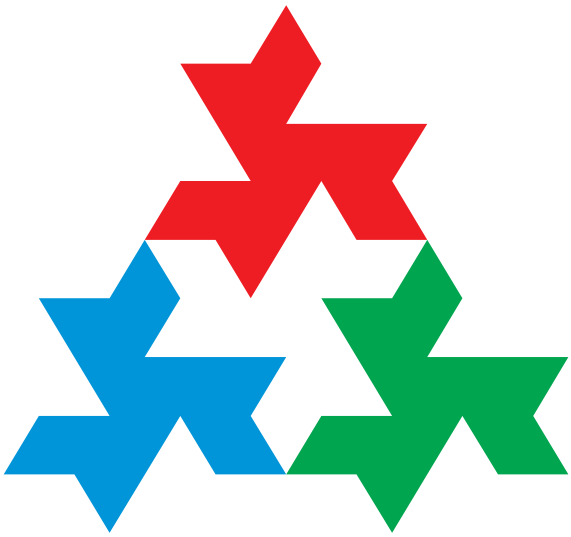
MINISTERSTWO EDUKACJI NARODOWEJ
FUNDACJA ROZWOJU INFORMATYKI
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ



XXIII OLIMPIADA INFORMATYCZNA
2015/2016

WARSZAWA, 2017

Autorzy tekstów:

Marek Cygan
Krzysztof Diks
Tomasz Idziaszek
Krzysztof Kiljan
Jakub Łącki
Wojciech Nadara
Paweł Parys
Karol Pokorski
Jakub Radoszewski
Wojciech Rytter
Marek Sokołowski
Juliusz Straszyński
Tomasz Syposz
Jacek Tomasiewicz
Michał Włodarczyk

Autorzy programów:

Paweł Czerwiński
Kamil Dębowski
Michał Glapa
Bartosz Kostka
Aleksander Łukasiewicz
Wojciech Nadara
Paweł Parys
Karol Pokorski
Jakub Radoszewski
Piotr Smulewicz
Marek Sokołowski
Marek Sommer

Opracowanie i redakcja:

Tomasz Idziaszek
Jakub Radoszewski

Skład:

Jakub Radoszewski

Tłumaczenie treści zadań:

Kamil Dębowski
Michał Glapa
Karol Kaszuba
Dominik Klemba
Bartosz Kostka
Jakub Radoszewski

Pozycja dotowana przez Ministerstwo Edukacji Narodowej.

Sponsorzy Olimpiady:

ISBN 978-83-64292-03-3

Spis treści

Krzysztof Diks

Wstęp

Drodzy Czytelnicy,

dwudziesta trzecia Olimpiada Informatyczna za nami. Była to chyba jedna z najtrudniejszych edycji Olimpiady. W pierwszym etapie wzięło udział tylko 481 zawodników i nikt nie zdobył maksymalnej liczby punktów, a tylko 81 uczniów uzyskało co najmniej połowę możliwych do zdobycia punktów. Należy się zastanowić, co zrobić, żeby takie wyniki nie pojawiały się w przyszłości. Duża w tym rola organizatorów Olimpiady. Powinni pamiętać o dobieraniu zadań o różnej skali trudności. Olimpiada powinna zachęcać do szlifowania umiejętności i pogłębiania wiedzy, a nie zniechęcać. My, jako organizatorzy, postaramy się stosować do tych wytycznych.

Z drugiej strony uczniowie powinni pamiętać, że żeby przejść do II etapu, nie zawsze trzeba rozwiązać wszystkie zadania, jak też nie każde zadanie musi być rozwiązane na poziomie rozwiązań wzorcowych. Liczy się każdy pomysł. Nawet nieefektywne rozwiązania mogą liczyć na pewną liczbę punktów, co w konsekwencji może zaprowadzić do kolejnego etapu. Inna rada to ćwiczyć, ćwiczyć i jeszcze raz ćwiczyć! Jest tak wiele różnych miejsc, gdzie można doskonalić swoje umiejętności i pogłębiać wiedzę. My ze swej strony polecamy portal z zadaniami `szkopul.edu.pl`, w którym w czasie rzeczywistym można trenować rozwiązywanie zadań olimpijskich, a w razie problemów z pokonywaniem trudności zachęcamy do zaglądania do „niebieskich książeczek".

Pamiętajmy, że sukces w Olimpiadzie Informatycznej nobilituje i pozwala znaleźć się w światowej elicie informatycznej, która zmienia świat. Żeby nie być gołosłownym, wspomnijmy chociażby grupę polskich olimpijczyków (Przemysław Dębiak, Rafał Józefowicz, Jakub Pachocki, Szymon Sidor), którzy pracowali w zespole twórców programu z OPEN AI, który pokonał człowieka w grze komputerowej DOTA 2 – zob. `https://openai.com/the-international/`.

Chcesz dołączyć do najlepszych – wystartuj w Olimpiadzie Informatycznej. Niebieska książeczka, którą trzymasz w ręku, może Ci w tym pomóc.

Krzysztof Diks
Przewodniczący Komitetu Głównego Olimpiady Informatycznej

Sprawozdanie z przebiegu XXIII Olimpiady Informatycznej w roku szkolnym 2015/2016

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 XXIII Olimpiady Informatycznej była 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.

Do szkół i zespołów szkół młodzieżowych ponadgimnazjalnych, przed rozpoczęciem zawodów, wysłano e-maile informujące o rozpoczęciu XXIII Olimpiady.

Zawody I stopnia rozpoczęły się 19 października 2015 roku. Ostatecznym terminem nadsyłania prac konkursowych był 16 listopada 2015 roku.

Zawody II i III stopnia były dwudniowymi sesjami stacjonarnymi, poprzedzonymi jednodniowymi sesjami próbnymi. Zawody II stopnia odbyły się w sześciu okręgach: Białymstoku, Gdańsku, Krakowie, Poznaniu, Warszawie i Wrocławiu w dniach 9-11 lutego 2016 roku, natomiast zawody III stopnia odbyły się w Warszawie na Wydziale Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego, w dniach 12-16 kwietnia 2016 roku.

Uroczystość zakończenia XXIII Olimpiady Informatycznej odbyła się 16 kwietnia 2016 roku w auli Centrum Nowych Technologii Uniwersytetu Warszawskiego w Warszawie.

