustalonego wierzchołka $v \in V$ mamy $\mu_{P_1}(v) = \mu_{P_2}(v)$, to w grafie H ten wierzchołek ma parzysty stopień. Istotnie, skoro $\mu_{P_1}(v) = \mu_{P_2}(v)$, to równoważnie $\mu_{P_1}(v)\mu_{P_2}(v) = 1$. Wobec tego

$$1 = \mu_{P_1}(v)\mu_{P_2}(v) = \prod_{vw \in E} P_1(vw) \prod_{vw \in E} P_2(vw) = \prod_{vw \in E} \left(P_1(vw)P_2(vw)\right).$$

W ostatnim iloczynie czynniki są równe 1 dla tych krawędzi, dla których P_1 i P_2 się zgadzają, i -1 dla tych, dla których się nie zgadzają. Skoro cały iloczyn ma być równy 1, tych drugich jest parzyście wiele.

Zauważmy, że aby przerobić etykietowanie P_1 na etykietowanie P_2, musimy co najmniej raz zmienić etykietę na każdej krawędzi z F, czyli suma długości cykli musi wynieść co najmniej $|F|$. Okazuje się, że $|F|$ już wystarcza — da się tak zestawić kursy śmieciarki, żeby każdą krawędzią z F przejechać dokładnie raz. Jest to ściślej sformułowane w następującym lemacie.

Lemat 1. Jeśli każdy wierzchołek grafu $H = (V, F)$ ma parzysty stopień, to H da się rozłożyć na krawędziowo rozłączne cykle proste. Formalnie, istnieje taki zbiór cykli prostych $\mathcal{C}$, że każda krawędź $e \in F$ należy do dokładnie jednego cyklu $C \in \mathcal{C}$.

Dowód: Przeprowadzimy dowód indukcyjny (względem liczby krawędzi). Jeśli graf nie ma żadnej krawędzi, teza jest w sposób trywialny prawdziwa. Załóżmy wobec tego, że w H jest krawędź.

Wpierw pokażmy, że w H jest wówczas pewien cykl prosty. Weźmy dowolny wierzchołek v_0, z którego wychodzi jakaś krawędź. Przejdźmy tą krawędzią do jego sąsiada v_1. Skoro v_1 ma stopień parzysty, to musi z niego wychodzić jeszcze jakaś inna krawędź, prowadząca, powiedzmy, do v_2. Przejdźmy więc wzdłuż tej krawędzi. Teraz z kolei v_2 ma stopień parzysty, więc wychodzi z niego jakaś krawędź różna od tej, którą przyszliśmy. Kontynuujemy tak budowę ścieżki $v_0, v_1, v_2, \ldots$ aż do momentu, gdy dla któregoś z wierzchołków v_i okaże się, że leży on na już zbudowanej ścieżce — powtórzyliśmy wierzchołek, czyli $v_i = v_j$ dla pewnego $j < i$. Wierzchołków jest skończenie wiele, więc w końcu musi tak się stać. Wówczas krawędzie łączące kolejno wierzchołki $v_j, v_{j+1}, \ldots, v_i = v_j$ tworzą oczywiście cykl. Co więcej, jest to cykl prosty, gdyż cały czas budowaliśmy ścieżkę prostą.

Mając już cykl w garści, możemy usunąć go z grafu H. Zauważmy, że po usunięciu cyklu stopień każdego wierzchołka nie zmienił się lub spadł o 2, więc w szczególności pozostał parzysty. Wobec tego nasz cykl możemy dołączyć do rozkładu, którego istnienie wynika z założenia indukcyjnego. ■

Widzimy zatem, że po obliczeniu wartości $\mu_{P_1}(v)$, $\mu_{P_2}(v)$ i sprawdzeniu ich równości dla każdego v, wystarczy efektywnie zaimplementować znajdowanie rozkładu na cykle, którego istnienie gwarantuje Lemat 1. Przeprowadzony dowód daje już pewien algorytm. Zastanówmy się nad jego złożonością. Pesymistycznie może okazać się, że w każdym kroku z bardzo długiej ścieżki odcinamy bardzo krótki cykl. Oznacza to, że aż $\Theta(m)$ razy możemy budować ścieżkę długości $\Theta(n)$, czyli czas działania całego algorytmu możemy oszacować jedynie przez $O(mn)$. Programy implementujące taki algorytm, jak na przykład `smis2.cpp` i `smis4.pas`, otrzymywały około 20 punktów.

Rozwiązanie wzorcowe

Zauważmy, że w poprzednim rozwiązaniu zupełnie niepotrzebnie po odcięciu cyklu wyrzucaliśmy już zbudowany odcinek ścieżki. Zamiast tego, możemy budować kolejny cykl, rozpoczynając nie od pustej ścieżki, ale od fragmentu już zbudowanego, powstałego po odcięciu odcinka pomiędzy v_i a jego poprzednim wystąpieniem v_j. W ten sposób każda krawędź grafu H jest dokładana do ścieżki dokładnie raz — ponieważ gdy jest usuwana ze ścieżki, to razem z całym cyklem jest też wyrzucana w ogóle z grafu H. Możemy zresztą usuwać krawędź z grafu już przy dodawaniu jej do ścieżki

— wtedy nie musimy dodatkowo sprawdzać, czy przypadkiem nie cofnęliśmy się na ścieżce. Cały algorytm wygląda teraz następująco:

```
1:  procedure Śmieci
2:  begin
3:    Wczytaj graf G oraz etykietowania P_1, P_2;
4:    Dla każdego v ∈ V oblicz wartości μ_{P_1}(v), μ_{P_2}(v);
5:    if dla pewnego v ∈ V zachodzi μ_{P_1}(v) ≠ μ_{P_2}(v) then return NIE;
6:    Zbuduj graf H;
7:    foreach v in V do begin
8:      S := v;
9:      while deg_H(v) > 0 do begin
10:       w := sąsiad końca ścieżki S;
11:       Usuń użytą krawędź z grafu;
12:       S := S.append(w);
13:       if w zaznaczony jako odwiedzony then begin
14:           Przejrzyj S od końca, znajdując poprzednie wystąpienie w;
15:           Odetnij fragment późniejszy niż to wystąpienie;
16:           Odznacz odwiedzenie wierzchołków z fragmentu;
17:           Wypisz fragment jako kolejny cykl;
18:       end else Zaznacz w jako odwiedzony;
19:     end
20:   end
21: end
```

Oczywiście, skoro na wyjściu wpierw musimy wypisać liczbę uzyskanych cykli, to algorytm tak naprawdę musi wpisywać znajdowane cykle do pomocniczej tablicy, potem wyznaczyć ich liczbę i wypisać wszystkie naraz.

Przejdźmy do analizy złożoności. Linie 3–6 oczywiście możemy wykonać w czasie $O(n+m)$. Zgodnie z poprzednimi obserwacjami, w pętli rozpoczynającej się w linii 7 każda krawędź grafu jest dokładana do ścieżki S dokładnie raz oraz usuwana dokładnie raz. Oznacza to, że cała pętla działa w czasie $O(n+m)$. Stąd wniosek, że cały algorytm działa również w czasie $O(n+m)$.

Jak usuwać krawędzie?

Wnikliwy Czytelnik na pewno zauważył, że w opisie algorytmu prześlizgnęliśmy się nad kilkoma szczegółami implementacyjnymi. Najważniejszym z nich jest usuwanie krawędzi grafu.

Naiwna implementacja wymaga przejrzenia wszystkich krawędzi znajdujących się na liście sąsiedztwa jednego z końców usuwanej krawędzi, co prowadzi do rozwiązania zadania w czasie $O(n^2)$. Takie rozwiązania dostawały ok. 60 punktów, przykładowe zaimplementowano w plikach `smis1.cpp` i `smis3.pas`. Nieco lepszym pomysłem jest przechowywanie sąsiadów każdego wierzchołka w strukturze słownikowej (dowolne zrównoważone drzewo poszukiwań binarnych, np. kontener `set` z biblioteki STL w C++). Taki algorytm działa w czasie $O((n+m)\log n)$ i pozwalał zdobyć ponad 90 punktów.

Nas jednak interesuje rozwiązanie liniowe, a więc chcielibyśmy pojedynczą operację usunięcia krawędzi wykonać w czasie stałym. Można to uczynić na kilka sposobów. Jednym z nich jest trzymanie sąsiadów każdego wierzchołka na liście dwukierunkowej (o tej strukturze danych można dowiedzieć się więcej m.in. w książce [20] lub [22]). Jeśli mamy krawędź vw, to zarówno na liście wierzchołka v jak i wierzchołka w trzymamy informację o istnieniu krawędzi vw, jako pojedynczy element listy. Dodatkowo, przy tworzeniu tej krawędzi dbamy o zapamiętanie tzw. *dowiązań krzyżowych*: przy informacji o vw na liście v trzymamy wskaźnik na odpowiadający element na liście w i vice versa. Teraz, chcąc usunąć krawędź vw, mając ją daną jako element listy wierzchołka v, nie tylko umiemy usunąć ten element z listy v, lecz także dzięki dowiązaniu krzyżowemu wiemy, który element z listy w usunąć, bez potrzeby przeglądania całej listy wierzchołka w.

Istnieje jeszcze jedno warte uwagi asymptotycznie optymalne rozwiązanie. Krawędzie wychodzące z każdego wierzchołka przechowywane są w (dynamicznie alokowanej) tablicy[1]. Element tej tablicy zawiera trzy wartości — numer wierzchołka w będącego drugim końcem krawędzi, indeks, pod którym ta krawędź jest przechowywana w tablicy odpowiadającej wierzchołkowi w, a także flaga, która informuje, czy krawędź została usunięta.

Samo usuwanie jest proste, ponieważ przechowujemy coś na kształt dowiązań krzyżowych i tylko zaznaczamy odpowiednie flagi. Jednakże gdy chcemy znaleźć kolejną krawędź, którą możemy wyjść z wierzchołka, musimy przebijać się przez potencjalnie wiele „usuniętych" krawędzi. Zauważmy jednak, że gdybyśmy za każdym razem szukali wolnej krawędzi, sprawdzając wszystkie po kolei od początku tablicy, znajdowalibyśmy krawędzie o coraz większych indeksach w tej tablicy, ponieważ raz usunięta krawędź nigdy nie jest wstawiana ponownie. Można zatem za każdym razem zaczynać w miejscu, w którym zatrzymaliśmy się poprzednio. Polecamy Czytelnikowi zastanowić się, dlaczego to już wystarcza do osiągnięcia liniowej złożoności czasowej całego rozwiązania.

Przedstawione właśnie podejście zostało zaimplementowane w pliku `smi3.cpp`. W praktyce jest nieco oszczędniejsze zarówno pod względem zużycia czasu, jak i pamięci.

Rozwiązanie alternatywne – cykl Eulera

Czytelnik, który zapoznał się bliżej z pojęciem cyklu Eulera, zapewne zauważył jego silny związek z zadaniem. Przypomnijmy, że cyklem Eulera w grafie nazywamy taki cykl, który przechodzi przez każdą krawędź dokładnie raz (zwykle nie jest to cykl prosty). Oczywiście warunkiem koniecznym na istnienie cyklu Eulera jest, aby graf był krawędziowo spójny oraz aby każdy wierzchołek miał stopień parzysty: za każdym razem, gdy cykl wchodzi do wierzchołka, musi też wyjść. Okazuje się, że jest to również warunek wystarczający. Co więcej, mając dany spójny graf o parzystych stopniach wszystkich wierzchołków, cykl Eulera można znaleźć w czasie $O(n+m)$ (można o tym przeczytać m.in. w książce [27] lub [22]).

[1] W bibliotece STL używa się do tego kontenera `vector`, gdyż dzięki niemu można łatwiej (w szczególności jednoprzebiegowo) skonstruować taką reprezentację grafu.

Rozwiązanie alternatywne wygląda zatem następująco. Po upewnieniu się, że warunek konieczny na istnienie rozwiązania jest spełniony, i zbudowaniu grafu H, w każdej spójnej składowej H znajdujemy cykl Eulera. Teraz każdy z tych cykli trzeba rozłożyć na cykle proste — kursy śmieciarki. Robimy to podobnie jak w rozwiązaniu wzorcowym. Wziąwszy jeden z tych cykli, ustalamy wierzchołek początkowy i po kolei przeglądamy wierzchołki zgodnie z kolejnością na cyklu. Gdy któryś z nich się powtórzy, odcinamy fragment pomiędzy powtórzeniami i wypisujemy jako kolejny znaleziony cykl prosty. Postępujemy tak, aż wytniemy cały cykl Eulera.

Rozwiązanie to również działa w czasie $O(n+m)$ i tak naprawdę jest w pewnym sensie równoważne rozwiązaniu wzorcowemu. Zostało zaimplementowane w plikach `smi.cpp` oraz `smi2.pas`.

Testy

Rozwiązania zostały sprawdzone na 10 grupach testów, każda z nich obejmowała jeden lub dwa testy. W testach 7*b*, 8*b*, 9*b* i 10*b* graf H jest silnie niezrównoważonym grafem dwudzielnym. Takie grafy są bowiem szczególnie trudne dla rozwiązań używających nieoptymalnych algorytmów usuwania krawędzi. Wszystkie testy, dla których nie zaznaczono inaczej, mają odpowiedź pozytywną.

Nazwa	n	m	Opis
smi1.in	8	9	mały test stworzony ręcznie
smi2.in	20	23	mały test stworzony ręcznie
smi3a.in	500	1 500	test losowy
smi3b.in	500	1 500	test losowy z odpowiedzią negatywną
smi4a.in	1 200	5 000	test losowy
smi4b.in	1 200	5 000	test losowy z odpowiedzią negatywną
smi5a.in	200	3 405	test losowy
smi5b.in	200	3 468	test losowy z odpowiedzią negatywną
smi6a.in	2 000	9 921	test losowy
smi6b.in	1 000	9 522	test losowy z odpowiedzią negatywną
smi7a.in	10 000	150 000	test losowy
smi7b.in	100 000	1 000 000	graf dwudzielny
smi8a.in	50 000	550 000	test losowy
smi8b.in	100 000	1 000 000	graf dwudzielny
smi9a.in	75 000	850 000	test losowy
smi9b.in	100 000	999 900	graf dwudzielny
smi10a.in	100 000	1 000 000	test losowy
smi10b.in	100 000	999 900	graf dwudzielny

Tomasz Waleń
Treść zadania, Opracowanie

Adam Karczmarz
Program

Dostępna pamięć: 64 MB. *OI, Etap II, dzień drugi, 10.02.2011*

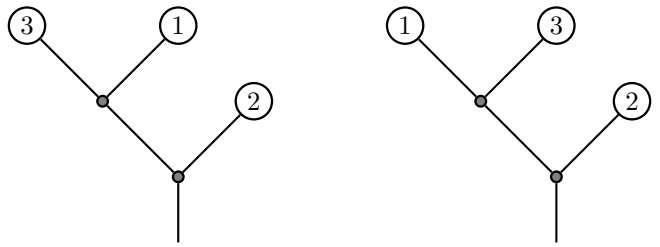
Rotacje na drzewie

Ogrodnik Bajtazar zajął się hodowlą rzadkiego drzewa o nazwie **Rotatus Informatikus**. *Ma ono bardzo ciekawe własności:*

- *Drzewo składa się z prostych gałęzi, rozgałęzień i liści. Wyrastający z ziemi pień drzewa jest również gałęzią.*
- *Każda gałąź jest zakończona u góry rozgałęzieniem lub liściem.*
- *Z rozgałęzienia na końcu gałęzi wyrastają dokładnie dwie dalsze gałęzie — lewa i prawa.*
- *Każdy liść drzewa zawiera jedną liczbę całkowitą z zakresu $1..n$. Liczby w liściach nie powtarzają się.*
- *Za pomocą pewnych zabiegów ogrodniczych, dla dowolnego rozgałęzienia można wykonać tzw.* **rotację**, *czyli zamienić miejscami lewą i prawą gałąź.*

Korona drzewa *to ciąg liczb całkowitych, który otrzymujemy, czytając liczby zawarte w liściach drzewa od lewej do prawej.*

Bajtazar pochodzi ze starego miasta Bajtogrodu i jak wszyscy jego mieszkańcy bardzo lubi porządek. Zastanawia się, jak za pomocą rotacji jak najlepiej uporządkować swoje drzewo. Uporządkowanie drzewa mierzymy liczbą **inwersji** *zawartych w jego koronie, tj. dla korony $a_1, a_2, \ldots, a_n$ wyznaczamy liczbę takich par (i, j), $1 \leqslant i < j \leqslant n$, dla których $a_i > a_j$.*



Rys. 1: Oryginalne drzewo (po lewej) o koronie $3, 1, 2$ zawiera dwie inwersje. Po rotacji otrzymujemy drzewo (po prawej) o koronie $1, 3, 2$, które zawiera tylko jedną inwersję. Każde z tych drzew ma 5 gałęzi.

Napisz program, który wyznaczy minimalną liczbę inwersji zawartych w koronie drzewa, które można otrzymać za pomocą rotacji wyjściowego drzewa Bajtazara.

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita n ($2 \leqslant n \leqslant 200\,000$), oznaczająca liczbę liści drzewa Bajtazara. W kolejnych wierszach znajduje się opis drzewa. Drzewo definiujemy rekurencyjnie:

- *jeśli na końcu pnia (czyli gałęzi, z której wyrasta drzewo) znajduje się liść z liczbą całkowitą p ($1 \leqslant p \leqslant n$), to opis drzewa składa się z jednego wiersza zawierającego jedną liczbę całkowitą p,*
- *jeśli na końcu pnia znajduje się rozgałęzienie, to opis drzewa składa się z trzech części:*
  - *pierwszy wiersz opisu zawiera jedną liczbę 0,*
  - *po tym następuje opis lewego poddrzewa (tak, jakby lewa gałąź wyrastająca z rozgałęzienia była jego pniem),*
  - *a następnie opis prawego poddrzewa (tak, jakby prawa gałąź wyrastająca z rozgałęzienia była jego pniem).*

W testach wartych przynajmniej 30% punktów zachodzi dodatkowy warunek $n \leqslant 5\,000$.

Wyjście

W jedynym wierszu standardowego wyjścia należy wypisać jedną liczbę całkowitą: minimalną liczbę inwersji w koronie drzewa, które można otrzymać za pomocą jakiegoś ciągu rotacji wyjściowego drzewa.

Przykład

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

Wyjaśnienie do przykładu: *Rysunek 1 ilustruje drzewo z przykładu.*

Rozwiązanie

Zliczanie inwersji w permutacjach

Zadanie ma bliski związek z bardzo znanym problemem: zliczaniem inwersji w permutacjach. Jak wskazuje nazwa, polega on na zliczeniu, dla danej permutacji p liczb od 1 do n, wszystkich par (i, j), takich że $1 \leqslant i < j \leqslant n$ oraz $p(i) > p(j)$.

Takie klasyczne zadanie ma wiele ciekawych rozwiązań, w szczególności algorytm korzystający z techniki „dziel i zwyciężaj”. Jak możemy podejść do problemu tym sposobem? Zapiszmy permutację w postaci ciągu i podzielmy go na dwa spójne podciągi możliwie równej długości. Wówczas inwersje będzie można podzielić na trzy grupy: leżące w lewym podciągu, leżące w prawym podciągu oraz te inwersje (i, j), dla których i jest indeksem w lewym, a j w prawym podciągu. Pierwsze dwa przypadki

chcielibyśmy rozważyć przez wywołania rekurencyjne konstruowanej funkcji. Aby było to możliwe, musimy jednak nieco uogólnić nasz problem — będziemy zliczać inwersje w dowolnym różnowartościowym ciągu liczb naturalnych. Szczęśliwie nie wpływa to negatywnie na dotychczas przeprowadzone rozumowanie.

Pozostaje nam zliczenie inwersji trzeciego rodzaju. Przydatne będzie do tego proste spostrzeżenie. Otóż zbiór takich inwersji zależy tylko od tego, jaki jest zbiór wartości w lewym, a jaki w prawym podciągu, nie zależy zaś od uporządkowania tych wartości. Co więcej, jeśli obydwa ciągi byłyby uporządkowane rosnąco, zliczenie tych inwersji byłoby łatwe: można byłoby przystosować do tego zadania algorytm scalania dwóch posortowanych ciągów. Dzięki temu, niejako przy okazji, zapewniamy sobie również, że wywołania rekurencyjne posortują lewy i prawy podciąg. Pożądane przez nas założenie o uporządkowaniu obydwu złączanych ciągów będzie więc spełnione.

Otrzymaliśmy tym samym algorytm, który tak naprawdę jest nieskomplikowaną modyfikacją algorytmu sortowania przez scalanie (*MergeSort*). W szczególności, działa w czasie $O(n \log n)$ i pamięci $O(n)$.

Analiza właściwego problemu

Wróćmy jednak do naszego wyjściowego zadania. W jaki sposób możemy wyliczyć minimalną liczbę inwersji dla drzewa? Na początku bardzo nieznacznie zmodyfikujmy definicję inwersji w tym przypadku. Niech nie będzie to para (i, j) indeksów w koronie, lecz para etykiet (x, y), taka że $x > y$ i liść o etykiecie x jest w koronie wcześniej niż ten o etykiecie y.

Spróbujmy podejść do problemu podobnie jak poprzednio. Tym razem nie będziemy jednak dzielić na pół, lecz z skorzystamy z naturalnego podziału zadanego przez strukturę drzewa — na prawe i lewe poddrzewa. Inwersje dzielą się więc na trzy grupy: te w lewym poddrzewie, te w prawym poddrzewie i te, których pierwszy liść leży w lewym poddrzewie, a drugi w prawym. Spoglądając na taki algorytm zliczania inwersji z globalnej perspektywy, dostrzegamy, że inwersję (x, y) liczymy wtedy, gdy znajdujemy się w najniższym wspólnym przodku liści o tych etykietach.

Wprowadźmy wobec tego użyteczną notację. Dla każdego wierzchołka v wejściowego drzewa przez inv_v oznaczmy liczbę inwersji (x, y) takich, że x jest etykietą liścia w lewym poddrzewie v, a y w prawym poddrzewie v. Oczywiście dla v będących liśćmi drzewa zachodzi $inv_v = 0$, a liczba inwersji w całym drzewie to suma wartości inv_v po wszystkich węzłach v.

W naszym zadaniu nie mamy jednak po prostu policzyć inwersji, lecz zminimalizować ich liczbę, mając do dyspozycji rotacje. Zauważmy jednak, że pojedyncza rotacja w wierzchołku v wpływa jedynie na wartość inv_v, a wszystkie wartości inv_u dla pozostałych wierzchołków ($u \neq v$) pozostają bez zmian. Stąd każdy wierzchołek możemy traktować oddzielnie. Aby w pełni opisać wpływ rotacji na liczbę inwersji, pozostaje stwierdzić, jak zmienia się inv_v przy rotacji w węźle v. Rozważamy oczywiście pary liści, których najniższym wspólnym przodkiem jest v. Nietrudno dostrzec, że wśród nich inwersjami stają się wówczas dokładnie te pary, które nie tworzyły wcześniej inwersji. Wobec tego, jeśli lewe poddrzewo v ma l_v, a prawe r_v liści, to po wykonaniu rotacji liczba inwersji w v jest równa $l_v \cdot r_v - inv_v$.

Te obserwacje pozwalają nam spostrzec, że stosując zachłanną strategię rotacji w każdym z węzłów, otrzymamy optymalne rozwiązanie (dla każdego wierzchołka wykonujemy rotację wtedy, gdy powoduje to zmniejszenie wartości inv_v). Podsumowując, minimalną liczbę inwersji drzewa możemy obliczyć za pomocą wzoru:

$$\sum_{v \in T} \min(inv_v,\, l_v \cdot r_v - inv_v).$$

Niestety nie wszystkie składniki powyższej sumy są łatwe do obliczenia. O ile wartości l_v oraz r_v można wyznaczyć prostym obejściem drzewa, o tyle efektywne wyznaczenie wartości inv_v jest bardziej skomplikowane.

Wyznaczanie liczby inwersji w węzłach

Pozostaje nam teraz stworzyć możliwie efektywny algorytm, który wyznaczy wartości inv_v dla wszystkich wierzchołków drzewa. Zauważmy, że przytoczony na początku opisu algorytm wyznaczający liczbę inwersji w ciągu tak naprawdę wyznacza liczbę inwersji w węzłach pewnego zrównoważonego drzewa binarnego zbudowanego nad tym ciągiem. Stanowi tym samym dobry punkt wyjściowy do naszego algorytmu.

Zdefiniujemy rekurencyjną funkcję *MergeTree*. Funkcja ta dla zadanego wierzchołka v obliczy wartości inv_v w całym poddrzewie, a także zwróci strukturę danych reprezentującą wartości wszystkich liści poddrzewa. Poniżej przedstawiamy szkielet takiego rozwiązania:

```
1:  function MergeTree(v)
2:  begin
3:    niech l oznacza lewego syna v, a r prawego syna v;
4:    Val_l := MergeTree(l);
5:    Val_r := MergeTree(r);
6:    { zgodnie z wcześniejszą definicją inv_v oznacza liczbę par (x, y) }
7:    { takich, że x > y i x ∈ Val_l, y ∈ Val_r }
8:    (inv_v, Val_v) := MergeValues(Val_l, Val_r);
9:    wynik := wynik + min(inv_v, l_v · r_v − inv_v);
10:   return Val_v;
11: end
```

Nadal jednak nie jest to kompletny opis, brakuje realizacji funkcji *MergeValues*.

Proste scalanie

Spróbujmy zastosować podejście, które sprawdziło się przy zliczaniu inwersji w ciągu, czyli do reprezentacji wartości liści użyjmy posortowanych ciągów (przechowywanych w listach lub tablicach dynamicznych). Funkcję *MergeValues* możemy wówczas wykonać w czasie $O(l_v + r_v)$. Niestety prowadzi to do algorytmu, który jest pesymistycznie kwadratowy. Przykładowo, na drzewie o głębokości $\Theta(n)$, w którym każdy lewy syn

jest liściem, algorytm działa w czasie $\Theta(n^2)$. Przedstawione rozwiązanie zostało zaimplementowane w plikach `rots1.cpp` oraz `rots2.pas`. Programy te otrzymywały tylko ok. 25 punktów, ponieważ wykorzystywały listy, które działają dość wolno.

Czas kwadratowy jest jednak o wiele łatwiejszy do osiągnięcia. Zauważmy, że gdyby nasza funkcja *MergeValues* działała w czasie $O(l_v \cdot r_v)$ i siłowo sprawdzała każdą parę liści o najniższym wspólnym przodku w v, wciąż pesymistyczny czas działania pozostałby kwadratowy. Każdą parę liści w całym drzewie sprawdzalibyśmy bowiem dokładnie raz. Takie rozwiązanie można znaleźć w plikach `rots3.cpp` i `rots4.pas`. Ze względu na prostotę sprawowało się nieco lepiej i dostawało ok. 30 punktów.

Efektywne scalanie

Skoncentrujmy się zatem na stworzeniu struktury danych, która pozwoli na efektywniejszą realizację funkcji *MergeValues*. Zacznijmy od opisu cech, które musi posiadać taka struktura danych.

Chcemy przechowywać zbiory kluczy o wartościach ze zbioru $\{1, \ldots, n\}$. Dodatkowo, struktura powinna umożliwiać operację *MergeValues*, która dla danych struktur A i B zwróci nową strukturę zawierającą wszystkie klucze z A i B. Pewnym ułatwieniem może być założenie, że zbiory kluczy z A i B są rozłączne.

Pierwszym rozwiązaniem problemu może być użycie drzew zrównoważonych (np. drzew czerwono-czarnych czy AVL, opisanych np. w książce [22] oraz na stronie `http://wazniak.mimuw.edu.pl`). Drzewa te umożliwiają wykonywanie w czasie $O(\log n)$ m.in. następujących operacji:

- wstawienie nowego elementu do struktury,
- wyznaczenie liczby kluczy w strukturze o wartościach większych niż x.

Możemy też zaimplementować operację scalania drzew A i B, tak by wymagała czasu $O(\min(|A|, |B|) \cdot \log(|A|+|B|))$. Nie jest to trudne — wystarczy, że wstawimy wszystkie klucze z mniejszego drzewa do większego i zwrócimy tak zmodyfikowane drzewo. W takim samym czasie możemy również zliczać liczbę inwersji za pomocą drugiej spośród operacji dostarczanych przez drzewo.

Zastanówmy się chwilę, jak zastosowanie takiej struktury wpłynie na całkowitą złożoność rozwiązania. Operacją dominującą jest wstawienie klucza do struktury. Możemy jednak zauważyć, że w trakcie działania całego algorytmu każdy element może być wstawiany co najwyżej $\log n$ razy. Faktycznie, jeśli wstawiamy jakiś element, to łączymy jego strukturę ze strukturą co najmniej tego samego rozmiaru, czyli po każdym kolejnym wstawieniu tego konkretnego elementu, jego struktura jest przynajmniej dwukrotnie większa. Ponieważ koszt wstawienia elementu szacuje się przez $O(\log n)$, więc całkowity koszt algorytmu wynosi $O(n \log^2 n)$. Złożoność pamięciowa jest jednak bardzo dobra — liniowa.

Niestety, struktura słownikowa dostępna w bibliotece STL języka C++ nie udostępnia operacji zapytania o liczbę kluczy o wartościach mniejszych/większych niż x, co powoduje, że jesteśmy skazani na własną implementację. Jest to niestety skomplikowane i pracochłonne zadanie jak na pięciogodzinną sesję. Jego rozwiązanie pozwalało

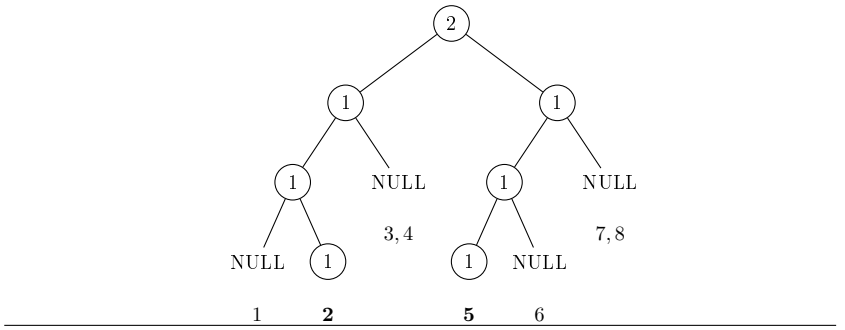
jednak cieszyć się maksymalną punktacją. Przykładowa implementacja znajduje się w plikach `rot2.cpp` oraz `rot3.pas`.

Prostsza struktura danych – drzewo przedziałowe

Spróbujmy wobec tego użyć struktury znacznie łatwiejszej w implementacji, której wykorzystanie często pozwala uchronić się przed implementacją drzew zrównoważonych. Mowa o drzewie przedziałowym[1]. Tym razem nie będzie to zwyczajne statyczne drzewo przedziałowe implementowane standardowo w tablicy, lecz dynamicznie alokowane, reprezentowane jako struktura wskaźnikowa. Jest to w zasadzie zwykłe drzewo przedziałowe, w którym nie przechowujemy poddrzew zawierających 0 kluczy (a dokładniej, całe takie poddrzewa są reprezentowane przez wartość NULL). Każdy węzeł drzewa utrzymuje trzy atrybuty:

- *count* — liczba kluczy w poddrzewie,
- *left* — wskaźnik do lewego syna (lub NULL),
- *right* — wskaźnik do prawego syna (lub NULL).

Przykładowe drzewo tego typu można zobaczyć na rysunku 1. Dzięki takiemu oszczędnemu gospodarowaniu pamięcią reprezentacja k kluczy z przedziału $\{1, \ldots, n\}$ wymaga jedynie pamięci rzędu $O(k \log n)$. W szczególności, struktura zawierająca 1 klucz zajmuje pamięć rzędu $\Theta(\log n)$.



Rys. 1: Przykładowe drzewo przedziałowe dla zakresu $\{1, \ldots, 8\}$ zawierające klucze $\{2, 5\}$. Wewnątrz węzłów podane są wartości atrybutów *count*.

Zauważmy, że skonstruowana przez nas struktura potrafi wykonać dokładnie te operacje, o których pisaliśmy w poprzedniej sekcji, w dokładnie takim samym czasie jak struktury słownikowe. W czym, skoro nie w ograniczeniu funkcjonalności, tkwi wobec tego sekret jej prostoty? Musimy z góry znać zbiór kluczy, jakie pojawiają się w całym algorytmie, i jeśli rozmiar tego zbioru to N, czas operacji na strukturze A jest dla drzew przedziałowych rzędu $O(\log N)$, zaś np. dla drzew AVL rzędu $O(\log |A|)$.

[1] Więcej o drzewach przedziałowych można dowiedzieć się np. w opracowaniu zadania *Tetris 3D* z XIII Olimpiady Informatycznej [13] lub na stronie `http://was.zaa.mimuw.edu.pl/?q=node/8`. Drugie ze źródeł porusza również tematykę dynamicznie alokowanych drzew przedziałowych.

W naszym przypadku nie odgrywa to jednak roli przy oszacowaniu złożoności czasowej — pozostaje równa $O(n \log^2 n)$. Poświęćmy jeszcze chwilę złożoności pamięciowej: jeśli przy łączeniu drzew od razu zwolnimy pamięć po niepotrzebnych węzłach, żaden klucz nie będzie nigdy w dwóch różnych drzewach. Stąd łącznie będziemy potrzebować $O(n \log n)$ węzłów. Pozwala to, co prawda z niewielkim zapasem, zmieścić się w limicie pamięci. Większa złożoność pamięciowa to jednak wyraźna wada w porównaniu ze strukturami słownikowymi.

Efektywniejsze scalanie drzew przedziałowych

Poprzednio scalaliśmy dwa drzewa przedziałowe przez wstawianie elementów mniejszego do większego. Spróbujmy zastosować nieco inne podejście, którego idea jest bardzo naturalna dla scalania statycznych drzew przedziałowych (cały czas pracujemy nad ustalonym uniwersum kluczy). W takich drzewach każdy element tablicy reprezentującej drzewo odpowiada zawsze za określony przedział wartości uniwersum. Zatem wartość *count* w węźle drzewa po scaleniu to po prostu suma wartości *count* w odpowiednich węzłach scalanych drzew.

W drzewach dynamicznie alokowanych zachodzi ta sama zależność — trzeba tylko wziąć pod uwagę, że część poddrzew jest zastąpionych przez NULL. Innymi słowy, jeśli kształt ścieżki z korzenia do węzła dla pewnych dwóch węzłów różnych drzew jest taki sam, to węzły te odpowiadają za ten sam przedział. Na przykład, lewy syn prawego syna korzenia odpowiada za trzecią ćwiartkę uniwersum.

Oparty na tych spostrzeżeniach algorytm scalania wygodnie zapisać rekurencyjnie. Dla danych dwóch drzew przedziałowych A i B tworzymy nowy węzeł, którego lewe poddrzewo jest efektem scalenia lewych poddrzew A, B, zaś prawe — prawych. Musimy jeszcze ustawić atrybut *count* w nowym węźle, który powinien przyjąć wartość $count(A) + count(B)$. Po wykonaniu scalenia węzły reprezentujące korzenie A i B nie są już dalej potrzebne, więc możemy zwolnić zajmowaną przez nie pamięć[2].

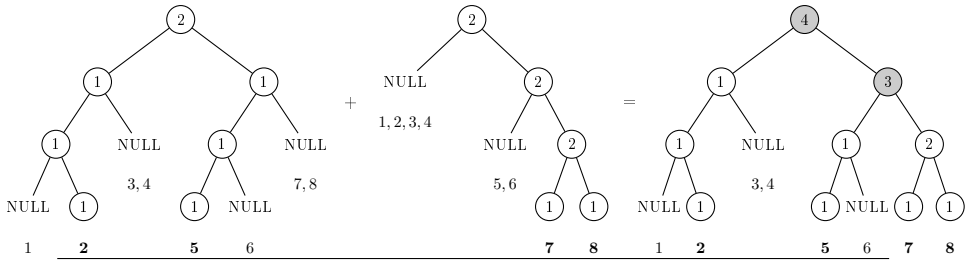
Na razie nie widać, czemu nasze rozwiązanie miałoby być choćby równie efektywne jak poprzednie. Jednak nie opisaliśmy jeszcze scalania poddrzew, z których przynajmniej jedno jest puste (reprezentowane przez NULL). Jest to oczywiście bardzo proste — wynikiem jest drugie ze scalanych poddrzew. Aby podać ten wynik, nie potrzeba schodzić w głąb tego poddrzewa, wystarczy więc do tego czas stały.

Funkcja *MergeValues* powinna również zwracać liczbę inwersji pomiędzy strukturami. Możemy to osiągnąć, sumując (dla wszystkich wywołań rekurencyjnych) iloczyny liczby kluczy w prawym poddrzewie drzewa A i lewym poddrzewie drzewa B. Jeśli jedno z poddrzew jest puste, cały iloczyn to oczywiście zero. Również aby podać tę wartość, nie musimy zagłębiać się w drugie drzewo.

Poniżej przedstawiamy pseudokod opisanego rozwiązania, natomiast na rysunku 2 zobrazowany jest efekt działania funkcji.

[2]W praktyce warto nie tworzyć nowego węzła, lecz wykorzystać ponownie jeden z usuwanych. Dzięki temu w całym algorytmie nowe węzły będziemy tworzyli tylko podczas budowania drzew reprezentujących jednoelementowy zbiór kluczy. To zaś pozwoli zaalokować na początku działania odpowiednią ich liczbę. Takie podejście znacząco przyspiesza program.

```
1:  function MergeValues(A,B)
2:  begin
3:    if A = NULL then return (0, B);
4:    if B = NULL then return (0, A);
5:    inv := count(A.right) · count(B.left);
6:    (inv_l, l) := MergeValues(A.left, B.left);
7:    (inv_r, r) := MergeValues(A.right, B.right);
8:    t := nowy węzeł drzewa przedziałowego;
9:    t.count := count(A) + count(B);
10:   t.left := l;
11:   t.right := r;
12:   zwolnij pamięć odpowiadającą węzłom A i B;
13:   return (inv_l + inv_r + inv, t);
14: end
```



Rys. 2: Przykład działania funkcji *MergeValues* dla drzew zawierających klucze $A = \{2, 5\}$ oraz $B = \{7, 8\}$. Wynikiem jest drzewo zawierające klucze $C = \{2, 5, 7, 8\}$. Kolorem szarym zostały oznaczone nowe węzły utworzone w trakcie scalania.

Zastanówmy się teraz, jaka jest złożoność powyższego rozwiązania. Nietrudno dostrzec, że złożoność pojedynczego wykonania funkcji *MergeValues* szacuje się przez liczbę węzłów mniejszego z drzew, tzn. $O(\min(|A|, |B|) \cdot \log n)$, więc złożoność całości nie jest gorsza niż poprzednio. Można jednak pokazać, że jest istotnie lepsza. Tutaj przychodzi z pomocą analiza kosztu zamortyzowanego z użyciem funkcji potencjału (więcej informacji na temat tej techniki dowodzenia można odnaleźć w książce [22]). W naszym przypadku jako funkcję potencjału możemy wybrać liczbę węzłów drzew przedziałowych. Początkowo mamy n drzew przedziałowych, z których każde zawiera dokładnie jeden klucz (każde z takich drzew odpowiada liściowi z drzewa danego w treści zadania), stąd początkowy potencjał wynosi $\Theta(n \log n)$.