SKŁAD OSOBOWY KOMITETÓW OLIMPIADY INFORMATYCZNEJ

Komitet Główny:

przewodniczący:

prof. dr hab. Krzysztof Diks (Uniwersytet Warszawski)

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

sekretarz naukowy:
dr Tomasz Idziaszek (Codility)

kierownik Jury:
dr Jakub Radoszewski (Uniwersytet Warszawski)

kierownik techniczny:
mgr Szymon Acedański (Uniwersytet Warszawski)

kierownik organizacyjny:
Tadeusz Kuran

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

sekretarz Komitetu Głównego:
mgr Monika Kozłowska-Zając (Ośrodek Edukacji Informatycznej i Zastosowań Komputerów w Warszawie)

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

Komitet Główny odbył 4 posiedzenia.

Komitety okręgowe:

Komitet Okręgowy w Białymstoku
przewodniczący:
dr hab. inż. Marek Krętowski, prof. PB (Politechnika Białostocka)
zastępca przewodniczącego:
dr inż. Krzysztof Jurczuk (Politechnika Białostocka)
sekretarz:
Jacek Tomasiewicz (Codility)
członkowie:
Joanna Bujnowska (studentka Uniwersytetu Warszawskiego)
Adam Iwaniuk (student Uniwersytetu Warszawskiego)

Adrian Jaskółka
mgr inż. Konrad Kozłowski (Politechnika Białostocka)

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

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

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

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

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

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

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

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

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

Komitet Okręgowy w Toruniu
przewodniczący:
prof. dr hab. Grzegorz Jarzembski (Uniwersytet Mikołaja Kopernika w Toruniu)
zastępca przewodniczącego:
dr Bartosz Ziemkiewicz (Uniwersytet Mikołaja Kopernika w Toruniu)
sekretarz:
dr Kamila Barylska (Uniwersytet Mikołaja Kopernika w Toruniu)
członkowie:
dr Łukasz Mikulski (Uniwersytet Mikołaja Kopernika w Toruniu)
mgr Marek Nowicki (Uniwersytet Mikołaja Kopernika w Toruniu)
dr Marcin Piątkowski (Uniwersytet Mikołaja Kopernika w Toruniu)

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

Komitet Okręgowy w Warszawie
przewodniczący:
dr Jakub Pawlewicz (Uniwersytet Warszawski)
zastępca przewodniczącego:
dr Tomasz Idziaszek (Codility)
sekretarz:
mgr Monika Kozłowska-Zając (Ośrodek Edukacji Informatycznej i Zastosowań Komputerów w Warszawie)
członkowie:
mgr Szymon Acedański (Uniwersytet Warszawski)
prof. dr hab. Krzysztof Diks (Uniwersytet Warszawski)
dr Jakub Radoszewski (Uniwersytet Warszawski)

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

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

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

JURY OLIMPIADY INFORMATYCZNEJ

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

Michał Adamczyk
Paweł Czerwiński
Kamil Dębowski
Maciej Dębski
Lech Duraj
Michał Glapa
Artur Jamro
Krzysztof Kiljan
Dominik Klemba
Bartosz Kostka
Przemysław Jakub Kozłowski
Aleksander Łukasiewicz
Bartosz Łukasiewicz
Maciej Matraszek
Wojciech Nadara
Paweł Parys
Karol Pokorski
Piotr Smulewicz
Marek Sokołowski
Marek Sommer

ZAWODY I STOPNIA

Zawody I stopnia XXIII Olimpiady Informatycznej odbyły się w dniach 19 października – 16 listopada 2015 roku. Wzięło w nich udział 481 zawodników. Decyzją Komitetu Głównego zdyskwalifikowano pięciu zawodników. Powodem dyskwalifikacji była niesamodzielność rozwiązań zadań konkursowych. Sklasyfikowano 476 zawodników.

Decyzją Komitetu Głównego do zawodów zostało dopuszczonych 14 uczniów gimnazjów. Pochodzili oni z następujących szkół:

nazwa gimnazjum	miejscowość	liczba uczniów
Gimnazjum nr 24 z Oddziałami Dwujęzycznymi oraz Oddziałami Międzynarodowymi w Zespole Szkół Ogólnokształcących nr 1	Gdynia	3
Gimnazjum nr 50 w Zespole Szkół Ogólnokształcących nr 6	Bydgoszcz	2
Gimnazjum nr 42 z Oddziałami Dwujęzycznymi	Warszawa	2
Publiczne Gimnazjum nr 23 z Oddziałami Dwujęzycznymi w Zespole Szkół Ogólnokształcących nr 6 im. Jana Kochanowskiego	Radom	2
Pallotyńskie Gimnazjum im. Stefana Batorego	Lublin	1
Prywatne Gimnazjum im. Królowej Jadwigi	Lublin	1
Gimnazjum nr 16 im. Króla Stefana Batorego	Kraków	1
Publiczne Gimnazjum z Oddziałami Integracyjnymi nr 2 im. Adama Mickiewicza	Pionki	1
Gimnazjum przy Społecznym LO nr 4 im. Batalionu AK „Parasol”	Warszawa	1

Kolejność województw pod względem liczby zawodników była następująca:

mazowieckie	104	wielkopolskie	17
dolnośląskie	63	podkarpackie	15
małopolskie	57	lubelskie	14
pomorskie	56	łódzkie	12
podlaskie	46	świętokrzyskie	8
śląskie	31	warmińsko-mazurskie	7
kujawsko-pomorskie	23	opolskie	4
zachodniopomorskie	17	lubuskie	2

W zawodach I stopnia najliczniej reprezentowane były szkoły:

nazwa szkoły	miejscowość	liczba uczniów
XIV Liceum Ogólnokształcące im. Stanisława Staszica w Zespole Szkół nr 82	Warszawa	53
Zespół Szkół Ogólnokształcących nr 1 (III Liceum Ogólnokształcące im. Marynarki Wojennej RP z Oddziałami Dwujęzycznymi oraz Oddziałami Międzynarodowymi i Gimnazjum nr 24 z Oddziałami Dwujęzycznymi oraz Oddziałami Międzynarodowymi)	Gdynia	40
I Liceum Ogólnokształcące im. Adama Mickiewicza	Białystok	39
Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej w Zespole Szkół nr 14	Wrocław	38
V Liceum Ogólnokształcące im. Augusta Witkowskiego	Kraków	30

Liceum Ogólnokształcące nr III im. Adama Mickiewicza	Wrocław	16
III Liceum Ogólnokształcące im. Adama Mickiewicza	Tarnów	14
Zespół Szkół i Placówek Oświatowych – V Liceum Ogólnokształcące	Bielsko-Biała	13
Zespół Szkół Ogólnokształcących nr 6 (VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich oraz Gimnazjum nr 50)	Bydgoszcz	13
XIII Liceum Ogólnokształcące w Zespole Szkół Ogólnokształcących nr 7	Szczecin	13
Zespół Szkół Ogólnokształcących nr 6 im. Jana Kochanowskiego (VI Liceum Ogólnokształcące z Oddziałami Dwujęzycznymi im. Jana Kochanowskiego oraz Publiczne Gimnazjum nr 23 z Oddziałami Dwujęzycznymi)	Radom	12
VIII Liceum Ogólnokształcące im. Króla Władysława IV w Zespole Szkół nr 15	Warszawa	9
I Liceum Ogólnokształcące im. Stanisława Staszica	Lublin	8
Zespół Szkół Uniwersytetu Mikołaja Kopernika (Gimnazjum i Liceum Akademickie)	Toruń	8
VIII Liceum Ogólnokształcące im. Adama Mickiewicza	Poznań	6
XX Liceum Ogólnokształcące im. Zbigniewa Herberta	Gdańsk	4
I Liceum Ogólnokształcące im. Tadeusza Kościuszki	Legnica	4
Zespół Szkół Komunikacji im. Hipolita Cegielskiego	Poznań	4
I Liceum Ogólnokształcące im. Marii Konopnickiej	Suwałki	4
XVIII Liceum Ogólnokształcące im. Jana Zamoyskiego	Warszawa	4
XXVII Liceum Ogólnokształcące im. Tadeusza Czackiego	Warszawa	4
I Liceum Ogólnokształcące im. Juliusza Słowackiego w Akademickim Zespole Szkół Ogólnokształcących	Chorzów	3
IX Liceum Ogólnokształcące im. Cypriana Kamila Norwida	Częstochowa	3
V Liceum Ogólnokształcące im. Stefana Żeromskiego	Gdańsk	3
VIII Liceum Ogólnokształcące z Oddziałami Dwujęzycznymi im. Marii Skłodowskiej-Curie	Katowice	3
I Liceum Ogólnokształcące im. Mikołaja Kopernika	Łódź	3
IV Liceum Ogólnokształcące im. Marii Skłodowskiej-Curie w Zespole Szkół Ogólnokształcących nr 2	Olsztyn	3

Najliczniej reprezentowane były następujące miejscowości:

Warszawa	79	Białystok	41
Wrocław	57	Gdynia	41

Kraków	35	Łódź	7
Tarnów	16	Katowice	6
Radom	15	Rzeszów	6
Szczecin	15	Olsztyn	5
Bielsko-Biała	13	Częstochowa	4
Bydgoszcz	13	Kielce	4
Poznań	13	Legnica	4
Gdańsk	11	Suwałki	4
Lublin	11	Chorzów	3
Toruń	9	Opole	3

Zawodnicy uczęszczali do następujących klas:

do klasy II gimnazjum	3 uczniów
do klasy III gimnazjum	11
do klasy I szkoły ponadgimnazjalnej	92
do klasy II szkoły ponadgimnazjalnej	169
do klasy III szkoły ponadgimnazjalnej	193
do klasy IV szkoły ponadgimnazjalnej	8

W zawodach I stopnia zawodnicy mieli do rozwiązania pięć zadań:

- „Hydrorozgrywka" autorstwa Michała Włodarczyka,
- „Korale" autorstwa Karola Pokorskiego,
- „Nadajniki" autorstwa Jacka Tomasiewicza,
- „Nim z utrudnieniem" autorstwa Marka Sokołowskiego,
- „Park wodny" autorstwa Jacka Tomasiewicza,

każde oceniane maksymalnie po 100 punktów.

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

- **HYD** – Hydrorozgrywka

	HYD	
	liczba zawodników	czyli
100 pkt.	11	2,31%
75–99 pkt.	1	0,21%
50–74 pkt.	10	2,1%
1–49 pkt.	35	7,35%
0 pkt.	80	16,81%
brak rozwiązania	339	71,22%

- **KOR** – Korale

	KOR	
	liczba zawodników	czyli
100 pkt.	17	3,57%
75–99 pkt.	56	11,77%
50–74 pkt.	68	14,29%
1–49 pkt.	161	33,82%
0 pkt.	36	7,56%
brak rozwiązania	138	28,99%

- **NAD** – Nadajniki

	NAD	
	liczba zawodników	czyli
100 pkt.	79	16,6%
75–99 pkt.	5	1,05%
50–74 pkt.	3	0,63%
1–49 pkt.	57	11,97%
0 pkt.	99	20,8%
brak rozwiązania	233	48,95%

- **NIM** – Nim z utrudnieniem

	NIM	
	liczba zawodników	czyli
100 pkt.	41	8,62%
75–99 pkt.	36	7,56%
50–74 pkt.	15	3,15%
1–49 pkt.	150	31,51%
0 pkt.	25	5,25%
brak rozwiązania	209	43,91%

- **PAR** – Park wodny

	PAR	
	liczba zawodników	czyli
100 pkt.	165	34,66%
75–99 pkt.	35	7,35%
50–74 pkt.	63	13,23%
1–49 pkt.	68	14,29%
0 pkt.	69	14,5%
brak rozwiązania	76	15,97%

W sumie za wszystkie 5 zadań konkursowych:

SUMA	liczba zawodników	czyli
500 pkt.	0	0,00%
375–499 pkt.	25	5,25%
250–374 pkt.	56	11,76%
125–249 pkt.	123	25,84%
1–124 pkt.	211	44,33%
0 pkt.	61	12,82%

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

ZAWODY II STOPNIA

Do zawodów II stopnia, które odbyły się w dniach 9–11 lutego 2016 roku w sześciu okręgach, zakwalifikowano 296 zawodników, którzy osiągnęli w zawodach I stopnia wynik nie mniejszy niż 66 pkt.

Zawodnicy zostali przydzieleni do okręgów w następującej liczbie:

Białystok	39 zawodników	Poznań	31
Gdańsk	38	Warszawa	72
Kraków	69	Wrocław	47

Dwóch zawodników nie stawiło się na zawody. W zawodach wzięło więc udział 294 zawodników.