Musimy teraz wykazać, że żadne wywołanie funkcji *MergeValues* nigdy nie zwiększy potencjału, a jeśli po jej wykonaniu potencjał zmniejszy się o wartość Δ, to wykonana praca nie będzie większa niż $c \cdot (\Delta + 1)$ dla pewnej stałej $c > 0$.

Rozważmy dwa możliwe przypadki:

- jeden z argumentów A lub B przyjmuje wartość NULL, w tym przypadku zmiana potencjału Δ wynosi 0, jednak tutaj wykonujemy stałą pracę; wystarczy więc dobrać odpowiednio dużą stałą c, tak aby zachodziło żądane ograniczenie na wykonaną pracę;

- w przypadku, gdy oba drzewa są niepuste, wykonujemy rekurencyjnie pracę co najwyżej $c \cdot (\Delta_l + 1)$ (dla lewych poddrzew) oraz $c \cdot (\Delta_r + 1)$ (dla prawych poddrzew), przy czym $\Delta_l + \Delta_r = \Delta - 1$ (usuwamy dwa węzły A i B, a tworzymy nowy węzeł t). Stąd także w tym przypadku wykonana praca nie przekracza $c \cdot (\Delta + 1)$.

Podsumowując, całkowity czas poświęcony wywołaniom *MergeValues* nie przekracza początkowej wartości potencjału, czyli $O(n \log n)$. Taka jest również i złożoność pamięciowa tego rozwiązania. Opisany algorytm zaimplementowano w plikach `rot5.cpp` oraz `rot7.cpp`. Oba programy dostają 100 punktów. Drugi jest jednak blisko trzykrotnie szybszy dzięki lepszemu zarządzaniu alokacją i zwalnianiem pamięci.

A może jeszcze coś da się poprawić?

Co ciekawe, to nie jest jeszcze koniec optymalizacji, które możemy uzyskać w tym zadaniu. Można bowiem zredukować złożoność pamięciową do $O(n)$, przy zachowaniu złożoności czasowej $O(n \log n)$. Takie rozwiązanie nie było oczywiście wymagane, ale stanowi wartą wzmianki ciekawostkę.

Jeden z przykładów takiego podejścia opiera się na nieznacznie bardziej skomplikowanej implementacji drzew przedziałowych. Wystarczy bowiem zredukować pamięć potrzebną do przechowywania k kluczy z $O(k \log n)$ do $O(k)$. Ten cel możemy zrealizować przez kompresję do jednej krawędzi ścieżek utworzonych z węzłów o dokładnie jednym synu. Ta sama technika jest wykorzystywana np. w drzewach sufiksowych.

Oczywiście, trudności implementacyjne przy scalaniu struktur są o wiele większe, program nie jest jednak bardzo długi. Aby dało się odtworzyć skompresowaną ścieżkę, w węźle trzeba dodatkowo pamiętać np. indeks, jaki otrzymałby w statycznej implementacji drzewa. To podejście, dzięki zastosowaniu niestandardowych operacji na bitach, pozwala nawet na łączenie drzew bez rozwijania ścieżek, co nie jest wymagane do osiągnięcia pożądanej złożoności czasowej i pamięciowej, ale w praktyce wyraźnie przyspiesza program. Opisane rozwiązanie zostało zaimplementowane w pliku `rot6.cpp`.

Testy

Zadanie było oceniane przy użyciu 11 zestawów danych testowych. Specyfika zadania wymagała zastosowania dosyć agresywnego grupowania testów.

Nazwa	n	Opis
rot1a.in	20	mały test poprawnościowy
rot1b.in	19	małe drzewo słabo zrównoważone
rot1c.in	20	drzewa o zbliżonych bądź skrajnie różnych wielkościach poddrzew (w każdym miejscu)
rot1d.in	20	zrównoważone drzewo o względnie dużej liczbie koniecznych inwersji

Nazwa	n	Opis
rot1e.in	2	minimalny test, odpowiedź 0
rot2a.in	101	drzewo zrównoważone u góry, u dołu zakończone krótkimi „ścieżkami”
rot2b.in	110	ścieżka z przyłączonymi ścieżkami
rot2c.in	126	losowe drzewo zrównoważone
rot2d.in	115	ścieżka z doczepionymi parami liści
rot3a.in	756	drzewo losowe, słabo zrównoważone
rot3b.in	879	drzewo jeszcze słabiej zrównoważone, z doczepionymi krótkimi ścieżkami
rot3c.in	856	drzewo dobrze zrównoważone
rot3d.in	832	rozgałęziona ścieżka, w korzeniu połączona z losowym drzewem
rot4a.in	3 885	pojedyncza ścieżka, która rozgałęzia się na mniejsze losowe drzewa
rot4b.in	4 159	drzewo słabo zrównoważone, ze sporą liczbą krótkich ścieżek
rot4c.in	4 387	dwie różnie skomplikowane ścieżki, połączone w korzeniu
rot4d.in	4 291	długa ścieżka z kilkoma małymi drzewkami
rot5a.in	11 009	losowa ścieżka w dół, test na poprawność
rot5b.in	12 474	losowe drzewo słabo zrównoważone z odrostami
rot5c.in	30 129	drzewo bardzo dobrze zrównoważone
rot5d.in	13 045	skomplikowane drzewo, zawierające ścieżki zarówno losowe, jak i posortowane
rot6a.in	50 768	drzewo z dużą liczbą odstających ścieżek
rot6b.in	47 809	drzewo rozrasta się ostatecznie w 4 długie ścieżki
rot6c.in	53 002	duży test poprawnościowy, drzewo losowe z długą ścieżką na końcu
rot6d.in	57 009	duży test przeciwko drzewom BST bez równoważenia
rot7a.in	80 987	ścieżka rozgałęzia się na kilkadziesiąt innych ścieżek
rot7b.in	83 998	korzeń rozdziela się na posortowaną ścieżkę z nieposortowanymi odrostami i na pomieszaną ścieżkę z posortowanymi odrostami
rot7c.in	82 345	ścieżka z odrastającymi sporymi losowymi drzewami
rot7d.in	87 509	test przeciwko słabym BST, kilkanaście odrastających posortowanych ścieżek
rot8a.in	120 859	duże drzewo ze sporą liczbą dosyć krótkich odrastających ścieżek

Nazwa	n	Opis
rot8b.in	117 098	także duże drzewo, lecz teraz odrasta od niego więcej mniejszych ścieżek
rot8c.in	119 877	dwie ścieżki z odrastającymi małymi drzewkami, połączone w korzeniu
rot8d.in	118 567	test przeciwko słabym BST, posortowane odrosty
rot9a.in	172 098	skomplikowane drzewo o dużej minimalnej liczbie inwersji
rot9b.in	173 790	proste drzewo losowe o małej minimalnej liczbie inwersji
rot9c.in	169 123	skomplikowany test poprawnościowy
rot9d.in	174 072	długie posortowane odrosty, przeciwko słabym BST
rot9e.in	170 000	drzewo zrównoważone
rot10a.in	189 680	10 długich ścieżek, niektóre posortowane, inne losowe
rot10b.in	191 998	większy wariant testu *7b*
rot10c.in	190 067	większy wariant testu *7c*
rot10d.in	190 800	test przeciwko słabym BST, podobny do *6d*, lecz dużo większy
rot10e.in	193 983	drzewo zrównoważone
rot11a.in	199 998	na początku drzewo „dzieli się” po równo, ostatecznie jednak rozdziela się na 16 długich ścieżek
rot11b.in	200 000	test poprawnościowy, większy wariant testu *8b*
rot11c.in	200 000	duży test na drzewo zrównoważone
rot11d.in	200 000	skomplikowany test podobny do *11a*, lecz na dole więcej możliwych różnych scenariuszy
rot11e.in	200 000	prosta, losowa ścieżka maksymalnej długości
rot11f.in	200 000	drzewo doskonale zrównoważone

Jacek Tomasiewicz
Treść zadania, Opracowanie

Bartłomiej Wołowiec
Program

Dostępna pamięć: 32 MB.

OI, Etap II, dzień drugi, 10.02.2011

Temperatura

W Bajtockim Instytucie Meteorologicznym (BIM) codziennie mierzy się temperaturę powietrza. Pomiaru dokonuje się automatycznie, po czym jest on drukowany. Niestety, tusz w drukarce dawno wysechł... Pracownicy BIM przekonali się o tym jednak dopiero, gdy Bajtocka Organizacja Meteorologiczna (BOM) zwróciła się z zapytaniem dotyczącym temperatury.

Na szczęście stażysta Bajtazar systematycznie notował temperatury wskazywane przez dwa zwykłe termometry okienne, znajdujące się na północnej i południowej ścianie budynku BIM. Wiadomo, że temperatura wskazywana przez termometr na południowej ścianie budynku nigdy nie jest niższa niż faktyczna, a wskazywana przez termometr na północnej ścianie budynku nigdy nie jest wyższa niż faktyczna. Nie wiadomo więc dokładnie, jaka danego dnia była temperatura, ale wiadomo, w jakim mieściła się przedziale.

Na szczęście, zapytanie BOM nie dotyczy dokładnych temperatur, a jedynie najdłuższego przedziału czasu, w którym temperatura była niemalejąca (tj. każdego kolejnego dnia była taka sama jak poprzedniego lub wyższa). Szef BIM wie, że BOM zależy na tym, aby znaleźć możliwie najdłuższy taki okres czasu. Postanowił więc zatuszować swoją wpadkę i polecił Bajtazarowi znaleźć, na podstawie jego notatek, najdłuższy taki przedział czasu, w którym temperatura **mogła być** *niemalejąca. Bajtazar nie bardzo wie, jak sobie poradzić z tym zadaniem, więc poprosił Cię o napisanie programu, który znajdzie taki najdłuższy przedział czasu.*

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita n ($1 \leqslant n \leqslant 1\,000\,000$) oznaczająca liczbę dni, przez które Bajtazar notował temperatury. Wyniki pomiarów z dnia i są zapisane w wierszu o numerze $i + 1$. Każdy z takich wierszy zawiera dwie liczby całkowite x i y ($-10^9 \leqslant x \leqslant y \leqslant 10^9$). Oznaczają one, odpowiednio, minimalną i maksymalną temperaturę, jaka mogła być danego dnia.

W pewnej liczbie testów, wartych łącznie 50 punktów, temperatury nigdy nie spadają poniżej -50 stopni i nigdy nie wzrastają powyżej 50 stopni ($-50 \leqslant x \leqslant y \leqslant 50$).

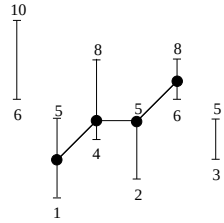
Wyjście

W pierwszym i jedynym wierszu standardowego wyjścia Twój program powinien wypisać jedną liczbę całkowitą, oznaczającą maksymalną liczbę dni, przez które temperatura w Bajtocji mogła być niemalejąca.

Przykład

Dla danych wejściowych:

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



poprawnym wynikiem jest:

```
4
```

Rozwiązanie

Kiedy przedział jest dobry?

Zanim przejdziemy do rozwiązania zadania, spróbujmy zastanowić się nad nieco prostszym problemem. Jak sprawdzić, czy w całym okresie czasu, od pierwszego do ostatniego dnia, temperatura mogła być niemalejąca?

Do odpowiedzi na to pytanie posłuży nam łatwa obserwacja. Załóżmy, że mamy niemalejący ciąg temperatur $(t_i)_{i=1}^n$, spełniających zadane ograniczenia $x_i \leqslant t_i \leqslant y_i$. Wówczas możemy zmniejszyć t_i, tak aby zachodziła równość $t_i = \max(x_i, t_{i-1})$. Po wykonaniu tej operacji dla wszystkich wyrazów otrzymamy ciąg, który wciąż spełnia wszystkie żądane warunki.

Teraz wiemy, że aby sprawdzić, czy istnieje jakikolwiek dobry ciąg temperatur, wystarczy sprawdzić, czy dobry jest ciąg dany wzorem

$$t_1 = x_1, \qquad t_i = \max(x_i, t_{i-1}) \text{ dla } i > 1.$$

Można to więc bardzo łatwo stwierdzić w czasie $O(n)$.

W ten sposób uzyskujemy pierwsze rozwiązanie właściwego zadania. Wystarczy dla każdego przedziału sprawdzić, czy jest dobry, i zapamiętać najdłuższy przedział, dla którego odpowiedź okazała się pozytywna. Oto pseudokod ilustrujący taki algorytm:

```
1:  wynik := 0;
2:  for j := 1 to n do
3:    for k := j to n do begin
4:      dobry := true;
5:      temp := x[j];
6:      for i := j + 1 to k do
7:        if y[i] ⩾ temp then
8:          temp := max(temp, x[i])
9:        else
10:         dobry := false;
```

```
11:       if dobry then
12:         wynik := max(wynik, k − j + 1);
13:     end
14:   return wynik;
```

Złożoność czasowa tego rozwiązania to $O(n^3)$, ponieważ wszystkich przedziałów czasu jest $O(n^2)$, a ich łączna długość to $O(n^3)$. Można było za nie uzyskać około 16 punktów.

Nieco szybsze rozwiązanie

Złożoność sześcienna jest bardzo daleka od naszych oczekiwań. Zauważmy jednak, że przedstawiony algorytm dla kolejnych fragmentów o tym samym początku wykonuje dokładnie te same obliczenia. Jeśli rozpoczynamy konstrukcję ciągu w j-tym dniu i w pewnym momencie natrafiamy na dzień, dla którego nie możemy wybrać temperatury, to znamy najdłuższy dobry przedział czasu o początku w j-tym dniu. Możemy więc, dla każdego dnia z osobna (uznając, że jest początkiem przedziału), zachłannie sprawdzać, jak długo temperatura mogła być niemalejąca.

```
1:  wynik := 1;
2:  for j := 1 to n do begin
3:    temp := x[j];
4:    i := j + 1;
5:    while i ⩽ n and y[i] ⩾ temp do begin
6:      temp := max(temp, x[i]);
7:      i := i + 1;
8:    end
9:    wynik := max(wynik, i − j);
10: end
11: return wynik;
```

Powyższy algorytm działa w czasie $O(n^2)$, gdyż dla każdego początku przedziału sprawdzamy $O(n)$ kolejnych dni. Za takie rozwiązanie można było uzyskać około 25 punktów. Przykładowa implementacja znajduje się w plikach `tems1.cpp` oraz `tems5.pas`.

Ciekawe rozwiązanie nieoptymalne

Okazuje się, że poprzedni algorytm można jeszcze przyspieszyć. To przyspieszenie samo w sobie jest interesujące, natomiast nie jest ono potrzebne do uzyskania rozwiązania wzorcowego.

Otóż w algorytmie kwadratowym dla każdego j wyznaczamy największe takie i, że przedział $[j, i]$ jest dobry. Zauważmy, że gdy obliczamy tę wartość dla różnych j, otrzymany wynik zależy tylko od bieżących wartości zmiennych i oraz *temp*. W szczególności, gdy w linii 6 zachodzi $temp \leqslant x[i]$ i *de facto* dokonujemy przypisania $temp := x[i]$,

możemy z góry stwierdzić, że najdłuższy przedział zaczynający się w j kończy się dokładnie tam, gdzie najdłuższy przedział zaczynający się w i. Kolejne obroty pętli będą bowiem w obu przypadkach dokładnie takie same.

Możemy wobec tego wyznaczać wyniki dla j w kolejności malejącej, spamiętując je i wykorzystując do dalszych obliczeń. Przy okazji pozbywamy się zmiennej *temp*, bo i tak nie zmienialibyśmy jej wartości. Innymi słowy, kiedy w poprzednim rozwiązaniu wartość *temp* wzrasta do wartości $x[i]$ (lub $x[i] = temp$), w poniższym rozwiązaniu wykorzystujemy spamiętaną wartość $najdl[i]$.

```
1:  wynik := 1;
2:  for j := n downto 1 do begin
3:    i := j + 1;
4:    while i ⩽ n and y[i] ⩾ x[j] do begin
5:      if x[j] ⩽ x[i] then begin
6:        i := najdl[i] + 1;
7:        break;
8:      end else i := i + 1;
9:    end
10:   najdl[j] := i − 1;
11:   wynik := max(wynik, i − j);
12: end
13: return wynik;
```

Zastanówmy się, jaka jest złożoność takiego algorytmu. Na pewno daje się oszacować przez $O(n^2)$. Spróbujmy jednak uzależnić ją również od Δ — liczby różnych wartości w tablicy x. Oczywiście cały program poza pętlą w liniach 4–9 wykonuje się w czasie liniowym. Wystarczy więc oszacować łączną liczbę obrotów tej pętli.

Ustalmy pewną konkretną wartość M. Niech $j_1 < j_2 < \ldots < j_m$ będą wszystkimi indeksami, dla których wartość tablicy x wynosi M. Zauważmy, że gdy dla $j = j_k$ w pętli dojdziemy do $i = j_{k+1}$, natychmiast wyjdziemy z pętli. Stąd obliczenie wyników dla $j_1, \ldots, j_m$ zajmuje czas $O(n)$, a złożoność czasowa całego algorytmu to $O(n\Delta)$.

Oczywiście $\Delta \leqslant \max_i x[i] - \min_i x[i] + 1$. Dzięki temu taki algorytm przechodzi testy z ograniczonymi wartościami temperatur, co sprawia, że dostaje 50 punktów. Przykładowy implementujący go program można znaleźć w pliku `tems11.cpp`.

Podobną złożoność można uzyskać na kilka innych sposobów, przykładowe implementacje są w plikach `tems[2,3,6,7,8,9,10].[cpp|pas]`. Można, na przykład, zastosować programowanie dynamiczne, w którym dla danego j i wartości *temp* obliczamy (także w kolejności malejących j) najdłuższy przedział zaczynający się w j, dla którego istnieje niemalejący ciąg temperatur spełniających odpowiednie warunki, a przy tym rozpoczynający się temperaturą równą *temp*. To rozwiązanie, w podstawowej wersji, zużywa aż $\Theta(n\Delta)$ pamięci. Jednakże zawsze przy obliczaniu wartości dla j korzystamy tylko z wartości dla $j + 1$, więc można używać na zmianę tylko dwóch tablic rozmiaru $\Theta(\Delta)$.

Rozwiązanie wzorcowe

Poprzednie rozwiązanie ciągle nie jest satysfakcjonujące, ponieważ dla $\Delta = \Theta(n)$ zdarza mu się działać w czasie $\Theta(n^2)$. Aby otrzymać istotnie szybszy algorytm, zastosujemy technikę nazywaną czasami *metodą gąsienicy*. Służy ona właśnie do znajdowania dla każdego j najdłuższego w jakimś sensie dobrego przedziału zaczynającego się w j. Wymagana jest przy tym monotoniczność, tzn. aby każdy podprzedział dobrego przedziału był dobry (co ma miejsce w naszym przypadku).

Przedstawmy zarys tej metody. Zaczynamy od jednoelementowego przedziału zawierającego pierwszy element. Następnie przesuwamy prawy koniec przedziału, dopóki przedział pozostaje dobry, w ten sposób uzyskujemy maksymalny przedział zaczynający się na pierwszej pozycji. Teraz zwiększamy lewy koniec przedziału o 1 i znów zaczynamy przesuwać prawy koniec przedziału, począwszy od miejsca, w którym skończyliśmy wcześniej, itd.

Na cały proces składa się liniowa liczba przesunięć końców przedziałów. Po każdym przesunięciu prawego końca musimy sprawdzić, czy ta operacja nie spowoduje, że przedział przestanie być dobry. Podobną operację wykonywaliśmy już w poprzednich algorytmach. Potrzebowaliśmy do tego zmiennej *temp*, którą aktualizowaliśmy przy przesuwaniu prawego końca przedziału. W tym rozwiązaniu musimy jeszcze umieć ją aktualizować przy przesuwaniu lewego końca. Wystarczy nam do tego proste spostrzeżenie, że jeśli aktualnie rozważany przedział to $[j, i]$, to bieżąca wartość zmiennej *temp* powinna być równa $\max(x[j], x[j+1], \ldots, x[i])$. Możemy wobec tego zapisać następujący pseudokod, w którym już nie używamy *explicite* zmiennej *temp*:

```
1: wynik := 1;
2: i := 1;
3: for j := 1 to n do begin
4:    while i ⩽ n and max(x[j], x[j + 1], ..., x[i − 1]) ⩽ y[i] do
5:       i := i + 1;
6:    wynik := max(wynik, i − j);
7: end
8: return wynik;
```

Obliczanie maksimów

Teraz pozostaje nam tylko szybkie obliczanie wartości postaci

$$\max(x_i, x_{i+1}, \ldots, x_{j-1}).$$

Znajdowanie maksymalnego (lub minimalnego) elementu w zadanym fragmencie ciągu jest dosyć standardowym problemem, znanym też pod nazwą RMQ (ang. *Range Minimum Query*). Jest kilka klasycznych struktur danych pozwalających na rozwiązanie tego problemu. Są to drzewa przedziałowe, zwane też licznikowymi (rozmiar struktury $O(n)$, koszt zapytania $O(\log n)$), struktura danych podobna do słownika podsłów bazowych (rozmiar $O(n \log n)$, koszt zapytania $O(1)$), jest wreszcie znane rozwiązanie optymalne (niestety trudne w implementacji i dlatego raczej niestosowane w praktyce), w którym po wstępnych obliczeniach w czasie $O(n)$ możemy odpowiadać na

zapytania w czasie stałym[1]. Ze względu na małe ograniczenie pamięciowe w tym zadaniu, najlepsza z wymienionych metod to zastosowanie drzew przedziałowych — już za takie rozwiązanie można było uzyskać 100 punktów. Przykładowa implementacja znajduje się w plikach `tem2.cpp` oraz `tem3.pas`.

Maksima szczególnej postaci

Aby otrzymać prostsze i efektywniejsze rozwiązanie naszego zadania, należy zauważyć, że występujące w nim zapytania są bardzo szczególnej postaci — przedział, o który pytamy, „pełznie" przez tablicę. Wystarczy nam zatem struktura danych podobna do kolejki, wspierająca operacje: wstawiania na koniec (*wstaw*), zdejmowania z początku (*zdejmij*) oraz odczytywania maksimum (*max*). Pseudokod używający takiej struktury wygląda następująco[2]:

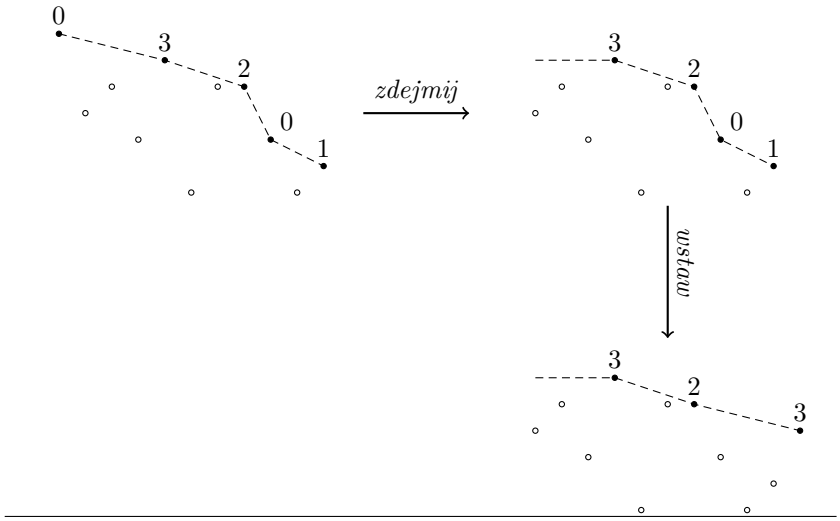
```
1:  i := 1;
2:  for j := 1 to n do begin
3:    while i ⩽ n and Q.max() ⩽ y[i] do begin
4:      Q.wstaw(x[i]);
5:      i := i + 1;
6:    end
7:    wynik := max(wynik, i − j);
8:    Q.zdejmij();
9:  end
10: return wynik;
```

Jeśli udałoby nam się wykonywać operacje dodawania elementu, usuwania najdawniej wstawionego elementu i znajdowania maksimum w zamortyzowanym czasie stałym, uzyskalibyśmy algorytm liniowy. Taka struktura danych wystąpiła już w rozwiązaniach zadań olimpijskich: w zadaniu *Piloci* z III etapu XVII Olimpiady Informatycznej [17] oraz w zadaniu *BBB* z II etapu XV Olimpiady [15]. Aby zachować kompletność opisu rozwiązania wzorcowego, omawiamy ją także poniżej.

Dla ustalenia uwagi przypuśćmy, że wstawiane elementy są parami różne, można to osiągnąć np. przyjmując, że spośród dwóch równych elementów większy jest ten wstawiony później. Zastanówmy się, jakie elementy kolejki mają szansę stać się kiedykolwiek maksimum. Oczywiście, jeśli w kolejce jest jakaś liczba, a potem wstawiona została większa liczba, ta pierwsza nigdy już nie będzie maksimum. Co więcej, nie będziemy potrzebować jej wartości, ponieważ przy operacji *zdejmij* nie interesuje nas wartość, którą usuwamy. Liczbę zawartą w kolejce nazwiemy *ciekawą*, jeśli żadna później wstawiona do kolejki liczba nie jest od niej większa. Liczby ciekawe tworzą ciąg malejący, który łatwo aktualizować przy wstawianiu. Maksimum ($Q.max()$) jest, oczywiście, pierwszy element kolejki. Problematyczne pozostaje usuwanie, czasem bowiem usuwamy liczby ciekawe, a czasem nieciekawe. Wystarczy jednak zapamiętać,

[1] Krótki opis pierwszych dwóch wspomnianych metod wraz z odnośnikami można znaleźć w opracowaniu zadania *Piorunochron* w tej książeczce. Opis trzeciej metody można znaleźć w dwuczęściowym artykule poświęconym problemom RMQ i LCA, w numerach 9/2007 i 11/2007 czasopisma *Delta* (artykuły dostępne także na stronie `http://www.deltami.edu.pl`).

[2] Zakładamy, że dla pustej kolejki mamy $Q.max() = -\infty$.



Rys. 1: Ilustracja działania struktury: pełne kółka reprezentują elementy kolejki D, puste to elementy nieciekawe, współrzędne kółka to odpowiednio indeks w ciągu i wartość, a liczba przy kółku to liczba nieciekawych elementów w ciągu po poprzednim ciekawym (albo od początku ciągu, jeśli rozważamy pierwszy ciekawy element).

ile jest nieciekawych liczb przed maksimum i między każdymi dwiema sąsiednimi ciekawymi.

Taką strukturę łatwo zapisać w terminologii kolejki dwustronnej par liczb całkowitych, np. kontenera `deque` z biblioteki STL w C++. Załóżymy, że mamy do dyspozycji taką kolejkę dwustronną D z operacjami *pusta*(), *pierwszy*(), *ostatni*(), *zdejmij_pierwszy*(), *zdejmij_ostatni*(), *wstaw_ostatni*(x, y). Do elementów pary będziemy odwoływać się przez *.wartość* oraz *.ile*.

Wstawienie elementu. Operacja wstawiania działa według schematu: dopóki ostatni element kolejki nie przekracza aktualnie dodawanego, usuwaj go. Następnie wstaw aktualny element na początek kolejki.

```
1: procedure wstaw(wartość)
2: begin
3:   ile := 0;
4:   while (not D.pusta()) and D.ostatni().wartość ⩽ wartość do begin
5:     ile := ile + D.ostatni().ile + 1;
6:     D.zdejmij_ostatni();
7:   end
8:   D.wstaw_ostatni(wartość, ile);
9: end
```

Koszt zamortyzowany tej operacji jest stały. Każdy element wyjściowego ciągu zostaje w algorytmie wstawiony dokładnie raz, a łączna liczba usunięć jest nie większa niż liczba wstawień.

Usunięcie elementu. Sprawdzamy pierwszy element kolejki — jeśli jest ciekawy, usuwamy go, jeśli nie, zmniejszamy odpowiedni licznik.

```
1: procedure zdejmij()
2: begin
3:   if D.pierwszy().ile = 0 then
4:     D.zdejmij_pierwszy()
5:   else
6:     D.pierwszy().ile := D.pierwszy().ile − 1;
7: end
```

Usuwanie odbywa się w czasie stałym.

Sprawdzenie maksimum. Zwracamy wartość pierwszego elementu.

```
1: function max()
2: begin
3:   if D.pusta() then
4:     return −∞;
5:   return D.pierwszy().wartość;
6: end
```

Znów ewidentnie mamy czas stały operacji.

To kończy opis rozwiązania wzorcowego. Implementacje znajdują się w plikach `tem.cpp`, `tem1.pas` oraz `tem5.cpp`.

Utrudnione zadanie

Podczas zawodów część zawodników błędnie zrozumiała treść zadania — zamiast szukać spójnego przedziału, szukali niespójnego podciągu. Zawodnicy mogli zasugerować się znanym problemem znajdywania najdłuższego podciągu rosnącego i tym samym utrudnili sobie zadanie. Rozwiązanie takiej wersji pozostawiamy Czytelnikowi jako ćwiczenie.

Testy

Testy zostały podzielone na trzy główne grupy:

- *mały losowy* — małe losowe testy poprawnościowe,
- *duży wynik* — wynikiem jest długi przedział czasu,
- *duży stos* — w strukturze z rozwiązania wzorcowego znajduje się równocześnie wiele elementów.

W testach oznaczonych jako *małe temperatury* temperatury mieszczą się w przedziale od -50 do 50.

Nazwa	n	Opis
tem1a.in	92	mały test losowy (małe temperatury)
tem1b.in	93	mały test losowy (małe temperatury)
tem1c.in	94	mały test losowy (małe temperatury)
tem2a.in	100	mały test losowy (małe temperatury)
tem2b.in	98	duży wynik (małe temperatury)
tem2c.in	82	duży wynik (małe temperatury)
tem2d.in	102	duży stos (małe temperatury)
tem3a.in	10 000	duży wynik (małe temperatury)
tem3b.in	12 101	duży wynik (małe temperatury)
tem3c.in	13 221	duży stos (małe temperatury)
tem4a.in	458 325	duży wynik (małe temperatury)
tem4b.in	522 312	duży wynik (małe temperatury)
tem4c.in	593 721	duży stos (małe temperatury)
tem5a.in	724 855	duży wynik
tem5b.in	785 775	duży wynik
tem5c.in	793 722	duży stos
tem6a.in	763 538	duży wynik
tem6b.in	812 289	duży wynik
tem6c.in	813 783	duży stos
tem7a.in	891 379	duży wynik
tem7b.in	832 280	duży wynik
tem7c.in	999 999	duży stos
tem8a.in	813 780	duży stos
tem8b.in	828 183	duży stos
tem8c.in	812 289	duży wynik
tem9a.in	813 780	duży stos (małe temperatury)
tem9b.in	828 183	duży stos (małe temperatury)
tem9c.in	812 289	duży wynik (małe temperatury)
tem10a.in	765 324	duży wynik (małe temperatury)
tem10b.in	850 626	duży stos (małe temperatury)
tem11a.in	713 964	duży wynik
tem11b.in	956 352	duży stos

Nazwa	n	Opis
tem12a.in	687 555	duży wynik
tem12b.in	854 694	duży stos

Zawody III stopnia

opracowania zadań

Jacek Tomasiewicz
Treść zadania, Opracowanie

Mateusz Baranowski
Program

Dostępna pamięć: 64 MB.

OI, Etap III, dzień próbny, 05.04.2011

Dynamit

Jaskinia Bajtocka składa się z n komór oraz z $n-1$ łączących je korytarzy. Nie wychodząc z jaskini, pomiędzy każdą parą komór można przejść bez zawracania na dokładnie jeden sposób. W niektórych komorach pozostawiono dynamit. Wzdłuż każdego korytarza rozciągnięto lont. W każdej komorze wszystkie lonty z prowadzących do niej korytarzy są połączone, a jeżeli w danej komorze znajduje się dynamit, to jest on połączony z lontem. Lont biegnący korytarzem pomiędzy sąsiednimi komorami spala się w jednej jednostce czasu, a dynamit wybucha dokładnie w chwili, gdy ogień znajdzie się w komorze zawierającej ten dynamit.

Chcielibyśmy równocześnie podpalić lonty w jakichś m komorach (w miejscu połączenia lontów) w taki sposób, aby wszystkie ładunki dynamitu wybuchły w jak najkrótszym czasie, licząc od momentu podpalenia lontów. Napisz program, który wyznaczy najkrótszy możliwy taki czas.

Wejście

Pierwszy wiersz standardowego wejścia zawiera dwie liczby całkowite n i m ($1 \leqslant m \leqslant n \leqslant 300\,000$), oddzielone pojedynczym odstępem i oznaczające odpowiednio liczbę komór w jaskini oraz liczbę komór, w których możemy podpalić lonty. Komory są ponumerowane od 1 do n. Kolejny wiersz zawiera n liczb całkowitych $d_1, d_2, \ldots, d_n$ ($d_i \in \{0, 1\}$), pooddzielanych pojedynczymi odstępami. Jeśli $d_i = 1$, to w i-tej komorze znajduje się dynamit, a jeśli $d_i = 0$, to nie ma w niej dynamitu. Kolejne $n-1$ wierszy opisuje korytarze w jaskini. W każdym z nich znajdują się dwie liczby całkowite a, b ($1 \leqslant a < b \leqslant n$) oddzielone pojedynczym odstępem i oznaczające, że istnieje korytarz łączący komory a i b. Każdy korytarz pojawia się w opisie dokładnie raz.

Możesz założyć, że w testach wartych łącznie 10% punktów zachodzi dodatkowy warunek $n \leqslant 10$, a w testach wartych łącznie 40% punktów zachodzi $n \leqslant 1\,000$.

Wyjście

Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą, równą minimalnemu czasowi od podpalenia lontów, po jakim wybuchną wszystkie ładunki dynamitu.

Przykład

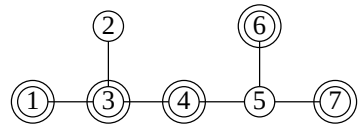
Dla danych wejściowych:

```
7 2
1 0 1 1 0 1 1
1 3
2 3
```

```
3 4
4 5
5 6
5 7
```

poprawnym wynikiem jest:

```
1
```



Wyjaśnienie do przykładu: *Podpalamy lonty w komorach 3 i 5. W chwili zero wybucha dynamit w komorze 3, a po jednostce czasu zapalone zostają dynamity w komorach 1, 4, 6 i 7.*

Rozwiązanie

Wprowadzenie

Niestety wszystkie trywialne rozwiązania tego zadania działają w czasie wykładniczym. Spróbujmy wobec tego na początek włożyć nieco wysiłku w stworzenie jakiegokolwiek rozwiązania wielomianowego. Później przyjdzie czas na jego przyspieszanie.

Skoncentrujmy się na razie na nieco prostszym problemie — sprawdzaniu, czy da się podpalić lonty w co najwyżej m komorach, tak aby wszystkie ładunki wybuchły w pierwszych t sekundach po podpaleniu lontów. Oczywiście, wielomianowe rozwiązanie tego problemu natychmiast daje wielomianowe rozwiązanie całego zadania. Możemy sprawdzać po kolei wszystkie możliwe t.

Problem decyzyjny dla ustalonego t

Dla uproszczenia języka w naszych rozważaniach, komory, w których podpalamy lonty (oraz odpowiadające im wierzchołki), nazwiemy *startowymi*. Układ co najwyżej m komór startowych spełniających opisany warunek nazwiemy zaś *rozwiązaniem*.

Wciąż nie narzuca nam się żaden trywialny wielomianowy algorytm. W takich przypadkach zwykle pozostaje szukać obserwacji, które pozwolą np. zredukować rozmiar danych czy też nałożyć dodatkowe warunki, które musi spełniać choć jedno rozwiązanie, jeśli w ogóle jakiekolwiek rozwiązanie istnieje.

Zastanówmy się, czy są jakieś wierzchołki, o których możemy bez straty ogólności przyjąć, że są lub nie są startowe. Nietrudno na przykład dostrzec, że jeśli w naszym drzewie jest liść, w którym nie ma dynamitu, nie ma sensu wybierać go jako startowego. Ogień zaprószony w liściu obejmie w ciągu pierwszych t sekund mniej wierzchołków (w sensie inkluzji, czyli zawierania zbiorów) niż zaprószony w jego sąsiedzie. Można zatem przerobić każde rozwiązanie, w którym rozważany liść jest wierzchołkiem startowym, tak aby zamiast niego wierzchołkiem startowym był jedyny sąsiad tego liścia. Podobnie jest wtedy, gdy w liściu jest dynamit, lecz $t > 0$.

Powyższe spostrzeżenie to krok w dobrą stronę. Pierwszą część można nieco wzmocnić — jeśli w liściu nie ma dynamitu, możemy w ogóle usunąć go z drzewa z gwarancją, że odpowiedź na postawione przed nami pytanie pozostanie bez zmiany. Usuwając do skutku takie liście, w końcu dostaniemy drzewo, w którym w każdym liściu znajduje się ładunek dynamitu. Być może będzie to drzewo puste lub mające

tylko pojedynczy wierzchołek. Jeśli tak się stanie, z odpowiedzią poradzimy sobie bez problemu, więc dalej będziemy zakładać przeciwnie.

Z drugą częścią spostrzeżenia jest pewien problem — nie pozwala krok po kroku eliminować wierzchołków jako potencjalnych startowych. Taka sytuacja występuje na przykład dla drzewa składającego się z dwóch wierzchołków zawierających dynamit i połączonych krawędzią. Dla $t > 0$ musimy wybrać któryś z nich, choć obydwa są liśćmi. Aby zapobiec tego rodzaju sytuacjom, ukorzenimy drzewo. Teraz zamiast mówić, że dla każdego liścia istnieje rozwiązanie, w którym ten liść nie będzie startowy, można sformułować silniejszy fakt. Istnieje rozwiązanie, w którym żaden liść, być może z wyjątkiem korzenia, nie jest startowy.

Analizując uzasadnienie wyjściowego spostrzeżenia i idąc dalej tą drogą, możemy otrzymać silniejszą obserwację.

Obserwacja 1. Niech $N_t(v)$ oznacza zbiór wierzchołków, które są odległe od v o co najwyżej t. Niech U będzie zbiorem wierzchołków drzewa o tej własności, że dla każdego $u \in U$ zachodzi $N_t(u) \subseteq N_t(\mathit{parent}(u))$, gdzie $\mathit{parent}(u)$ oznacza ojca w drzewie. Wówczas każde rozwiązanie da się przerobić tak, aby nie zawierało wierzchołków ze zbioru U.

Dowód: Dowodem będzie algorytm odpowiednio przerabiający rozwiązanie. Wystarczy bardzo prosty — dopóki jakiś $u \in U$ jest startowy, zamiast u bierzemy jego ojca. Jeśli ojciec już był wierzchołkiem startowym, po prostu usuwamy u z rozwiązania. Z założeń jest jasne, że po każdym kroku zbiór wierzchołków startowych wciąż jest rozwiązaniem. Ponadto w każdym kroku elementy zbioru startowego wędrują w górę drzewa (lub w ogóle znikają), więc nasza procedura ma własność stopu. ■

Zastanówmy się, co wiemy o zbiorze U. Na pewno nie należy do niego korzeń (nie ma zdefiniowanego ojca), zaś dla $t > 0$ należą do niego liście. Chwila pomyślunku pozwala w pełni scharakteryzować ten zbiór — należą do niego te wierzchołki różne od korzenia, których wszyscy potomkowie są od nich odlegli o co najwyżej $t-1$. Innymi słowy, poddrzewa zaczepione w tych wierzchołkach mają wysokość co najwyżej $t-1$ (uznajemy tu, że poddrzewo puste ma wysokość 0, jednowierzchołkowe 1 itd.).

Dla dość dużego zbioru wierzchołków U ustaliliśmy, że nie będziemy wybierać ich jako startowe. Wcześniej przez ucinanie liści doprowadziliśmy nasze drzewo do sytuacji, w której w każdym liściu jest dynamit. Okazuje się, że w takiej sytuacji zawsze jesteśmy w stanie wskazać co najmniej jeden wierzchołek, który musimy wybrać jako startowy. Jeśli wysokość drzewa (oznaczmy ją przez h) nie przekracza t, jedynym wierzchołkiem poza U jest korzeń, więc sprawa jest jasna. W przeciwnym przypadku rozpatrzmy najgłębszy liść w drzewie, czyli dowolny liść na głębokości $h-1$. Wiemy, że jest tam ładunek, który musi znaleźć się w zasięgu pewnego wierzchołka startowego. Wszystkie wierzchołki o głębokości $h-t$ lub większej są jednak w zbiorze U. Stąd jedynym wierzchołkiem spoza U, w którego zasięgu jest nasz liść, jest jego przodek o głębokości $h-t-1$. Ten właśnie wierzchołek musi być jednym ze startowych.

Istnienie wierzchołka, który musi być startowy, to dla nas znakomita wiadomość. Możemy bowiem zaznaczyć, że zużyliśmy jeden taki wierzchołek, usunąć ładunki znajdujące się w jego zasięgu i kontynuować obliczenia. Usuwamy w szczególności ładunek w liściu, więc poprzednia redukcja pozwoli zmniejszyć drzewo. Wobec tego nigdy nie

utkniemy w martwym punkcie. Otrzymaliśmy tym samym pierwszy algorytm rozwiązujący nasz problem. Oto jego schematyczny zapis:

```
1:  procedure Sprawdź(t)
2:  begin
3:    while |T| > 0 do begin
4:      if pewien liść nie zawiera ładunku then
5:        usuń ten liść
6:      else if m = 0 then
7:        return NIE
8:      else begin
9:        h := wysokość drzewa T;
10:       if h ⩽ t then return TAK;
11:       v := wierzchołek o t poziomów ponad najgłębszym liściem;
12:       m := m − 1;
13:       usuń ładunki wybuchowe w zbiorze N_t(v);
14:     end
15:   end
16:   return TAK;
17: end
```

Zbadajmy, jak szybko działa otrzymany algorytm. Najwięcej zajmie usuwanie ładunków wybuchowych — odwiedzanie wszystkich wierzchołków z $N_t(v)$ może za każdym razem zajmować czas liniowy. Wszystkie inne operacje w pojedynczym obrocie pętli też oczywiście łatwo wykonamy liniowo, a obrotów jest liniowo wiele (w każdym usuwamy liść, ładunek z liścia lub wychodzimy z pętli). Stąd łączny czas działania możemy oszacować przez $O(n^2)$.

Przypomnijmy, że skonstruowany właśnie algorytm musimy do rozwiązania całego zadania wykonać $O(n)$, a dokładniej $O(wynik)$ razy. Stąd łączna złożoność to $O(n^2 \cdot wynik)$.

Przyspieszamy algorytm

Ograniczamy liczbę sprawdzeń

Jak na razie nie przykładaliśmy wagi do dobrej złożoności, tylko do otrzymania jakiegokolwiek wielomianowego algorytmu. W szczególności, bardzo nieoszczędnie zredukowaliśmy zadanie do sprawdzania, czy istnieje rozwiązanie dla konkretnych t. Oczywiście, ta własność jest monotoniczna — jeśli jest prawdziwa dla t_0, to także dla każdego $t \geqslant t_0$.

Taka sytuacja to typowe okoliczności przemawiające za zastosowaniem wyszukiwania binarnego. Wystarczy zatem wywołać funkcję sprawdzającą $O(\log n)$ razy. Zredukowaliśmy tym samym złożoność do $O(n^2 \log n)$. Programy implementujące taki algorytm zdobywały ok. 40 punktów. Przykładowe rozwiązania można znaleźć w plikach `dyns3.c`, `dyns4.cpp` oraz `dyns5.pas`.

Dociekliwy Czytelnik może zauważyć, że dla małych wyników tak naprawdę nasz algorytm jest teraz wolniejszy. Okazuje się jednak, że można tak przerobić wyszukiwanie binarne kosztem stałej ukrytej w notacji O, aby działało w $O(\log(wynik + 2))$ krokach. Pozostawiamy tę modyfikację jako ćwiczenie dla Czytelnika. W zadaniu nie była oczywiście wymagana.

Przyspieszamy sprawdzenie

Doszliśmy do momentu, w którym nie jesteśmy w stanie istotnie poprawić złożoności całego rozwiązania, nie przyspieszając funkcji sprawdzającej. Poprawmy wobec tego jej złożoność.

Nasz poprzedni algorytm idzie z grubsza w kierunku od liści do korzenia, zapewniając, że wszystkie ładunki na kolejnych poziomach znajdują się w zasięgu pewnego wierzchołka startowego. Staramy się przy tym umieścić wierzchołki startowe najbliżej korzenia jak tylko się da. Na razie nie idziemy jednak po prostu z dołu do góry, czasami skaczemy t poziomów w górę, by ustawić wierzchołek startowy i usunąć pewne ładunki. Spróbujemy „opóźnić" te operacje i postępować jednoprzebiegowo. Cały czas chcemy umieszczać wierzchołki startowe najwyżej jak się da, czyli tak, jak czyni to poprzedni algorytm. Na takie podejście można również spojrzeć jak na programowanie dynamiczne — zdefiniujemy pewną funkcję i wyznaczymy jej wartość dla wierzchołka na podstawie wartości dla synów.

Zastanówmy się, kiedy wybieramy jakiś wierzchołek jako startowy. Na pewno wówczas, gdy pewien ładunek znajduje się w poddrzewie tego wierzchołka dokładnie t poziomów niżej i wciąż nie zapewniliśmy jego detonacji (w skrócie: nie zdetonowaliśmy go). Również wtedy, gdy taki ładunek jest nieco wyżej, ale dotarliśmy już do korzenia. Ta sytuacja odpowiada umieszczeniu w poprzednim algorytmie ładunku o t poziomów wyżej niż najgłębszy dotychczas nierozpatrzony. Nie dopuścimy do sytuacji, gdy pewien ładunek pozostanie o więcej niż t poziomów głębiej niż aktualnie rozważany wierzchołek, więc przy okazji tej operacji zdetonujemy wszystkie ładunki w naszym poddrzewie. Nie trzeba zatem pamiętać wszystkich niezdetonowanych ładunków — wystarczy najgłębszy z nich, a właściwie jego głębokość względem aktualnie rozpatrywanego wierzchołka.

Wybierając wierzchołek jako startowy, możemy również zdetonować pewne ładunki spoza poddrzewa. Ogień dociera do nich przez ojca wierzchołka, w którym zaczepione jest poddrzewo, więc ich rozpatrzenie zostawimy na później. W tym celu musimy zapamiętać, gdzie umieściliśmy wierzchołek startowy. Wystarczy nam względna głębokość najwyżej położonego wierzchołka startowego w rozważanym poddrzewie. Dzięki temu, gdy będziemy przetwarzać pewien wierzchołek, będziemy mogli sprawdzić, czy najniżej położony niezdetonowany ładunek w poddrzewie pewnego syna jest w zasięgu wierzchołka startowego w poddrzewie innego syna.

Opisane dwie wartości wystarczają zatem do przeprowadzenia obliczeń. Pierwszą, czyli głębokość najniższego niezdetonowanego ładunku (lub $-\infty$, gdy takiego nie ma), zapiszmy w tablicy *dynamit*. Drugą, czyli głębokość najwyższego wierzchołka startowego (lub ∞, gdy takiego nie ma) — w tablicy *ogień*. Zbierzmy dotychczasowe rozważania w postaci pseudokodu, w którym zilustrujemy pojedynczy krok algorytmu. Warto pamiętać o szczególnym traktowaniu korzenia. Ten detal może łatwo umknąć przy implementacji.

```
 1: procedure Obsłuż(v)
 2: begin
 3:   { niech v oznacza bieżący wierzchołek }
 4:   ogień[v] := ∞;
 5:   dynamit[v] := −∞;
 6:   if w v jest dynamit then
 7:     dynamit[v] := 0;
 8:   foreach w : syn v do begin
 9:     dynamit[v] := max(dynamit[v], dynamit[w] + 1);
10:     ogień[v] := min(ogień[v], ogień[w] + 1);
11:   end
12:   if dynamit[v] + ogień[v] ⩽ t then begin
13:   { istniejące wierzchołki startowe zdetonują wszystkie pozostałe ładunki }
14:     dynamit[v] := −∞;
15:   end
16:   if dynamit[v] = t or (v to korzeń and dynamit[v] ≠ −∞) then begin
17:   { wybieramy v jako startowy }
18:     m := m − 1;
19:     dynamit[v] := −∞;
20:     ogień[v] := 0;
21:   end
22: end
```

Cały algorytm sprawdzania, czy istnieje rozwiązanie (czyli m wierzchołków, w zasięgu których jest każdy ładunek wybuchowy), polega po prostu na wykonaniu przedstawionego algorytmu dla każdego wierzchołka. Czynimy to od liści do korzenia, ponieważ korzystamy z wartości dla synów. Rozwiązanie istnieje wtedy i tylko wtedy, gdy udało nam się zużyć co najwyżej m wierzchołków startowych, czyli gdy na końcu m jest nieujemne.

Złożoność czasowa takiej funkcji sprawdzającej jest liniowa. Wobec tego cały algorytm, jeśli korzysta z wyszukiwania binarnego, działa w czasie $O(n \log n)$. Na tej zasadzie działało rozwiązanie wzorcowe, zaimplementowane je w plikach `dyn.c`, `dyn1.cpp` i `dyn2.pas`.

Rozwiązanie siłowe

Jak głosi treść zadania, do otrzymania 10 punktów wystarcza program działający dla $n \leqslant 10$, czyli w praktyce brutalne rozwiązanie wykładnicze. Dobrze napisane takie rozwiązanie działa w czasie $O(n \cdot \binom{n}{m})$. Wystarczała jednak złożoność czasowa $O(2^n + nm \cdot \binom{n}{m})$. Takie rozwiązanie polegało na wygenerowaniu wszystkich podzbiorów zbioru wierzchołków, odfiltrowaniu m-elementowych i dla każdego z nich przeprowadzeniu symulacji. Można ją przeprowadzić przy użyciu jednego wywołania algorytmu BFS o wielu źródłach (w czasie $O(n)$) lub m zwykłych wywołań tego algorytmu (czas łącznie $O(mn)$). Przykładowe rozwiązania wykładnicze można znaleźć w plikach `dyns6.c` i `dyns7.pas`.

Testy

Zadanie było sprawdzane na 10 grupach testów, zawierających łącznie 34 przypadki testowe. Krótkie opisy testów są zawarte w następującej tabeli.

Nazwa	n	m	Opis
dyn1a.in	1	1	minimalny
dyn1b.in	1	1	minimalny
dyn1c.in	6	2	prosty poprawnościowy
dyn1d.in	10	3	poprawnościowy
dyn2a.in	50	3	mało rozgałęzień i podpaleń
dyn2b.in	97	8	dużo rozgałęzień
dyn2c.in	80	10	dużo podpaleń
dyn3a.in	500	10	mało rozgałęzień i podpaleń
dyn3b.in	600	40	dużo rozgałęzień i dynamitu
dyn4a.in	800	8	mały test losowy, mało podpaleń
dyn4b.in	1000	50	mały test, dużo podpaleń
dyn5a.in	10 000	100	losowy
dyn5b.in	23 450	1 000	dużo dynamitu
dyn5c.in	30 000	2	mało dynamitu, drzewo binarne
dyn5d.in	30 000	30	linia
dyn6a.in	50 000	5	losowy
dyn6b.in	70 000	13	dużo dynamitu
dyn6c.in	80 000	1 000	jedno duże rozgałęzienie
dyn6d.in	76 500	66	kilka wierzchołków z dużą liczbą krawędzi
dyn7a.in	100 000	600	losowy
dyn7b.in	100 000	16	dużo dynamitu
dyn7c.in	100 000	11	mało dynamitu, małe rozgałęzienia
dyn7d.in	110 000	23	losowy, dynamit w każdej komnacie
dyn8a.in	200 000	17	losowy
dyn8b.in	230 000	27	dużo dynamitu
dyn8c.in	250 000	14	dynamit w każdej komnacie
dyn8d.in	250 000	14	dużo dynamitu
dyn9a.in	299 900	7	dużo rozgałęzień
dyn9b.in	299 990	33	dużo dynamitu
dyn9c.in	299 999	12 033	regularne drzewo złożone ze ścieżek długości 5

Nazwa	n	m	Opis
dyn10a.in	300 000	1	maksymalny test
dyn10b.in	300 000	49	dużo dynamitu
dyn10c.in	300 000	30	drzewo binarne o dużej wysokości
dyn10d.in	300 000	299 999	maksymalna gwiazda

Jakub Wojtaszczyk — Treść zadania, Opracowanie
Alan Kutniewski — Program

Dostępna pamięć: 64 MB. *OI, Etap III, dzień pierwszy, 06.04.2011*

Impreza

Bajtazar chciałby zorganizować imprezę. Zależy mu na tym, żeby impreza była udana, i sądzi, że najlepszym sposobem na to jest, żeby wszyscy zaproszeni goście się znali. Teraz Bajtazar musi ustalić listę swoich znajomych, których ma zaprosić.

Bajtazar ma n znajomych, przy czym n jest podzielne przez 3. Szczęśliwie, większość znajomych Bajtazara zna się nawzajem. Więcej — Bajtazar pamięta, że był na imprezie, na której było $\frac{2}{3}n$ jego znajomych i na której wszyscy znali się nawzajem. Niestety, poza tym jednym faktem Bajtazar nieszczególnie wiele z tej imprezy pamięta... W szczególności nie ma pojęcia, którzy znajomi byli wtedy obecni.

Bajtazar nie musi organizować wielkiej imprezy, ale chciałby zaprosić przynajmniej $\frac{n}{3}$ spośród swoich znajomych. Nie ma za bardzo pomysłu, jak ich dobrać, więc poprosił Cię o pomoc.

Wejście

W pierwszym wierszu standardowego wejścia podane są dwie liczby całkowite n i m ($3 \leqslant n \leqslant 3\,000$, $\frac{\frac{2}{3}n(\frac{2}{3}n-1)}{2} \leqslant m \leqslant \frac{n(n-1)}{2}$), oddzielone pojedynczym odstępem, oznaczające odpowiednio liczbę znajomych Bajtazara oraz liczbę par znajomych Bajtazara, którzy znają się nawzajem. Znajomi Bajtazara są ponumerowani od 1 do n. W kolejnych m wierszach znajdują się po dwie liczby całkowite oddzielone pojedynczym odstępem. W wierszu $i+1$ (dla $i = 1, 2, \ldots, m$) znajdują się liczby a_i i b_i ($1 \leqslant a_i < b_i \leqslant n$), oddzielone pojedynczym odstępem, które oznaczają, że osoby a_i oraz b_i znają się. Każda para liczb pojawia się na wejściu co najwyżej raz.

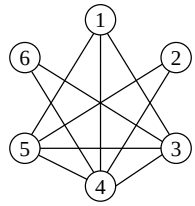
Wyjście

Twój program powinien wypisać w pierwszym i jedynym wierszu standardowego wyjścia $\frac{n}{3}$ liczb, w kolejności rosnącej, pooddzielanych pojedynczymi odstępami, oznaczających numery znajomych, których Bajtazar ma zaprosić na imprezę. Jako że istnieje wiele rozwiązań, należy wypisać dowolne z nich.

Przykład

Dla danych wejściowych:

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



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

poprawnym wynikiem jest:

```
2 4
```

Wyjaśnienie do przykładu: *Znajomi nr 1, 3, 4, 5 znają się nawzajem. Prawidłowym wynikiem jest dowolna dwójka znajomych, którzy się znają. Nie muszą oni nawet być częścią jakiejkolwiek czwórki znającej się wzajemnie.*

Rozwiązanie

Wprowadzenie

Postawiony przed nami problem łatwo wyrazić w terminologii grafowej. Mamy dany graf nieskierowany $G = (V, E)$, którego wierzchołki reprezentują znajomych Bajtazara, zaś krawędź łącząca dwa wierzchołki oznacza, że odpowiednie dwie osoby się znają. Warunki zadania gwarantują nam, że w grafie istnieje zbiór przynajmniej $2|V|/3$ wierzchołków, wśród których każda para jest połączona krawędzią; taki zbiór wierzchołków w teorii grafów nazywa się *kliką*. Naszym zadaniem jest znaleźć w tym grafie dowolną klikę rozmiaru $|V|/3$. Przypomnijmy też założenie, że $3 \mid |V|$.

Rozwiązanie wzorcowe

Niech $n = |V|$ oznacza liczbę wierzchołków w wejściowym grafie. Rozwiązanie wzorcowe działa następująco:

1. Jeżeli w grafie występują dwa wierzchołki, które nie są połączone krawędzią, to usuwamy obydwa te wierzchołki z grafu i kontynuujemy.
2. Jeżeli w grafie (po usunięciu pewnych wierzchołków zgodnie z krokiem 1) każde dwa wierzchołki są połączone krawędzią, to wybieramy dowolne $n/3$ wierzchołków i zwracamy je jako rozwiązanie.

Zbiór wierzchołków, który zwrócimy w drugim kroku, oczywiście będzie kliką. Jedyne, co trzeba sprawdzić, by zweryfikować poprawność powyższego rozwiązania, to to, czy w momencie dojścia do drugiego kroku w grafie faktycznie pozostanie nam co najmniej $n/3$ wierzchołków.

Aby się o tym upewnić, rozważmy dowolny zbiór wierzchołków W będący kliką rozmiaru $2n/3$ — nie umiemy takiego zbioru znaleźć, ale z warunków zadania wynika, że taki zbiór istnieje. Niech $U = V \setminus W$ będzie zbiorem pozostałych wierzchołków. Początkowo $|W| = 2n/3$, $|U| = n/3$. Zauważmy, że wśród dowolnych dwóch wierzchołków usuniętych w pierwszym kroku algorytmu zawsze co najmniej jeden jest z U — w przeciwnym razie w zbiorze W znajdowałyby się dwa wierzchołki niepołączone krawędzią, co przeczy temu, że W jest kliką. Wobec tego w każdym kroku algorytmu rozmiar zbioru U maleje co najmniej o jeden, a zatem algorytm wykona nie więcej

niż $n/3$ kroków. Jednocześnie w każdym kroku usuwamy dwa wierzchołki — a zatem na końcu pozostanie nam co najmniej $n/3$ wierzchołków, wobec czego algorytm faktycznie jest poprawny.

Implementacja

Naiwna implementacja algorytmu wzorcowego w każdym kroku przegląda wszystkie pary wierzchołków w poszukiwaniu jakiejś, którą można by usunąć — jako że par wierzchołków jest $O(n^2)$, a kroków — $O(n)$, to łączna złożoność takiego algorytmu to $O(n^3)$. Nietrudno jednak zobaczyć, jak tę złożoność poprawić: każdą parę wierzchołków warto przeglądać tylko raz. Faktycznie, jeżeli przy pierwszym przejrzeniu danej pary wierzchołków nie usuwamy z grafu, to znaczy, że są one połączone krawędzią, więc już nigdy nie będziemy mogli tej pary usunąć. Wobec tego możemy utrzymywać dla każdego wierzchołka informację, czy wciąż znajduje się on w grafie, czy też nie, a następnie przeglądać po kolei wszystkie pary wierzchołków. Jeśli dwa wierzchołki nie są połączone krawędzią oraz żadnego z nich jeszcze nie usunęliśmy z grafu, to usuwamy obydwa. To rozwiązanie ma złożoność czasową $O(n^2)$, a zatem liniową względem rozmiaru grafu, tzn. liczby krawędzi. Można się spodziewać, że dla $n \leqslant 3\,000$ takie rozwiązanie będzie działało dostatecznie szybko. Implementacje tego rozwiązania można znaleźć w plikach `imp.cpp`, `imp1.pas` i `imp2.cpp`.

Uwagi

Czytelnika, który trochę interesuje się teorią grafów, niniejsze zadanie może nieco zaskoczyć. Dość znanym faktem jest to, że problem rozstrzygnięcia, czy w danym grafie istnieje klika danego rozmiaru, albo też znalezienia kliki o danym rozmiarze, nawet jeśli wiadomo, że ona istnieje, jest *NP-trudny* — co oznacza, że nie jest znany algorytm o złożoności wielomianowej, który rozwiązywałby ten problem, i byłoby sporym zaskoczeniem, gdyby taki algorytm istniał.

Co więcej, NP-trudny jest również problem aproksymacji kliki, tj. znalezienia w grafie kliki, która byłaby co najwyżej C razy mniejsza od największej istniejącej w tym grafie. Co może być zaskakujące, problem ten jest trudny nie tylko dla ustalonej liczby C, jak na przykład 2, ale też dla C zależnego od liczby n wierzchołków w grafie, np. dla $C(n) = \sqrt{n}$. Przykładowo, bardzo trudno jest znaleźć w grafie klikę rozmiaru $n^{1/4}$, nawet jeśli wiemy, że jest w nim klika rozmiaru $n^{3/4}$. W tym kontekście treść zadania może wydawać się bardzo zaskakująca, jako że przybliżamy w nim — co prawda w bardzo specyficznym przypadku — maksymalną klikę z dokładnością co do połowy jej rozmiaru.

Testy

Programy zgłoszone przez zawodników były sprawdzane na jedenastu zestawach testowych. Każdy zestaw składał się z pięciu testów.

W każdym z zestawów graf zawiera klikę rozmiaru $2n/3$, zaś pozostałe wierzchołki są połączone ze sobą w sposób pseudolosowy i zależny od oznaczenia testu:

- Testy oznaczone literą *a* zawierają kilka wierzchołków o bardzo wysokich stopniach, które znają dużą część (ale nie wszystkie) wierzchołki kliki, choć same do kliki nie należą. Pozostałe wierzchołki są połączone losowo.
- W testach oznaczonych literami *b*, *c*, *d* oraz *e* większość wierzchołków spoza głównej kliki jest również połączona w klikę, krawędzie pomiędzy klikami są mniej więcej losowe, a do tego jest dodanych kilka wierzchołków izolowanych.

Nazwa	n	m
imp1a.in	30	297
imp1b.in	48	784
imp1c.in	99	3 465
imp1d.in	99	3 465
imp1e.in	99	3 429
imp2a.in	198	13 357
imp2b.in	498	96 287
imp2c.in	498	96 287
imp2d.in	498	96 287
imp2e.in	498	95 864
imp3a.in	399	54 459
imp3b.in	999	387 154
imp3c.in	999	387 154
imp3d.in	999	387 154
imp3e.in	999	385 937
imp4a.in	600	123 293
imp4b.in	1 500	870 939
imp4c.in	1 500	870 939
imp4d.in	1 500	870 939
imp4e.in	1 500	873 750
imp5a.in	900	277 948
imp5b.in	1 800	1 255 786
imp5c.in	1 800	1 255 786
imp5d.in	1 800	1 255 786
imp5e.in	1 800	1 257 899
imp6a.in	1 500	781 719
imp6b.in	1 998	1 547 054
imp6c.in	1 998	1 547 054
imp6d.in	1 998	1 547 054
imp6e.in	1 998	1 547 436
imp7a.in	2 001	1 383 666
imp7b.in	2 298	2 049 890
imp7c.in	2 298	2 049 890
imp7d.in	2 298	2 049 890
imp7e.in	2 298	2 051 731
imp8a.in	2 298	1 808 780
imp8b.in	2 499	2 421 079
imp8c.in	2 499	2 421 079
imp8d.in	2 499	2 421 079
imp8e.in	2 499	2 421 510
imp9a.in	2 499	2 141 401
imp9b.in	2 700	2 829 686
imp9c.in	2 700	2 829 686
imp9d.in	2 700	2 829 686
imp9e.in	2 700	2 824 605
imp10a.in	2 700	2 504 388
imp10b.in	2 898	3 257 199
imp10c.in	2 898	3 257 199
imp10d.in	2 898	3 257 199
imp10e.in	2 898	3 255 867
imp11a.in	3 000	3 087 131
imp11b.in	3 000	3 494 958
imp11c.in	3 000	3 494 958
imp11d.in	3 000	3 494 958
imp11e.in	3 000	3 493 490

Wojciech Rytter, Bartosz Szreder
Treść zadania, Opracowanie

Dawid Dąbrowski
Program

Dostępna pamięć: 128 MB.

OI, Etap III, dzień pierwszy, 06.04.2011

Inspekcja

Sieć kolejowa Bajtockich Kolei Bitowych (BKB) składa się z dwukierunkowych odcinków torów łączących pewne pary stacji. Każda para stacji jest połączona co najwyżej jednym odcinkiem torów. Ponadto wiadomo, że z każdej stacji kolejowej można dojechać do każdej innej dokładnie jedną trasą. (Trasa może być złożona z kilku odcinków torów, ale nigdy nie przechodzi przez żadną stację więcej niż raz).

Bajtazar jest tajnym inspektorem BKB. W celu przeprowadzenia inspekcji wybiera jedną ze stacji (oznaczmy ją S), w której organizuje sobie centralę, i rozpoczyna podróż po wszystkich innych stacjach. Podróż ma następujący przebieg:

- *Bajtazar zaczyna na stacji S.*
- *Następnie wybiera jedną ze stacji jeszcze nie skontrolowanych i przemieszcza się do niej po najkrótszej ścieżce, dokonuje tam inspekcji i wraca do S.*
- *Nieuczciwi pracownicy BKB ostrzegają się nawzajem o przyjeździe Bajtazara. Aby ich zmylić, następną stację do skontrolowania Bajtazar wybiera w taki sposób, aby wyjechać w inną stronę ze stacji S niż poprzednio, tzn. innym odcinkiem torów wychodzącym z S.*
- *Każda stacja (poza stacją S) jest kontrolowana dokładnie raz.*
- *Po skontrolowaniu ostatniej stacji Bajtazar* **nie wraca** *do S.*

Przejazd każdym odcinkiem torów trwa tyle samo — jedną godzinę.

Bajtazar pragnie rozważyć wszystkie możliwe stacje jako stacje początkowe S. Dla każdej z nich chcemy wyznaczyć kolejność, w jakiej Bajtazar powinien kontrolować pozostałe stacje, tak aby łącznie jak najmniej czasu spędził na przejazdach, o ile dla danej stacji S w ogóle taka kolejność istnieje.

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą n ($1 \leqslant n \leqslant 1\,000\,000$), oznaczającą liczbę stacji. Stacje są ponumerowane od 1 do n. Kolejne $n-1$ wierszy opisuje odcinki torów, po jednym w wierszu. W każdym z nich znajdują się dwie liczby całkowite a, b ($1 \leqslant a, b \leqslant n$, $a \neq b$), oddzielone pojedynczym odstępem, oznaczające, że istnieje odcinek torów łączący stacje a i b. Każdy odcinek torów pojawia się w opisie dokładnie raz.

W testach wartych przynajmniej 30% punktów zachodzi dodatkowy warunek $n \leqslant 2\,000$.

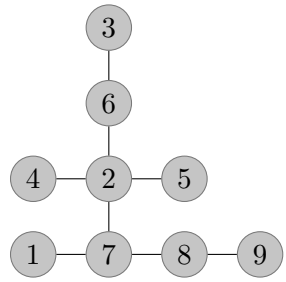
Wyjście

Twój program powinien wypisać na standardowe wyjście n wierszy, a w każdym z nich po jednej liczbie całkowitej. Liczba w i-tym wierszu powinna być równa minimalnej liczbie godzin, jakie Bajtazar musi spędzić na przejazdach, aby skontrolować stacje, dla $S = i$ — o ile dla $S = i$ szukana kolejność stacji istnieje. W przeciwnym przypadku, w i-tym wierszu powinna znaleźć się liczba -1.

Przykład

Dla danych wejściowych:

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



poprawnym wynikiem jest:

```
-1
23
-1
-1
-1
-1
-1
-1
-1
```

Rysunek pokazuje przykładową sieć połączeń. Szukana kolejność, w jakiej Bajtazar powinien kontrolować stacje, istnieje tylko dla $S = 2$. Może to być na przykład: 7, 4, 8, 6, 1, 5, 3, 9. Przy takiej kolejności kontrolowanych stacji Bajtazar spędzi na przejazdach łącznie 23 godziny.

Rozwiązanie

Wprowadzenie

Przetłumaczmy najpierw treść zadania na język matematyki. Dane jest *drzewo* T o n wierzchołkach, czyli spójny n-wierzchołkowy graf nieskierowany niezawierający żadnych cykli. Definiujemy *przechadzkę* po drzewie T rozpoczynającą się w wierzchołku S jako taką permutację $P = (v_1, v_2, \ldots, v_{n-1})$ pozostałych wierzchołków drzewa, że dla każdych dwóch kolejnych wierzchołków jedyna łącząca je ścieżka przechodzi przez S. Długością przechadzki P nazywamy liczbę

$$w(P) := d(S, v_1) + d(v_1, S) + d(S, v_2) + d(v_2, S) + \ldots + d(S, v_{n-1}).$$

Przez $d(a, b)$ oznaczamy tu odległość między wierzchołkami a i b w drzewie T (odległość, czyli długość najkrótszej, a w wypadku drzewa jedynej, ścieżki). Zwróćmy uwagę, że w definicji nie ma składnika $d(v_{n-1}, S)$.

Celem zadania jest stwierdzenie, dla każdego wierzchołka w drzewie T, czy istnieje przechadzka rozpoczynająca się w tym wierzchołku, a jeśli tak, to jaka jest jej

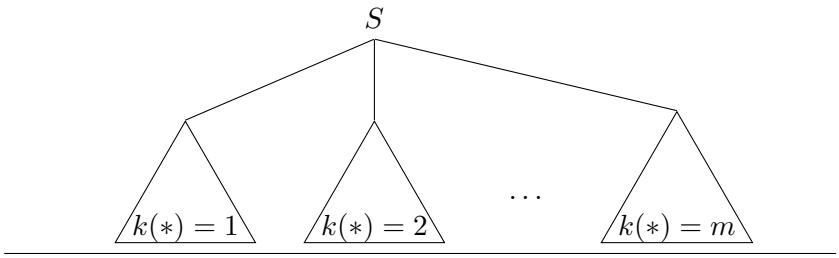
minimalna długość. Ograniczenia zadania wymagają rozwiązania działającego w złożoności czasowej liniowej lub ewentualnie nieznacznie gorszej.

Krok po kroku do rozwiązania wzorcowego

Analizujemy prostszy problem, czyli ustalamy S

Rozwiązanie wzorcowe opiszemy, rozpoczynając od przeanalizowania prostszego problemu. Załóżmy, że mamy jako początek przechadzki rozważyć tylko jeden, ustalony wierzchołek S, a przy tym chcemy tylko stwierdzić, czy taka przechadzka istnieje, bez podawania minimalnej długości.

Wierzchołki do odwiedzenia tworzą zbiór $V_S = \{1, \ldots, n\} \setminus \{S\}$. Niech m oznacza liczbę sąsiadów wierzchołka S, ponumerujmy ich liczbami od 1 do m. Niech wreszcie $k(x)$ będzie (dla $x \in V_S$) numerem sąsiada wierzchołka S, przez którego prowadzi ścieżka z S do x, patrz rys. 1.



Rys. 1: Ilustracja drzewa ukorzenionego w S oraz przyporządkowania k.

Weźmy multizbiór (czyli zbiór elementów wraz z dodatnią *krotnością* występowania każdego z nich) $C = \{k(x) : x \in V_S\}$. Przechadzka istnieje wtedy i tylko wtedy, gdy elementy multizbioru C można ustawić w ciąg, tak aby dwie równe liczby nigdy nie stały na sąsiednich pozycjach (takie uporządkowanie nazwiemy *prawidłowym*). Permutacja $(v_1, \ldots, v_{n-1})$ zbioru V_S jest bowiem przechadzką dokładnie wtedy, gdy ciąg $k(v_1), \ldots, k(v_{n-1})$ jest prawidłowym uporządkowaniem C.

Aby zwięźle opisać, kiedy multizbiór da się prawidłowo uporządkować, wprowadźmy pojęcie *lidera*, czyli elementu o maksymalnej krotności. Jeśli istnieje kilka różnych elementów występujących największą liczbę razy, każdy z nich jest liderem.

Twierdzenie 1. *Jeśli $n \geqslant 2$, $A = \{a_1, a_2, \ldots, a_{n-1}\}$ jest multizbiorem liczb, zaś N liczbą wystąpień lidera w tym ciągu, to A da się prawidłowo uporządkować wtedy i tylko wtedy, gdy $2N \leqslant n$.*

Jeśli zachodzi ostra nierówność, owo prawidłowe uporządkowanie może mieć jako ostatni wyraz dowolny element $a \in A$. W przeciwnym razie ostatni wyraz musi być liderem.

Dowód: Na razie wykażemy indukcyjnie pozytywną część twierdzenia, tzn. istnienie odpowiednich ciągów. Dla $n = 2$ teza jest jasna.

Niech więc $n \geqslant 3$. Udowodnimy tezę dla n, korzystając z jej prawdziwości dla $n' = n - 1$. Jeśli $2N < n$, niech $a \in A$ będzie dowolne. Wówczas multizbiór

$A' = A \setminus \{a\}$ ma $n' - 1$ elementów, zaś jego lider występuje $N' \leqslant N$ razy. Wobec tego $2N' \leqslant 2N \leqslant n - 1 = n'$, a więc A' da się prawidłowo uporządkować.

Przy tym, jeśli a był liderem A, to albo $N' < N$ i uporządkowanie A' może mieć dowolny ostatni wyraz, albo $N' = N$, lecz a nie jest liderem A'. Jeśli zaś a nie był liderem A, to nie jest też liderem A' i $N' = N$. W obydwu przypadkach można prawidłowo uporządkować A', tak aby ostatni wyraz był różny od a, co pozwala stworzyć prawidłowe uporządkowanie A kończące się na a (które wybraliśmy jako dowolny element A).

Jeśli $2N = n$, niech a będzie liderem. Łatwo widać, że jest to wówczas jedyny lider, czyli multizbiór $A' = A \setminus \{a\}$ ma $n' - 1$ elementów, zaś jego lider występuje $N' = N - 1$ razy. Stąd $2N' < 2N - 1 = n - 1 = n'$, czyli A' można prawidłowo uporządkować tak, aby ostatni wyraz był dowolny, w szczególności różny od a. Zatem istnieje prawidłowe uporządkowanie A, którego ostatnim wyrazem jest lider.

Przejdźmy teraz do dowodu części negatywnej. Niech ℓ będzie (dowolnym) liderem A. Rozważmy hipotetyczne prawidłowe uporządkowanie A. Po każdym wystąpieniu ℓ, być może z wyjątkiem jednego, na ostatniej pozycji, musi być inny wyraz ciągu. Stąd liczba wszystkich wyrazów (czyli $n - 1$) musi wynosić przynajmniej $N + N - 1$, czyli $2N \leqslant n$. Jeśli zaś ℓ nie występuje na ostatniej pozycji, to przynajmniej $N + N$, skąd $2N < n$. ■

Mając tak udowodnione twierdzenie, możemy śmiało stwierdzić, że przechadzka z wierzchołka S istnieje wtedy i tylko wtedy, gdy rozmiar największego poddrzewa zaczepionego w pewnym synu S nie przekracza $\frac{n}{2}$.

Możemy ukorzenić graf w wierzchołku S i przeszukać go (np. w głąb), licząc wielkości poddrzew zaczepionych we wszystkich wierzchołkach w czasie $O(n)$, a następnie sprawdzić, czy największe z nich spełnia podane ograniczenie.

Utrudniamy problem – zbliżamy się do rozwiązania

Rozwiązaliśmy prostszy problem, teraz pora go utrudnić. Zapytajmy o minimalną możliwą długość przechadzki przy ustalonym S i założeniu, że taka przechadzka istnieje. Spójrzmy na definicję długości przechadzki. Możemy ją zapisać prościej:

$$w(P) = 2D - d(S, v_{n-1}),$$

gdzie D jest sumą odległości od wierzchołka S do pozostałych wierzchołków drzewa.

Zauważmy, że dla ustalonego punktu początkowego przechadzki wartość D jest stała. Widzimy więc, że chcielibyśmy skończyć przechadzkę w wierzchołku położonym jak najdalej od S.

Zastanówmy się, w jakich wierzchołkach możemy skończyć przechadzkę. Z pomocą przychodzi udowodnione twierdzenie, które w naszym języku mówi, że ostatni wierzchołek przechadzki może być dowolny, o ile tylko nie zachodzi równość w ograniczeniu. W przeciwnym przypadku jesteśmy zobligowani skończyć w największym poddrzewie.

Możemy więc sprawdzić, która z sytuacji ma miejsce, uruchomić ponownie przeszukiwanie grafu i obliczyć odległość od S do najdalszego wierzchołka w drzewie bądź do najdalszego wierzchołka w największym poddrzewie.

Ten algorytm działa w czasie $O(n)$, a zatem na jego podstawie jesteśmy w stanie rozwiązać całe zadanie w czasie $O(n^2)$ (po kolei ustalając $S = 1, 2, \ldots, n$). Takie

rozwiązanie zdobywało około 30 punktów. Jego implementacja znajduje się w plikach `inss0.cpp` oraz `inss1.pas`.

Rozwiązania wzorcowe

Przyspieszamy rozwiązanie nieoptymalne

Rozwiązanie wzorcowe opiera się na tych samych spostrzeżeniach, jednak istotnie szybciej oblicza dla każdego wierzchołka wielkości poddrzew oraz najdalsze wierzchołki w odpowiednich poddrzewach.

Ukorzeńmy drzewo w dowolnym wybranym wierzchołku, np. tym o numerze 1. Na początek obliczymy dla każdego wierzchołka rozmiary poddrzew.

Możemy rekurencyjnie wyznaczyć dla każdego u rozmiary poddrzew zaczepionych w *synach* u (według hierarchii ustalonej względem wierzchołka wybranego na samym początku). Najniższym poziomem rekursji będzie liść drzewa, który nie ma żadnych synów.

Dla każdego wierzchołka u zostaje zatem do wyznaczenia tylko rozmiar poddrzewa, które, gdy spojrzymy od strony u, jest zaczepione w rodzicu u. Jeśli suma rozmiarów poddrzew zaczepionych w synach u wynosi $t(u)$, szukany rozmiar to oczywiście $n-t(u)-1$. Daje to liniowy algorytm obliczania rozmiarów poddrzew wszystkich wierzchołków w drzewie.

Teraz zajmijmy się wyznaczaniem najdalszego wierzchołka dla każdego z wierzchołków drzewa. Podzielimy tę procedurę na dwie fazy. W pierwszej znajdziemy najdalszego *potomka* każdego wierzchołka. Można to zrealizować w czasie $O(n)$ dla całego drzewa, w podobny sposób, w jaki wyznaczaliśmy wielkości poddrzew. Oznaczmy najdalszego potomka wierzchołka u przez $A(u)$.