Zawodnicy uczęszczali do szkół w następujących województwach:

mazowieckie	64	podkarpackie	7
dolnośląskie	43	wielkopolskie	6
małopolskie	41	śląskie	5
podlaskie	41	opolskie	4
pomorskie	38	łódzkie	3
zachodniopomorskie	14	świętokrzyskie	3
lubelskie	13	warmińsko-mazurskie	2
kujawsko-pomorskie	10		

W zawodach II stopnia najliczniej reprezentowane były szkoły:

nazwa szkoły	miejscowość	liczba uczniów
I Liceum Ogólnokształcące im. Adama Mickiewicza	Białystok	37
XIV Liceum Ogólnokształcące im. Stanisława Staszica w Zespole Szkół nr 82	Warszawa	37
Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej w Zespole Szkół nr 14	Wrocław	32
Zespół Szkół Ogólnokształcących nr 1 (III Liceum Ogólnokształcące im. Marynarki Wojennej RP z Oddziałami Dwujęzycznymi oraz Oddziałami Międzynarodowymi i Gimnazjum nr 24 z Oddziałami Dwujęzycznymi oraz Oddziałami Międzynarodowymi)	Gdynia	31
V Liceum Ogólnokształcące im. Augusta Witkowskiego	Kraków	24
XIII Liceum Ogólnokształcące w Zespole Szkół Ogólnokształcących nr 7	Szczecin	12
III Liceum Ogólnokształcące im. Adama Mickiewicza	Tarnów	12
Zespół Szkół Ogólnokształcących nr 6 im. Jana Kochanowskiego (VI Liceum Ogólnokształcące z Oddziałami Dwujęzycznymi im. Jana Kochanowskiego i Publiczne Gimnazjum nr 23 z Oddziałami Dwujęzycznymi)	Radom	10
Liceum Ogólnokształcące nr III im. Adama Mickiewicza	Wrocław	8
I Liceum Ogólnokształcące im. Stanisława Staszica	Lublin	8
VIII Liceum Ogólnokształcące im. Króla Władysława IV w Zespole Szkół nr 15	Warszawa	7
Zespół Szkół Ogólnokształcących nr 6 (VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich oraz Gimnazjum nr 50)	Bydgoszcz	5

Zespół Szkół Uniwersytetu Mikołaja Kopernika (Liceum Akademickie)	Toruń	5
VIII Liceum Ogólnokształcące im. Adama Mickiewicza	Poznań	4
I Liceum Ogólnokształcące im. Tadeusza Kościuszki	Legnica	3
XVIII Liceum Ogólnokształcące im. Jana Zamoyskiego	Warszawa	3

Najliczniej reprezentowane były miejscowości:

Warszawa	51	Radom	10
Wrocław	40	Bydgoszcz	5
Białystok	39	Gdańsk	5
Gdynia	32	Poznań	5
Kraków	26	Toruń	5
Szczecin	14	Rzeszów	4
Tarnów	12	Legnica	3
Lublin	11	Opole	3

9 lutego odbyła się sesja próbna, podczas której zawodnicy rozwiązywali nieliczące się do ogólnej klasyfikacji zadanie „Świąteczny łańcuch" autorstwa Jakuba Radoszewskiego. W dniach konkursowych zawodnicy rozwiązywali następujące zadania:

- w pierwszym dniu zawodów 10 lutego:
  - „Drogi jednokierunkowe" autorstwa Marka Cygana,
  - „Zająknięcia" autorstwa Tomasza Syposza,
- w drugim dniu zawodów 11 lutego:
  - „Arkanoid" autorstwa Tomasza Idziaszka,
  - „Wcale nie Nim" autorstwa Tomasza Idziaszka,

każde oceniane maksymalnie po 100 punktów.

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

- **SWI – próbne** – Świąteczny łańcuch

	SWI – próbne	
	liczba zawodników	czyli
100 pkt.	9	3,06%
75–99 pkt.	2	0,68%
50–74 pkt.	189	64,29%
1–49 pkt.	20	6,8%
0 pkt.	63	21,43%
brak rozwiązania	11	3,74%

- **DRO** – Drogi jednokierunkowe

	DRO	
	liczba zawodników	czyli
100 pkt.	13	4,42%
75–99 pkt.	1	0,34%
50–74 pkt.	43	14,63%
1–49 pkt.	190	64,63%
0 pkt.	39	13,26%
brak rozwiązania	8	2,72%

- **ZAJ** – Zająknięcia

	ZAJ	
	liczba zawodników	czyli
100 pkt.	14	4,76%
75–99 pkt.	2	0,68%
50–74 pkt.	20	6,8%
1–49 pkt.	76	25,85%
0 pkt.	104	35,38%
brak rozwiązania	78	26,53%

- **ARK** – Arkanoid

	ARK	
	liczba zawodników	czyli
100 pkt.	4	1,36%
75–99 pkt.	0	0%
50–74 pkt.	9	3,06%
1–49 pkt.	149	50,68%
0 pkt.	24	8,16%
brak rozwiązania	108	36,74%

- **WCA** – Wcale nie Nim

	WCA	
	liczba zawodników	czyli
100 pkt.	10	3,4%
75–99 pkt.	3	1,02%
50–74 pkt.	9	3,06%
1–49 pkt.	247	84,02%
0 pkt.	9	3,06%
brak rozwiązania	16	5,44%

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

SUMA	liczba zawodników	czyli
400 pkt.	1	0,34%
300–399 pkt.	2	0,68%
200–299 pkt.	9	3,06%
100–199 pkt.	58	19,73%
1–99 pkt.	217	73,81%
0 pkt.	7	2,38%

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

ZAWODY III STOPNIA

Zawody III stopnia odbyły się na Wydziale Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego w dniach 12-16 kwietnia 2016 roku. Do zawodów III stopnia zakwalifikowano 96 najlepszych uczestników zawodów II stopnia, którzy uzyskali wynik nie mniejszy niż 87 pkt.

Zawodnicy uczęszczali do szkół w następujących województwach:

mazowieckie	27	zachodniopomorskie	3
małopolskie	17	podkarpackie	2
dolnośląskie	16	świętokrzyskie	2
pomorskie	14	wielkopolskie	2
podlaskie	7	kujawsko-pomorskie	1
lubelskie	5		

Niżej wymienione szkoły miały w finale więcej niż jednego zawodnika:

nazwa szkoły	miejscowość	liczba uczniów
XIV Liceum Ogólnokształcące im. Stanisława Staszica w Zespole Szkół nr 82	Warszawa	17
V Liceum Ogólnokształcące im. Augusta Witkowskiego	Kraków	16
Zespół Szkół Ogólnokształcących nr 1 (III Liceum Ogólnokształcące im. Marynarki Wojennej RP z Oddziałami Dwujęzycznymi oraz Oddziałami Międzynarodowymi i Gimnazjum nr 24 z Oddziałami Dwujęzycznymi oraz Oddziałami Międzynarodowymi)	Gdynia	13
Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej w Zespole Szkół nr 14	Wrocław	13
I Liceum Ogólnokształcące im. Adama Mickiewicza	Białystok	7
VI Liceum Ogólnokształcące z Oddziałami Dwujęzycznymi im. Jana Kochanowskiego w Zespole Szkół Ogólnokształcących nr 6	Radom	4
XIII Liceum Ogólnokształcące w Zespole Szkół Ogólnokształcących nr 7	Szczecin	3
I Liceum Ogólnokształcące im. Stanisława Staszica	Lublin	3
VIII Liceum Ogólnokształcące im. Króla Władysława IV w Zespole Szkół nr 15	Warszawa	2
XVIII Liceum Ogólnokształcące im. Jana Zamoyskiego	Warszawa	2
Liceum Ogólnokształcące nr III im. Adama Mickiewicza	Wrocław	2
VIII Liceum Ogólnokształcące im. Adama Mickiewicza	Poznań	2