Najbardziej odległy punkt od danego wierzchołka u nie musi być jednak jego potomkiem. W tym przypadku ścieżka od u do tego wierzchołka składa się z kilku krawędzi w górę drzewa (co najmniej jednej), a następnie z pewnej ścieżki w dół drzewa, rozłącznej krawędziowo ze ścieżką w górę drzewa. Taki najdalszy wierzchołek oznaczmy przez $B(u)$.

Uruchomimy drugie przeszukiwanie w głąb, które będzie korzystało z wyników pierwszego. Dla korzenia znamy wynik, gdyż nie ma wierzchołków wyżej od niego. Załóżmy teraz, że jesteśmy w pewnym wierzchołku u oraz że znamy wynik dla niego, natomiast chcemy poznać wynik dla pewnego syna v. Możliwe są dwa przypadki.

1. Ścieżka z v do $B(v)$ biegnie przez u dalej *w górę*. W tym wypadku $B(v) = B(u)$.
2. Ścieżka z v do $B(v)$ prowadzi krawędzią z v do u, a następnie *w dół* z u do $B(v)$. W tym wypadku $B(v) = A(u)$, chyba że v oraz $A(u)$ leżą w tym samym poddrzewie, wówczas musimy sprawdzić wierzchołki $A(w)$ dla w będących synami u różnymi od v i wybrać najgłębszy z nich. Oczywiście, taką bardziej kosztowną operację trzeba wykonać tylko dla dokładnie jednego z synów u.

Korzystając z tych spostrzeżeń, wartości B możemy policzyć w czasie $O(n)$ dla wszystkich wierzchołków drzewa, o ile będziemy także pamiętali odległości każdego wierzchołka u od wierzchołków $A(u)$ i $B(u)$.

Brakuje nam jeszcze tylko wartości $D(u)$, czyli sum odległości. Oznaczmy przez T_u drzewo ukorzenione w wierzchołku u. W przeszukiwaniu w głąb możemy posłużyć się następującym spostrzeżeniem: przechodząc krawędzią między wierzchołkami u oraz v, oddalamy się o 1 od wszystkich wierzchołków z poddrzewa u w drzewie T_v oraz przybliżamy o 1 do wszystkich z poddrzewa v w drzewie T_u. Możemy więc dla wierzchołka numer 1 obliczyć sumę odległości prostym przeszukiwaniem drzewa, a następnie uruchomić kolejne przeszukiwanie, które będzie, korzystając z powyższego spostrzeżenia, liczyć wartości $D(u)$ dla wszystkich wierzchołków grafu w czasie $O(n)$.

Mając tak policzone wartości dla wierzchołków drzewa, możemy prosto odpowiadać na pytania o istnienie przechadzki oraz jej minimalną długość. Rozwiązanie to zostało zaimplementowane w plikach `ins.cpp` oraz `ins2.pas`.

Po co się męczyć?

Uruchamiając rozwiązanie (chociażby siłowe) na różnych testach, można dostrzec pewną prawidłowość w generowanych odpowiedziach. Zazwyczaj tylko jeden wierzchołek drzewa spełnia nierówność przedstawioną we wcześniejszym twierdzeniu, a w niektórych przypadkach są dwa takie wierzchołki. Okazuje się, że dzieje się tak nieprzypadkowo.

Twierdzenie 2. *Niech T będzie dowolnym drzewem. Każde dwa wierzchołki T, które są początkami pewnych przechadzek, są sąsiadami. W szczególności, przechadzka w T istnieje dla co najwyżej dwóch wierzchołków początkowych.*

Dowód: Niech u i v będą dwoma różnymi wierzchołkami T, które stanowią początki pewnych przechadzek.

Niech T_1 oznacza zbiór wierzchołków, z których ścieżka do u *nie* wiedzie przez v. Zauważmy, że po ukorzenieniu drzewa w wierzchołku v zbiór T_1 stanowi zbiór wierzchołków poddrzewa zaczepionego w u. Wobec tego $|T_1| \leqslant \frac{n}{2}$. Analogicznie niech T_2 będzie zbiorem wierzchołków, z których ścieżka do v nie wiedzie przez u. Podobnie jak poprzednio, mamy $|T_2| \leqslant \frac{n}{2}$.

Zauważmy, że w drzewie nie jest możliwe, aby z pewnego wierzchołka ścieżka do u prowadziła przez v, a ścieżka do v przez u. Oznacza to, że każdy wierzchołek drzewa T leży w przynajmniej jednym ze zbiorów T_1, T_2. Ponieważ oba liczą po co najwyżej $\frac{n}{2}$ elementów, więc musi zachodzić $|T_1| = \frac{n}{2}$ i $|T_2| = \frac{n}{2}$, a ich część wspólna musi być pusta. Do tej części wspólnej należą jednak wszystkie, poza końcami, wierzchołki ścieżki z u do v. Stąd wniosek, że u i v to sąsiedzi.

Każdy co najmniej trójelementowy zbiór wierzchołków drzewa zawiera parę niesąsiadujących wierzchołków, więc druga część twierdzenia wynika bezpośrednio z pierwszej. ■

Mając takie twierdzenie, widzimy, że dla wszystkich wierzchołków wystarczy wyliczyć jedynie wielkości poddrzew. Jeśli na podstawie tych rozmiarów stwierdzimy, że że dla pewnego wierzchołka u istnieje przechadzka, możemy uruchomić przeszukiwanie w głąb w drzewie w nim ukorzenionym i łatwo policzyć potrzebne wartości $A(u)$, $B(u)$ i $D(u)$. Bardzo upraszcza to strukturę programu, o czym można się przekonać, spoglądając na rozwiązania zaimplementowane w plikach `ins1.cpp`, `ins3.pas` oraz `ins4.cpp`.

Testy

Większość zestawów składała się z testów należących do czterech następujących kategorii:

(*a*) Testy obalające rozwiązania korzystające z błędnych warunków na istnienie przechadzki. Składają się z korzenia, poddrzewa o wielkości $\frac{n}{2}$ oraz losowej reszty grafu.

(*b*) Testy obalające rozwiązanie zakładające, że odpowiedź dla jednego tylko wierzchołka jest niezerowa. Składają się z dwóch wierzchołków połączonych krawędzią oraz dwóch poddrzew tej samej wielkości doczepionych do każdego z nich.

(*c*) Testy obalające rozwiązanie zakładające, że zawsze kończymy w najgłębszym wierzchołku. Składają się z płytkiego poddrzewa o wielkości $\frac{n}{2}$ oraz długiej ścieżki w drugim poddrzewie.

(*d*) Losowo generowane drzewa, w których najdłuższe ścieżki są rzędu $\sqrt{n}$.

Nazwa	n	Opis
ins1[abcd].in	20	testy typu (*a*)-(*d*)
ins1e.in	1	przypadek brzegowy
ins1f.in	2	przypadek brzegowy
ins2[abcd].in	80	testy typu (*a*)-(*d*)
ins3[abcd].in	2 000	testy typu (*a*)-(*d*)
ins4[abcd].in	20 000	testy typu (*a*)-(*d*)
ins5[abcd].in	39 514	testy typu (*a*)-(*d*)
ins6[abcd].in	100 000	testy typu (*a*)-(*d*)
ins7[abcd].in	500 000	testy typu (*a*)-(*d*)
ins8[abcd].in	1 000 000	testy typu (*a*)-(*d*)
ins9[abcd].in	1 000 000	testy typu (*a*)-(*d*)
ins10[abcd].in	1 000 000	testy typu (*a*)-(*d*)

Wojciech Rytter
Treść zadania, Opracowanie

Marcin Pilipczuk
Program

Dostępna pamięć: 32 MB.

OI, Etap III, dzień pierwszy, 06.04.2011

Okresowość

Bajtazar, król Bitocji, zarządził reformę nazwisk swoich poddanych. Nazwiska mieszkańców Bitocji często zawierają powtarzające się frazy, np. w nazwisku `Abiabuabiab` *dwukrotnie występuje fragment* `abiab`*. Bajtazar chce zamienić nazwiska swoich poddanych na ciągi bitów takiej samej długości jak ich oryginalne nazwiska. Chciałby przy tym w jakimś stopniu zachować sposób, w jaki w oryginalnych nazwiskach powtarzają się te same frazy.*

W dalszej części zadania, dla prostoty, będziemy utożsamiać wielkie i małe litery w nazwiskach. Dla dowolnego ciągu znaków (liter lub bitów) $w = w_1 w_1 \ldots w_k$ powiemy, że liczba naturalna p $(1 \leqslant p < k)$ jest okresem w, jeżeli $w_i = w_{i+p}$ dla wszystkich $i = 1, \ldots, k-p$. Przez $Okr(w)$ będziemy oznaczać zbiór wszystkich okresów w. Na przykład, $Okr($`ABIABUABIAB`$) = \{6, 9\}$, $Okr($`01001010010`$) = \{5, 8, 10\}$ *oraz* $Okr($`0000`$) = \{1, 2, 3\}$.

Bajtazar zdecydował, że każde nazwisko ma zostać zamienione na ciąg bitów:

- *tej samej długości co oryginalne nazwisko,*
- *o dokładnie takim samym zbiorze okresów co oryginalne nazwisko,*
- *ma to być najmniejszy (w porządku leksykograficznym[1]) ciąg bitów spełniający powyższe warunki.*

Na przykład, nazwisko `ABIABUABIAB` *powinno zostać zamienione na* `01001101001`, `BABBAB` *na* `010010`*, a* `BABURBAB` *na* `01000010`*.*

Bajtazar poprosił Cię o napisanie programu, który pomógłby w tłumaczeniu dotychczasowych nazwisk jego poddanych na nowe. W nagrodę będziesz mógł zachować swoje oryginalne nazwisko!

Wejście

W pierwszym wejściu standardowego wejścia znajduje się jedna liczba całkowita k — liczba nazwisk do przetworzenia $(1 \leqslant k \leqslant 20)$. Nazwiska są podane w kolejnych wierszach, po jednym w wierszu. Każde z nazwisk składa się z od 1 do 200 000 wielkich liter (alfabetu angielskiego).

W testach wartych łącznie 30% punktów każde nazwisko składa się z co najwyżej 20 liter.

Wyjście

Twój program powinien wypisać na standardowe wyjście k wierszy. W kolejnych wierszach powinny znajdować się ciągi zer i jedynek (niezawierające odstępów) odpowiadające kolejnym nazwiskom z wejścia. W przypadku, gdy dla danego nazwiska nie istnieje ciąg bitów zgodny z warunkami zadania, należy dla tego nazwiska wypisać „`XXX`*" (bez cudzysłowów).*

[1] Ciąg bitów $x_1 x_2 \ldots x_k$ jest mniejszy w porządku leksykograficznym od ciągu bitów $y_1 y_2 \ldots y_k$, jeżeli dla pewnego i, $1 \leqslant i \leqslant k$, mamy $x_i < y_i$ oraz dla wszystkich $j = 1, \ldots, i-1$ mamy $x_j = y_j$.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`3`	`01001101001`
`ABIABUABIAB`	`010010`
`BABBAB`	`01000010`
`BABURBAB`	

Rozwiązanie

Informacją wejściową jest słowo w długości n, pierwszym krokiem jest obliczenie zbioru okresów $Okr(w)$ tego słowa, po czym właściwie zapominamy o słowie w. Zbiór ten można łatwo obliczyć, korzystając z algorytmu na wyznaczanie tablicy tzw. prefikso-sufiksów związanej z algorytmem Knutha-Morrisa-Pratta (szukania wzorca). Tablica ta wielokrotnie pojawiała się w zadaniach z Olimpiady Informatycznej, patrz także opracowania zadań *Szablon* z XII Olimpiady i *Palindromy* z XIII Olimpiady. W drugim z podanych opisów jest uzasadniona własność, że słowo w ma okres p wtedy i tylko wtedy, kiedy ma prefikso-sufiks długości $n-p$, tzn. prefiks słowa długości $n-p$ jest również sufiksem słowa. Zakładamy zatem odtąd, że znamy zbiór okresów słowa wejściowego. Inaczej niż w treści zadania przyjmiemy, że $n \in Okr(w)$.

Zbiory okresów mają wiele ciekawych własności, w szczególności zachodzi następująca implikacja (*nwd* oznacza tutaj *najmniejszy wspólny dzielnik*).

Lemat 1 (o okresowości). $p, q \in Okr(w)$ *oraz* $p + q \leqslant n \;\Rightarrow\; nwd(p, q) \in Okr(w)$.

Lemat ten wystąpił już poprzednio w rozwiązaniach zadań olimpijskich, np. w drugim z wyżej wymienionych.

Tekst u nazywamy **pierwotnym** albo *nierozkładalnym*, gdy u nie ma okresu będącego jego właściwym dzielnikiem (mniejszym od długości u), zapisujemy to jako funkcję logiczną $Pierwotny(u)$. Na przykład $Pierwotny($`1010`$) =$ **false**, $Pierwotny($`1011`$) =$ **true**. Pozostawiamy Czytelnikowi jako proste ćwiczenie wyprowadzenie z lematu o okresowości następującego faktu.

Lemat 2 (o słowach pierwotnych). *Jeśli słowo nie jest pierwotne, to zmiana pojedynczego symbolu na dowolnej pozycji zamienia to słowo na pierwotne.*

Zamiast przetwarzać okresy, wygodniej robić to dla niepustych prefikso-sufiksów tekstu. Zakładamy, że cały tekst też jest swoim prefikso-sufiksem.

Oznaczmy ciąg długości kolejnych prefikso-sufiksów słowa w, posortowany rosnąco, przez $PS(w) = (p_1, p_2, \ldots, p_k)$, w szczególności $p_k = n$. Ciąg $PS(w)$ łatwo obliczyć, znając zbiór okresów, zachodzi bowiem: $Okr(w) = Okr(v) \;\Leftrightarrow\; PS(w) = PS(v)$.

Przykład 1. $Okr(w) = \{10,\ 20,\ 25,\ 27\} \;\Longrightarrow\; PS(w) = (2,\ 7,\ 17,\ 27)$.

Oznaczmy leksykograficznie minimalne słowo, którego ciągiem długości prefikso-sufiksów jest $(p_1, p_2, \ldots, p_i)$, przez:

$$P_i \;=\; MinLex(p_1, p_2, \ldots, p_i).$$

Słowo to jest prefikso-sufiksem wyniku długości p_i.

Opis algorytmu

Zasadnicza koncepcja algorytmu jest naturalna: ze względu na leksykograficzną minimalność narzuca się dopychanie tekstu jak największą liczbą zer. Takie podejście łatwo wymyślić na intuicję i zastosować, pomimo tego, że poprawność nie jest oczywista.

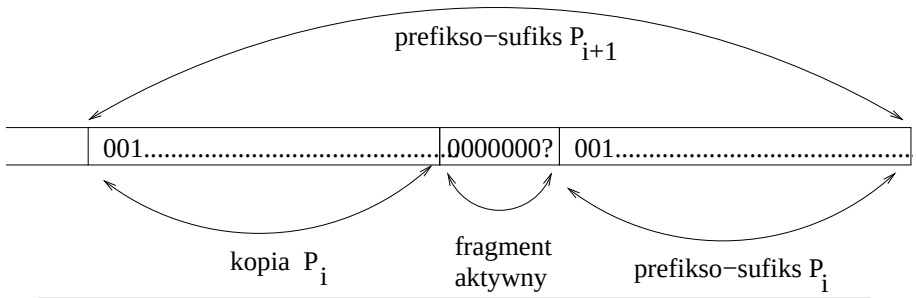
Algorytm opiera się na *zachłannym leksykograficznie* wpisywaniu fragmentów postaci 0^j lub $0^{j-1}1$ w te fragmenty tekstu, które nie są jednoznacznie wyznaczone przez prefikso-sufiksy w następującym sensie: każdy P_{i+1} musi zaczynać się od słowa P_i i kończyć się słowem P_i.

Początkowo mamy tekst długości n, którego wszystkie pola są „puste". Wypełniamy kolejne puste pola od końca, wpisując prefikso-sufiksy, poczynając od najkrótszego z nich, tzn. P_1.

Dla $PS = (p_1, p_2, \ldots, p_k)$, słowo P_{i+1} konstruujemy, startując od P_i i dopisując na początku P_i lub jego prefiks, jeśli jest za mało miejsca na całe P_i, patrz rysunek 1. Niezapełnione pola tekstu zastępujemy leksykograficznie pierwszym ciągiem bitów Δ, który nie wygeneruje nadmiarowego prefikso-sufiksu.

Obserwacja 1. Najdziwniejszą własnością algorytmu jest to, że wystarczy zawsze sprawdzić tylko dwie możliwości na brakujący fragment: $\Delta \in \{0^j,\ 0^{j-1}1\}$ dla $j = p_{i+1} - 2p_i$.

Dzięki tej obserwacji możliwy jest algorytm działający w czasie liniowym. Podstawowym elementem algorytmu jest sprawdzenie, którą z opcji na Δ wybrać. Można to zrealizować na wiele sposobów, w naszej wersji korzystamy z funkcji *Pierwotny*, inną możliwością jest sprawdzenie dla każdej alternatywy wprost, czy nie generuje się nadmiarowy prefikso-sufiks.



Rys. 1: Konstrukcja kolejnego prefikso-sufiksu w sytuacji, gdy $2p_i < p_{i+1}$. Końcową częścią tekstu jest P_i, zapełniamy P_{i+1}, z definicji prefikso-sufiksu prefiks P_{i+1} długości p_i jest równy P_i, wolne pola (fragment aktywny) zapełniamy jak największą liczbą zer, w miejsce '?' wstawiamy 0 lub 1.

Przykład 2. Opiszemy, w jaki sposób konstruujemy minimalne leksykograficznie słowo $MinLex(PS(w))$ dla naszego przykładowego ciągu prefikso-sufiksów $PS(w) = (2, 7, 17, 27)$:

Krok 1. Konstruujemy $MinLex(2) = \texttt{01}$.

Krok 2. Wiemy, że $MinLex(2,7) =$ `01???01`. Próbujemy wstawić minimalny leksykograficznie ciąg `000`, zastępując znaki '`?`'. Otrzymujemy $MinLex(2,7) =$ `0100001`, tekst ten jest zgodny z ciągiem prefikso-sufiksów $2, 7$.

Krok 3. $MinLex(2,7,17) = MinLex(2,7)$`???`$MinLex(2,7)$. Jeśli zastąpimy znaki '`?`' przez `000`, to otrzymany tekst `01000 01000 01000 01` ma nadmiarowy prefikso-sufiks długości 12. Zatem próbujemy zastąpić '`???`' przez następny w kolejności minimalnej ciąg `001`. Tym razem otrzymany tekst jest zgodny z ciągiem prefikso-sufiksów $2, 7, 17$. Zatem $MinLex(2,7,17) =$ `01000010010100001`.

Krok 4. Ponieważ $27 - 17 < 17$, więc wiemy, że wynikiem jest tekst, którego prefiksem i sufiksem jest $MinLex(p_1, \ldots, p_{i+1})$. Zatem końcowym wynikiem jest:

$$\begin{aligned} MinLex(2,7,17,27) &= \texttt{0100001001}\ MinLex(2,7,17) \\ &= \texttt{0100001001 01000010010100001}. \end{aligned}$$

Podaną metodę ilustruje poniższy pseudokod.

```
1:  function MinLex(p_1, p_2, ..., p_k)
2:  begin
3:    if p_1 = 1 then P_1 := 0 else P_1 := 0^{p_1 - 1}1;
4:    for i := 1 to k - 1 do begin
5:      j := p_{i+1} - 2p_i;
6:      if j ⩽ 0 then P_{i+1} := vP_i  { v jest prefiksem P_i długości p_{i+1} - p_i }
7:      else if Pierwotny(P_i 0^j) then P_{i+1} := P_i 0^j P_i
8:      else P_{i+1} := P_i 0^{j-1}1 P_i;
9:    end
10:   return P_k;
11: end
```

Poprawność algorytmu

Chcemy pokazać indukcyjnie (po i), że zbiorem prefikso-sufiksów skonstruowanego przez nas słowa P_i jest właśnie $\{p_1, p_2, \ldots, p_i\}$. Zauważmy, że w kroku indukcyjnym dla P_{i+1} wystarczy pokazać, że to słowo ma prefikso-sufiks długości p_i oraz że nie ma dłuższych nietrywialnych prefikso-sufiksów.

Na początku rozważmy przypadek $j > 0$. Załóżmy, że wówczas P_i zawiera co najmniej jedną jedynkę — w przeciwnym przypadku $P_i = 0^{p_i}$ i nie ma czego dowodzić. Poprawność algorytmu wynika z dwóch następujących faktów.

Fakt 1. *Niech $j > 0$ oraz P będzie dowolnym binarnym tekstem zawierającym co najmniej jedną jedynkę. Słowo $u = P\,0^j P$ ma (nadmiarowy) prefikso-sufiks o długości q, $|P| < q < |u|$, wtedy i tylko wtedy, gdy $P\,0^j$ nie jest pierwotny.*

Dowód: Przypuśćmy, że słowo u ma nadmiarowy prefikso-sufiks P' długości q. Ten prefikso-sufiks nie może się zacząć wewnątrz aktywnego fragmentu 0^j, bo P zawiera

jedynkę. Faktycznie, wówczas mielibyśmy $P' = P\ 0^i = 0^i P$ dla $i = q - |P|$, czyli P' miałoby okres długości i złożony z samych zer.

Z drugiej strony, jeśli P' zaczyna się w prefiksie P słowa u, to, na mocy lematu o okresowości, słowo u ma (mały) okres, który jest krótszy od $P\,0^j$ i jest dzielnikiem długości tego słowa. Wynika to stąd, że tekst u miałby okresy o długościach $|P\ 0^j|$ oraz $|u| - q \leqslant |P|$. Długość tekstu u jest nie mniejsza od sumy tych okresów i można wtedy zastosować lemat o okresowości.

Zatem jeśli $P\,0^j\,P$ ma nadmiarowy prefikso-sufiks, to $P\,0^j$ nie jest pierwotne. Łatwo widać implikację odwrotną, wtedy u ma nadmiarowy prefikso-sufiks o długości $|u| - p$, gdzie p jest (małym) pełnym okresem $P\,0^j$. To kończy uzasadnienie faktu. ■

Fakt 2. *Jeśli $P\,0^j$ nie jest pierwotny, to $P\,0^{j-1}1$ jest pierwotny.*

Dowód: Wynika to z lematu o tekstach pierwotnych. ■

Aby zakończyć uzasadnienie w tym przypadku, wystarczy zauważyć, że fakt 1 zachodzi także dla słów postaci $u' = P\ 0^{j-1}1\ P$.

Teraz pozostał nam do rozpatrzenia przypadek, że $j \leqslant 0$. W tym celu wystarczy wykazać następujący fakt, implikujący krok indukcyjny w tym przypadku. W jego dowodzie wykorzystujemy to, że wyjściowy zbiór prefikso-sufiksów $\{p_1, p_2, \ldots, p_k\}$ odpowiada jakiemuś istniejącemu słowu, tj. początkowemu słowu w.

Fakt 3. *Załóżmy, że słowo P_i ma zbiór prefikso-sufiksów $\{p_1, p_2, \ldots, p_i\}$. Jeśli $p_{i+1} - 2p_i \leqslant 0$, to słowo $P_{i+1} = vP_i$ ma prefiks P_i oraz nie ma nadmiarowych prefikso-sufiksów długości q, $p_i < q < p_{i+1}$.*

Dowód: Najpierw pokażemy pierwszą część tezy, czyli że P_{i+1} ma prefikso-sufiks P_i. Niech w_i i w_{i+1} oznaczają sufiksy słowa w długości, odpowiednio, p_i i p_{i+1}. Wystarczy przeprowadzić następujące rozumowanie. Jeśli chcemy pokazać, że P_{i+1} ma prefikso-sufiks P_i, czyli równoważnie, że P_{i+1} ma okres $p_{i+1} - p_i$, to wystarczy uzasadnić, że P_i ma taki właśnie okres, gdyż P_{i+1} zaczyna się prefiksem słowa P_i długości $p_{i+1} - p_i$. Na mocy założenia faktu, słowo P_i ma dokładnie taki sam zbiór okresów jak w_i, więc wystarczy pokazać, że w_i ma okres $p_{i+1} - p_i$. Tak jednak jest, gdyż w_i jest prefikso-sufiksem słowa w_{i+1}, które całe ma, w związku z tym, okres $p_{i+1} - p_i$. Rozumowanie zakończone sukcesem.

Musimy jeszcze pokazać, że P_{i+1} nie może mieć nadmiarowego prefikso-sufiksu o długości q, $q > p_i$. Wówczas P_{i+1} miałoby okresy $p_{i+1} - q$ oraz $p_{i+1} - p_i$, więc na mocy lematu o okresowości (ponieważ $j \leqslant 0$) P_{i+1} miałoby okres d będący dzielnikiem tych dwóch wartości. Stąd, stosując rozumowanie takie jak w pierwszej części dowodu, pokazujemy, że d byłoby także okresem słowa u_{i+1}, a więc także $p_{i+1} - q$ byłoby okresem tego słowa, czyli q byłoby jego prefikso-sufiksem, co daje sprzeczność. ■

Złożoność algorytmu

Podstawowym problemem jest sprawdzenie, czy tekst $P_i\ 0^j$ jest pierwotny. W miarę konstruowania coraz dłuższych prefikso-sufiksów, a więc jednocześnie prefiksów całego tekstu wynikowego, budujemy on-line tablicę *wszystkich* najdłuższych prefikso-sufiksów z algorytmu Knutha-Morrisa-Pratta. Dzięki tej tablicy znamy najkrótszy

okres i możemy sprawdzić, czy dany prefiks jest pierwotny — za pomocą lematu o okresowości można pokazać, że słowo jest pierwotne wtedy i tylko wtedy, gdy jego najkrótszy okres jest nim samym lub nie dzieli jego długości (patrz także wspomniane już opracowanie zadania *Palindromy* w książce [13]).

Jeśli obliczyliśmy już tablicę prefikso-sufiksów dla $P_i\ 0^{j-1}$, to sprawdzenie, czy $P_i\ 0^j$ jest niepierwotne, wykonujemy w czasie stałym: można pokazać, że jeśli $P_i\ 0^j$ jest niepierwotne, to najdłuższy prefikso-sufiks słowa $P_i\ 0^{j-1}$ przedłuża się do prefikso-sufiksu słowa $P_i\ 0^j$. Jeśli odpowiedź jest pozytywna, to obliczamy prefikso-sufiks on-line dla $P_i\ 0^{j-1}1$, a jeśli negatywna, to dla $P_i\ 0^{j-1}0$. Inaczej mówiąc, koszt „cofania się" jest za każdym razem $O(1)$, w sumie liniowy. Pozostaje „na czysto" koszt liczenia on-line tablicy prefikso-sufiksów dla całego słowa, jest on liniowy.

Testy

Każdy test składał się z 20 przypadków testowych. W poniższej tabeli n to maksymalna długość słowa w teście.

Nazwa	n	Opis
okr1.in	14	mały test poprawnościowy, 10 pkt
okr2.in	17	mały test poprawnościowy, 10 pkt
okr3.in	20	mały test poprawnościowy, 10 pkt
okr4.in	200	średni test, 20 pkt
okr5.in	2 000	średni test, 20 pkt
okr6.in	200 000	duży test, 30 pkt

Tomasz Idziaszek
Treść zadania, Opracowanie

Paweł Parys
Program

Dostępna pamięć: 32 MB. *OI, Etap III, dzień drugi, 07.04.2011*

Konkurs programistyczny

Bartuś i jego koledzy startują w Drużynowym Konkursie Programistycznym. Każda drużyna składa się z n zawodników i ma do dyspozycji n komputerów. Zawody trwają t minut i w tym czasie zawodnicy mają do rozwiązania m zadań programistycznych. Dodatkowo, drużynom przyznawane są punkty karne — za rozwiązanie zadania po upływie s minut od początku konkursu drużyna otrzymuje s punktów karnych. Wygrywa ta drużyna, która rozwiąże największą liczbę zadań, a w przypadku remisu ta, która ma najmniej punktów karnych.

W dniu zawodów Bartuś szybko przegląda treści zadań i rozdziela je między kolegów. Zna on dobrze swoją drużynę i potrafi bezbłędnie ocenić, kto jest w stanie zrobić które zadanie. Każdemu zawodnikowi rozwiązanie zadania, które umie on zrobić, zajmuje dokładnie r minut pracy przy komputerze.

Drużynie Bartusia nie poszło zbyt dobrze na tegorocznej edycji Konkursu. Zastanawia się on, jaki wpływ na ten przykry fakt miały jego decyzje dotyczące przydziału zadań. Poprosił Cię, abyś napisał program, który na podstawie danych, jakie posiadał Bartuś na początku konkursu, obliczy maksymalny możliwy wynik drużyny Bartusia, a także przydział zadań członkom drużyny, który umożliwi osiągnięcie tego wyniku.

Wejście

W pierwszym wierszu standardowego wejścia znajduje się pięć liczb całkowitych n, m, r, t oraz k ($1 \leqslant n, m \leqslant 500$, $1 \leqslant r, t \leqslant 1\,000\,000$), pooddzielanych pojedynczymi odstępami i oznaczających odpowiednio: liczbę zawodników, liczbę zadań, czas rozwiązywania zadania przez zawodnika, czas trwania zawodów i liczbę par zawodnik–zadanie (podanych dalej na wejściu). Każdy z kolejnych k wierszy zawiera dwie liczby całkowite a i b ($1 \leqslant a \leqslant n$, $1 \leqslant b \leqslant m$), oddzielone pojedynczym odstępem, oznaczające, że zawodnik a jest w stanie rozwiązać zadanie b. Każda taka para może pojawić się na wejściu co najwyżej raz.

W testach wartych łącznie 30% punktów zachodzi dodatkowy warunek $n, m \leqslant 100$.

Wyjście

W pierwszym wierszu standardowego wyjścia należy wypisać najlepszy możliwy wynik drużyny Bartusia w postaci dwóch liczb całkowitych oddzielonych pojedynczym odstępem: liczby rozwiązanych zadań z oraz liczby punktów karnych. W kolejnych z wierszach należy wypisać przykładowy przydział zadań: w każdym wierszu mają znaleźć się trzy liczby całkowite a, b i c ($1 \leqslant a \leqslant n$, $1 \leqslant b \leqslant m$, $0 \leqslant c \leqslant t - r$), pooddzielane pojedynczymi odstępami i oznaczające, że zawodnik a powinien zacząć rozwiązywać zadanie b w chwili c (konkurs rozpoczyna się w chwili 0). Nie należy nikomu przydzielać zadań, których dana osoba nie potrafi rozwiązać. Jeśli istnieje więcej niż jedna poprawna odpowiedź, Twój program powinien wypisać dowolną z nich.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`2 4 3 15 4`	`3 12`
`1 1`	`1 4 0`
`2 3`	`2 3 0`
`1 4`	`1 1 3`
`1 3`	

Rozwiązanie

Zacznijmy od kilku prostych obserwacji. Załóżmy, że Bartuś dokonał już przydziału zadań pomiędzy zawodników swojej drużyny, i zastanówmy się, jak powinni oni postępować, by zminimalizować liczbę zdobytych punktów karnych. Jasne jest, że każdy zawodnik powinien rozpocząć rozwiązywanie nowego zadania tak szybko, jak to możliwe, zatem opłaca się zaczynać rozwiązywanie zadań w chwilach, które są wielokrotnościami r. Wynika z tego, że każdy zawodnik może rozwiązać co najwyżej $\lfloor t/r \rfloor$ zadań. Nie ma też znaczenia, w jakiej kolejności zawodnik będzie rozwiązywał przydzielone mu zadania — liczba punktów karnych naliczonych temu zawodnikowi zależy tylko od liczby zadań, które on rozwiązał. Jeśli zawodnik rozwiąże d zadań, to do wyniku dołoży $r(1 + 2 + \ldots + d)$ punktów karnych.

To pozwala nam na sformułowanie problemu w języku teorii grafów: rozważmy graf dwudzielny $G = (O \cup Z, E)$, $|O| = n$, $|Z| = m$, $|E| = k$. Wierzchołki O reprezentują zawodników (osoby), wierzchołki Z — zadania, a krawędź między $o \in O$ a $z \in Z$ istnieje, jeśli zawodnik o potrafi rozwiązać zadanie z.

Przydział zadań możemy przedstawić jako podzbiór zbioru krawędzi $M \subseteq E$. Niech $d_M(o)$ oznacza stopień wierzchołka $o \in O$ w podgrafie $(O \cup Z, M)$; stopień ten odpowiada liczbie zadań przydzielonych zawodnikowi o. Wprowadźmy też oznaczenie $c_M(o) = 1 + 2 + \ldots + d_M(o)$, które nazwiemy *kosztem* zawodnika o; koszt pomnożony przez r jest liczbą punktów karnych naliczonych zawodnikowi o.

Zadanie polega na wyznaczeniu takiego podzbioru $M \subseteq E$ (oznaczającego przydział zadań), który spełnia następujące warunki:

1. każdy wierzchołek z Z jest incydentny z co najwyżej jedną krawędzią z M;
2. dla każdego $o \in O$ mamy $d_M(o) \leqslant t/r$;
3. liczba krawędzi z M jest jak największa;
4. przy założeniu z poprzedniego punktu, sumaryczny koszt $c(M) = \sum_{o \in O} c_M(o)$ jest jak najmniejszy.

Pierwszy warunek oznacza, że każde zadanie jest rozwiązywane przez co najwyżej jednego zawodnika. Drugi warunek gwarantuje, że każdy zawodnik zdąży rozwiązać przydzielone mu zadania. Ostatnie dwa warunki stwierdzają, że przydział zadań M maksymalizuje wynik drużyny.

Rozwiązanie pierwsze: ścieżki polepszające koszt

Na początku rozwiążemy prostsze zadanie, bez zakładania drugiego warunku. Później zobaczymy, że z rozwiązania dla tej wersji łatwo wynika rozwiązanie oryginalnego zadania. Wykorzystamy metodę ścieżek powiększających, znaną przede wszystkim z algorytmu wyznaczania najliczniejszego skojarzenia w grafie.

Zauważmy, że jeśli istnieją zadania, których żaden zawodnik nie umie rozwiązać, to możemy je usunąć. Załóżmy zatem bez straty ogólności, że w zbiorze Z nie ma wierzchołków izolowanych. Dla każdego zadania $z \in Z$ wybierzmy do przydziału M dowolną krawędź wychodzącą z z. Dostaliśmy $|M| = m$, zatem w przydziale M wszystkie zadania zostały rozwiązane. Więcej oczywiście się nie da, postarajmy się teraz uzyskać optymalny koszt.

Ścieżkę w grafie G nazwiemy *naprzemienną*, jeśli zawiera parzystą liczbę krawędzi, zaczyna się w wierzchołku ze zbioru O krawędzią z przydziału M i z każdych dwóch kolejnych krawędzi na ścieżce dokładnie jedna należy do M. Naprzemienną ścieżkę $P = (o_1, z_1, o_2, z_2, \ldots, o_l)$, $o_i \in O$, $z_i \in Z$, $(o_i, z_i) \in M$, $(z_i, o_{i+1}) \in E \setminus M$ nazwiemy *polepszającą koszt*, jeżeli

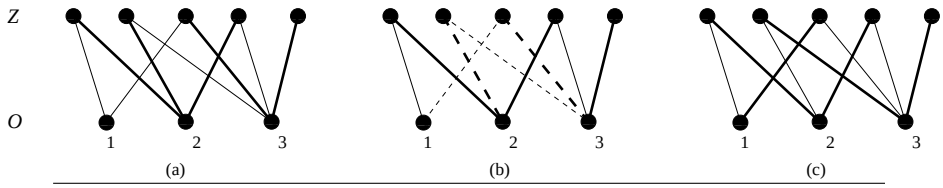
$$d_M(o_1) > d_M(o_l) + 1, \qquad (\star)$$

czyli stopień pierwszego wierzchołka ścieżki jest o co najmniej 2 większy niż stopień ostatniego wierzchołka. Pokażemy, że odwracając przydział krawędzi na ścieżce P, uzyskamy lepszy koszt.

Oznaczmy przez $M' = (M \setminus P) \cup (P \setminus M)$ przydział odwrócony wzdłuż ścieżki P — taką operację nazwiemy *dodaniem* ścieżki P do przydziału M. Jasne jest, że $|M'| = |M|$ oraz że zmieniły się stopnie krańcowych wierzchołków ścieżki: $d_{M'}(o_1) = d_M(o_1) - 1$, $d_{M'}(o_l) = d_M(o_l) + 1$, natomiast stopnie reszty wierzchołków pozostały bez zmian. Koszt nowego przydziału M' jest w istocie mniejszy niż koszt M (w ostatniej nierówności korzystamy z $(\star)$):

$$\begin{aligned} c(M') &= c_{M'}(o_1) + c_{M'}(o_l) + \textstyle\sum_{o \in O \setminus \{o_1, o_l\}} c_{M'}(o) = \\ &= (c_M(o_1) - d_M(o_1)) + (c_M(o_l) + d_M(o_l) + 1) + \textstyle\sum_{o \in O \setminus \{o_1, o_l\}} c_M(o) = \\ &= c(M) - d_M(o_1) + d_M(o_l) + 1 < c(M). \end{aligned}$$

Przychodzi więc na myśl algorytm, który znajduje kolejne ścieżki proste (czyli ścieżki bez cykli) polepszające koszt, aż już żadnej nie będzie. Pojedynczy krok algorytmu — znalezienie ścieżki polepszającej koszt lub stwierdzenie, że taka ścieżka nie



Rys. 1: Dla przydziału zadań M (pogrubione krawędzie w (a)) znajdujemy ścieżkę polepszającą koszt z wierzchołka o_2 do wierzchołka o_1 (przerywane krawędzie w (b)) i odwracamy przydział wzdłuż tej ścieżki, uzyskując M' (c).

istnieje — możemy wykonać w czasie $O(k)$. W tym celu rozważamy wierzchołki ze zbioru O w kolejności nierosnących stopni i z każdego z nich przeszukujemy graf w głąb (oczywiście z wierzchołków z O wychodzimy krawędziami należącymi do przydziału M, a z wierzchołków z Z wychodzimy krawędziami z $E \setminus M$). Ponadto zaznaczamy odwiedzane wierzchołki i tak zaznaczonych wierzchołków nie odwiedzamy ponownie. Dlaczego możemy tak zrobić? Otóż powiedzmy, że przenumerowaliśmy wierzchołki w kolejności nierosnących stopni ($d_M(o_1) \geqslant d_M(o_2) \geqslant \ldots \geqslant d_M(o_n)$) i ostatnim rozważonym wierzchołkiem był wierzchołek o_i. Jeśli nie znaleźliśmy ścieżki polepszającej koszt zaczynającej się w tym wierzchołku, to znaczy, że nie ma ścieżki naprzemiennej zaczynającej się w o_i i kończącej się w wierzchołku z O o stopniu mniejszym niż $d_M(o_i) - 1$. Tak więc na pewno wśród dotychczas odwiedzonych wierzchołków nie znajdziemy też wierzchołka o stopniu mniejszym niż $d_M(o_j) - 1$ dla $j > i$.

Początkowy koszt może wynosić co najwyżej $m(m+1)/2$ (jest tak w przypadku, gdy wszystkie zadania przydzielimy jednemu zawodnikowi), a w każdym kroku algorytmu koszt zmniejsza się co najmniej o 1, zatem uzyskujemy czas $O(m^2k) = O(m^3n)$.

Aby wykazać poprawność naszego algorytmu, wystarczy udowodnić następujące twierdzenie:

Twierdzenie 1. *Koszt przydziału o rozmiarze m jest optymalny wtedy i tylko wtedy, gdy nie istnieje ścieżka prosta polepszająca koszt.*

Dowód: Implikacja w prawo jest oczywista: gdyby istniała ścieżka polepszająca koszt, to dodanie jej do przydziału zmniejszyłoby jego koszt. Udowodnimy implikację w lewo. Niech M będzie przydziałem, którego koszt nie jest optymalny; pokażemy, że istnieje ścieżka prosta polepszająca koszt tego przydziału. Rozważmy przydział N taki, że $|N| = m$ oraz koszt $c(N)$ jest optymalny. Jeśli jest więcej niż jeden kandydat na N, to weźmy tego, dla którego zbiór $D = (M \setminus N) \cup (N \setminus M)$ ma jak najmniej krawędzi. Ścieżką *naprzemienną* względem D nazwiemy ścieżkę $P = (o_1, z_1, \ldots, o_l)$, taką, że $(o_i, z_i) \in M \setminus N$, $(z_i, o_{i+1}) \in N \setminus M$. Z minimalności D wynika, że nie istnieją cykle (ścieżki spełniające $o_1 = o_l$) naprzemienne względem D, gdyż inaczej dodalibyśmy taki cykl do przydziału N i dostalibyśmy również optymalny przydział N, tyle że z mniejszą liczbą krawędzi w zbiorze D. Ponadto, każda ścieżka naprzemienna względem D, prowadząca z o_1 do o_l, spełnia $d_N(o_1) \geqslant d_N(o_l)$. Gdyby bowiem było $d_N(o_1)+1 < d_N(o_l)$, to odwracając kolejność krawędzi na tej ścieżce (tzn. rozważając tę ścieżkę od o_l do o_1), uzyskalibyśmy ścieżkę polepszającą koszt w N. Gdyby zaś zachodziło $d_N(o_1) + 1 = d_N(o_l)$, to dodanie tej ścieżki do N nie zmieniłoby kosztu, ale zmniejszyłoby rozmiar D, co znowu przeczy minimalności zbioru D.

Ścieżkę prostą polepszającą koszt w M znajdujemy następująco: wybieramy wierzchołek $o_1 \in O$ taki, że $d_M(o_1) > d_N(o_1)$ (skoro $c(M) > c(N)$, to taki wierzchołek musi istnieć). Konstruujemy teraz ścieżkę naprzemienną względem D do wierzchołka o_l takiego, że $d_{M \setminus N}(o_l) = 0$. Konkretnie, z wierzchołka o_i wychodzimy dowolną krawędzią z $M \setminus N$ (jeśli $d_{M \setminus N}(o_i) \neq 0$, to taka krawędź istnieje), natomiast z wierzchołka z_i wychodzimy jedyną krawędzią z $N \setminus M$ (z każdego wierzchołka ze zbioru Z wychodzi jedna krawędź z N i jedna z M, a ponieważ do z_i weszliśmy krawędzią z $M \setminus N$, to w przypadku tego wierzchołka są to dwie różne krawędzie). Nie może się zdarzyć, że w którymś momencie wrócimy do wierzchołka, z którego wyszliśmy, bo wtedy

otrzymalibyśmy cykl naprzemienny względem D. Zauważmy również, że ścieżka jest niepusta ($l > 1$), bo $d_{M \setminus N}(o_1) > 0$. Widzimy wreszcie, że do ostatniego wierzchołka o_l weszliśmy krawędzią z $N \setminus M$, a nie wychodzą z niego krawędzie z $M \setminus N$, a więc $d_M(o_l) \leqslant d_N(o_l) - 1$. Dostajemy zatem

$$d_M(o_l) \leqslant d_N(o_l) - 1 \leqslant d_N(o_1) - 1 < d_M(o_1) - 1,$$

przy czym środkową nierówność uzasadniliśmy w poprzednim akapicie. To kończy dowód — skonstruowana ścieżka jest ścieżką polepszającą koszt w przydziale M, gdyż spełnia warunek ($\star$). ■

Co z czasem trwania zawodów?

Rozwiązanie uzyskane powyższą metodą nie uwzględnia warunku, że żaden zawodnik nie może rozwiązać więcej niż t/r zadań. Może się zatem zdarzyć, że w optymalnym przydziale M dla niektórych wierzchołków $o \in O$ będzie $d_M(o) > t/r$. Okazuje się jednak, że aby uzyskać rozwiązanie pierwotnego zadania, wystarczy w optymalnym przydziale M usunąć dowolne nadmiarowe krawędzie wychodzące z tych wierzchołków. Poniżej udowodnimy ten fakt.

Przez $M_{|d}$ oznaczmy (pewien) przydział uzyskany z M poprzez ograniczenie z góry stopnia każdego z wierzchołków ze zbioru O przez d. Wprowadźmy też oznaczenie na jakość rozwiązania obciętego. Mianowicie *d-kosztem* przydziału zadań M nazwiemy parę

$$c_d(M) = (|M_{|d}|, c(M_{|d})),$$

w której pierwsza współrzędna oznacza rozmiar obciętego przydziału, a druga jego koszt. Ponieważ

$$\begin{aligned} |M_{|d}| &= \textstyle\sum_{o \in O} \min(d_M(o), d), \\ c(M_{|d}) &= \textstyle\sum_{o \in O} 1 + 2 + \ldots + \min(d_M(o), d), \end{aligned}$$

więc definicja $c_d(M)$ nie zależy od wyboru konkretnego przydziału $M_{|d}$.

Powiemy, że przydział M ma d-koszt większy niż przydział N, jeśli albo $|M_{|d}| < |N_{|d}|$, albo $|M_{|d}| = |N_{|d}|$ i $c(M_{|d}) > c(N_{|d})$. Jasne jest, że w zadaniu chodzi o zminimalizowanie $\lfloor t/r \rfloor$-kosztu przydziału zadań.

Twierdzenie 2. *Jeśli nie istnieje ścieżka prosta polepszająca koszt, to d-koszt przydziału o rozmiarze m jest optymalny.*

Dowód: Dowód przebiega w analogiczny sposób jak dowód implikacji w lewo twierdzenia 1. Niech M będzie przydziałem, którego d-koszt nie jest optymalny — wskażemy ścieżkę prostą polepszającą koszt tego przydziału. Analogicznie rozważamy przydział N o optymalnym d-koszcie i minimalnym zbiorze $D = (M \setminus N) \cup (N \setminus M)$.

Jedynym miejscem w dowodzie, w którym korzystaliśmy z nieoptymalności kosztu przydziału M, było pokazanie istnienia wierzchołka początkowego ścieżki, tzn. wierzchołka $o_1 \in O$ takiego, że $d_M(o_1) > d_N(o_1)$. Pokażemy, że do znalezienia go wystarczy nieoptymalność d-kosztu M.

Istotnie, jeśli $|M_{|d}| < |N_{|d}|$, to mamy $\sum_{o\in O} \min(d_M(o), d) < \sum_{o\in O} \min(d_N(o), d)$, zatem musi istnieć wierzchołek $o'_1 \in O$ taki, że $d_M(o'_1) < d_N(o'_1)$. Ale jednocześnie mamy $\sum_{o\in O} d_M(o) = m = \sum_{o\in O} d_N(o)$, co wymusza istnienie wierzchołka o_1.

Jeśli z kolei $|M_{|d}| = |N_{|d}|$, to musi być $c(M_{|d}) > c(N_{|d})$. W takim wypadku również łatwo widać, że wierzchołek o_1 musi istnieć. ■

Powyższe rozwiązanie zostało zaimplementowane w pliku `pro3.cpp`. Można je przyspieszyć, stosując pewne proste optymalizacje, które nie zmniejszają teoretycznej złożoności algorytmu, ale w praktyce sprawdzają się całkiem dobrze. Przykładowo, dużo daje, jeśli nie zaczynamy od dowolnego rozwiązania, lecz od już dość dobrego. Może to być, na przykład, wynik zachłannego poprawiania poprzez zmianę jednej krawędzi. Wtedy ścieżki polepszające koszt trzeba znajdować już stosunkowo mało razy. Dodatkowo, można też nie zaczynać szukania ścieżki polepszającej koszt z wierzchołków o najmniejszym lub o jeden większym stopniu, bo wiadomo, że i tak żadnej nie znajdziemy.

Rozwiązanie drugie: ścieżki powiększające

Przedstawimy teraz szybszy algorytm. Podobnie jak poprzednio, zakładamy, że zbiór Z nie zawiera wierzchołków izolowanych, oraz nie bierzemy pod uwagę ograniczenia na czas trwania konkursu. Zaczynamy od pustego przydziału M. W każdym kroku koszt $c(M)$ będzie optymalny dla mniejszego grafu, zawierającego tylko zadania incydentne z którąś krawędzią z M. W kroku i (dla $i = 1, 2, \ldots, m$) będziemy chcieli dodać krawędź wychodzącą z zadania z_i. Ścieżkę w grafie nazwiemy *powiększającą*, jeśli zawiera nieparzystą liczbę krawędzi, zaczyna się w wierzchołku ze zbioru Z krawędzią nienależącą do M i z każdych dwóch kolejnych krawędzi na ścieżce dokładnie jedna należy do M. Szukamy ścieżki powiększającej z wierzchołka z_i, która kończy się w wierzchołku $o \in O$ o najmniejszym stopniu $d_M(o)$. Dodajemy tę ścieżkę do M, uzyskując M', takie że $|M'| = |M| + 1$ i koszt $c(M')$ jest optymalny. Pojedynczy krok wykonujemy przeszukiwaniem grafu w głąb w czasie $O(k)$, zatem cały algorytm działa w czasie $O(mk) = O(m^2 n)$.

Uzasadnimy teraz poprawność tego algorytmu. Oznaczmy przez R ścieżkę powiększającą, którą dodaliśmy do przydziału M, tworząc przydział M', oraz niech $o \in O$ i $z \in Z$ będą końcami ścieżki R. Pokażemy, że jeśli przydział M miał optymalny koszt, to M' także ma optymalny koszt, a konkretnie, że dla M' nie istnieje ścieżka polepszająca koszt.

Załóżmy nie wprost, że ścieżka taka istnieje. Możemy przyjąć, że jest ona prosta, oznaczmy ją przez $P = (o_1, z_1, \ldots, o_l)$. Załóżmy na początek, że $o_1 \neq o$.

Ponieważ P jest ścieżką polepszającą koszt dla M', więc $d_{M'}(o_1) > d_{M'}(o_l) + 1$, a ponieważ przez dodanie R nie zmieniliśmy stopnia wierzchołkowi o_1, a wierzchołkowi o_l mogliśmy go co najwyżej podnieść (gdy $o_l = o$), zatem $d_M(o_1) > d_M(o_l) + 1$. Ponadto, ścieżka P musi przecinać ścieżkę R, w przeciwnym bowiem wypadku byłaby ścieżką polepszającą koszt dla M. Niech v' i v'' będą, odpowiednio, pierwszym i ostatnim wierzchołkiem ścieżki P spośród tych, które należą również do R. Można zauważyć, że $v' \in O$. Ponieważ ścieżka naprzemienna $o_1 \overset{P}{\rightsquigarrow} v' \overset{R}{\rightsquigarrow} o$ nie może być

ścieżką polepszającą koszt dla M, zatem $d_M(o_1) \leqslant d_M(o) + 1$. Z tych dwóch nierówności wynika, że $d_M(o_l) + 1 < d_M(o_1) \leqslant d_M(o) + 1$.

Jeśli $v'' \in O \setminus \{o_l\}$, to ścieżka P opuszcza wierzchołek v'' krawędzią z M, a jeśli $v'' \in Z$, to krawędzią spoza M. Wynika z tego, że ścieżka $z \overset{R}{\rightsquigarrow} v'' \overset{P}{\rightsquigarrow} o_l$ jest ścieżką powiększającą dla M. Co więcej, z faktu $d_M(o_l) < d_M(o)$ wynika, że wierzchołek o_l jest lepszym kandydatem na koniec ścieżki R niż wierzchołek o. Sprzeczność — zatem ścieżka P nie istnieje, co dowodzi optymalności kosztu $c(M')$.

Pozostał przypadek, gdy $o_1 = o$. Wtedy jednak mamy $d_M(o) + 1 = d_{M'}(o)$ oraz $d_M(o_l) = d_{M'}(o_l)$, a z własności P jest $d_{M'}(o) > d_{M'}(o_l) + 1$, zatem ostatecznie $d_M(o) > d_M(o_l)$ i dalej działa ten sam argument.

Podobnie jak poprzednio, możemy na koniec usuwać nadmiarowe krawędzie wychodzące od osób, które rozwiązują więcej niż t/r zadań. Równoważnie, możemy jednak od razu nie dodawać ścieżki do M, jeśli jej dodanie powoduje zwiększenie stopnia jakiegoś wierzchołka ponad t/r (dowód podobny do poprzednich nie jest trudny).

To rozwiązanie zostało zaimplementowane w pliku `pro4.cpp`. Także w tym rozwiązaniu możemy poczynić pewne optymalizacje. Często zdarza się tak, że ścieżka powiększająca jest bardzo krótka, nawet jednokrawędziowa. Mimo to nasze rozwiązanie przejdzie cały graf w jej poszukiwaniu. Program dużo lepiej zachowuje się, jeśli najpierw sprawdzimy, czy może któryś z sąsiednich wierzchołków z O nie należy jeszcze w ogóle do żadnej krawędzi z M. Wtedy możemy ich po prostu skojarzyć i nie przeszukiwać całego grafu. Optymalizacja ta spisuje się najsłabiej, gdy zadań jest istotnie więcej niż osób. Wtedy (choć graf musi być mniejszy) dla wielu zadań nie będzie sąsiada, który nie rozwiązywałby jeszcze żadnego zadania, i będzie trzeba dla nich przejść cały graf.

Najsłabszym punktem naszego rozwiązania $O(m^2n)$ jest to, że w zasadzie dla znalezienia najlepszej ścieżki powiększającej trzeba przejrzeć cały graf (bo ma to być najlepsza ścieżka, więc może kończyć się wszędzie). Możemy jednak spojrzeć na to od drugiej strony: zamiast od zadania, będziemy szukać najlepszej ścieżki powiększającej od osoby. Wtedy musimy zacząć od osoby o najmniejszym możliwym stopniu (pomijając osoby, o których już wiemy, że żadna ścieżka z nich nie istnieje). Przy tym założeniu możemy dotrzeć do dowolnego zadania i ścieżka będzie najlepsza. Możemy zatem szukać najkrótszej takiej ścieżki, przeszukując graf wszerz. Jeśli jest ona krótka (co zdarzy się w większości przypadków), to przeszukiwanie zakończy się szybko i nie będzie przechodziło całego grafu. Takie rozwiązanie jest zaimplementowane w plikach `pro.cpp` i `pro2.pas` oraz, z drobnymi usprawnieniami, w pliku `pro1.cpp`.

Rozwiązanie trzecie: przepływ w sieci

Zadanie można sprowadzić do problemu znajdowania maksymalnego przepływu o minimalnym koszcie w sieci (ang. *min-cost max-flow problem*). Oznaczmy przez $d = \min(m, \lfloor t/r \rfloor)$ maksymalną liczbę zadań, którą może rozwiązać pojedynczy zawodnik. Tworzymy sieć H o następujących wierzchołkach:

- źródło s i ujście t;
- wierzchołki $z_1, \ldots, z_m$ odpowiadające zadaniom;

- wierzchołki $o_1, \ldots, o_n$ odpowiadające zawodnikom.

Ponadto w sieci (dla każdego $1 \leqslant i \leqslant m$, $1 \leqslant j \leqslant n$):

- istnieje krawędź o koszcie 0 ze źródła s do wierzchołka z_i;
- jeśli i-te zadanie może być rozwiązane przez j-tego zawodnika, to istnieje krawędź o koszcie 0 z wierzchołka z_i do wierzchołka o_j;
- istnieje d krawędzi o rosnących kosztach $1, 2, \ldots, d$ z wierzchołka o_j do ujścia t.

Wszystkie krawędzie mają przepustowość jednostkową. Każdy przepływ w sieci H odpowiada pewnemu przydziałowi zadań (i na odwrót). Wartość przepływu jest równa liczbie rozwiązanych zadań, natomiast koszt przepływu pomnożony przez r jest równy liczbie punktów karnych, zatem maksymalny przepływ o minimalnym koszcie odpowiada optymalnemu rozwiązaniu.

Liczba wierzchołków w sieci wynosi $n' = 2 + m + n = O(m + n)$, natomiast liczba krawędzi $k' = m + k + nd = O(mn)$. W pliku `pros5.cpp` zaimplementowano to rozwiązanie, korzystając z algorytmu znajdowania przepływu działającego w czasie $O(fn'k') = O(m^2n(m + n))$, gdzie $f = m$ jest maksymalną wartością przepływu.

Rozwiązania niepoprawne

Najbardziej narzucającym się rozwiązaniem niepoprawnym jest zachłanne poprawianie przydziału zadań: szukamy zadania, które możemy przydzielić komuś innemu, polepszając koszt. Takie rozwiązanie znajduje się w pliku `prob6.cpp` i przechodzi jedynie dwa testy: 7 i 8. Nie działa ono już na dwóch testach przykładowych.

Można też zaproponować następujące rozwiązanie zachłanne. Obliczamy w nim:

- trudność każdego zadania — im więcej zawodników umie zrobić dane zadanie, tym jest ono łatwiejsze,
- doświadczenie każdego zawodnika — im więcej zadań umie zrobić dany zawodnik, tym jest bardziej doświadczony.

Teraz próbujemy zachłannie przydzielać zadania w kolejności od najtrudniejszego, za każdym razem wybierając możliwie najmniej doświadczonego zawodnika. Rozwiązanie znajduje się w pliku `prob7.cpp` i przechodzi testy 7 i 8.

Kolejne rozwiązanie niepoprawne opiera się na spostrzeżeniu, że każdy przydział zadań M jest sumą pewnej liczby skojarzeń w grafie dwudzielnym G (dalej będziemy tę liczbę oznaczali przez Δ). Pomysł jest więc taki, by znajdować kolejne maksymalne skojarzenia w grafie, dokładać je do M i usuwać z G. Rozwiązanie to ma złożoność $O(m^2n)$ i w praktyce działa sprawnie. Zapisane zostało w pliku `prob8.cpp`. Rozwiązanie daje błędne odpowiedzi w przypadku grafów rzadkich lub takich, które wymagają dużego Δ. Przechodzi wszystkie testy przykładowe i tylko dwa testy punktowane (1 i 6). W pliku `prob9.cpp` znajduje się program, który 10 razy próbuje wykonać powyższy algorytm, za każdym razem permutując losowo listy sąsiedztwa grafu. Ten program nie wykazuje żadnej poprawy względem programu `prob8.cpp`.

Testy

Do zadania przygotowano 26 testów połączonych w 10 grup (testy 8b, 9b, 10b, 10c to testy proste pod względem złożoności, jednak sprawdzające różne skrajne przypadki). Część testów specjalnie odsiewa rozwiązanie `prob8.cpp`. Poniżej szczegółowe zestawienie testów.

Nazwa	n	m	k	⌊t/r⌋	wynik	Opis
pro1a.in	20	100	816	11	100	losowy
pro1b.in	20	100	817	11	100	losowy
pro2.in	50	100	269	2	95	losowy
pro3a.in	80	80	2 125	1	80	losowy
pro3b.in	8	80	33	7	27	losowy
pro4a.in	200	200	16 201	10	200	losowy
pro4b.in	200	200	15 971	10	200	losowy
pro4c.in	20	200	152	9	121	losowy
pro5a.in	252	499	97 624	2	494	losowy plus wierzchołki stopnia 0
pro5b.in	252	499	97 588	2	494	losowy plus wierzchołki stopnia 0
pro5c.in	7	494	103	400	91	losowy
pro6a.in	500	500	74 870	2	500	losowy
pro6b.in	497	486	30 381	419	478	rozłączne części 487 : 243 i 10 : 243 ($n : m$)
pro6c.in	6	495	471	469	469	jeden zawodnik rozwiązuje
pro7a.in	500	500	123 295	3	500	rozłączne części 249 : 498 i 251 : 2
pro7b.in	479	493	47 283	117	493	losowy
pro7c.in	3	500	439	120	324	losowy
pro8a.in	500	500	17 629	3	390	luźno połączone części 95 : 385 i 395 : 85 plus dodatki
pro8b.in	500	500	74 893	0	0	losowy, $r > t$
pro8c.in	430	500	107 652	497	500	losowy
pro9a.in	500	500	80 367	10^6	500	losowy
pro9b.in	500	500	0	5	0	bez krawędzi

Nazwa	n	m	k	$\lfloor t/r \rfloor$	wynik	Opis
pro9c.in	20	500	409	314	294	rozłączne części 2 : 100, 3 : 200, 2 : 200
pro10a.in	500	500	25634	3	452	luźno połączone części 100 : 200, 200 : 100, 192 : 191 plus dodatki
pro10b.in	500	500	250 000	1	500	każdy umie wszystko
pro10c.in	500	500	999	1	500	wąż

Paweł Mechliński, Jakub Pachocki
Treść zadania, Opracowanie

Błażej Osiński
Program

Dostępna pamięć: 64 MB.

OI, Etap III, dzień drugi, 07.04.2011

Meteory

Bajtocka Unia Międzygwiezdna (BUM) odkryła niedawno w pobliskiej galaktyce nową, wyjątkowo interesującą planetę. Co prawda nie nadaje się ona do kolonizacji, lecz co jakiś czas nawiedzają ją deszcze nietypowych, nieznanych wcześniej meteorów.

Państwa członkowskie BUM umieściły tuż przy orbicie nowo odkrytej planety stacje badawcze, których głównym zadaniem jest zbieranie próbek przelatujących skał. Komisja BUM podzieliła orbitę na m sektorów ponumerowanych kolejno od 1 do m, przy czym sektor 1 sąsiaduje z sektorem m. W każdym sektorze znajduje się dokładnie jedna stacja badawcza, należąca do jednego z n państw członkowskich BUM.

Każde państwo wyznaczyło liczbę próbek meteorów, jakie pragnie zebrać przed zakończeniem misji. Twoim zadaniem jest, na podstawie prognozy deszczów meteorów na najbliższe lata, wyznaczyć dla każdego państwa, kiedy będzie mogło zakończyć badania.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie liczby całkowite n oraz m ($1 \leqslant n, m \leqslant 300\,000$), oddzielone pojedynczym odstępem i oznaczające odpowiednio liczbę państw członkowskich BUM oraz liczbę sektorów, na jakie podzielona została orbita.

W drugim wierszu znajduje się m liczb całkowitych o_i ($1 \leqslant o_i \leqslant n$), pooddzielanych pojedynczymi odstępami i oznaczających państwa będące właścicielami poszczególnych sektorów.

W trzecim wierszu znajduje się n liczb całkowitych p_i ($1 \leqslant p_i \leqslant 10^9$), pooddzielanych pojedynczymi odstępami i oznaczających liczby próbek meteorów, jakich poszczególne państwa potrzebują do zakończenia badań.

W czwartym wierszu znajduje się jedna liczba całkowita k ($1 \leqslant k \leqslant 300\,000$) oznaczająca liczbę przewidywanych opadów. W kolejnych k wierszach znajdują się opisy deszczów meteorów w kolejności chronologicznej. W i-tym z tych wierszy znajdują się trzy liczby l_i, r_i, a_i (pooddzielane pojedynczymi odstępami), oznaczające, że w sektorach $l_i, l_{i+1}, \ldots, r_i$ (jeśli $l_i \leqslant r_i$) lub sektorach $l_i, l_{i+1}, \ldots, m, 1, \ldots, r_i$ (jeśli $l_i > r_i$) wystąpi deszcz meteorów, w wyniku którego wszystkie znajdujące się tam stacje badawcze uzyskają po a_i próbek skalnych ($1 \leqslant a_i \leqslant 10^9$).

W testach wartych przynajmniej 20% punktów zachodzi dodatkowy warunek $n, m, k \leqslant 1\,000$.

Wyjście

Twój program powinien wypisać na standardowe wyjście n wierszy. W i-tym wierszu powinna znaleźć się liczba całkowita w_i, oznaczająca numer deszczu, po którym stacje badawcze i-tego państwa zgromadzą łącznie co najmniej p_i próbek skalnych, lub słowo `NIE`*, jeśli dane państwo nie będzie mogło skończyć badań w przewidywalnej przyszłości.*

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`3 5`	`3`
`1 3 2 1 3`	`NIE`
`10 5 7`	`1`
`3`	
`4 2 4`	
`1 3 1`	
`3 5 2`	

Rozwiązanie

Wprowadzenie

Mimo dość naturalnego przedstawienia problemu i względnie nieskomplikowanego rozwiązania, korzystającego jedynie ze struktury danych, która jest już dla wielu olimpijczyków klasyczną, zadanie o deszczach meteorów było jednym z najtrudniejszych zadań finału XVIII Olimpiady Informatycznej. Wymagało ono od zawodników znalezienia zaskakującego, nietypowego zastosowania wyszukiwania binarnego, które może być także zastosowane do szerszej gamy problemów.

Rozwiązanie naiwne

Jednym z najprostszych algorytmów rozwiązujących to zadanie jest przeglądanie dla każdego deszczu wszystkich stacji, na które on spada, i aktualizowanie wymagań ich posiadaczy. Rozwiązanie takie działa w złożoności czasowej $O(mk)$, czyli zdecydowanie zbyt wolno.

Symulacja opadów

Nie ulega wątpliwości, że aby uzyskać rozwiązanie o złożoności czasowej lepszej niż $O(mk)$, będziemy musieli zrezygnować z przeglądania wszystkich stacji, na które spada dany deszcz. W tym celu możemy skorzystać z tego, że deszcze zawsze padają na stacje położone w spójnym fragmencie orbity, będzie to więc zawsze albo przedział postaci $[l_i, r_i]$, albo suma przedziałów postaci $[l_i, m] \cup [1, r_i]$. Strukturą danych, która umożliwia szybkie operacje na tego typu przedziałach, jest na przykład *drzewo przedziałowe*, opisane między innymi w opracowaniach zadań *Koleje* z IX Olimpiady Informatycznej [9] czy *Tetris 3D* z XIII Olimpiady Informatycznej [13]. Używając drzewa przedziałowego zbudowanego nad punktami $1, 2, \ldots, m$, można symulować kolejne deszcze oraz udzielać odpowiedzi na pytania o liczbę próbek zebranych dotychczas przez wskazaną stację w czasie $O(\log m)$. Niestety, ponieważ państwa mogą posiadać więcej niż jedną stację, dalej nie widać sposobu na szybkie identyfikowanie państw,

które zebrały już wystarczająco wiele próbek — tutaj jedyną metodą pozostaje wciąż powolne przeglądanie wszystkich stacji.

Rozwiązanie wzorcowe

Problemy, w których należy wskazać najmniejszą wartość w spełniającą pewną własność (tutaj: najmniejsze w, dla którego po w-tym deszczu dane państwo zebrało już dostatecznie wiele próbek), często stają się prostsze, gdy zastosujemy wyszukiwanie binarne. Wtedy wystarczy znaleźć sposób udzielania odpowiedzi na prostsze pytanie: „czy dane w spełnia żądaną własność?". W naszym przypadku, dla ustalonego państwa i konkretnej wartości w nietrudno sprawdzić stosowną własność w czasie $O((m+k)\log m)$, symulując deszcze meteorów aż do w-tego i zliczając opady.

Tego typu algorytm może wydawać się niepotrzebnie skomplikowany i nieefektywny jako rozwiązanie problemu dla jednego państwa. Kluczowa różnica z opisanym wyżej algorytmem naiwnym tkwi jednak w kolejności stawiania pytań. I tu ujawnia się główny pomysł rozwiązania wzorcowego: odpowiedzi na pojedyncze zapytanie z wyszukiwania binarnego, niekoniecznie dla równych wartości w, jesteśmy w stanie udzielić w czasie $O((m+k)\log m)$ dla *wszystkich* państw naraz!

Załóżmy, że chcemy udzielić odpowiedzi na zapytania z wyszukiwania binarnego dla państw $P_1, P_2, \ldots, P_n$ i wartości wynoszących odpowiednio $w_1, w_2, \ldots, w_n$ (przy czym $w_1 \leqslant w_2 \leqslant \ldots \leqslant w_n$). Możemy wykonywać symulację z wcześniejszego podrozdziału i dla każdego i, po spadnięciu w_i-tego deszczu znaleźć odpowiedź na zapytanie dla państwa P_i — poprzez proste przejrzenie i zsumowanie liczb uzyskanych próbek ze stacji, których jest ono właścicielem. Ponieważ każdą stację przejrzymy dokładnie raz, a złożoność czasowa zasymulowania pojedynczego deszczu, jak i ustalenia liczby próbek uzyskanych dotąd przez stację, dzięki zastosowaniu drzewa przedziałowego jest rzędu $O(\log m)$, więc cała operacja zajmie w istocie czas $O((m+k)\log m)$.

Aby rozwiązać nasze zadanie, wystarczy więc przeprowadzić równoległe wyszukiwanie binarne odpowiedzi dla wszystkich państw naraz. Ponieważ etapów takiego wyszukiwania będzie $O(\log k)$, złożoność czasowa całego rozwiązania wyniesie $O((m+k)\log m \log k)$. Implementacje można znaleźć w plikach `met.cpp` i `met1.pas`.

Przykład

Przeanalizujmy na przykładzie, jak działa podana wersja wyszukiwania binarnego. Rozważmy cztery państwa, o wymaganiach odpowiednio 11, 3, 10 i 8 próbek, i pięć deszczów meteorów, zgodnie z poniższą tabelką:

piąty deszcz:			1	1	1	1		
czwarty deszcz:	4	4	4					4
trzeci deszcz:		2	2	2	2	2		
drugi deszcz:	3	3	3	3				
pierwszy deszcz:	2				2	2	2	2
numery państw	**1**	**3**	**2**	**1**	**3**	**4**	**4**	**1**

Dla każdego państwa początkowy przedział możliwych (pozytywnych) wyników to $[1, 5]$. W pierwszej fazie wyszukiwania binarnego dla każdego państwa sprawdzamy

wartość $w = 3$ (ciąg wartości to 3, 3, 3, 3). Za pomocą drzewa przedziałowego obliczamy, że po trzech deszczach pierwsze państwo zebrało 12, drugie 5, trzecie 9, a czwarte 6 próbek. Widzimy, że pierwsze dwa państwa są usatysfakcjonowane, a drugie dwa nie. Dlatego w drugiej fazie wyszukiwania binarnego pierwsze dwa państwa będą miały przedział wartości $[1, 3]$, a drugie dwa — przedział $[4, 5]$.

W drugiej fazie dla pierwszych dwóch państw rozważamy wartość $w = 2$, a dla dwóch pozostałych wartość $w = 4$, czyli ciąg wyszukiwanych wartości to 2, 2, 4, 4. Teraz symulujemy pierwsze dwa deszcze, po czym sprawdzamy, że pierwsze państwo zebrało 10 próbek, a drugie 3 próbki. Następnie symulujemy kolejne dwa deszcze i obliczamy, że trzecie państwo zebrało 13, a czwarte 6 próbek. Widzimy, że państwa drugie i trzecie są usatysfakcjonowane, skąd wnosimy, że przedział możliwych wyników dla nich to, odpowiednio, $[1, 2]$ i $[4, 4]$. Z kolei państwom pierwszemu i czwartemu brakuje próbek, czyli ich przedziały wartości to $[3, 3]$ i $[5, 5]$. To oznacza, że znamy już wyniki dla pierwszego i trzeciego państwa (dla czwartego jeszcze nie — musimy jeszcze sprawdzić, czy po zakończeniu wszystkich opadów będzie usatysfakcjonowane).

W ostatniej fazie wyszukiwania binarnego rozważamy ciąg wartości 1, 5 dla państw drugiego i czwartego i widzimy, że żadne z nich nie jest usatysfakcjonowane: drugie państwo nie zebrało żadnej próbki, a czwartemu brakuje jednej do żądanych 8 próbek. To oznacza, że wynikiem dla drugiego państwa jest 2, a dla czwartego odpowiedź brzmi „`NIE`”. Ostateczny ciąg wyników w tym przykładzie to zatem: 3, 2, 4, `NIE`.

Testy

Nazwa	n	m	k
met1a.in	142	1 000	1 000
met1b.in	500	1 000	1 000
met1c.in	225	1 000	1 000
met2a.in	333	1 000	957
met2b.in	333	1 000	919
met2c.in	750	1 000	1 000
met3a.in	5 000	50 000	50 000
met3b.in	25 000	50 000	50 000
met3c.in	12 976	50 000	50 000
met4a.in	9 043	50 000	50 000
met4b.in	9 111	50 000	50 000
met4c.in	25 000	50 000	49 913
met4d.in	15 340	50 000	5 000
met5a.in	5 000	50 000	49 419
met5b.in	25 000	50 000	49 704

Nazwa	n	m	k
met5c.in	500	500	50 000
met5d.in	16 855	50 000	50 000
met6a.in	500	50 000	50 000
met6b.in	10 000	50 000	50 000
met6c.in	1 961	50 000	50 000
met6d.in	25 044	50 000	50 000
met7a.in	99 901	300 000	300 000
met7b.in	4	300 000	300 000
met7c.in	500	500	250 000
met7d.in	199 840	300 000	300 000
met8a.in	85 652	300 000	300 000
met8b.in	3	300 000	300 000
met8c.in	500	500	200 000
met8d.in	233 410	300 000	300 000

Michał Pilipczuk
Treść zadania, Opracowanie

Alan Kutniewski
Program

Dostępna pamięć: 32 MB. *OI, Etap III, dzień drugi, 07.04.2011*

Patyczki

Mały Jaś dostał od babci i dziadka prezent na urodziny. Jest nim pudełko pełne patyczków różnej długości i różnych kolorów. Jaś zastanawia się, czy z pewnych trzech patyczków z zestawu da się zbudować trójkąt o wszystkich bokach różnych kolorów. Jasia interesują tylko trójkąty niezdegenerowane, czyli takie o dodatnim polu.

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita k ($3 \leqslant k \leqslant 50$) — jest to liczba różnych kolorów patyczków. Kolory numerujemy od 1 do k.

W kolejnych k wierszach znajdują się opisy patyczków poszczególnych kolorów. W wierszu o numerze $i + 1$ znajdują się liczby całkowite pooddzielane pojedynczymi odstępami, opisujące patyczki koloru i. Pierwsza z tych liczb, n_i ($1 \leqslant n_i \leqslant 1\,000\,000$), oznacza liczbę patyczków koloru i. Po niej następuje n_i liczb całkowitych oznaczających długości patyczków. Są to liczby całkowite dodatnie nie większe niż 1 000 000 000. Łączna liczba wszystkich patyczków nie przekracza 1 000 000.

W testach wartych przynajmniej 30% punktów zachodzi dodatkowy warunek: sumaryczna liczba patyczków nie przekracza 250.

Wyjście

Twój program powinien wypisać (w pierwszym i jedynym wierszu standardowego wyjścia):

- *albo sześć liczb całkowitych pooddzielanych pojedynczymi odstępami, opisujących sposób zbudowania trójkąta o różnokolorowych bokach w następującym formacie: kolor i długość pierwszego patyczka, kolor i długość drugiego patyczka oraz kolor i długość trzeciego patyczka,*
- *albo słowo* `NIE`*, jeżeli takie trzy patyczki nie istnieją.*

Jeżeli istnieje wiele trójek patyczków w różnych kolorach, z których można zbudować trójkąt, Twój program może wypisać dowolną z nich.

Przykład

Dla danych wejściowych:

```
4
1 42
2 6 9
3 8 4 8
1 12
```

jednym z poprawnych wyników jest:

```
3 8 4 12 2 9
```

natomiast dla danych wejściowych:	*poprawnym wynikiem jest:*

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

Rozwiązanie

Przypadek trzech kolorów

Rozważymy najpierw prostsze zadanie, w którym patyczki są dostępne jedynie w trzech kolorach, tzn. $k = 3$. Oznaczmy te kolory jako czerwony, niebieski i żółty. Przyjmijmy, że trójkąt, którego szukamy, ma najdłuższy bok czerwony, długości c; średni niebieski, długości n; oraz najkrótszy żółty, długości z. Wówczas jedyny warunek potrzebny i wystarczający do jego konstrukcji to $n + z > c$.

Załóżmy, że dobraliśmy już patyczek czerwony długości c. Zauważmy, że jako patyczek niebieski możemy bez straty ogólności dobrać *najdłuższy* patyczek tego koloru, który jest *nie dłuższy* niż c. Istotnie, jeśli dla jakiejś trójki długości c, n, z spełniony jest warunek $n + z > c$, to jeśli zwiększymy n, będzie on tym bardziej spełniony. Podobnie, wybrawszy już patyczek niebieski długości n, z patyczków żółtych możemy dobrać najdłuższy nie dłuższy niż n.

Rozumowanie to prowadzi już do algorytmu. Wpierw sortujemy patyczki każdego koloru względem ich długości i ustawiamy je na trzech listach. Następnie mamy 6 możliwości na to, którego koloru patyczek ma być najdłuższy, którego średni, a którego najkrótszy. Sprawdzamy każdą z tych możliwości; dla ustalenia uwagi przyjmijmy tę, którą rozważaliśmy w poprzednich akapitach. Po kolei iterujemy przez patyczki czerwone, poczynając od najkrótszych. Na listach niebieskich oraz żółtych patyczków trzymamy dwa wskaźniki, wskazujące w każdym momencie na najlepsze możliwe dobory patyczków tych kolorów dla rozważanego patyczka czerwonego. Za każdym razem, gdy rozważamy kolejny patyczek czerwony, zwiększamy wskaźniki na pozostałych listach: przesuwamy wskaźnik na liście niebieskiej, by wskazywał na najdłuższy patyczek nie dłuższy niż czerwony, a potem przesuwamy wskaźnik na liście żółtej, by wskazywał na najdłuższy patyczek nie dłuższy niż niebieski. Następnie sprawdzamy, czy z wskazywanej trójki patyczków da się utworzyć trójkąt. Jeśli tak, to go wypisujemy i kończymy działanie algorytmu, w przeciwnym razie sprawdzamy dalej. Rozumowanie z poprzednich akapitów dowodzi, że jeśli istnieje co najmniej jedna trójka tworząca trójkąt, to którąś z nich znajdziemy.

Oznaczmy przez $N = \sum_{i=1}^{k} n_i$ łączną liczbę patyczków. Pierwsza faza opisanego algorytmu (sortowanie) działa w czasie $O(N \log N)$, zaś druga w czasie $O(N)$, gdyż dla każdej z sześciu kombinacji kolorów wskaźniki przesuwają się po listach jedynie wprzód. Stąd cała procedura ma złożoność $O(N \log N)$. Zauważmy, że bardzo prosto da się ją przerobić na, niestety zbyt wolny, algorytm dla większych wartości k. Po prostu sortujemy każdy z kolorów z osobna, a następnie na k sposobów ustalamy kolor, z którego wybierzemy najdłuższy patyczek, na $k - 1$ — z którego wybierzemy średni, oraz na $k - 2$ — z którego wybierzemy najkrótszy. Dla każdej możliwości stosujemy znany już nam algorytm rozstrzygający, czy przy takim doborze możemy

zbudować szukany trójkąt. Metoda ma więc złożoność czasową $O(N \log N + k^3 N)$. Jej implementacja znajduje się w plikach `pats1.cpp` i `pats2.pas`.