12 kwietnia odbyła się sesja próbna, podczas której zawodnicy rozwiązywali nieliczące się do ogólnej klasyfikacji zadanie „Równoważne programy" autorstwa Michała Włodarczyka. W dniach konkursowych zawodnicy rozwiązywali następujące zadania:

- w pierwszym dniu zawodów 13 kwietnia:
  - „Posłaniec" autorstwa Wojciecha Nadary,
  - „Pracowity Jaś" autorstwa Wojciecha Nadary,
  - „Żywopłot" autorstwa Tomasza Idziaszka,
- w drugim dniu zawodów 14 kwietnia:
  - „Klubowicze" autorstwa Wojciecha Ryttera,
  - „Niebanalne podróże" autorstwa Wojciecha Nadary,
  - „Parada" autorstwa Jacka Tomasiewicza,

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:

- **ROW – próbne** – Równoważne programy

	ROW – próbne	
	liczba zawodników	czyli
100 pkt.	57	59,38%
75–99 pkt.	4	4,17%
50–74 pkt.	3	3,12%
1–49 pkt.	25	26,04%
0 pkt.	0	0%
brak rozwiązania	7	7,29%

- **POS** – Posłaniec

	POS	
	liczba zawodników	czyli
100 pkt.	11	11,46%
75–99 pkt.	7	7,29%
50–74 pkt.	4	4,17%
1–49 pkt.	42	43,75%
0 pkt.	14	14,58%
brak rozwiązania	18	18,75%

- **PRA** – Pracowity Jaś

	PRA	
	liczba zawodników	czyli
100 pkt.	44	45,83%
75–99 pkt.	0	0%
50–74 pkt.	3	3,13%
1–49 pkt.	28	29,17%
0 pkt.	7	7,29%
brak rozwiązania	14	14,58%

- **ZYW** – Żywopłot

	ZYW	
	liczba zawodników	czyli
100 pkt.	48	50%
75–99 pkt.	2	2,08%
50–74 pkt.	4	4,17%
1–49 pkt.	6	6,25%
0 pkt.	12	12,5%
brak rozwiązania	24	25%

- **KLU** – Klubowicze

	KLU	
	liczba zawodników	czyli
100 pkt.	0	0%
75–99 pkt.	2	2,08%
50–74 pkt.	2	2,08%
1–49 pkt.	69	71,88%
0 pkt.	6	6,25%
brak rozwiązania	17	17,71%

- **NIE** – Niebanalne podróże

	NIE	
	liczba zawodników	czyli
100 pkt.	7	7,29%
75–99 pkt.	2	2,08%
50–74 pkt.	2	2,08%
1–49 pkt.	28	29,17%
0 pkt.	33	34,38%
brak rozwiązania	24	25%

- **PAR** – Parada

	PAR	
	liczba zawodników	czyli
100 pkt.	74	77,08%
75–99 pkt.	5	5,21%
50–74 pkt.	10	10,42%
1–49 pkt.	5	5,21%
0 pkt.	0	0%
brak rozwiązania	2	2,08%

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

SUMA	liczba zawodników	czyli
450–600 pkt.	11	11,46%
300–449 pkt.	21	21,88%
150–299 pkt.	43	44,79%
1–149 pkt.	19	19,79%
0 pkt.	2	2,08%

16 kwietnia, w auli Centrum Nowych Technologii Uniwersytetu Warszawskiego w Warszawie przy ul. S. Banacha 2c, odbyła się uroczystość zakończenia XXIII Olimpiady Informatycznej, na której ogłoszono wyniki zawodów III stopnia.

Poniżej zestawiono listę wszystkich laureatów i wyróżnionych finalistów:

(1) **Jarosław Kwiecień**, 3 klasa, Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej, Wrocław, 591 pkt., laureat I miejsca

(2) **Mateusz Radecki**, 3 klasa, VI Liceum Ogólnokształcące im. Jana Kochanowskiego, Radom, 584 pkt., laureat I miejsca

(3) **Paweł Burzyński**, 2 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia, 538 pkt., laureat I miejsca

(4) **Juliusz Pham**, 1 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia, 533 pkt., laureat I miejsca

(5) **Bartłomiej Karasek**, 3 klasa, Zespół Szkół nr 1, Grodzisk Mazowiecki, 519 pkt., laureat I miejsca

(6) **Mariusz Trela**, 1 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków, 516 pkt., laureat I miejsca

(7) **Maciej Sypetkowski**, 3 klasa, I Liceum Ogólnokształcące im. Władysława Jagiełły, Krasnystaw, 513 pkt., laureat I miejsca

(8) **Stanisław Szcześniak**, 1 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica, Warszawa, 470 pkt., laureat II miejsca

(9) **Franciszek Budrowski**, 2 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza, Białystok, 462 pkt., laureat II miejsca

(10) **Piotr Kowalewski**, 1 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia, 458 pkt., laureat II miejsca

(11) **Łukasz Kondraciuk**, 3 klasa, IV Liceum Ogólnokształcące im. Mikołaja Kopernika, Rzeszów, 457 pkt., laureat II miejsca

(12) **Jan Tabaszewski**, 3 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica, Warszawa, 447 pkt., laureat II miejsca

(13) **Michał Tepper**, 3 klasa, VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich, Bydgoszcz, 434 pkt., laureat II miejsca

(14) **Tomasz Kościuszko**, 3 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica, Warszawa, 431 pkt., laureat II miejsca

(15) **Michał Górniak**, 2 klasa, I Liceum Ogólnokształcące im. Tadeusza Kościuszki, Legnica, 428 pkt., laureat II miejsca

(16) **Piotr Grabowski**, 2 klasa, I Liceum Ogólnokształcące im. Stanisława Staszica, Lublin, 419 pkt., laureat II miejsca

(17) **Jakub Boguta**, 2 klasa, I Liceum Ogólnokształcące im. Stanisława Staszica, Lublin, 416 pkt., laureat II miejsca

(18) **Stanisław Strzelecki**, 1 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica, Warszawa, 385 pkt., laureat III miejsca

(19) **Anadi Agrawal**, 1 klasa, Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej, Wrocław, 375 pkt., laureat III miejsca

(20) **Marek Żochowski**, 2 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia, 374 pkt., laureat III miejsca

(21) **Anna Białokozowicz**, 3 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza, Białystok, 366 pkt., laureatka III miejsca

(22) **Kacper Kluk**, 1 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia, 357 pkt., laureat III miejsca

(23) **Rafał Łyżwa**, 2 klasa, VI Liceum Ogólnokształcące im. Jana Kochanowskiego, Radom, 344 pkt., laureat III miejsca

(23) **Kacper Walentynowicz**, 1 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia, 344 pkt., laureat III miejsca

(25) **Piotr Pawlak**, 3 klasa, Ogólnokształcąca Szkoła Muzyczna I i II stopnia im. Feliksa Nowowiejskiego, Gdańsk, 338 pkt., laureat III miejsca

(26) **Jan Lebioda**, 2 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica, Warszawa, 328 pkt., laureat III miejsca

(27) **Piotr Borowski**, 2 klasa, I Liceum Ogólnokształcące im. Stanisława Staszica, Lublin, 316 pkt., laureat III miejsca

(28) **Tomasz Grześkiewicz**, 2 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica, Warszawa, 311 pkt., laureat III miejsca

(29) **Mateusz Gienieczko**, 2 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia, 309 pkt., laureat III miejsca

(30) **Mateusz Hazy**, 3 klasa, Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej, Wrocław, 308 pkt., laureat III miejsca

(31) **Cyryl Waśkiewicz**, 3 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica, Warszawa, 304 pkt., laureat III miejsca

(32) **Mateusz Szpyrka**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków, 303 pkt., laureat III miejsca

(33) **Tomasz Nowak**, 1 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica, Warszawa, 296 pkt., laureat III miejsca

(34) **Jan Olkowski**, 2 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica, Warszawa, 292 pkt., laureat III miejsca

(35) **Konrad Staniszewski**, 2 klasa, VI Liceum Ogólnokształcące im. Jana Kochanowskiego, Radom, 288 pkt., laureat III miejsca

(36) **Maciej Nadolski**, 2 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia, 284 pkt., laureat III miejsca

(37) **Piotr Kuczko**, 2 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza, Białystok, 280 pkt., laureat III miejsca

(38) **Krzysztof Potępa**, 2 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków, 279 pkt., laureat III miejsca

(39) **Jakub Bartmiński**, 2 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica, Warszawa, 273 pkt., laureat III miejsca

(40) **Agnieszka Dudek**, 3 klasa, Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej, Wrocław, 270 pkt., laureatka III miejsca

(41) **Adam Pawłowski**, 1 klasa, VIII Liceum Ogólnokształcące im. Adama Mickiewicza, Poznań, 268 pkt., laureat III miejsca

(42) **Łukasz Janeczko**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków, 266 pkt., laureat III miejsca

(43) **Michał Niciejewski**, 2 klasa, XIII Liceum Ogólnokształcące, Szczecin, 258 pkt., laureat III miejsca

(44) **Kamil Ćwintal**, 3 klasa, Zespół Szkół Ogólnokształcących im. Edwarda Szylki, Ożarów, 253 pkt., laureat III miejsca

(45) **Igor Dolecki**, 2 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica, Warszawa, 251 pkt., laureat III miejsca

(46) **Michał Siennicki**, 1 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica, Warszawa, 250 pkt., laureat III miejsca

(47) **Jakub Zadrożny**, 3 klasa, Liceum Ogólnokształcące nr III im. Adama Mickiewicza, Wrocław, 248 pkt., finalista z wyróżnieniem

(48) **Krzysztof Piesiewicz**, 3 klasa, VIII Liceum Ogólnokształcące im. Władysława IV, Warszawa, 247 pkt., finalista z wyróżnieniem

(49) **Artur Puzio**, 2 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica, Warszawa, 245 pkt., finalista z wyróżnieniem

(50) **Kamil Piechowiak**, 2 klasa, VIII Liceum Ogólnokształcące im. Adama Mickiewicza, Poznań, 242 pkt., finalista z wyróżnieniem

(50) **Rafał Pragacz**, 3 klasa, Zespół Szkół Sióstr Nazaretanek, Kielce, 242 pkt., finalista z wyróżnieniem

(52) **Iwona Kotlarska**, 2 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica, Warszawa, 237 pkt., finalistka z wyróżnieniem

(53) **Piotr Bujakowski**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków, 232 pkt., finalista z wyróżnieniem

(53) **Jakub Obuchowski**, 3 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza, Białystok, 232 pkt., finalista z wyróżnieniem

(55) **Alicja Chaszczewicz**, 3 klasa, Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej, Wrocław, 226 pkt., finalistka z wyróżnieniem

(55) **Tomasz Ponitka**, 1 klasa, Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej, Wrocław, 226 pkt., finalista z wyróżnieniem

(57) **Tomasz Kanas**, 3 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia, 220 pkt., finalista z wyróżnieniem

(58) **Mateusz Maciej Masłowski**, 3 klasa, Gimnazjum nr 24 (Zespół Szkół Ogólnokształcących nr 1), Gdynia, 215 pkt., finalista z wyróżnieniem

(59) **Krzysztof Małysa**, 2 klasa, XIII Liceum Ogólnokształcące, Szczecin, 214 pkt., finalista z wyróżnieniem

(59) **Bartosz Rudzki**, 3 klasa, Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej, Wrocław, 214 pkt., finalista z wyróżnieniem

(61) **Bartłomiej Wrona**, 3 klasa, XXVII Liceum Ogólnokształcące im. Tadeusza Czackiego, Warszawa, 211 pkt., finalista z wyróżnieniem

(62) **Mateusz Orda**, 1 klasa, Liceum Ogólnokształcące nr III im. Adama Mickiewicza, Wrocław, 206 pkt., finalista z wyróżnieniem

(62) **Marek Skiba**, 3 klasa, Prywatne Gimnazjum i Liceum Ogólnokształcące im. Królowej Jadwigi, Lublin, 206 pkt., finalista z wyróżnieniem

(64) **Wojciech Gołaszewski**, 3 klasa, Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej, Wrocław, 205 pkt., finalista z wyróżnieniem

(64) **Michał Łopata**, 2 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza, Białystok, 205 pkt., finalista z wyróżnieniem

(64) **Julia Majkowska**, 3 klasa, Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej, Wrocław, 205 pkt., finalistka z wyróżnieniem

(64) **Konstanty Subbotko**, 2 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica, Warszawa, 205 pkt., finalista z wyróżnieniem