Rozwiązanie wzorcowe

Spróbujemy teraz poprawić algorytm tak, by również w ogólnym przypadku działał w czasie $O(N \log N)$, czyli niezależnie od liczby kolorów. Wyobraźmy sobie, że posortowaliśmy niemalejąco długości wszystkich patyczków naraz, i przy każdym patyczku zapamiętaliśmy, jaki ma kolor. Załóżmy, że istnieje różnokolorowy trójkąt, w którym najdłuższy bok stanowi patyczek P_1, długości d_1 i koloru k_1. Niech P'_2, P'_3 będą pozostałymi dwoma bokami tego trójkąta, odpowiednio długości d'_2, d'_3 ($d'_2 \geqslant d'_3$) oraz kolorów k'_2, k'_3. Wówczas wszystkie kolory k_1, k'_2 i k'_3 są różne oraz $d'_2 + d'_3 > d_1$. Niech dalej d_2 będzie długością najdłuższego patyczka P_2 nie dłuższego niż d_1, koloru innego niż k_1 (powiedzmy k_2). Ponadto, niech d_3 będzie długością najdłuższego patyczka P_3 nie dłuższego niż d_2, koloru innego niż k_1 oraz k_2 (powiedzmy k_3). Pokażemy, że wówczas patyczki P_1, P_2 i P_3 również tworzą poprawny trójkąt.

Wystarczy uzasadnić, że $d_2 \geqslant d'_2$ oraz $d_3 \geqslant d'_3$. Pierwsza nierówność jest oczywista, gdyż $k'_2 \neq k_1$, więc patyczek P'_2 jest brany pod uwagę przy znajdowaniu P_2. Aby dowieść drugiej, załóżmy najpierw, że $k'_2 \neq k_2$. Wówczas przy znajdowaniu P_3 patyczek P'_2 jest brany pod uwagę, gdyż jest nie dłuższy niż d_2 oraz innego koloru niż zarówno P_1, jak i P_2. Stąd $d_3 \geqslant d'_2$, więc też $d_3 \geqslant d'_3$. Teraz załóżmy, że jednak $k'_2 = k_2$. Ale wtedy z kolei patyczek P'_3 jest brany pod uwagę przy doborze P_3, gdyż jest innego koloru niż P_1 oraz P_2, więc w tym przypadku również $d_3 \geqslant d'_3$.

Dowiedziony fakt pozwala już wykonać stosowną modyfikację poprzedniego algorytmu. Jako potencjalne najdłuższe boki trójkąta rozważamy kolejne patyczki P_1 na posortowanej liście długości. Obserwacja z poprzedniego akapitu pozwala dla takiego P_1 sprawdzać tylko jedną parę patyczków jako dwa pozostałe boki: P_2, najdłuższy nie dłuższy od P_1 innego koloru niż P_1; oraz P_3, najdłuższy nie dłuższy od P_2, innego koloru niż P_1 i P_2. Po każdym kroku algorytmu pamiętamy takie trzy patyczki P_1, P_2, P_3 (lub informację, że danego patyczka nie ma). Powiedzmy, że rozważamy kolejny patyczek P. Wówczas aktualizujemy zapamiętane patyczki w następujący sposób:

- jeśli P jest koloru innego niż P_1 i P_2, to $P_3 := P_2$, $P_2 := P_1$, $P_1 := P$;
- jeśli P jest tego samego koloru co P_2, to $P_2 := P_1$, $P_1 := P$;
- jeśli P jest tego samego koloru co P_1, to $P_1 := P$.

Oznacza to, że po wstępnym sortowaniu o złożoności czasowej $O(N \log N)$ możemy już w czasie liniowym sprawdzić, czy z którejś takiej trójki patyczków P_1, P_2, P_3 możemy skonstruować trójkąt, co jest równoważne istnieniu jakiejkolwiek trójki spełniającej warunki zadania.

Implementacje tego rozwiązania można znaleźć w plikach `pat2.cpp` oraz `pat3.pas`.

Testy

Rozwiązania zawodników były sprawdzane z użyciem 13 zestawów składających się jednego, dwóch lub trzech testów.

Nazwa	k	N	Opis
pat1.in	3	15	test losowy, odpowiedź pozytywna
pat2.in	3	50	test losowy, odpowiedź pozytywna
pat3.in	4	150	test losowy, odpowiedź pozytywna
pat4a.in	5	250	test losowy, odpowiedź pozytywna
pat4b.in	4	180	test strukturalny, odpowiedź negatywna
pat5a.in	5	485	test strukturalny, odpowiedź negatywna
pat5b.in	7	32 344	test strukturalny, jednoznaczna odpowiedź pozytywna
pat6a.in	5	1 370	test strukturalny, odpowiedź negatywna
pat6b.in	7	42 695	test strukturalny, jednoznaczna odpowiedź pozytywna
pat7a.in	5	6 881	test strukturalny, odpowiedź negatywna
pat7b.in	6	103 077	test strukturalny, jednoznaczna odpowiedź pozytywna
pat8a.in	11	25 738	test strukturalny, odpowiedź negatywna
pat8b.in	9	155 756	test strukturalny, jednoznaczna odpowiedź pozytywna
pat8c.in	10	101 702	test strukturalny, odpowiedź negatywna
pat9a.in	14	68 830	test strukturalny, odpowiedź negatywna
pat9b.in	9	263 487	test strukturalny, jednoznaczna odpowiedź pozytywna
pat9c.in	11	153 583	test strukturalny, odpowiedź negatywna
pat10a.in	28	500 000	test losowy, odpowiedź pozytywna
pat10b.in	9	323 145	test strukturalny, jednoznaczna odpowiedź pozytywna
pat10c.in	9	236 090	test strukturalny, odpowiedź negatywna
pat11a.in	35	500 000	test losowy, odpowiedź pozytywna
pat11b.in	11	357 247	test strukturalny, jednoznaczna odpowiedź pozytywna
pat11c.in	12	305 079	test strukturalny, odpowiedź negatywna
pat12a.in	45	10^6	test losowy, odpowiedź pozytywna
pat12b.in	10	406 820	test strukturalny, jednoznaczna odpowiedź pozytywna
pat12c.in	9	441 183	test strukturalny, odpowiedź negatywna
pat13a.in	50	10^6	test losowy, odpowiedź pozytywna
pat13b.in	15	474 711	test strukturalny, jednoznaczna odpowiedź pozytywna
pat13c.in	12	531 982	test strukturalny, odpowiedź negatywna

XXIII Międzynarodowa Olimpiada Informatyczna,

Pattaya, Tajlandia 2011

Paweł Idziak, Jakub Łącki
Tłumaczenie

IOI 2011, dzień pierwszy, 24.07.2011

Ogród tropikalny

Zapalony botanik Bajtazar często zabiera grupy studentów do jednego z największych ogrodów tropikalnych w Tajlandii. W ogrodzie znajduje się N fontann ponumerowanych liczbami od 0 do $N-1$. Fontanny połączone są M ścieżkami. Każda ścieżka jest dwukierunkowa i łączy inną parę różnych fontann. Do każdej fontanny prowadzi co najmniej jedna ścieżka. Wokół ścieżek rosną fantastyczne rośliny, które Bajtazar chciałby zobaczyć. Spacer z różnymi grupami studentów może rozpoczynać się przy różnych fontannach.

Bajtazar uwielbia piękne rośliny tropikalne. Od każdej fontanny prowadzi studentów najpiękniejszą ze ścieżek, **chyba że** *jest to ścieżka, którą właśnie przyszli, i mogą pójść w innym kierunku. W takim przypadku grupa idzie drugą pod względem urody ścieżką. Jeśli nie ma wyboru, wraca ona ze skrzyżowania, idąc drugi raz po tej samej ścieżce. Bajtazar jest profesjonalnym botanikiem i żadne dwie ścieżki nie są dla niego równie piękne.*

Studenci nie są zbytnio zainteresowani roślinami, jednak z radością zjedliby obiad w luksusowej restauracji przy fontannie numer P. Bajtazar wie, że studenci zgłodnieją po przejściu dokładnie K ścieżek, przy czym wartość ta może być inna dla każdej grupy studentów. Bajtazar zastanawia się, ile różnych tras spaceru może wybrać dla każdej z grup, przy czym:

- *każda grupa może rozpocząć spacer przy dowolnej fontannie,*
- *kolejne ścieżki na trasie muszą być wybierane zgodnie z powyższymi zasadami, oraz*
- *każda grupa musi zakończyć spacer przy fontannie numer P po przejściu dokładnie K ścieżek.*

Zwróć uwagę, że grupy mogą przechodzić koło fontanny numer P w trakcie swojego spaceru, jednak liczy się tylko to, by cały spacer zakończył się przy tej fontannie.

Zadanie

Dla podanego opisu fontann i ścieżek w ogrodzie, znajdź odpowiedzi na pytanie nurtujące Bajtazara dla Q grup studentów, czyli Q wartości K. Zaimplementuj procedurę **count_routes***(N, M, P, R, Q, G). Jej parametry to:*

- *N — liczba fontann. Fontanny są ponumerowane liczbami od 0 do $N-1$.*
- *M — liczba ścieżek. Ścieżki są ponumerowane liczbami od 0 do $M-1$. Kolejne ścieżki są coraz mniej ładne, tj. dla $0 \leqslant i < M-1$, ścieżka numer i jest ładniejsza niż ścieżka numer $i+1$.*
- *P — fontanna, przy której znajduje się luksusowa restauracja.*
- *R — dwuwymiarowa tablica opisująca ścieżki. Ścieżka numer i ($0 \leqslant i < M$) łączy fontanny o numerach $R[i][0]$ oraz $R[i][1]$. Każda ścieżka łączy parę różnych fontann. Pomiędzy parą fontann istnieje co najwyżej jedna ścieżka.*

- Q — *liczba grup studentów.*
- G — *jednowymiarowa tablica liczb całkowitych zawierająca wartości* K. *Dla* $0 \leqslant i < Q$, $G[i]$ *określa liczbę ścieżek, które przejdzie* i*-ta grupa.*

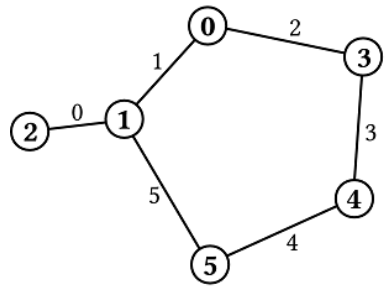
Dla $0 \leqslant i < Q$, *Twoja procedura powinna znaleźć liczbę możliwych tras składających się z dokładnie* $G[i]$ *ścieżek. Każda z tych tras powinna zaprowadzić grupę numer* i *do fontanny numer* P. *Dla każdej grupy, Twoja procedura powinna wywołać procedurę* **answer***(*X*), gdzie* X *to wyznaczona liczba tras. Odpowiedzi należy udzielać w porządku zgodnym z kolejnością grup. Jeśli nie istnieje żadna dopuszczalna trasa, Twoja procedura powinna wywołać procedurę* **answer***(0).*

Przykłady

Przykład 1.

Rozważmy przykład z rysunku numer 1, gdzie $N = 6$, $M = 6$, $P = 0$, $Q = 1$, $G[0] = 3$ *oraz*

$$R = \begin{matrix} 1 & 2 \\ 0 & 1 \\ 0 & 3 \\ 3 & 4 \\ 4 & 5 \\ 1 & 5 \end{matrix}$$



Rysunek 1.

Zauważ, że ścieżki są podane w kolejności od najpiękniejszej do najmniej ładnej.
Istnieją dwie poprawne trasy długości 3:

- $1 \to 2 \to 1 \to 0$ *oraz*
- $5 \to 4 \to 3 \to 0$.

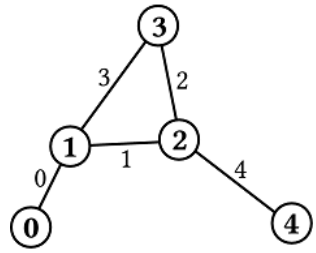
Pierwsza trasa rozpoczyna się przy fontannie numer 1. Najpiękniejsza ścieżka prowadzi do fontanny numer 2. Przy fontannie numer 2 grupa nie ma wyboru, zatem wraca po tej samej ścieżce. Po powrocie do fontanny numer 1 grupa ominie ścieżkę numer 0 i zamiast niej wybierze ścieżkę numer 1. Zaprowadzi ona grupę do fontanny numer $P = 0$.

Procedura powinna zatem wywołać **answer***(2).*

Przykład 2.

Rozważmy przykład z rysunku numer 2, gdzie $N = 5$, $M = 5$, $P = 2$, $Q = 2$, $G[0] = 3$, $G[1] = 1$ *oraz*

$$R = \begin{array}{cc} 1 & 0 \\ 1 & 2 \\ 3 & 2 \\ 1 & 3 \\ 4 & 2 \end{array}$$



Rysunek 2.

Dla pierwszej grupy studentów istnieje tylko jedna poprawna trasa kończąca się przy fontannie numer 2 po przejściu 3 ścieżek: $1 \to 0 \to 1 \to 2$.

Dla drugiej grupy studentów istnieją dwie poprawne trasy prowadzące do fontanny numer 2 w jednym kroku: $3 \to 2$ *oraz* $4 \to 2$.

Zatem poprawna implementacja procedury **count_routes** *powinna wywołać najpierw* **answer***(1), aby udzielić odpowiedzi dla pierwszej grupy studentów, a następnie wywołać* **answer***(2), by udzielić odpowiedzi dla drugiej grupy studentów.*

Podzadania

Podzadanie 1. (49 punktów)

- $2 \leqslant N \leqslant 1\,000$
- $1 \leqslant M \leqslant 10\,000$
- $Q = 1$
- *Każdy element tablicy* G *to liczba całkowita z przedziału* $[1,\ 100]$.

Podzadanie 2. (20 punktów)

- $2 \leqslant N \leqslant 150\,000$
- $1 \leqslant M \leqslant 150\,000$
- $Q = 1$
- *Każdy element tablicy* G *to liczba całkowita z przedziału* $[1,\ 1\,000\,000\,000]$.

Podzadanie 3. (31 punktów)

- $2 \leqslant N \leqslant 150\,000$
- $1 \leqslant M \leqslant 150\,000$
- $1 \leqslant Q \leqslant 2\,000$
- *Każdy element tablicy* G *to liczba całkowita z przedziału* $[1,\ 1\,000\,000\,000]$.

Szczegóły techniczne

Ograniczenia

- *Limit czasu procesora: 5 sekund*

- *Dostępna pamięć: 256 MB*
  Uwaga: *Nie ma osobnego ograniczenia na rozmiar stosu. Pamięć użyta przez stos wlicza się w całkowity rozmiar używanej pamięci.*

Interfejs (API)

- *Katalog na pliki źródłowe:* `garden/`
- *Nazwa pliku z rozwiązaniem:* `garden.c` *lub* `garden.cpp`, *lub* `garden.pas`
- *Interfejs procedur zawodnika:* `garden.h` *lub* `garden.pas`
- *Interfejs procedur modułu oceniającego:* `gardenlib.h` *lub* `gardenlib.pas`
- *Przykładowy moduł oceniający:* `grader.c` *lub* `grader.cpp`, *lub* `grader.pas`
- *Pliki wejściowe dla przykładowego modułu oceniającego:* `grader.in.1`, `grader.in.2`, ...
  Uwaga: *Przykładowy moduł oceniający wczytuje dane zgodnie z poniższym formatem:*
  - *Wiersz 1:* N, M *oraz* P.
  - *Wiersze od 2 do* $M + 1$*: opis ścieżek; wiersz* $i + 2$ $(0 \leqslant i < M)$ *zawiera liczby* $R[i][0]$ *oraz* $R[i][1]$ *oddzielone pojedynczym odstępem.*
  - *Wiersz* $M + 2$*:* Q.
  - *Wiersz* $M + 3$*: tablica* G *podana jako ciąg liczb całkowitych, poodzielanych pojedynczymi odstępami.*
  - *Wiersz* $M + 4$*: tablica oczekiwanych odpowiedzi podana jako ciąg liczb całkowitych poodzielanych pojedynczymi odstępami.*
- *Pliki z oczekiwaną odpowiedzią dla przykładowych wejść przykładowego modułu oceniającego:* `grader.expect.1`, `grader.expect.2`, ...
  W tym zadaniu powyższe pliki powinny zawierać jedynie napis „Correct".

Paweł Idziak, Jakub Łącki
Tłumaczenie

IOI 2011, dzień pierwszy, 24.07.2011

Silos ryżowy

Rzecz dzieje się na długiej, prostej drodze, zwanej Drogą Ryżową. Wzdłuż tej drogi rozmieszczonych jest R pól ryżowych. Położenie każdego z tych pól jest określone jedną współrzędną — liczbą całkowitą, mieszczącą się w zakresie od 1 do L. Pola ryżowe są podane w kolejności niemalejących współrzędnych. Formalnie, dla $0 \leqslant i < R$, pole ryżowe o numerze i ma współrzędną $X[i]$, przy czym $1 \leqslant X[0] \leqslant \ldots \leqslant X[R-1] \leqslant L$.

Może zdarzyć się, że **wiele pól ryżowych będzie miało tę samą współrzędną**.

Planujemy budowę jednego **silosu ryżowego**, *w którym będziemy przechowywać możliwie dużo zebranego ryżu. Podobnie jak w przypadku pól ryżowych,* **silos ryżowy jest wyznaczony przez całkowitoliczbową współrzędną** *z zakresu od 1 do L. Silos ryżowy może być położony w dowolnym miejscu, w szczególności na takiej samej współrzędnej jak jedno lub wiele pól ryżowych.*

Z każdego pola ryżowego można co roku zebrać **dokładnie jedną ciężarówkę** *ryżu. Miasto planuje zatrudnienie kierowcy do przewozu ryżu do silosu. Kierowca pobiera opłatę 1 bata za każdą jednostkę odległości przebytą w kierunku silosu. Innymi słowy, koszt transportu jednej ciężarówki ryżu z danego pola do silosu jest równy różnicy między ich współrzędnymi.*

Niestety, nasz budżet w tym roku jest napięty: możemy wydać jedynie B batów na przewóz ryżu. Twoim zadaniem jest pomóc nam tak umieścić silos, by przewieźć do niego jak najwięcej ryżu.

Zadanie

Zaimplementuj procedurę **besthub**(R, L, X, B). *Jej parametry to:*

- R — *liczba pól ryżowych. Pola ryżowe są ponumerowane liczbami od 0 do $R-1$.*
- L — *maksymalna wielkość współrzędnych.*
- X — *jednowymiarowa tablica liczb całkowitych, uporządkowana w kolejności niemalejącej. Dla każdego $0 \leqslant i < R$, pole ryżowe numer i ma współrzędną $X[i]$.*
- B — *wielkość budżetu.*

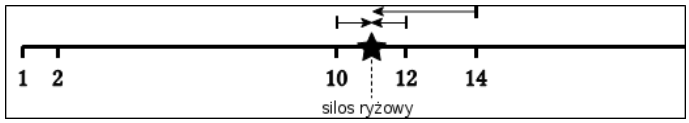
Twoja procedura ma za zadanie znaleźć optymalną lokalizację silosu i zwrócić maksymalną liczbę ciężarówek ryżu, które mogą być do niego przewiezione.

Zauważ, że całkowity koszt przewozu ryżu może być bardzo duży. Budżet podany jest jako 64-bitowa liczba całkowita, dlatego zalecamy wykonywanie obliczeń na 64-bitowych liczbach całkowitych. W C/C++ używaj typu `long long`, *zaś w Pascalu — typu* `Int64`.

Przykład

Rozważmy przypadek, gdy $R = 5$, $L = 20$, $B = 6$ oraz

$X = \begin{matrix} 1 \\ 2 \\ 10 \\ 12 \\ 14 \end{matrix}$



Rysunek 1.

W tym przypadku istnieje wiele optymalnych lokalizacji silosu: może on być ulokowany na dowolnej współrzędnej o wartości pomiędzy 10 a 14. Powyższy rysunek pokazuje jedną z takich optymalnych lokalizacji silosu. Będziemy wtedy mogli przewieźć do niego ryż z pól o współrzędnych 10, 12 i 14. Dla każdej takiej optymalnej lokalizacji silosu, łączny koszt przewozu ryżu nie przekroczy wtedy sześciu batów. Oczywiście, żadna inna lokalizacja silosu nie umożliwi przewozu ryżu z więcej niż trzech pól i dlatego to rozwiązanie jest optymalne. Procedura **besthub** *powinna więc zwrócić 3.*

Podzadania

Podzadanie 1. (17 punktów)

- $1 \leqslant R \leqslant 100$
- $1 \leqslant L \leqslant 100$
- $0 \leqslant B \leqslant 10\,000$
- *Żadne dwa pola nie mają tej samej współrzędnej (tylko w tym podzadaniu).*

Podzadanie 2. (25 punktów)

- $1 \leqslant R \leqslant 500$
- $1 \leqslant L \leqslant 10\,000$
- $0 \leqslant B \leqslant 1\,000\,000$

Podzadanie 3. (26 punktów)

- $1 \leqslant R \leqslant 5\,000$
- $1 \leqslant L \leqslant 1\,000\,000$
- $0 \leqslant B \leqslant 2\,000\,000\,000$

Podzadanie 4. (32 punkty)

- $1 \leqslant R \leqslant 100\,000$
- $1 \leqslant L \leqslant 1\,000\,000\,000$
- $0 \leqslant B \leqslant 2\,000\,000\,000\,000\,000$

Szczegóły techniczne

Ograniczenia

- *Limit czasu procesora: 1 sekunda*
- *Dostępna pamięć: 256 MB*
  Uwaga: *Nie ma osobnego ograniczenia na rozmiar stosu. Pamięć użyta przez stos wlicza się w całkowity rozmiar używanej pamięci.*

Interfejs (API)

- *Katalog na pliki źródłowe:* `ricehub/`
- *Nazwa pliku z rozwiązaniem:* `ricehub.c` *lub* `ricehub.cpp`*, lub* `ricehub.pas`

- *Interfejs procedur zawodnika:* `ricehub.h` *lub* `ricehub.pas`
- *Przykładowy moduł oceniający:* `grader.c` *lub* `grader.cpp`*, lub* `grader.pas`
- *Pliki wejściowe dla przykładowego modułu oceniającego:* `grader.in.1`*,* `grader.in.2`*,* ...
  Uwaga: *Przykładowy moduł oceniający wczytuje dane zgodnie z poniższym formatem:*
  - *Wiersz 1:* R*,* L *oraz* B*.*
  - *Wiersze od 2 do* $R + 1$*: informacje o lokalizacji pól ryżowych; wiersz* $i + 2$ *zawiera liczbę* $X[i]$*, dla* $0 \leqslant i < R$*.*
  - *Wiersz* $R + 2$*: oczekiwana odpowiedź.*
- *Pliki z oczekiwaną odpowiedzią dla przykładowych wejść przykładowego modułu oceniającego:* `grader.expect.1`*,* `grader.expect.2`*,* ...
  W tym zadaniu powyższe pliki powinny zawierać jedynie napis „Correct".

Paweł Idziak, Jakub Łącki
Tłumaczenie

IOI 2011, dzień pierwszy, 24.07.2011

Wyścig

W tym roku, równolegle do IOI, w Pattaya odbędzie się Międzynarodowy Wyścig Samochodowy (MWS) 2011. Obowiązkiem organizatorów jest wytyczenie jak najlepszej trasy wyścigu.

W regionie Pattaya-Chonburi leży N miast połączonych siecią $N-1$ autostrad. Każda autostrada jest dwukierunkowa, łączy dwa różne miasta, a jej długość w kilometrach wyraża się liczbą całkowitą. Dodatkowo, istnieje **dokładnie jedna** *ścieżka łącząca każdą parę miast. Innymi słowy, pomiędzy każdą parą miast istnieje dokładnie jedna trasa składająca się z ciągu autostrad, która nie przebiega dwukrotnie przez żadne miasto.*

Regulamin MWS mówi, że trasa wyścigu powinna być ścieżką o długości **dokładnie** *K kilometrów i powinna się zaczynać i kończyć w różnych miastach. Aby uniknąć wypadków, trasa nie może dwukrotnie przebiegać po żadnej autostradzie (ani przez żadne miasto). Wyścig może spowodować utrudnienia w ruchu, dlatego jego trasa powinna składać się z jak najmniejszej liczby autostrad.*

Zadanie

Zaimplementuj procedurę **best_path**(N, K, H, L)*. Jej parametry to:*

- N — *liczba miast. Miasta są ponumerowane liczbami od 0 do $N-1$.*
- K — *długość trasy.*
- H — *dwuwymiarowa tablica opisująca autostrady. Dla $0 \leqslant i < N-1$, autostrada i łączy miasta $H[i][0]$ oraz $H[i][1]$.*
- L — *jednowymiarowa tablica opisująca długości autostrad. Dla $0 \leqslant i < N-1$, długością autostrady i jest $L[i]$.*

Możesz założyć, że wszystkie wartości w tablicy H zawierają się w przedziale $[0, N-1]$ oraz że autostrady łączą wszystkie miasta w sposób opisany powyżej. Ponadto, wartości w tablicy L są liczbami całkowitymi z przedziału $[0,\ 1\,000\,000]$.

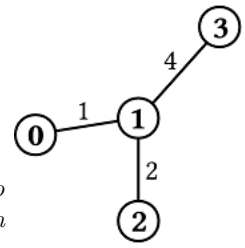
Twoja procedura powinna zwrócić **najmniejszą możliwą liczbę autostrad***, z których może składać się poprawna trasa o długości dokładnie K. Jeśli taka trasa nie istnieje, Twoja procedura powinna zwrócić -1.*

Przykłady

Przykład 1

Rozważmy przykład z rysunku numer 1, gdzie $N = 4$, $K = 3$,

$$H = \begin{matrix} 0 & 1 \\ 1 & 2 \\ 1 & 3 \end{matrix} \qquad L = \begin{matrix} 1 \\ 2 \\ 4 \end{matrix}$$



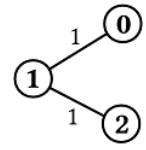
Rysunek 1.

Wyścig może zacząć się w mieście numer 0 i prowadzić przez miasto numer 1 do miasta numer 2. Trasa będzie miała długość 1 km + 2 km = 3 km i składać się będzie z dwu autostrad. Jest to najlepsza możliwa trasa wyścigu, dlatego **best_path***(N, K, H, L) powinno zwrócić 2.*

Przykład 2

Rozważmy przykład z rysunku numer 2, gdzie $N = 3$, $K = 3$,

$$H = \begin{matrix} 0 & 1 \\ 1 & 2 \end{matrix} \qquad L = \begin{matrix} 1 \\ 1 \end{matrix}$$



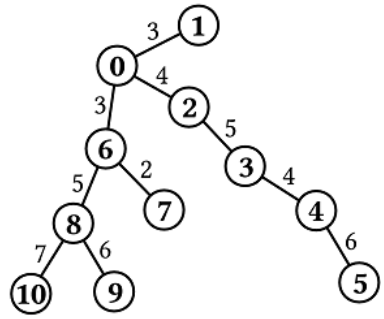
Rysunek 2.

Nie istnieje poprawna trasa wyścigu. W tym przypadku **best_path***(N, K, H, L) powinno zwrócić -1.*

Przykład 3

Rozważmy przykład z rysunku numer 3, gdzie $N = 11$, $K = 12$,

$$H = \begin{matrix} 0 & 1 \\ 0 & 2 \\ 2 & 3 \\ 3 & 4 \\ 4 & 5 \\ 0 & 6 \\ 6 & 7 \\ 6 & 8 \\ 8 & 9 \\ 8 & 10 \end{matrix} \qquad L = \begin{matrix} 3 \\ 4 \\ 5 \\ 4 \\ 6 \\ 3 \\ 2 \\ 5 \\ 6 \\ 7 \end{matrix}$$



Rysunek 3.

Jedna z możliwych tras składa się z 3 autostrad: z miasta 6 przez miasta 0 i 2 do miasta 3. Można też poprowadzić trasę z miasta 10 przez miasto 8 do miasta 6. Obydwie trasy wyścigu mają długość 12 km, zgodnie z wymaganiami. Druga z nich jest optymalna, bo nie istnieje trasa składająca się z jednej autostrady. Zatem **best_path***(N, K, H, L) powinno zwrócić 2.*

Podzadania

Podzadanie 1. (9 punktów)

- $1 \leqslant N \leqslant 100$
- $1 \leqslant K \leqslant 100$
- *Sieć drogowa tworzy linię, tj. dla* $0 \leqslant i < N - 1$, *autostrada* i *łączy miasta* i *oraz* $i + 1$.

Podzadanie 2. (12 punktów)

- $1 \leqslant N \leqslant 1\,000$
- $1 \leqslant K \leqslant 1\,000\,000$

Podzadanie 3. (22 punkty)

- $1 \leqslant N \leqslant 200\,000$
- $1 \leqslant K \leqslant 100$

Podzadanie 4. (57 punktów)

- $1 \leqslant N \leqslant 200\,000$
- $1 \leqslant K \leqslant 1\,000\,000$

Szczegóły techniczne

Ograniczenia

- *Limit czasu procesora: 3 sekundy*
- *Dostępna pamięć: 256 MB*
  Uwaga: *Nie ma osobnego ograniczenia na rozmiar stosu. Pamięć użyta przez stos wlicza się w całkowity rozmiar używanej pamięci.*

Interfejs (API)

- *Katalog na pliki źródłowe:* `race/`
- *Nazwa pliku z rozwiązaniem:* `race.c` *lub* `race.cpp`, *lub* `race.pas`
- *Interfejs procedur zawodnika:* `race.h` *lub* `race.pas`
- *Interfejs procedur modułu oceniającego:* `race.h` *lub* `racelib.pas`
- *Przykładowy moduł oceniający:* `grader.c` *lub* `grader.cpp`, *lub* `grader.pas`
- *Pliki wejściowe dla przykładowego modułu oceniającego:* `grader.in.1`, `grader.in.2`, ...
  Uwaga: *Przykładowy moduł oceniający wczytuje dane zgodnie z poniższym formatem:*

  - *Wiersz 1: N oraz K.*
  - *Wiersze od 2 do N: opis autostrad; wiersz $i + 2$ (dla $0 \leqslant i < N - 1$) zawiera liczby $H[i][0]$, $H[i][1]$ oraz $L[i]$ pooddzielane pojedynczymi odstępami.*
  - *Wiersz $N + 1$: oczekiwana odpowiedź.*

- *Pliki z oczekiwaną odpowiedzią dla przykładowych wejść przykładowego modułu oceniającego:* `grader.expect.1`, `grader.expect.2`, ...
  W tym zadaniu powyższe pliki powinny zawierać jedynie napis „Correct”.

Paweł Idziak, Jakub Łącki
Tłumaczenie

IOI 2011, dzień drugi, 26.07.2011

Krokodyl

Archeolog Bajtazar ryzykuje życie, starając się poznać tajemniczą, podziemną kryjówkę Krokodyla. Kryjówka ta składa się z N komnat połączonych M dwukierunkowymi korytarzami. Każdy korytarz łączy dwie różne komnaty. Pomiędzy każdą parą komnat istnieje co najwyżej jeden korytarz. Ponadto, dla każdego korytarza wiemy, ile czasu zajmuje jego przebycie. W K spośród N komnat znajdują się wyjścia na powierzchnię ziemi. Bajtazar przebywa początkowo w komnacie 0 i chciałby się jak najszybciej dostać do komnaty z wyjściem.

Krokodyl chce przeszkodzić Bajtazarowi w ucieczce. Ze swojej jamy obsługuje mechanizm pozwalający na zablokowanie dowolnego korytarza, przy czym system pozwala, by co najwyżej jeden korytarz był w danym momencie zamknięty. Gdy Krokodyl chce zamknąć pewien korytarz, poprzednio zablokowany korytarz zostaje natychmiast otwarty.

Z punktu widzenia Bajtazara sytuacja wygląda następująco. Za każdym razem, gdy chce on opuścić pewną komnatę, Krokodyl może zamknąć jeden z wychodzących z niej korytarzy. Bajtazar wybiera wtedy jeden z niezablokowanych korytarzy i zmierza nim do innej komnaty. Korytarz, do którego wszedł Bajtazar, nie może być zablokowany przez Krokodyla aż do momentu, gdy Bajtazar dojdzie do komnaty na jego końcu. Gdy Bajtazar dotrze do następnej komnaty, Krokodyl może znów zamknąć jeden z wychodzących z niej korytarzy (być może ten, którym Bajtazar właśnie przyszedł), i tak dalej.

Bajtazar chciałby zawczasu przygotować plan ucieczki z podziemi. Mówiąc dokładniej, chciałby przygotować zestaw zasad poruszania się pomiędzy komnatami. Niech A będzie jedną z komnat. Jeśli jest to komnata z wyjściem na powierzchnię ziemi, Bajtazar nie potrzebuje żadnego planu, bo może natychmiast wydostać się z podziemi. W przeciwnym przypadku, instrukcja dla komnaty A może mieć jedną z poniższych postaci:

- *Jeśli jesteś w komnacie A, idź korytarzem do komnaty B. Jeśli jednak ten korytarz jest zablokowany, wybierz korytarz do komnaty C.*
- *Nie przejmuj się komnatą A. Nie ma możliwości, byś do niej dotarł, jeśli podążasz zgodnie z planem ucieczki.*

Zwróć uwagę, że Krokodyl może czasem (na przykład, jeśli plan każe Bajtazarowi poruszać się po cyklu) uniemożliwić ucieczkę Bajtazarowi. Plan ucieczki jest **dobry**, *jeśli Bajtazar dotrze do wyjścia w skończonym czasie, niezależnie od tego, co zrobi Krokodyl. Dla dobrego planu ucieczki, oznaczmy przez T najmniejszą taką liczbę, że w ciągu czasu T Bajtazar* **na pewno** *dostanie się do komnaty z wyjściem na powierzchnię. Powiemy wówczas, że* **dobry plan ucieczki wymaga czasu** *T.*

Zadanie

Zaimplementuj procedurę **travel_plan**(N, M, R, L, K, P). *Jej parametry to:*

- N — *liczba komnat. Komnaty są ponumerowane liczbami od 0 do $N-1$.*
- M — *liczba korytarzy. Korytarze są ponumerowane liczbami od 0 do $M-1$.*

- R — *dwuwymiarowa tablica liczb całkowitych opisująca korytarze. Dla* $0 \leqslant i < M$, *korytarz* i *łączy dwie różne komnaty* $R[i][0]$ *oraz* $R[i][1]$. *Każdy korytarz łączy inną parę komnat.*
- L — *jednowymiarowa tablica liczb całkowitych, w której wartość* $L[i]$ *opisuje czas potrzebny na przebycie korytarza* i. *Dla* $0 \leqslant i < M$ *zachodzi* $1 \leqslant L[i] \leqslant 1\,000\,000\,000$.
- K — *liczba komnat, w których znajdują się wyjścia na powierzchnię. Spełnia ona nierówności* $1 \leqslant K \leqslant N$.
- P — *jednowymiarowa tablica* K *różnych liczb całkowitych. Dla* $0 \leqslant i < K$, *wartość* $P[i]$ *jest równa numerowi* i*-tej komnaty wyjściowej. W komnacie 0 nigdy nie znajduje się wyjście.*

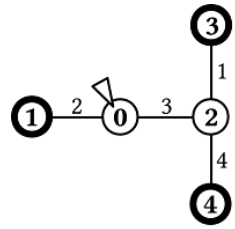
Twoja procedura powinna zwrócić najmniejszą możliwą liczbę T, *taką że istnieje dobry plan ucieczki wymagający czasu* T. *Możesz założyć, że z każdej komnaty, która nie jest komnatą wyjściową, wychodzą co najmniej dwa korytarze. Możesz też założyć, że zawsze będzie istniał plan ucieczki, dla którego* $T \leqslant 1\,000\,000\,000$.

Przykłady

Przykład 1

Rozważmy przykład przedstawiony na rysunku numer 1, w którym $N = 5$, $M = 4$, $K = 3$ *oraz*

$$R = \begin{matrix} 0 & 1 \\ 0 & 2 \\ 3 & 2 \\ 2 & 4 \end{matrix} \qquad L = \begin{matrix} 2 \\ 3 \\ 1 \\ 4 \end{matrix} \qquad P = \begin{matrix} 1 \\ 3 \\ 4 \end{matrix}$$



Rysunek 1.

Komnaty wyjściowe zostały zaznaczone pogrubionymi kółkami. Bajtazar znajduje się początkowo w komnacie 0 (dodatkowo zaznaczonej trójkątem). Optymalny plan ucieczki wygląda następująco:

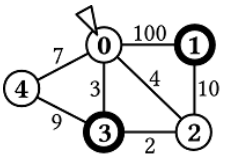
- *Jeśli jesteś w komnacie 0, idź korytarzem do komnaty 1.*
  Jeśli jednak ten korytarz jest zablokowany, wybierz korytarz do komnaty 2.
- *Jeśli jesteś w komnacie 2, idź korytarzem do komnaty 3.*
  Jeśli jednak ten korytarz jest zablokowany, wybierz korytarz do komnaty 4.

W najgorszym przypadku Bajtazar dotrze do komnaty wyjściowej po 7 jednostkach czasu. Zatem procedura **travel_plan** *powinna zwrócić 7.*

Przykład 2

Rozważmy przykład przedstawiony na rysunku numer 2, w którym $N = 5$, $M = 7$, $K = 2$ *oraz*

$$R = \begin{array}{cc} 0 & 2 \\ 0 & 3 \\ 3 & 2 \\ 2 & 1 \\ 0 & 1 \\ 0 & 4 \\ 3 & 4 \end{array} \qquad L = \begin{array}{c} 4 \\ 3 \\ 2 \\ 10 \\ 100 \\ 7 \\ 9 \end{array} \qquad P = \begin{array}{c} 1 \\ 3 \end{array}$$



Rysunek 2.

Optymalny plan ucieczki wygląda następująco:

- *Jeśli jesteś w komnacie 0, idź korytarzem do komnaty 3.*
  Jeśli jednak ten korytarz jest zablokowany, wybierz korytarz do komnaty 2.
- *Jeśli jesteś w komnacie 2, idź korytarzem do komnaty 3.*
  Jeśli jednak ten korytarz jest zablokowany, wybierz korytarz do komnaty 1.
- *Nie przejmuj się komnatą 4.*
  Nie ma możliwości, byś do niej dotarł, jeśli podążasz zgodnie z planem ucieczki.

Bajtazar dotrze do pewnej komnaty wyjściowej nie później niż po 14 jednostkach czasu. Zatem procedura **travel_plan** *powinna zwrócić 14.*

Podzadania

Podzadanie 1. (46 punktów)

- $3 \leqslant N \leqslant 1\,000$
- *Sieć korytarzy tworzy drzewo. Oznacza to, że* $M = N - 1$ *i dla każdej pary komnat* i *oraz* j*, istnieje ciąg korytarzy łączący* i *z* j*.*
- *Z każdej komnaty wyjściowej wychodzi dokładnie jeden korytarz.*
- *Każda inna komnata jest bezpośrednio połączona z trzema lub więcej innymi komnatami.*

Podzadanie 2. (43 punkty)

- $3 \leqslant N \leqslant 1\,000$
- $2 \leqslant M \leqslant 100\,000$

Podzadanie 3. (11 punktów)

- $3 \leqslant N \leqslant 100\,000$
- $2 \leqslant M \leqslant 1\,000\,000$

Szczegóły techniczne

Ograniczenia

- *Limit czasu procesora: 2 sekundy*
- *Dostępna pamięć: 256 MB*
  Uwaga: *Nie ma osobnego ograniczenia na rozmiar stosu. Pamięć użyta przez stos wlicza się w całkowity rozmiar używanej pamięci.*

Interfejs (API)

- *Katalog na pliki źródłowe:* `crocodile/`
- *Nazwa pliku z rozwiązaniem:* `crocodile.c` *lub* `crocodile.cpp`, *lub* `crocodile.pas`
- *Interfejs procedur zawodnika:* `crocodile.h` *lub* `crocodile.pas`
- *Interfejs procedur modułu oceniającego:* `crocodile.h` *lub* `crocodilelib.pas`
- *Przykładowy moduł oceniający:* `grader.c` *lub* `grader.cpp`, *lub* `grader.pas`
- *Pliki wejściowe dla przykładowego modułu oceniającego:* `grader.in.1`, `grader.in.2`, ...
  Uwaga: *Przykładowy moduł oceniający wczytuje dane zgodnie z poniższym formatem:*
  - *Wiersz 1:* N, M *oraz* K.
  - *Wiersze 2 do* $M + 1$*: Wiersz* $i + 2$ $(0 \leqslant i < M)$ *zawiera* $R[i][0]$, $R[i][1]$ *oraz* $L[i]$ *pooddzielane pojedynczymi odstępami.*
  - *Wiersz* $M + 2$*: lista* K *liczb całkowitych* $P[0], P[1], \ldots, P[K-1]$ *pooddzielanych pojedynczymi odstępami.*
  - *Wiersz* $M + 3$*: oczekiwana odpowiedź.*
- *Pliki z oczekiwaną odpowiedzią dla przykładowych wejść przykładowego modułu oceniającego:* `grader.expect.1`, `grader.expect.2`, ...
  W tym zadaniu powyższe pliki powinny zawierać jedynie napis „Correct".

Paweł Idziak, Jakub Łącki
Tłumaczenie

IOI 2011, dzień drugi, 26.07.2011

Papugi

Bajtazarka interesuje się ornitologią. Od kiedy przeczytała o protokole **IP over Avian Carriers** *(IPoAC, protokół transferu pakietów IP przez gołębie pocztowe), spędza sporo czasu, ucząc hordę inteligentnych papug przenoszenia wiadomości na długich dystansach.*

Marzeniem Bajtazarki jest użycie papug do wysłania wiadomości M do odległego kraju. Wiadomość M jest ciągiem N liczb całkowitych (niekoniecznie różnych), z przedziału $[0,\ 255]$. Bajtazarka hoduje K specjalnie przeszkolonych papug. Wszystkie papugi są identyczne i nawet Bajtazarka nie potrafi ich rozróżnić. Każdy ptak potrafi zapamiętać liczbę z przedziału $[0, R]$.

Pewnego razu Bajtazarka spróbowała przesłać wiadomość. W tym celu wypuszczała papugi z klatki jedna za drugą. Zanim każdy ptak wzbił się w powietrze, Bajtazarka uczyła go kolejnej liczby z ciągu. Niestety, pomysł nie wypalił. Wprawdzie ptaki dotarły do celu, jednak w innej kolejności niż powinny. To pozwoliło Bajtazarce odtworzyć wszystkie wysłane liczby, jednak nie potrafiła odtworzyć ich prawidłowej kolejności.

Aby spełnić swe marzenie, Bajtazarka potrzebuje lepszej metody i dlatego prosi Cię o pomoc. Dla podanej wiadomości M, chce ona wypuszczać ptaki jeden po drugim, tak jak poprzednio. Poprosiła Cię o napisanie programu, który wykona dwie operacje:

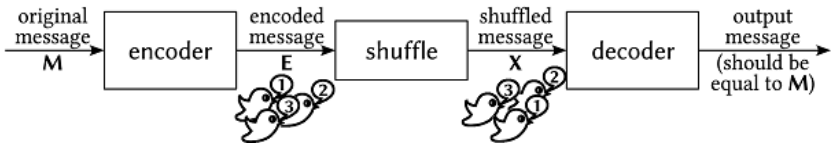
- *Po pierwsze, powinien umożliwić wczytanie wiadomości M i przetworzenie jej w ciąg co najwyżej K liczb całkowitych z przedziału $[0, R]$. Bajtazarka nauczy papugi kolejnych liczb z wyznaczonego ciągu.*
- *Po drugie, program powinien umożliwić wczytanie listy liczb całkowitych z przedziału $[0, R]$, która opisuje liczby dostarczone do celu przez papugi. Z podanej listy powinien odtworzyć oryginalną wiadomość M.*

Możesz założyć, że wszystkie papugi zawsze docierają do celu i że każda z nich poprawnie pamięta podany jej numer. Bajtazarka przypomina raz jeszcze, że papugi mogą przylecieć w **dowolnej** *kolejności. Zwróć uwagę, że Bajtazarka dysponuje K papugami, więc wyznaczony ciąg liczb całkowitych z przedziału $[0, R]$ musi składać się z co najwyżej K liczb.*

Zadanie

Napisz dwie osobne procedury. Jedną dla nadawcy (encoder), zaś drugą dla odbiorcy (decoder). Cały proces został pokazany na poniższym rysunku (shuffle oznacza pewne przetasowanie kolejności ptaków).

Tłumaczenie rysunku: oryginalna wiadomość M, zakodowana wiadomość E, przetasowana wiadomość X, wiadomość wyjściowa (powinna być równa M).



Procedury, które należy napisać, to:

- *Procedura* **encode***(N, M). Jej parametry są następujące:*
  - *N — długość wiadomości.*
  - *M — jednowymiarowa tablica N liczb całkowitych, które opisują wiadomość. Możesz założyć, że dla* $0 \leqslant i < N$ *zachodzi* $0 \leqslant M[i] \leqslant 255$.

  Ta procedura powinna zakodować wiadomość M do ciągu liczb całkowitych z przedziału $[0, R]$*, który należy nadać. Procedura* **encode** *podaje wyznaczony ciąg przez wywołanie* **send***(a) dla każdej liczby całkowitej a, którą trzeba wysłać przy pomocy papug.*
- *Procedura* **decode***(N, L, X). Jej parametry są następujące:*
  - *N — długość oryginalnej wiadomości.*
  - *L — długość otrzymanej wiadomości, tj. liczba wysłanych ptaków.*
  - *X — jednowymiarowa tablica L liczb całkowitych, które opisują otrzymane liczby. Liczby* $X[i]$ $(0 \leqslant i < L)$ *to dokładnie te same liczby, które wyznaczyła Twoja implementacja procedury* **encode***, jednak być może w innej kolejności.*

  Ta procedura powinna odtworzyć oryginalną wiadomość. Aby podać wynik, powinna ona wywołać **output***(b) dla kolejnych liczb b tworzących odkodowaną wiadomość, we właściwej kolejności.*

Zwróć uwagę, że R i K nie są podane jako parametry, lecz są ustalone w poszczególnych podzadaniach.

Aby poprawnie rozwiązać dane podzadanie, procedury muszą spełnić poniższe warunki:

- *Wszystkie liczby wyznaczone przez procedurę* **encode** *muszą mieścić się w zakresie określonym w podzadaniu.*
- *Liczba wywołań procedury* **send** *w procedurze* **encode** *nie może przekroczyć ograniczenia K określonego w podzadaniu.*
- *Procedura* **decode** *musi poprawnie odtworzyć oryginalną wiadomość M oraz wywołać* **output***(b) dokładnie N razy, dla b równego kolejno* $M[0], M[1], \ldots, M[N-1]$.

W ostatnim podzadaniu, Twój wynik zależy od ilorazu długości zakodowanej wiadomości oraz oryginalnej wiadomości.

Przykład

Rozważmy przykład, w którym $N = 3$ *oraz*

$$M = \begin{matrix} 10 \\ 30 \\ 20 \end{matrix}$$

Procedura **encode***(N, M), przy użyciu sobie znanego sposobu, może zakodować wiadomość do ciągu (7, 3, 2, 70, 15, 20, 3). Aby podać wyznaczony ciąg, powinna wywołać:*

send*(7)*
send*(3)*
send*(2)*
send*(70)*
send*(15)*
send*(20)*
send*(3)*

Załóżmy teraz, że wszystkie papugi doleciały do celu, a otrzymany ciąg liczb to (3, 20, 70, 15, 2, 3, 7). Procedura **decode** *zostanie wówczas wywołana z parametrami* $N = 3$, $L = 7$ *oraz*

$$X = \begin{matrix} 3 \\ 20 \\ 70 \\ 15 \\ 2 \\ 3 \\ 7 \end{matrix}$$

Procedura **decode** *musi odtworzyć oryginalną wiadomość (10, 30, 20). Powinna ona podać odpowiedź przez wywołanie*

output*(10)*
output*(30)*
output*(20)*

Podzadania

Podzadanie 1. (17 punktów)

- $N = 8$, *a każda liczba w tablicy* M *to 0 lub 1.*
- *Każda zakodowana liczba musi mieścić się w przedziale od 0 do* $R = 65535$.
- *Maksymalna dopuszczalna liczba wywołań procedury* **send** *wynosi* $K = 10 \cdot N$.

Podzadanie 2. (17 punktów)

- $1 \leqslant N \leqslant 16$
- *Każda zakodowana liczba musi mieścić się w przedziale od 0 do* $R = 65535$.
- *Maksymalna dopuszczalna liczba wywołań procedury* **send** *wynosi* $K = 10 \cdot N$.

Podzadanie 3. (18 punktów)

- $1 \leqslant N \leqslant 16$
- *Każda zakodowana liczba musi mieścić się w przedziale od 0 do* $R = 255$.
- *Maksymalna dopuszczalna liczba wywołań procedury* **send** *wynosi* $K = 10 \cdot N$.

Podzadanie 4. (29 punktów)

- $1 \leqslant N \leqslant 32$
- *Każda zakodowana liczba musi mieścić się w przedziale od 0 do* $R = 255$.
- *Maksymalna dopuszczalna liczba wywołań procedury* **send** *wynosi* $K = 10 \cdot N$.

Podzadanie 5. (co najwyżej 19 punktów)

- $16 \leqslant N \leqslant 64$
- *Każda zakodowana liczba musi mieścić się w przedziale od 0 do* $R = 255$.
- *Maksymalna dopuszczalna liczba wywołań procedury* **send** *wynosi* $K = 15 \cdot N$.
- **Ważne:** *wynik za to podzadanie zależy od ilorazu między długością zakodowanej wiadomości a długością oryginalnej wiadomości.*

  Dla danego testu t *w tym podzadaniu niech* $P_t = L_t/N_t$ *oznacza iloraz między długością zakodowanej wiadomości* L_t *a długością oryginalnej wiadomości* N_t. *Niech* P *będzie największą spośród wszystkich wartości* P_t. *Twój wynik w tym podzadaniu zostanie określony na podstawie następujących zasad:*

  - *Jeśli* $P \leqslant 5$, *otrzymasz 19 punktów.*
  - *Jeśli* $5 < P \leqslant 6$, *otrzymasz 18 punktów.*
  - *Jeśli* $6 < P \leqslant 7$, *otrzymasz 17 punktów.*
  - *Jeśli* $7 < P \leqslant 15$, *otrzymasz* $1 + 2 \cdot (15 - P)$ *punktów (zaokrąglone w dół).*
  - *Jeśli* $P > 15$ *lub* **dowolna** *z odpowiedzi jest niepoprawna, otrzymasz 0 punktów.*

Ważne: *Każde poprawne rozwiązanie podzadań od 1 do 4 jest w stanie rozwiązać poprzednie podzadania. Jednakże, z powodu większego ograniczenia na* K, *poprawne rozwiązanie podzadania 5 może nie być dobre dla podzadań od 1 do 4. Istnieje rozwiązanie, które rozwiązuje wszystkie podzadania tą samą metodą.*

Szczegóły techniczne

Ograniczenia

- *Podczas oceny, Twoje zgłoszenia zostaną skompilowane do dwóch programów* **e** *oraz* **d**, *które zostaną uruchomione osobno. Obydwie napisane przez Ciebie procedury zostaną włączone w każdy program (tj. zlinkowane), lecz tylko* **e** *będzie wywoływać procedurę* **encode**, *a tylko* **d** *będzie wykonywać* **decode**.
- *Limit czasu procesora: Program* **e** *wywoła procedurę* **encode** *50 razy i musi działać co najwyżej 2 sekundy. Podobnie, program* **d** *wywoła procedurę* **decode** *50 razy i powinien działać co najwyżej 2 sekundy.*
- *Dostępna pamięć: 256 MB*
  Uwaga: *Nie ma osobnego ograniczenia na rozmiar stosu. Pamięć użyta przez stos wlicza się w całkowity rozmiar używanej pamięci.*

Interfejs (API)

- *Katalog na pliki źródłowe:* `parrots/`
- *Nazwa plików z rozwiązaniami:*
  - `encoder.c` *lub* `encoder.cpp`*, lub* `encoder.pas`
  - `decoder.c` *lub* `decoder.cpp`*, lub* `decoder.pas`

  Uwaga dla fanów C/C++: *zarówno w przykładowym, jak i ostatecznym module oceniającym, pliki* `encoder.c[pp]` *i* `decoder.c[pp]` *zostaną ze sobą zlinkowane. Aby zapobiec konfliktom zmiennych globalnych w różnych plikach, możesz poprzedzić ich deklaracje modyfikatorem* `static`*.*
- *Interfejs procedur zawodnika:*
  - `encoder.h` *lub* `encoder.pas`
  - `decoder.h` *lub* `decoder.pas`
- *Interfejs procedur modułu oceniającego:*
  - `encoderlib.h` *lub* `encoderlib.pas`
  - `decoderlib.h` *lub* `decoderlib.pas`
- *Przykładowy moduł oceniający:* `grader.c` *lub* `grader.cpp`*, lub* `grader.pas`

  Przykładowy moduł oceniający wykonuje dwie osobne rundy. W każdej rundzie wywołuje najpierw procedurę **encode** *z podanymi danymi, a następnie procedurę* **decode***, podając jej wynik wyznaczony przez procedurę* **encode***. W pierwszej rundzie moduł oceniający nie zmienia kolejności liczb w zakodowanej wiadomości. W drugiej rundzie moduł zamienia liczby na pozycjach parzystych i nieparzystych. Prawdziwy moduł oceniający będzie tasować zakodowane wiadomości na różne sposoby. Możesz zmienić sposób tasowania w przykładowym module oceniającym przez modyfikację procedury* **shuffle** *(w C/C++) lub* **Shuffle** *(w Pascalu).*

  Przykładowy moduł oceniający sprawdza również zakres oraz długość zakodowanych danych. Domyślnie sprawdza on, czy zakodowane dane mieszczą się w przedziale od 0 do 65535 i czy długość nie przekracza $10 \cdot N$. Możesz to zmienić przez zmianę wartości stałych `channel_range` *(na przykład z 65535 na 255) oraz* `max_expansion` *(na przykład z 10 na 15 lub 7).*
- *Pliki wejściowe dla przykładowego modułu oceniającego:* `grader.in.1`*,* `grader.in.2`*, ...*

  Uwaga: *Przykładowy moduł oceniający wczytuje dane zgodnie z poniższym formatem:*
  - *Wiersz 1: N.*
  - *Wiersz 2: lista N liczb całkowitych: $M[0], M[1], \ldots, M[N-1]$.*
- *Pliki z oczekiwaną odpowiedzią dla przykładowych wejść przykładowego modułu oceniającego:* `grader.expect.1`*,* `grader.expect.2`*, ...*

  W tym zadaniu powyższe pliki powinny zawierać jedynie napis „Correct".

Paweł Idziak, Jakub Łącki
Tłumaczenie

IOI 2011, dzień drugi, 26.07.2011

Słonie

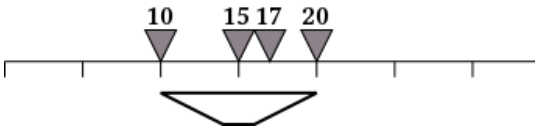
Pattaya słynie ze spektakularnego pokazu tańczących słoni. Bierze w nim udział N słoni tańczących w szeregu. Po wielu latach treningu słonie potrafią wykonywać zapierające dech w piersiach ewolucje. Cały pokaz składa się z serii aktów. W każdym z nich, dokładnie jeden słoń wykonuje cudowny taniec, podczas którego być może zmienia swoje położenie w szeregu.

Producenci pokazu chcą go upamiętnić albumem, w którym znajdą się zdjęcia przedstawiające to, co obserwują widzowie po każdym akcie.

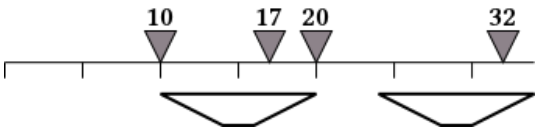
W każdym momencie pokazu wiele słoni może się znaleźć w tym samym miejscu w szeregu. W takim przypadku słonie stają jeden za drugim.

Pojedynczy aparat fotograficzny może objąć grupę słoni wtedy i tylko wtedy, gdy fragment szeregu zajmowany przez tę grupę słoni ma długość nie większą niż L. Dla uwiecznienia wszystkich słoni może być potrzebnych wiele aparatów fotograficznych, jako że słonie mogą rozproszyć się na scenie i nie mieścić w zakresie jednego obiektywu.

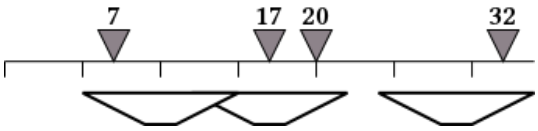
Dla przykładu, załóżmy, że L = 10 i że słonie stoją na pozycjach 10, 15, 17 i 20. W takim momencie jeden aparat ujmie je wszystkie, tak jak pokazano to na poniższym rysunku (słonie zaznaczone są trójkątami, a aparat fotograficzny — trapezem).



W kolejnym akcie słoń z pozycji 15 przedostał się na pozycję 32. Po tym akcie jeden aparat już nie wystarczy.



Gdy w następnym akcie słoń z pozycji 10 przeniesie się na pozycję 7, będziemy potrzebowali aż trzech aparatów.



W tym zadaniu powinieneś znaleźć **minimalną** *liczbę aparatów fotograficznych potrzebnych do zrobienia zdjęć po każdym akcie pokazu tańca słoni. Zadanie to jest zadaniem interaktywnym. Zauważ, że liczba potrzebnych aparatów z aktu na akt może rosnąć, maleć lub pozostawać bez zmian.*

Zadanie

Zaimplementuj następujące procedury:

- *Procedurę* **init***(N, L, X), której parametry to:*
  - *N — liczba słoni. Słonie są ponumerowane liczbami od 0 do N − 1.*
  - *L — długość fragmentu sceny obejmowanego pojedynczym aparatem fotograficznym. Możesz założyć, że L jest liczbą całkowitą i* $0 \leqslant L \leqslant 1\,000\,000\,000$.
  - *X — jednowymiarowa tablica liczb całkowitych opisująca wyjściowe pozycje słoni. Dla* $0 \leqslant i < N$, *słoń i zaczyna pokaz na pozycji* $X[i]$. *Początkowe pozycje są posortowane niemalejąco, tj.* $0 \leqslant X[0] \leqslant \ldots \leqslant X[N-1] \leqslant 1\,000\,000\,000$. *Zauważ, że w trakcie spektaklu słonie mogą zmienić swoją kolejność w szeregu.*

  Procedura ta zostanie wywołana tylko raz — przed wszystkimi wywołaniami procedury **update**. *Procedura* **init** *nie zwraca żadnej wartości.*

- *Procedurę* **update***(i, y), której parametry to:*
  - *i — numer słonia tańczącego w bieżącym akcie.*
  - *y — pozycja słonia i po zakończeniu bieżącego aktu. Możesz założyć, że y jest liczbą całkowitą i* $0 \leqslant y \leqslant 1\,000\,000\,000$.

  Ta procedura będzie wywoływana wielokrotnie. Każde wywołanie dotyczy kolejnego, jednego aktu. W każdym wywołaniu procedura musi zwrócić **minimalną liczbę aparatów fotograficznych potrzebnych** *do objęcia wszystkich słoni po akcie odpowiadającym temu wywołaniu.*

Przykład

Rozważmy przypadek, gdy $N = 4$, $L = 10$ *oraz wyjściowe pozycje słoni to:*

$$X = \begin{matrix} 10 \\ 15 \\ 17 \\ 20 \end{matrix}$$

Na początku zostanie wywołana procedura **init** *z wyżej podanymi parametrami. Następnie, dla każdego aktu zostanie wywołana Twoja procedura* **update**. *Poniżej znajdziesz przykładowy ciąg takich wywołań i poprawne odpowiedzi dla nich.*

akt	**parametry wywołania**	**zwrócona wartość**
1	**update***(2,16)*	*1*
2	**update***(1,25)*	*2*
3	**update***(3,35)*	*2*
4	**update***(0,38)*	*2*
5	**update***(2,0)*	*3*

Podzadania

Podzadanie 1. (10 punktów)

- $N = 2$, *tj. w spektaklu biorą udział dokładnie dwa słonie.*
- *Na początku i po każdym akcie na każdej pozycji znajduje się co najwyżej jeden słoń.*
- *Twoja procedura* **update** *zostanie wywołana co najwyżej 100 razy.*

Podzadanie 2. (16 punktów)

- $1 \leqslant N \leqslant 100$
- *Na początku i po każdym akcie na każdej pozycji znajduje się co najwyżej jeden słoń.*
- *Twoja procedura* **update** *zostanie wywołana co najwyżej 100 razy.*

Podzadanie 3. (24 punkty)

- $1 \leqslant N \leqslant 50\,000$
- *Na początku i po każdym akcie na każdej pozycji znajduje się co najwyżej jeden słoń.*
- *Twoja procedura* **update** *zostanie wywołana co najwyżej 50 000 razy.*

Podzadanie 4. (47 punktów)

- $1 \leqslant N \leqslant 70\,000$
- *Wiele słoni może znajdować się na tej samej pozycji.*
- *Twoja procedura* **update** *zostanie wywołana co najwyżej 70 000 razy.*

Podzadanie 5. (3 punkty)

- $1 \leqslant N \leqslant 150\,000$
- *Wiele słoni może znajdować się na tej samej pozycji.*
- *Twoja procedura* **update** *zostanie wywołana co najwyżej 150 000 razy.*
- *Przeczytaj uwagi dotyczące limitu czasu pracy procesora w sekcji* **Szczegóły techniczne**.

Szczegóły techniczne

Ograniczenia

- *Limit czasu procesora: 9 sekund*
  Uwaga: *Struktury danych zaimplementowane w STL dla C++ mogą być wolne. W szczególności mogą być zbyt wolne do rozwiązania podzadania 5.*

- *Dostępna pamięć: 256 MB*
  Uwaga: *Nie ma osobnego ograniczenia na rozmiar stosu. Pamięć użyta przez stos wlicza się w całkowity rozmiar używanej pamięci.*

Interfejs (API)

- *Katalog na pliki źródłowe:* `elephants/`
- *Nazwa pliku z rozwiązaniem:* `elephants.c` *lub* `elephants.cpp`, *lub* `elephants.pas`
- *Interfejs procedur zawodnika:* `elephants.h` *lub* `elephants.pas`
- *Przykładowy moduł oceniający:* `grader.c` *lub* `grader.cpp`, *lub* `grader.pas`
- *Pliki wejściowe dla przykładowego modułu oceniającego:* `grader.in.1`, `grader.in.2`, ...
  Uwaga: *Przykładowy moduł oceniający wczytuje dane zgodnie z poniższym formatem:*
  - *Wiersz 1: N, L oraz M, gdzie M to liczba aktów w pokazie tańców słoni.*
  - *Wiersze od 2 do $N+1$: pozycje wyjściowe słoni. Wiersz $k+2$ zawiera $X[k]$ dla $0 \leqslant k < N$.*
  - *Wiersze od $N+2$ do $N+M+1$: informacje o M kolejnych aktach. Dla $1 \leqslant j \leqslant M$ wiersz $N+1+j$ zawiera $i[j]$, $y[j]$ oraz $s[j]$, pooddzielane pojedynczymi odstępami. Wiersz ten mówi, że w j-tym akcie słoń $i[j]$ zmienił pozycję na $y[j]$ oraz że $s[j]$ jest minimalną liczbą aparatów fotograficznych potrzebnych do zrobienia zdjęć po tym akcie.*
- *Pliki z oczekiwaną odpowiedzią dla przykładowych wejść przykładowego modułu oceniającego:* `grader.expect.1`, `grader.expect.2`, ...
  W tym zadaniu powyższe pliki powinny zawierać jedynie napis „Correct”.

XVII Bałtycka Olimpiada Informatyczna,

Lyngby, Dania 2011

Jakub Łącki, Mirosław Michalski
Tłumaczenie

BOI 2011, dzień pierwszy, 30.04.2011

Jabłonie

Bajtazar pracuje w sadzie, w którym rośnie N jabłoni. Jego obowiązki obejmują dwa rodzaje zadań — nawożenie drzew oraz obliczanie statystyk dotyczących ich wysokości.

Do nawożenia drzew używa butelek z **MegaWypasNawozem**. *Drzewo, które zostanie nim potraktowane, rośnie natychmiast o jeden centymetr. Dla każdej butelki* **MegaWypasNawozu** *znamy jej objętość c_i, która określa, ile maksymalnie drzew można nawieźć jej zawartością. Ponadto, z każdą butelką związana jest liczba h_i, oznaczająca minimalną wysokość drzew, na których można zastosować nawóz z tej butelki. Ponieważ Bajtazar chciałby, żeby wszystkie jego drzewa były jak najwyższe, zawsze stosuje nawóz na c_i najniższych drzewach spośród drzew o wysokości nie mniejszej niż h_i centymetrów.*

Obliczanie statystyk dotyczących jabłoni polega na wyznaczaniu liczby drzew, których wysokość zawiera się w pewnym przedziale. Bajtazar poświęca mnóstwo czasu na pracę w ogrodzie, dlatego poprosił Cię o napisanie programu, który na podstawie listy zadań, jakie Bajtazar ma do wykonania, obliczy żądane statystyki za niego.

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby całkowite N i M określające, odpowiednio, liczbę jabłoni w ogrodzie Bajtazara oraz liczbę zdarzeń. W drugim wierszu znajduje się ciąg N liczb całkowitych z przedziału $[1, N]$ opisujący początkowe wysokości poszczególnych drzew, podane w centymetrach. Kolejne M wierszy opisuje zdarzenia, w kolejności chronologicznej. Każdy z tych wierszy rozpoczyna się znakiem t_i ($t_i =$ `F` *lub $t_i =$* `C`*) określającym typ zdarzenia.*

Jeśli $t_i =$ `F`*, to w dalszej części wiersza znajdują się jeszcze dwie liczby całkowite c_i i h_i. Taki wiersz oznacza, że Bajtazar stosuje butelkę* **MegaWypasNawozu** *na c_i najniższych drzewach spośród drzew o wysokości nie mniejszej niż h_i. Jeśli jest mniej niż c_i drzew o wymaganej wysokości, Bajtazar nawozi wszystkie z nich i wyrzuca butelkę z niewykorzystanym nawozem.*

Jeśli $t_i =$ `C`*, to w dalszej części wiersza znajdują się jeszcze dwie liczby całkowite min_i i max_i. Oznaczają one, że Bajtazar ma wyznaczyć liczbę drzew, których wysokość H jest pomiędzy min_i a max_i, tj. spełnia $min_i \leqslant H \leqslant max_i$.*

Wyjście

Dla każdego zdarzenia typu `C` *wypisz jeden wiersz zawierający jedną liczbę całkowitą — liczbę jabłoni o wymaganej wysokości. Kolejność wyników powinna odpowiadać kolejności zdarzeń typu* `C` *z wejścia.*

Ograniczenia

$1 \leqslant N, M \leqslant 100\,000$
$1 \leqslant c_i \leqslant N$, $0 \leqslant h_i \leqslant 1\,000\,000\,000$

$1 \leqslant min_i \leqslant max_i \leqslant 1\,000\,000\,000$
W testach wartych łącznie 40 punktów zachodzi $1 \leqslant N \leqslant 7\,000$ *oraz liczba zdarzeń typu* `F` *nie przekracza 7 000.*

Przykład

Dla danych wejściowych:

```
5 7
1 3 2 5 2
F 2 1
C 3 6
F 2 3
C 6 8
F 2 1
F 2 2
C 3 5
```

poprawnym wynikiem jest:

```
3
0
5
```

Jakub Łącki, Mirosław Michalski
Tłumaczenie

BOI 2011, dzień pierwszy, 30.04.2011

Lody

Bajtek wraz z przyjaciółmi udał się na wakacje do Włoch. Słońce dało im się we znaki, dlatego postanowili kupić lody na ochłodę. Do wyboru mają N smaków, ponumerowanych liczbami od 1 do N. Niektórych par smaków nie można łączyć, gdyż razem smakują okropnie. Bajtek chciałby dowiedzieć się, na ile sposobów może wybrać **trzy** *różne smaki tak, aby nie było wśród nich żadnej niedozwolonej pary. Kolejność smaków w wybranej trójce jest nieistotna.*

Wejście

W pierwszym wierszu wejścia znajdują się dwie nieujemne liczby całkowite N, M, oznaczające, odpowiednio, liczbę smaków oraz liczbę niedozwolonych par. Każdy z kolejnych M wierszy zawiera parę różnych liczb całkowitych, określającą dwa smaki, których nie należy wybierać jednocześnie. Każda niedozwolona para występuje w wejściu co najwyżej raz.

Wyjście

W pierwszym i jedynym wierszu powinieneś wypisać jedną liczbę całkowitą równą liczbie poprawnych wyborów trzech smaków lodów.

Ograniczenia

$1 \leqslant N \leqslant 200$
$0 \leqslant M \leqslant 10\,000$

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*

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

Wyjaśnienie do przykładu: *Dostępnych jest 5 różnych smaków, są 3 niedozwolone parowania. Smak pierwszy nie może zostać połączony ze smakami 2 oraz 3, zaś smak 3 nie może zostać połączony ze smakiem 4. Istnieją tylko trzy kombinacje spełniające powyższe warunki: (1, 4, 5), (2, 3, 5) i (2, 4, 5).*

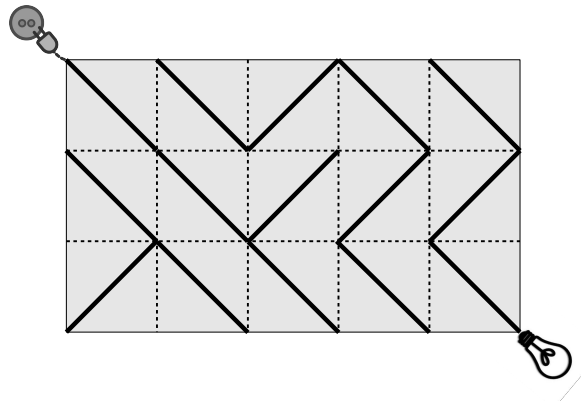
Jakub Łącki, Mirosław Michalski
Tłumaczenie

BOI 2011, dzień pierwszy, 30.04.2011

Włącz lampę

Bajtoni przygotowuje obwód elektryczny w kształcie prostokąta o wymiarach $N \times M$, podzielonego na kwadraciki jednostkowe. Dwa przeciwległe rogi każdego kwadracika są połączone przewodem.

Do lewego górnego rogu układu podłączone jest zasilanie, zaś do prawego dolnego jest podłączona lampa. Lampa jest włączona wtedy i tylko wtedy, gdy istnieje ścieżka złożona z przewodów, która prowadzi ze źródła zasilania do lampy. Aby włączyć lampę, można obrócić pewną liczbę kwadracików o 90° w dowolnym kierunku.



Na powyższym rysunku lampa jest wyłączona. Jeśli dowolny kwadracik w drugiej kolumnie od prawej obrócimy o 90°, lampa zostanie połączona ze źródłem zasilania i zapali się.

Napisz program, który wyznaczy najmniejszą możliwą liczbę obrotów kwadracików, jaką trzeba wykonać, by lampa zapaliła się.

Wejście

W pierwszym wierszu wejścia znajdują się dwie liczby całkowite N i M oznaczające wymiary obwodu. W każdym z kolejnych N wierszy znajduje się po M znaków `\` *lub* `/`*, które określają kierunek przewodu w odpowiadającym kwadraciku.*

Wyjście

Należy wypisać dokładnie jeden wiersz. Jeśli włączenie lampy jest możliwe, wiersz ten powinien zawierać dokładnie jedną liczbę całkowitą — minimalną liczbę obrotów potrzebnych do włączenia lampy. W przeciwnym przypadku należy wypisać `NO SOLUTION` .

Ograniczenia

$1 \leqslant N, M \leqslant 500$
W testach wartych 40 punktów zachodzą dodatkowe warunki $1 \leqslant N \leqslant 4$ *i* $1 \leqslant M \leqslant 5$.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*

`3 5`
`\\/\\`
`\\///`
`/\\\\`
	`1`

Przykładowe wejście opisuje obwód z rysunku.

Jakub Łącki, Mirosław Michalski
Tłumaczenie

BOI 2011, dzień pierwszy, 30.04.2011

Wikingowie i skarb

Znalazłeś mapę skarbów! Mapa jest siatką kwadratową o wymiarach $N \times M$, składającą się z kwadracików jednostkowych. Każdy kwadracik symbolizuje morze lub część lądu. Dodatkowo, na mapie zaznaczone jest miejsce, gdzie znajduje się skarb, oraz położenie statku wrogo nastawionych wikingów. Na mapie oznaczyłeś także swoją pozycję.

Teraz musisz ustalić ścieżkę, która zaprowadzi Cię do skarbu. Ścieżka musi zacząć się na Twojej obecnej pozycji, zakończyć w miejscu, w którym znajduje się skarb, i składać się z serii ruchów. W jednym ruchu możesz przemieścić się na sąsiednie pole (w pionie lub poziomie), niebędące częścią wyspy. Musisz być ostrożny: statek wikingów może Cię gonić, również przesuwając się o jedno pole w pionie lub poziomie. Po każdym Twoim ruchu wikingowie mogą ruszyć się lub nie. Twój ruch oraz następującą po nim reakcję wikingów nazywamy **rundą**.

Po każdej rundzie wykonywane są następujące sprawdzenia:

- *Jeśli jesteś w jednej linii ze statkiem wikingów (jesteś w tym samym wierszu lub tej samej kolumnie mapy oraz na polach pomiędzy wami jest wyłącznie morze), to giniesz.*
- *Jeśli jeszcze nie zginąłeś i jesteś na polu ze skarbem, zabierasz go.*

Napisz program, który sprawdzi, czy można **zawczasu ustalić** *ścieżkę, którą należy podążać, tak, aby nie zostać zabitym i zdobyć skarb, niezależnie od ruchów wykonywanych przez wikingów.*

Wejście

W pierwszym wierszu wejścia znajdują się dwie liczby całkowite N, M oznaczające wymiary mapy. W każdym z kolejnych N wierszy znajduje się ciąg M znaków opisujących zawartości poszczególnych pól mapy:

- `.` *oznacza morze,*
- `I` *oznacza część wyspy,*
- `V` *oznacza statek wikingów,*
- `Y` *oznacza Twoją pozycję,*
- `T` *oznacza miejsce, w którym znajduje się skarb.*

Każdy ze znaków `V`, `Y` *oraz* `T` *pojawi się w wejściu dokładnie raz.*

Wyjście

W pierwszym i jedynym wierszu należy wypisać słowo `YES`*, jeśli można wyznaczyć ścieżkę spełniającą założenia zadania, lub* `NO`*, w przeciwnym przypadku.*

Ograniczenia

$1 \leqslant N, M \leqslant 700$
W testach wartych 50 punktów zachodzi dodatkowy warunek $1 \leqslant N, M \leqslant 200$.

Przykład

Dla danych wejściowych:

```
5 7
Y.....V
..I....
..IIIII
.......
...T...
```

poprawnym wynikiem jest:

```
YES
```

a dla danych:

```
5 7
Y....V.
..I....
..IIIII
.......
...T...
```

poprawnym wynikiem jest:

```
NO
```

natomiast dla danych:

```
2 3
.YT
VII
```

poprawnym wynikiem jest:

```
NO
```

Wyjaśnienie do przykładów: *W pierwszym przykładzie ścieżka pozwalająca zdobyć skarb może składać się z następujących ruchów: dół, dół, dół, prawo, prawo, prawo, dół. W drugim i trzecim przykładzie nie istnieje ścieżka, która zaprowadzi Cię do skarbu i pozwoli Ci na pewno przeżyć.*

Jakub Łącki, Mirosław Michalski
Tłumaczenie

BOI 2011, dzień drugi, 1.05.2011

Spotkanie

Stowarzyszenie na rzecz Ochrony Świata zwołało N swoich członków na nadzwyczajny kongres, na którym ma zostać stworzony plan uratowania naszej planety. Aby ustalić wspólne stanowisko, uczestnicy kongresu postępują w następujący sposób:

- *Każdy uczestnik ma jedną propozycję i przedstawia ją innym uczestnikom. Każde wystąpienie zajmuje dokładnie P minut.*
- *Po wystąpieniach wszystkich członków następuje głosowanie, w którym wybierana jest najlepsza propozycja. Głosowanie trwa V minut.*

Na przykład, jeśli zarówno przedstawienie propozycji, jak i głosowanie trwa jedną minutę ($P = V = 1$), podjęcie decyzji w grupie 100 członków zajmie dokładnie 101 minut.

Uczestnicy kongresu chcą przyspieszyć proces podejmowania decyzji, dlatego zdecydowali się podzielić na grupy i pracować równolegle. Każda z grup wybiera najlepszą propozycję w sposób opisany powyżej, następnie reprezentanci każdej z grup spotykają się, żeby wybrać jedną propozycję spośród tych, które wygrały głosowania w grupach.

Na przykład, jeśli 100 członków podzielimy na dwie grupy z 40 i 60 członkami, to proces wyboru najlepszej propozycji może wyglądać następująco (ponownie zakładamy, że $P = V = 1$):

- *Większa grupa wybiera najlepszą propozycję w ciągu 61 minut.*
- *Mniejsza potrzebuje na to 41 minut i musi poczekać na zakończenie prac w większej grupie.*
- *Następnie dwoje reprezentantów grup przedstawia plany, co zajmuje 2 minuty, oraz wspólnie głosuje, co zajmuje dodatkową minutę.*

Całkowity czas poświęcony na podjęcie decyzji to $61 + 2 + 1 = 64$ minuty.

Mniejsze grupy mogą dalej dzielić się na jeszcze mniejsze w analogiczny sposób. Dozwolone są również podziały na więcej niż dwie podgrupy. Grupa złożona z jednego członka podejmuje decyzję natychmiast, bo nie musi on przedstawiać propozycji samemu sobie.

Napisz program, który obliczy, jaki jest minimalny czas potrzebny N uczestnikom kongresu na podjęcie wspólnej decyzji, przy założeniu, że podział na grupy zostanie wykonany optymalnie.

Wejście

W pierwszym wierszu wejścia znajdują się trzy liczby całkowite N, P i V oznaczające kolejno liczbę uczestników kongresu, czas potrzebny na przedstawienie jednej propozycji oraz czas, jaki zajmuje głosowanie. Czasy podane są w minutach.

Wyjście

W pierwszym i jedynym wierszu powinna znaleźć się jedna liczba całkowita określająca minimalny czas potrzebny na podjęcie wspólnej decyzji.