(68) **Paweł Anikiel**, 1 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia, 204 pkt., finalista z wyróżnieniem

(69) **Grzegorz Eugeniusz Uriasz**, 2 klasa, I Liceum Ogólnokształcące im. Mikołaja Kopernika, Krosno, 200 pkt., finalista z wyróżnieniem

Lista pozostałych finalistów w kolejności alfabetycznej:

- **Michał Balcerzak**, 3 klasa, VIII Liceum Ogólnokształcące im. Władysława IV, Warszawa
- **Jakub Buziewicz**, 2 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków
- **Paweł Charyło**, 2 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza, Białystok
- **Mateusz Dudziński**, 2 klasa, XVIII Liceum Ogólnokształcące im. Jana Zamoyskiego, Warszawa
- **Paweł Jasiak**, 1 klasa, Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej, Wrocław
- **Szymon Kapała**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków
- **Rafał Kaszuba**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków
- **Wiktoria Kośny**, 3 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica, Warszawa
- **Leopold Kozioł**, 2 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków
- **Krystian Król**, 1 klasa, VI Liceum Ogólnokształcące im. Jana Kochanowskiego, Radom
- **Sebastian Książczyk**, 3 klasa, Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej, Wrocław
- **Robert Laskowski**, 2 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica, Warszawa
- **Mateusz Maćkowski**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków
- **Tomasz Madej**, 3 klasa, XIII Liceum Ogólnokształcące, Szczecin
- **Jan Mirkiewicz**, 3 klasa, Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej, Wrocław
- **Maciej Nemś**, 2 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków
- **Jan Omeljaniuk**, 3 klasa, XVIII Liceum Ogólnokształcące im. Jana Zamoyskiego, Warszawa
- **Adam Pardyl**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków
- **Krzysztof Pióro**, 2 klasa, III Liceum Ogólnokształcące im. Adama Mickiewicza, Tarnów

- **Bartosz Podkanowicz**, 2 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków
- **Paweł Sawicki**, 1 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia
- **Marcin Serwin**, 2 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków
- **Michał Szostek**, 2 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia
- **Karol Waszczuk**, 3 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza, Białystok
- **Łukasz Wróblewski**, 2 klasa, Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej, Wrocław
- **Tomasz Zieliński**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków
- **Krzysztof Ziobro**, 1 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków

Komitet Główny Olimpiady Informatycznej przyznał następujące nagrody rzeczowe:

(1) puchar wręczono zwycięzcy XXIII Olimpiady Jarosławowi Kwietniowi,
(2) złote, srebrne i brązowe medale przyznano odpowiednio laureatom I, II i III miejsca,
(3) stypendia przyznano laureatom I i II miejsca według następującego klucza:

   - zwycięzca – 4000 zł,
   - laureaci I miejsca – 2000 zł,
   - laureaci II miejsca – 1000 zł.

   Stypendia zostały ufundowane przez Ministerstwo Edukacji Narodowej.
(4) książki ufundowane przez Wydawnictwo Naukowe PWN przyznano wszystkim laureatom i finalistom,
(5) książki „W poszukiwaniu wyzwań 2" przyznano wszystkim laureatom i finalistom,
(6) roczną prenumeratę miesięcznika „Delta" przyznano wszystkim laureatom.

Komitet Główny powołał reprezentacje na:

(1) **Międzynarodową Olimpiadę Informatyczną IOI'2016**, która odbyła się w terminie 12-19 sierpnia 2016 r. w Rosji w Kazaniu
(2) oraz **Olimpiadę Informatyczną Krajów Europy Środkowej CEOI'2016**, która odbyła się w terminie 18-23 lipca 2016 r. w Rumunii w miejscowości Piatra-Neamţ w składzie:

   - Jarosław Kwiecień
   - Mateusz Radecki

- Paweł Burzyński
- Juliusz Pham

rezerwowi:

- Bartłomiej Karasek
- Mariusz Trela

(3) Na **Bałtycką Olimpiadę Informatyczną BOI'2016**, która odbyła się w terminie 11-15 maja 2016 r. w Finlandii w Helsinkach. Zostali do niej powołani zawodnicy z klas programowo niższych niż ostatnia klasa szkoły ponadgimnazjalnej, w składzie:

- Paweł Burzyński
- Juliusz Pham
- Mariusz Trela
- Stanisław Szcześniak
- Franciszek Budrowski
- Piotr Kowalewski

rezerwowi:

- Michał Górniak
- Piotr Grabowski

Komitet Główny podjął następujące uchwały o udziale młodzieży w Obozie Naukowo-Treningowym im. A. Kreczmara oraz Obozie Czesko-Polsko-Słowackim, które odbyły się w Warszawie w terminach 27 czerwca – 7 lipca 2016 r. oraz 27 czerwca – 3 lipca 2016 r.:

- na Obóz Naukowo-Treningowy im. A. Kreczmara zostali zaproszeni reprezentanci na Międzynarodową Olimpiadę Informatyczną, wraz z zawodnikami rezerwowymi, oraz finaliści Olimpiady, którzy nie uczęszczali w tym roku szkolnym do programowo najwyższej klasy szkoły ponadgimnazjalnej,
- na Obóz Czesko-Polsko-Słowacki zostali zaproszeni reprezentanci na Międzynarodową Olimpiadę Informatyczną oraz zawodnicy rezerwowi.

Sekretariat wystawił łącznie 46 zaświadczeń o uzyskaniu tytułu laureata, 23 zaświadczenia o uzyskaniu tytułu wyróżnionego finalisty oraz 27 zaświadczeń o uzyskaniu tytułu finalisty XXIII Olimpiady Informatycznej.