Ograniczenia

$1 \leqslant N \leqslant 10^{15}$
$1 \leqslant P, V \leqslant 1\,000$
W testach wartych łącznie 70 punktów zachodzi dodatkowy warunek $1 \leqslant N \leqslant 50\,000$.
W testach wartych łącznie 40 punktów zachodzi dodatkowy warunek $1 \leqslant N \leqslant 5\,000$.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`9 1 1`	`8`
a dla danych:	*poprawnym wynikiem jest:*
`6 1 2`	`8`
natomiast dla danych:	*poprawnym wynikiem jest:*
`6 2 1`	`12`

Wyjaśnienie do przykładu: *W pierwszym przykładzie uczestnicy mogą podzielić się na trzy trzyosobowe grupy. Wówczas każda z grup pracuje przez 4 minuty, a następnie ich przedstawiciele potrzebują kolejnych 4 minut na ustalenie ostatecznego stanowiska.*

Jakub Łącki, Mirosław Michalski
Tłumaczenie

BOI 2011, dzień drugi, 1.05.2011

Szubrawcy

Uczestnicy Bajtockiego Konkursu Programistycznego zgłosili N rozwiązań w postaci kodów źródłowych. Przed ogłoszeniem wyników jury postanowiło sprawdzić samodzielność rozwiązań. Jury dysponuje programem, który dla dwóch rozwiązań jest w stanie stwierdzić, czy są one na tyle podobne, aby posądzić ich autorów o współpracę.

Ponieważ liczba zgłoszonych rozwiązań jest bardzo duża i sprawdzenie wszystkich par zajęłoby sporo czasu, postanowiono już na wstępie odrzucić takie pary rozwiązań, których rozmiary różnią się znacząco. Dokładniej, postanowiono pominąć sprawdzenie par, w których rozmiar mniejszego rozwiązania stanowi mniej niż 90% rozmiaru większego rozwiązania. Sprawdzanie nastąpi więc tylko dla takich par rozwiązań (f_i, f_j), które spełniają $i \neq j$, $rozmiar(f_i) \leqslant rozmiar(f_j)$ oraz $rozmiar(f_i) \geqslant 0.9 \cdot rozmiar(f_j)$.

Napisz program, który obliczy liczbę par, dla których zostanie wykonane porównanie.

Wejście

W pierwszym wierszu wejścia znajduje się jedna liczby całkowita N oznaczająca liczbę zgłoszonych rozwiązań. Kolejny wiersz zawiera ciąg N liczb całkowitych $rozmiar(f_1), \ldots, rozmiar(f_N)$. Wyrazy ciągu opisują rozmiary poszczególnych rozwiązań.

Wyjście

W pierwszym i jedynym wierszu wypisz liczbę par rozwiązań, które zostaną ze sobą porównane.

Ograniczenia

$1 \leqslant N \leqslant 100\,000$
$1 \leqslant rozmiar(f_i) \leqslant 100\,000\,000$
W testach wartych 50 punktów zachodzi dodatkowy warunek $1 \leqslant N \leqslant 2\,000$.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`2` `2 1`	`0`
natomiast dla danych:	*poprawnym wynikiem jest:*
`5` `1 1 1 1 1`	`10`

Wyjaśnienie do przykładu: *W sytuacji opisanej w drugim przykładzie wszystkie rozwiązania zostaną ze sobą porównane, przy czym każde dwa rozwiązania porównuje się dokładnie raz.*

Jakub Łącki, Mirosław Michalski
Tłumaczenie

BOI 2011, dzień drugi, 1.05.2011

Wielokąt

Na nieskończonej kartce w kratkę narysowany jest N-wierzchołkowy wielokąt prosty. W wielokącie prostym każde dwie sąsiednie krawędzie mają dokładnie jeden punkt wspólny, zaś wszystkie inne pary krawędzi nie przecinają ani nie dotykają się. Dodatkowo, wierzchołki wielokąta leżą w punktach kratowych, tj. ich obydwie współrzędne są całkowite.

Bajtazar zakreślił na kartce wszystkie fragmenty linii kratowych, które są zawarte **ściśle** *we wnętrzu wielokąta. Twoim zadaniem jest obliczenie łącznej długości fragmentów zaznaczonych przez Bajtazara.*

Wejście

W pierwszym wierszu wejścia znajduje się jedna liczba całkowita N oznaczająca liczbę wierzchołków wielokąta. W kolejnych N wierszach znajdują się pary liczb całkowitych x_i, y_i, które określają współrzędne poszczególnych wierzchołków wielokąta. Wierzchołki są podane zgodnie z kierunkiem ruchu wskazówek zegara lub w odwrotnej kolejności.

Wyjście

W pierwszym wierszu powinieneś wypisać jedną liczbę rzeczywistą równą całkowitej długości odcinków zaznaczonych przez Bajtazara, tj. należących do linii tworzących kratkę na kartce oraz zawartych we wnętrzu wielokąta. Wypisany wynik zostanie uznany za poprawny, jeśli będzie dostatecznie blisko oczekiwanej wartości.

Nieco bardziej formalny opis sprawdzania poprawności odpowiedzi: jeśli wypisana przez Ciebie liczba to L, a poprawny wynik wynosi R, to odpowiedź zostaje zaliczona, o ile zachodzi co najmniej jeden z poniższych warunków:

- $|L - R| \leqslant R \cdot 10^{-6}$ *(dokładność względna),*
- $|L - R| \leqslant 10^{-6}$ *(dokładność bezwzględna).*

Ograniczenia

$3 \leqslant N \leqslant 100\,000$
$-500\,000\,000 \leqslant x_i, y_i \leqslant 500\,000\,000$
W testach wartych łącznie 50 punktów każda krawędź wielokąta jest albo pionowa, albo pozioma.

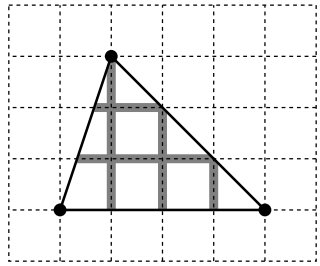
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
10.0
```



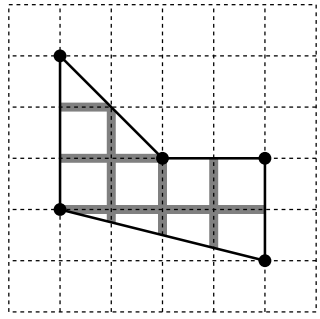
Rys. 1: Wielokąt z pierwszego testu przykładowego. Długość poziomych odcinków wynosi $\frac{4}{3} + \frac{8}{3} = 4$, zaś długość odcinków pionowych to $3+2+1 = 6$. Ich łączna długość jest równa $4 + 6 = 10$.

natomiast dla danych:

```
5
0 0
-2 2
-2 -1
2 -2
2 0
```

poprawnym wynikiem jest:

```
12.5
```



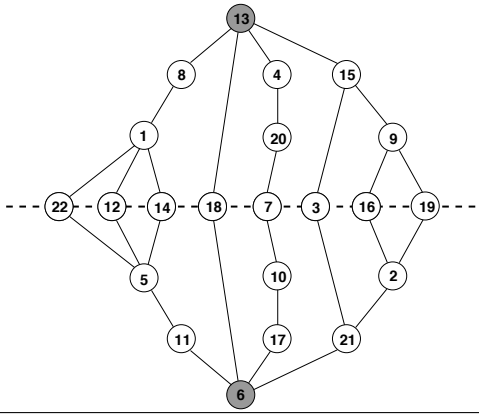
Rys. 2: Wielokąt z drugiego testu przykładowego. Długość odcinków poziomych wynosi $1+2+4 = 7$, zaś długość odcinków pionowych to $\frac{9}{4} + \frac{3}{2} + \frac{7}{4} = 5.5$. Łączna długość odcinków linii tworzących kratkę zawartych w wielokącie to $7 + 5.5 = 12.5$.

Jakub Łącki, Mirosław Michalski
Tłumaczenie

BOI 2011, dzień drugi, 1.05.2011

Drzewiaste odbicie

Niech T będzie ukorzenionym drzewem, a S jego kopią. Budujemy nowy graf przez wzięcie T i S oraz utożsamienie odpowiadających sobie liści[1]*. Powstały graf nazywamy* **drzewiastym odbiciem** *drzewa T.*



Rys. 1: Przykład drzewiastego odbicia. Graf ten jest opisany w trzecim teście przykładowym.

Napisz program, który stwierdzi, czy podany graf jest drzewiastym odbiciem jakiegoś drzewa.

Wejście

W pierwszym wierszu wejścia znajdują się dwie liczby całkowite N i M oznaczające odpowiednio liczbę wierzchołków oraz liczbę krawędzi w grafie. Wierzchołki są ponumerowane liczbami od 1 do N. Kolejne M wierszy zawiera opis krawędzi grafu, w i-tym spośród tych wierszy znajduje się para liczb całkowitych x_i, y_i ($x_i \neq y_i$, $1 \leqslant x_i, y_i \leqslant N$), która reprezentuje krawędź między wierzchołkami x_i oraz y_i. Pomiędzy dowolnymi dwoma wierzchołkami istnieje co najwyżej jedna krawędź.

Wyjście

W pierwszym wierszu wypisz jedno słowo `YES`*, jeśli graf jest drzewiastym odbiciem pewnego drzewa, lub* `NO` *w przeciwnym przypadku.*

[1] Korzeń drzewa nie jest liściem, nawet jeśli jego stopień jest równy 1.

Ograniczenia

$3 \leqslant N, M \leqslant 100\,000$
W testach wartych łącznie 60 punktów zachodzi dodatkowy warunek $3 \leqslant N, M \leqslant 3\,500$*, zaś w testach wartych łącznie 30 punktów zachodzi dodatkowy warunek* $3 \leqslant N, M \leqslant 300$*.*

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*

`7 7`
`1 2`
`2 3`
`3 4`
`4 5`
`5 6`
`6 7`
`7 1`
	`NO`

a dla danych:	*poprawnym wynikiem jest:*

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

natomiast dla danych:

```
22 28
13 8
8 1
1 22
1 12
1 14
13 18
13 4
4 20
20 7
13 15
15 3
15 9
9 16
9 19
22 5
12 5
14 5
5 11
11 6
18 6
7 10
10 17
17 6
3 21
21 6
16 2
19 2
2 21
```

poprawnym wynikiem jest:

```
YES
```

XVIII Olimpiada Informatyczna Krajów Europy Środkowej

Gdynia, 2011

Lech Duraj, Grzegorz Herman
Tłumaczenie

CEOI 2011, dzień pierwszy, 9.07.2011

Balony

Organizatorzy CEOI 2011 zamierzają urządzić przyjęcie z mnóstwem balonów. Będzie ich n, wszystkie w kształcie kuli, ułożone w jednym rzędzie na podłodze. Balony trzeba będzie napompować, na razie każdy z nich ma promień zero. Balon o numerze i jest przymocowany do podłogi w punkcie o współrzędnej x_i. Balony będą pompowane kolejno, od lewej do prawej. W czasie pompowania promień i-tego balonu zwiększa się w sposób ciągły, aż do momentu kiedy osiągnie określoną dla niego górną granicę r_i, lub balon dotknie któregoś z poprzednio napompowanych.

Rys. 1: Balony z testu przykładowego, po pełnym napompowaniu.

Organizatorzy chcieliby wiedzieć, ile powietrza będzie potrzebne do napełnienia balonów. Twoim zadaniem jest wyznaczenie promienia, który osiągnie każdy z nich.

Wejście

Pierwszy wiersz standardowego wejścia zawiera liczbę całkowitą n ($1 \leqslant n \leqslant 200\,000$) — liczbę balonów. Kolejne n wierszy opisuje balony: i-ty z nich zawiera liczby całkowite x_i oraz r_i ($0 \leqslant x_i \leqslant 10^9$, $1 \leqslant r_i \leqslant 10^9$). Możesz założyć, że balony są podane w kolejności rosnącej współrzędnej x_i.

W testach wartych przynajmniej 40 punktów zachodzi dodatkowa nierówność $n \leqslant 2\,000$.

Wyjście

Twój program powinien wypisać n wierszy, przy czym i-ty z nich powinien zawierać dokładnie jedną liczbę — promień i-tego balonu po napełnieniu. Odpowiedź będzie zaakceptowana, jeśli każda liczba będzie różniła się od prawidłowej o co najwyżej 0.001.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`3`	`9.000`
`0 9`	`1.000`
`8 1`	`4.694`
`13 7`	

Wskazówka: *Aby wypisać wartość* `a` *typu* `long double` *z dokładnością do trzech miejsc po przecinku w C/C++ możesz użyć instrukcji* `printf("%.3Lf\n", a);`. *Aby wypisać to samo z użyciem strumieni w C++, użyj* `cout << fixed << setprecision(3);` *przed wypisaniem wartości za pomocą* `cout << a << "\n";` *(nie zapomnij dołączyć pliku nagłówkowego* `iomanip`*). W Pascalu możesz użyć instrukcji* `writeln(a:0:3);`.

Zalecamy używanie typów `long double` *w C/C++ oraz* `extended` *w Pascalu, ze względu na ich większą dokładność (w szczególności, przy ich użyciu żaden z rozważanych przez organizatorów poprawnych algorytmów nie miał problemów z błędami zaokrągleń).*

Lech Duraj, Grzegorz Herman
Tłumaczenie

CEOI 2011, dzień pierwszy, 9.07.2011

Logo

W ramach nowej kampanii reklamowej, duża firma z Gdyni zamierza zaprezentować w mieście swoje logo. Firma przeznaczy na nie cały budżet kampanii, musi ono zatem być naprawdę imponujące. Jeden z menedżerów wpadł na pomysł użycia jako części logo całych budynków.

Logo składa się z n pionowych pasków różnej wysokości, numerowanych od lewej do prawej liczbami $1, 2, \ldots, n$. Logo można opisać permutacją $(s_1, s_2, \ldots, s_n)$ liczb $1, 2, \ldots, n$. Opis taki oznacza, że pasek o numerze s_1 jest najniższy wśród pasków logo, pasek o numerze s_2 drugi w kolejności... wreszcie pasek s_n jest najwyższy. Dokładne wysokości pasków z punktu widzenia opisu nie są istotne.

Wzdłuż głównej ulicy Gdyni stoi m budynków. Ku Twojemu zdziwieniu, ich wysokości są parami różne. Twoje zadanie polega na znalezieniu miejsc, w których logo pasuje do układu budynków.

Pomóż firmie znaleźć wszystkie spójne fragmenty ciągu budynków, które pasują do logo, tzn. s_1-szy budynek tego fragmentu jest najniższy, s_2-gi drugi najniższy, itd. Na przykład podciąg budynków o wysokościach 5, 10, 4 pasuje do logo opisanego permutacją $(3, 1, 2)$, jako że trzeci budynek (o wysokości 4) jest najniższy, pierwszy z lewej (5) — drugi co do wysokości, zaś drugi (10) najwyższy.

Wejście

Pierwszy wiersz standardowego wejścia zawiera dwie liczby całkowite n oraz m ($2 \leqslant n \leqslant m \leqslant 1\,000\,000$). Drugi wiersz zawiera n liczb $s_1, s_2, \ldots, s_n$ będących permutacją zbioru $\{1, 2, \ldots, n\}$. Oznacza to, że $1 \leqslant s_i \leqslant n$ oraz $s_i \neq s_j$ dla $i \neq j$. Trzeci wiersz zawiera m liczb całkowitych $h_1, h_2, \ldots, h_m$ ($1 \leqslant h_i \leqslant 10^9$ dla $1 \leqslant i \leqslant m$) — wysokości budynków. Liczby h_i są parami różne. W każdym wierszu liczby rozdzielone są pojedynczymi odstępami.

W testach wartych łącznie przynajmniej 35 punktów zachodzi $n \leqslant 5\,000$ oraz $m \leqslant 20\,000$.

W testach wartych przynajmniej 60 punktów, $n \leqslant 50\,000$ oraz $m \leqslant 200\,000$.

Wyjście

Pierwszy wiersz standardowego wyjścia powinien zawierać liczbę k pasujących fragmentów. Drugi wiersz powinien zawierać k liczb — indeksów (licząc od 1) budynków, od których te fragmenty się zaczynają. Indeksy należy wypisać w kolejności rosnącej, pooddzielane pojedynczymi odstępami. W wypadku gdy $k = 0$, drugi wiersz powinien pozostać pusty.

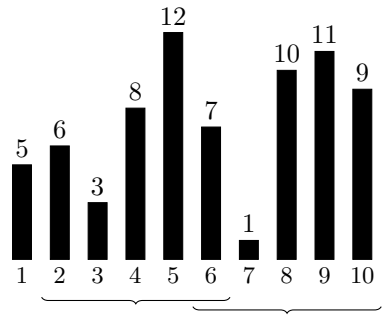
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
2
2 6
```



Wyjaśnienie do przykładu: *Ciągi* $(6, 3, 8, 12, 7)$ *oraz* $(7, 1, 10, 11, 9)$ *pasują do logo opisanego permutacją* $(2, 1, 5, 3, 4)$*. W szczególności w pierwszym z nich najniższy jest budynek numer 2 (o wysokości 3), drugim najniższym jest budynek numer 1 (o wysokości 6), trzecim — budynek numer 5 (o wysokości 7) i tak dalej.*

Lech Duraj, Grzegorz Herman
Tłumaczenie

CEOI 2011, dzień pierwszy, 9.07.2011

Skarb

Ahoj! Słyszałeś kiedykolwiek o piratach i ich skarbach? Podczas spaceru plażą w Gdyni Bajtek znalazł starą butelkę z listem w środku. List zawiera instrukcje pozwalające odnaleźć ukryty skarb, ciężko go jednak rozszyfrować. Jedno, co Bajtek wie na pewno, to to, że musi odnaleźć dwa szczególne punkty w pobliskim parku — skarb będzie znajdował się w połowie drogi między nimi.

W parku jest wiele szlaków, las zaś tak gęsty, że miejsca poza nimi pozostają całkowicie niedostępne dla człowieka. Struktura szlaków ma interesującą własność: każde dwa punkty są połączone dokładnie jedną drogą. Może ona biec wzdłuż wielu ścieżek, nigdy jednak nie przechodzi przez żaden punkt więcej niż raz.

Bajtek poprosił przyjaciół o pomoc w eksploracji parku. Grupa zamierza rozpocząć poszukiwania w pewnym punkcie, a następnie przeprowadzać je etapami. W każdym etapie jeden z przyjaciół wybierze pewien już odkryty punkt, a potem zagłębi się na pewną liczbę kroków w prowadzącą z niego, całkowicie dotąd nieznaną ścieżkę.

Równolegle, Bajtek będzie analizował strukturę parku. Od czasu do czasu może on próbować odgadnąć dwa specjalne punkty określające położenie skarbu. Dla każdej takiej pary punktów, chce poznać punkt leżący w połowie drogi między nimi. Twoim zadaniem jest mu w tym dopomóc.

Komunikacja

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

- `init` — *zostanie wywołana dokładnie raz, na początku sprawdzania. Wykorzystaj ją do inicjalizacji swoich struktur danych.*
  - *C/C++:* `void init();`
  - *Pascal:* `procedure init();`

  W momencie wywołania tej funkcji powinieneś założyć, że znany jest dokładnie jeden punkt w parku, oznaczony numerem 1.

- `path` — *wywoływana, gdy odkryta zostanie nowa ścieżka. Wykorzystaj ją do utrzymywania struktury danych opisującej park.*
  - *C/C++:* `void path(int a, int s);`
  - *Pascal:* `procedure path(a, s: longint);`

  Ścieżka rozpoczyna się w już znanym punkcie o numerze `a` *i ma długość* `s` *kroków. Po każdym kroku odkrywany jest punkt, który otrzymuje najmniejszy dotychczas niewykorzystany numer. Funkcja ta zostanie wywołana przynajmniej raz.*

- `dig` — *zapytanie o miejsce ukrycia skarbu.*
  - *C/C++:* `int dig(int a, int b);`
  - *Pascal:* `function dig(a, b: longint) : longint;`

  Funkcja ta powinna zwrócić numer punktu znajdującego się w połowie drogi między punktami o numerach `a` *i* `b`*. Możesz założyć, że zostały one już odkryte oraz że* `a`≠`b`*. Jeśli ścieżka ma nieparzystą długość (i jej środek nie leży w żadnym nazwanym punkcie), podaj punkt najbliższy środka leżący bliżej* `a` *(patrz też przykład na następnej stronie). Funkcja ta zostanie wywołana przynajmniej raz.*

Twoja biblioteka **nie może** *czytać żadnych danych (ani ze standardowego wejścia, ani z plików).* **Nie może** *również nic wypisywać do plików ani na standardowe wyjście. Może pisać na wyjście błędów (*`stderr`*) — pamiętaj jednak, że zużywa to cenny czas. Jeśli piszesz w C/C++, Twoja biblioteka* **nie może** *zawierać funkcji* `main`*. Jeśli piszesz w Pascalu, powinieneś dostarczyć moduł (patrz przykładowe programy na Twoim dysku).*

Kompilacja

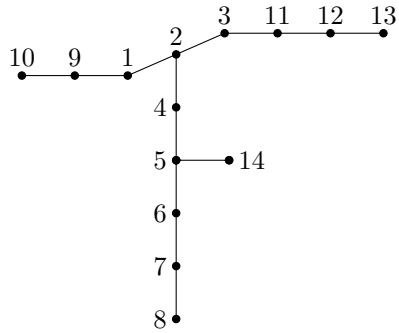
Twoja biblioteka — `tre.c`*,* `tre.cpp` *lub* `tre.pas` *— zostanie skompilowana z programem oceniającym przy użyciu następujących instrukcji:*

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

Przykładowe wykonanie

Poniższa tabela zawiera przykładowy ciąg wywołań funkcji oraz poprawne wyniki funkcji `dig`*. Odpowiadająca tym wywołaniom struktura ścieżek pokazana jest na rysunku.*

Wywołanie	Wynik	Dodane punkty
`init();`		1
`path(1, 2);`		2, 3
`dig(1, 3);`	2	
`path(2, 5);`		4, 5, 6, 7, 8
`dig(7, 3);`	5	
`dig(3, 7);`	4	
`path(1, 2);`		9, 10
`path(3, 3);`		11, 12, 13
`dig(10, 11);`	1	
`path(5, 1);`		14
`dig(14, 8);`	6	
`dig(2, 4);`	2	



Ograniczenia

Wywołań funkcji (`init`, `path` *i* `dig`*) będzie łącznie co najwyżej* 400 000. *Przyjaciele Bajtka odwiedzą co najwyżej* 1 000 000 000 *punktów.*

W testach wartych 50 *punktów, odwiedzonych punktów będzie co najwyżej* 400 000.

W testach wartych 20 *punktów będzie co najwyżej* 5 000 *odkrytych punktów, a także co najwyżej* 5 000 *wywołań funkcji* `init`, `path` *i* `dig` *łącznie.*

Eksperymenty

Aby umożliwić Ci eksperymentowanie z Twoim rozwiązaniem, dostarczyliśmy przykładowy program oceniający w plikach

`tregrader.c`, `tregrader.cpp` *oraz* `tregrader.pas`

znajdujących się w katalogu **/home/zawodnik/tre/** *na Twoim komputerze. W celu eksperymentowania, umieść swoje rozwiązanie w pliku*

`tre.c`, `tre.cpp` *lub* `tre.pas`

w odpowiednim katalogu (`c`, `cpp` *lub* `pas`*). Na początku konkursu w plikach tych będą przykładowe (niepoprawne) rozwiązania. Swoje rozwiązanie możesz skompilować poleceniem*

```
make tre
```

działającym zgodnie z opisem w sekcji Kompilacja. Kompilacja w C/C++ wymaga też pliku `treinc.h`*, który znajduje się już w odpowiednim katalogu.*

Powstały plik wykonywalny czyta listę nazw i argumentów funkcji ze standardowego wejścia, wywołuje implementacje z Twojej biblioteki i wypisuje na standardowe wyjście wyniki funkcji `dig`*. Lista wywołań powinna być następującej postaci:*

Pierwszy wiersz zawiera liczbę wywołań, q*. Każdy z następnych* q *wierszy zawiera pojedynczy znak* `i`, `p` *lub* `d`*, a po nim dwie nieujemne liczby całkowite. Znak określa, która z funkcji ma zostać wywołana:* `i` *dla* `init`, `p` *dla* `path` *oraz* `d` *dla* `dig`*. Liczby określają argumenty wywołania:* a *i* s *dla* `path`*,* a *i* b *dla* `dig`*. Gdy znakiem jest* `i`*, obie liczby są zerami. Program oceniający* **nie sprawdza***, czy wejście jest poprawnie sformatowane ani czy spełnia warunki z sekcji Komunikacja i Ograniczenia. Plik* `tre0.in` *opisuje podane wyżej przykładowe wykonanie:*

```
12
i 0 0
p 1 2
d 1 3
p 2 5
d 7 3
d 3 7
p 1 2
p 3 3
d 10 11
```

```
p 5 1
d 14 8
d 2 4
```

Aby wczytać dane z tego pliku, użyj polecenia

```
./tre < tre0.in
```

Wyniki wywołań funkcji `dig` *zostaną wypisane na standardowe wyjście. Poprawne odpowiedzi (zawarte również w pliku* `tre0.out`*) to:*

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

Możesz sprawdzić, czy Twoje rozwiązanie daje poprawną odpowiedź na teście przykładowym, wysyłając je do systemu oceniającego SIO.

Lech Duraj, Grzegorz Herman
Tłumaczenie

CEOI 2011, dzień drugi, 10.07.2011

Rezerwacje

Twój przyjaciel jest właścicielem hotelu w Gdyni. Właśnie rozpoczyna się sezon turystyczny i do hotelu napłynęła przytłaczająca liczba ofert od potencjalnych klientów. Przyjaciel poprosił Cię więc o pomoc w przygotowaniu systemu rezerwacji pokoi hotelowych.

Do wynajęcia jest n pokoi. Przygotowanie i-tego spośród nich będzie kosztowało Twojego przyjaciela c_i złotych (o ile zdecyduje się go wynająć). Wówczas pokój będzie można wynająć maksymalnie p_i osobom. Można założyć, że koszt przygotowania pokoju o większej pojemności jest nie mniejszy niż koszt przygotowania dowolnego pokoju o mniejszej pojemności.

System rezerwacji otrzyma pewną liczbę ofert. Każda z nich będzie zawierała minimalną wymaganą pojemność pokoju (d_j osób) oraz cenę, jaką wynajmujący chce zapłacić za pokój (v_j złotych). Każdemu oferentowi można wynająć tylko jeden pokój, nie można też wynająć kilku klientom tego samego pokoju. Twój przyjaciel chciałby mieć w wakacje trochę czasu dla siebie, nie zamierza zatem przyjmować więcej niż o ofert.

Skoro jesteś tak zdolnym programistą, Twój przyjaciel chciałby, abyś obliczył dla niego maksymalny możliwy zysk z wynajęcia pokoi. Zysk to całkowita suma, która wpłynie za wynajęte pokoje, pomniejszona o koszt ich przygotowania.

Wejście

Pierwszy wiersz standardowego wejścia zawiera trzy liczby całkowite n, m oraz o ($1 \leqslant n, m \leqslant 500\,000$, $1 \leqslant o \leqslant min(m, n)$), oznaczające odpowiednio liczbę pokoi w hotelu, liczbę złożonych ofert oraz maksymalną liczbę ofert, którą przyjaciel jest w stanie przyjąć. Następne n wierszy to opisy pokoi: i-ty wiersz zawiera dwie liczby całkowite c_i, p_i ($1 \leqslant c_i, p_i \leqslant 10^9$) — koszt przygotowania i pojemność i-tego pokoju. Kolejne m wierszy to opisy ofert, każdy zawierający dwie liczby całkowite v_j, d_j ($1 \leqslant v_j, d_j \leqslant 10^9$) — wysokość j-tej oferty oraz minimalny rozmiar wymaganego pokoju.

Możesz założyć, że w testach wartych 40 punktów zachodzi dodatkowa nierówność $n, m \leqslant 100$.

Wyjście

Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą — maksymalny możliwy zysk z wynajęcia co najwyżej o pokoi. Zauważ (i miej nadzieję!), że zysk może okazać się duży.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
3 2 2 150 2 400 3 100 2 200 1 700 3	400

Wyjaśnienie do przykładu: *Twój przyjaciel może przyjąć obydwie oferty, wynajmując pokoje numer 2 i 3.*

Lech Duraj, Grzegorz Herman
Tłumaczenie

CEOI 2011, dzień drugi, 10.07.2011

Drużyny

W jednej ze szkół podstawowych w Gdyni odbywa się Dzień Sportu. Najważniejszym punktem programu są Mistrzostwa w Piłce Nożnej.

Mnóstwo dzieci zgromadziło się na boisku, aby uformować drużyny. Jako że każde z nich chciało należeć do najlepszej drużyny, nie mogły dojść do porozumienia. Niektóre zagroziły, że w ogóle nie wezmą udziału w Mistrzostwach, inne zaczęły płakać... nikt już nie wiedział, czy turniej w ogóle się odbędzie.

Bajtazar, nauczyciel WF-u, musi zorganizować Mistrzostwa. Postanowił sam podzielić dzieci na drużyny tak, aby każde z nich było zadowolone. Aby tak się stało, i-te spośród dzieci musi znaleźć się w drużynie mającej przynajmniej a_i zawodników.

Poza spełnieniem wymagań wszystkich dzieci, Bajtazar chciałby zmaksymalizować liczbę drużyn biorących udział w Mistrzostwach. Jeśli nawet wtedy istniałoby wiele możliwości podziału, chciałby wybrać tę, w której rozmiar najliczniejszej drużyny jest najmniejszy. Zadanie okazało się całkiem niełatwe, Bajtazar poprosił Cię więc o pomoc.

Wejście

Pierwszy wiersz standardowego wejścia zawiera liczbę dzieci n ($1 \leqslant n \leqslant 1\,000\,000$). Każdy z kolejnych n wierszy zawiera jedną liczbę całkowitą a_i ($1 \leqslant a_i \leqslant n$) — minimalny rozmiar drużyny, który usatysfakcjonuje i-te dziecko.

W testach wartych przynajmniej 50 punktów n nie przekroczy 5 000.

Wyjście

W pierwszym wierszu standardowego wyjścia powinna znaleźć się jedna liczba całkowita t — maksymalna możliwa liczba drużyn. Każdy z kolejnych t wierszy powinien zawierać opis jednej z drużyn: najpierw liczbę całkowitą s_i ($1 \leqslant s_i \leqslant n$) — rozmiar i-tej drużyny, a następnie s_i liczb całkowitych $k_1, k_2, \ldots, k_{s_i}$ ($1 \leqslant k_j \leqslant n$) — numery dzieci należących do tej drużyny. Jeśli istnieje wiele poprawnych odpowiedzi, możesz wypisać dowolną, która minimalizuje rozmiar najliczniejszej spośród t drużyn.

Przykład

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

Lech Duraj, Grzegorz Herman
Tłumaczenie

CEOI 2011, dzień drugi, 10.07.2011

Wyspa

Centrum Gdyni znajduje się na wyspie pośrodku pięknej rzeki. Każdego ranka tysiące samochodów przejeżdżają przez wyspę z zachodu na wschód, z dzielnic mieszkalnych na jednym brzegu rzeki do obszarów przemysłowych po drugiej stronie. Mieszkańcy wjeżdżają na wyspę mostami na zachodnim krańcu i opuszczają ją mostami na wschodnim krańcu.

Wyspa ma kształt prostokąta o bokach równoległych do osi układu współrzędnych. Będziemy więc ją opisywać jako prostokąt $A \times B$ we współrzędnych kartezjańskich, którego przeciwległe narożniki mają współrzędne $(0, 0)$ oraz (A, B).

Na wyspie jest n skrzyżowań, ponumerowanych od 1 do n. Skrzyżowanie numer i ma współrzędne (x_i, y_i). Skrzyżowania o współrzędnych postaci $(0, y)$ są wjazdami na zachodni kraniec wyspy, zaś te o współrzędnych postaci (A, y) są wyjazdami po wschodniej stronie. Każda ulica jest odcinkiem łączącym dwa skrzyżowania. Niektóre spośród ulic są jedno-, inne zaś dwukierunkowe. Ulice, poza skrzyżowaniami, nie mają punktów wspólnych, nie ma też na wyspie tuneli ani wiaduktów. Nie zakładaj nic ponadto o sieci drogowej (w szczególności, drogi mogą biec brzegiem wyspy, mogą też istnieć skrzyżowania bez wchodzących czy wychodzących z nich ulic).

Z powodu rosnącego natężenia ruchu, prezydent Gdyni zatrudnił Cię w celu sprawdzenia, czy sieć dróg na wyspie jest wystarczająca. Na początek musisz sprawdzić, do ilu skrzyżowań na wschodnim brzegu wyspy można dojechać z każdego ze skrzyżowań na zachodnim brzegu.

Wejście

Pierwszy wiersz standardowego wejścia zawiera cztery liczby całkowite n, m, A oraz B ($1 \leqslant n \leqslant 300\,000$, $0 \leqslant m \leqslant 900\,000$, $1 \leqslant A, B \leqslant 10^9$). Są to odpowiednio liczba skrzyżowań w centrum Gdyni, liczba ulic oraz wymiary wyspy.

W każdym z kolejnych n wierszy znajdują się dwie liczby całkowite x_i, y_i ($0 \leqslant x_i \leqslant A$, $0 \leqslant y_i \leqslant B$) — współrzędne i-tego skrzyżowania. Żadne dwa skrzyżowania nie mają tych samych współrzędnych.

Kolejne m wierszy opisuje ulice. Każdy z tych wierszy zawiera trzy liczby c_i, d_i, k_i ($1 \leqslant c_i, d_i \leqslant n$, $c_i \neq d_i$, $k_i \in \{1, 2\}$). Opis ten oznacza, że skrzyżowania c_i i d_i są połączone ulicą. Jeśli $k_i = 1$, jest to jednokierunkowa ulica z c_i do d_i. W przeciwnym wypadku jest to ulica dwukierunkowa. Każda (nieuporządkowana) para $\{c_i, d_i\}$ pojawi się na wejściu co najwyżej raz.

Możesz założyć, że co najmniej jedno skrzyżowanie na wschodnim brzegu wyspy jest osiągalne z jakiegoś skrzyżowania na zachodnim brzegu.

W testach wartych przynajmniej 30 punktów zachodzi dodatkowy warunek $n, m \leqslant 6\,000$.

Wyjście

Twój program powinien wypisać na standardowe wyjście jeden wiersz dla każdego skrzyżowania z zachodniego krańca wyspy. Wiersz ten powinien zawierać liczbę osiągalnych z niego skrzyżowań po wschodniej stronie. Odpowiedzi uporządkuj w kolejności **malejącej** *współrzędnej y.*

Przykład

Dla danych wejściowych:

```
5 3 1 3
0 0
0 1
0 2
1 0
1 1
1 4 1
1 5 2
3 5 2
```

poprawnym wynikiem jest:

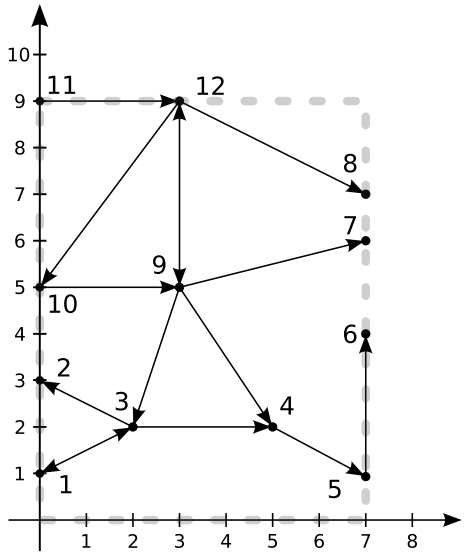
```
2
0
2
```

podczas gdy dla danych wejściowych:

```
12 13 7 9
0 1
0 3
2 2
5 2
7 1
7 4
7 6
7 7
3 5
0 5
0 9
3 9
1 3 2
3 2 1
3 4 1
4 5 1
5 6 1
9 3 1
9 4 1
9 7 1
9 12 2
10 9 1
11 12 1
12 8 1
12 10 1
```

poprawną odpowiedzią jest:

```
4
4
0
2
```



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] A. V. Aho, J. E. Hopcroft, J. D. Ullman. *Projektowanie i analiza algorytmów komputerowych*. PWN, Warszawa, 1983.

[19] L. Banachowski, A. Kreczmar. *Elementy analizy algorytmów*. WNT, Warszawa, 1982.

[20] L. Banachowski, K. Diks, W. Rytter. *Algorytmy i struktury danych*. WNT, Warszawa, 1996.

[21] J. Bentley. *Perełki oprogramowania.* WNT, Warszawa, 1992.

[22] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Wprowadzenie do algorytmów.* WNT, Warszawa, 2004.

[23] M. de Berg, M. van Kreveld, M. Overmars. *Geometria obliczeniowa. Algorytmy i zastosowania.* WNT, Warszawa, 2007.

[24] R. L. Graham, D. E. Knuth, O. Patashnik. *Matematyka konkretna.* PWN, Warszawa, 1996.

[25] D. Harel. *Algorytmika. Rzecz o istocie informatyki.* WNT, Warszawa, 1992.

[26] D. E. Knuth. *Sztuka programowania.* WNT, Warszawa, 2002.

[27] W. Lipski. *Kombinatoryka dla programistów.* WNT, Warszawa, 2004.

[28] E. M. Reingold, J. Nievergelt, N. Deo. *Algorytmy kombinatoryczne.* WNT, Warszawa, 1985.

[29] K. A. Ross, C. R. B. Wright. *Matematyka dyskretna.* PWN, Warszawa, 1996.

[30] R. Sedgewick. *Algorytmy w C++. Grafy.* RM, 2003.

[31] S. S. Skiena, M. A. Revilla. *Wyzwania programistyczne.* WSiP, Warszawa, 2004.

[32] P. Stańczyk. *Algorytmika praktyczna. Nie tylko dla mistrzów.* PWN, Warszawa, 2009.

[33] M. M. Sysło. *Algorytmy.* WSiP, Warszawa, 1998.

[34] M. M. Sysło. *Piramidy, szyszki i inne konstrukcje algorytmiczne.* WSiP, Warszawa, 1998.

[35] R. J. Wilson. *Wprowadzenie do teorii grafów.* PWN, Warszawa, 1998.

[36] N. Wirth. *Algorytmy + struktury danych = programy.* WNT, Warszawa, 1999.

[37] Rashid Bin Muhammad. Smallest enclosing circle problem. `http://www.personal.kent.edu/~rmuhamma/Compgeometry/MyCG/CG-Applets/Center/centercli.htm`.

Niniejsza publikacja stanowi wyczerpujące źródło informacji o zawodach XVIII Olimpiady Informatycznej, przeprowadzonych w roku szkolnym 2010/2011. 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 XXIII Międzynarodowej Olimpiady Informatycznej, XVII Bałtyckiej Olimpiady Informatycznej oraz XVIII Olimpiady Informatycznej Krajów Europy Środkowej.

XVIII Olimpiada Informatyczna to pozycja polecana szczególnie uczniom przygotowującym się do udziału w zawodach następnych Olimpiad oraz nauczycielom informatyki, którzy znajdą w niej interesujące i przystępnie sformułowane problemy algorytmiczne wraz z rozwiązaniami. Książka może być wykorzystywana także jako pomoc do zajęć z algorytmiki na studiach informatycznych.

Olimpiada Informatyczna
jest organizowana przy współudziale

ISBN 978–83–930856–7–5