Lista opiekunów naukowych wskazanych przez laureatów i finalistów XXIII Olimpiady Informatycznej przedstawia się następująco:

- Maciej Borowiecki (XXVII Liceum Ogólnokształcące im. Tadeusza Czackiego w Warszawie)
  - Bartłomiej Wrona – finalista z wyróżnieniem

- Iwona Bujnowska (I Liceum Ogólnokształcące im. Adama Mickiewicza w Białymstoku)
  - Franciszek Budrowski – laureat II miejsca
  - Anna Białokozowicz – laureatka III miejsca
  - Jakub Obuchowski – finalista z wyróżnieniem
  - Karol Waszczuk – finalista
- Ireneusz Bujnowski (I Liceum Ogólnokształcące im. Adama Mickiewicza w Białymstoku)
  - Franciszek Budrowski – laureat II miejsca
  - Anna Białokozowicz – laureatka III miejsca
  - Piotr Kuczko – laureat III miejsca
  - Michał Łopata – finalista z wyróżnieniem
  - Jakub Obuchowski – finalista z wyróżnieniem
  - Paweł Charyło – finalista
  - Karol Waszczuk – finalista
- Patryk Czajka (student Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego)
  - Jakub Bartmiński – laureat III miejsca
  - Igor Dolecki – laureat III miejsca
  - Jan Lebioda – laureat III miejsca
  - Tomasz Nowak – laureat III miejsca
  - Stanisław Strzelecki – laureat III miejsca
  - Artur Puzio – finalista z wyróżnieniem
  - Jan Omeljaniuk – finalista
- Jolanta Ćwintal (Zespół Szkół Ogólnokształcących im. Edwarda Szylki w Ożarowie)
  - Kamil Ćwintal – laureat III miejsca
- Czesław Drozdowski (XIII Liceum Ogólnokształcące w Szczecinie)
  - Michał Niciejewski – laureat III miejsca
  - Krzysztof Małysa – finalista z wyróżnieniem
  - Tomasz Madej – finalista
- Bartłomiej Dudek (student Wydziału Matematyki i Informatyki Uniwersytetu Wrocławskiego)
  - Jarosław Kwiecień – laureat I miejsca
  - Agnieszka Dudek – laureatka III miejsca
  - Mateusz Hazy – laureat III miejsca
  - Alicja Chaszczewicz – finalistka z wyróżnieniem
  - Wojciech Gołaszewski – finalista z wyróżnieniem
  - Julia Majkowska – finalistka z wyróżnieniem
  - Bartosz Rudzki – finalista z wyróżnieniem
  - Jakub Zadrożny – finalista z wyróżnieniem
  - Sebastian Książczyk – finalista
  - Jan Mirkiewicz – finalista

- Lech Duraj (Katedra Algorytmiki Uniwersytetu Jagiellońskiego)
  - Mariusz Trela – laureat I miejsca
  - Krzysztof Ziobro – finalista
- Paweł Dybiec (student Wydziału Matematyki i Informatyki Uniwersytetu Wrocławskiego)
  - Paweł Jasiak – finalista
- Andrzej Dyrek (V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie)
  - Łukasz Janeczko – laureat III miejsca
  - Krzysztof Potępa – laureat III miejsca
  - Mateusz Szpyrka – laureat III miejsca
  - Piotr Bujakowski – finalista z wyróżnieniem
  - Jakub Buziewicz – finalista
  - Szymon Kapała – finalista
  - Leopold Kozioł – finalista
  - Mateusz Maćkowski – finalista
  - Maciej Nemś – finalista
  - Adam Pardyl – finalista
  - Bartosz Podkanowicz – finalista
  - Marcin Serwin – finalista
- Karol Farbiś (student Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego)
  - Jan Tabaszewski – laureat II miejsca
- Michał Glapa (student Katedry Algorytmiki Uniwersytetu Jagiellońskiego)
  - Mariusz Trela – laureat I miejsca
  - Łukasz Janeczko – laureat III miejsca
  - Rafał Kaszuba – finalista
- Alina Gościniak (VIII Liceum Ogólnokształcące im. Adama Mickiewicza w Poznaniu)
  - Adam Pawłowski – laureat III miejsca
  - Kamil Piechowiak – finalista z wyróżnieniem
- Grzegorz Herman (Katedra Algorytmiki Uniwersytetu Jagiellońskiego)
  - Łukasz Janeczko – laureat III miejsca
  - Mateusz Szpyrka – laureat III miejsca
  - Piotr Bujakowski – finalista z wyróżnieniem
  - Szymon Kapała – finalista
  - Rafał Kaszuba – finalista
  - Mateusz Maćkowski – finalista
  - Adam Pardyl – finalista
  - Tomasz Zieliński – finalista
- Andrzej Jackowski (Społeczne Gimnazjum nr 2 STO w Białymstoku)
  - Franciszek Budrowski – laureat II miejsca

- Henryk Kawka (Zespół Szkół nr 7 w Lublinie)
  - Jakub Boguta – laureat II miejsca
  - Piotr Grabowski – laureat II miejsca
  - Piotr Borowski – laureat III miejsca
  - Marek Skiba – finalista z wyróżnieniem
- Bartosz Kostka (student Wydziału Matematyki i Informatyki Uniwersytetu Wrocławskiego)
  - Jarosław Kwiecień – laureat I miejsca
  - Michał Górniak – laureat II miejsca
  - Anadi Agrawal – laureat III miejsca
  - Agnieszka Dudek – laureatka III miejsca
  - Mateusz Hazy – laureat III miejsca
  - Alicja Chaszczewicz – finalistka z wyróżnieniem
  - Wojciech Gołaszewski – finalista z wyróżnieniem
  - Julia Majkowska – finalistka z wyróżnieniem
  - Tomasz Ponitka – finalista z wyróżnieniem
  - Bartosz Rudzki – finalista z wyróżnieniem
  - Jakub Zadrożny – finalista z wyróżnieniem
  - Sebastian Książczyk – finalista
  - Jan Mirkiewicz – finalista
- Katarzyna Kowalska (studentka Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego)
  - Igor Dolecki – laureat III miejsca
  - Stanisław Strzelecki – laureat III miejsca
- Przemysław Jakub Kozłowski (student Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego)
  - Bartłomiej Wrona – finalista z wyróżnieniem
- Jan Kwaśniak (student Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego)
  - Rafał Pragacz – finalista z wyróżnieniem
- Romualda Laskowska (I Liceum Ogólnokształcące im. Tadeusza Kościuszki w Legnicy)
  - Michał Górniak – laureat II miejsca
- Krzysztof Loryś (Wydział Matematyki i Informatyki Uniwersytetu Wrocławskiego)
  - Michał Górniak – laureat II miejsca
  - Jakub Zadrożny – finalista z wyróżnieniem
- Aleksander Łukasiewicz (student Wydziału Matematyki i Informatyki Uniwersytetu Wrocławskiego)
  - Tomasz Ponitka – finalista z wyróżnieniem
  - Paweł Jasiak – finalista
- Dawid Matla (XIV Liceum Ogólnokształcące im. Polonii Belgijskiej we Wrocławiu)
  - Agnieszka Dudek – laureatka III miejsca
  - Alicja Chaszczewicz – finalistka z wyróżnieniem

- Mirosław Mortka (VI Liceum Ogólnokształcące im. Jana Kochanowskiego w Radomiu)
  - Mateusz Radecki – laureat I miejsca
  - Rafał Łyżwa – laureat III miejsca
  - Konrad Staniszewski – laureat III miejsca
  - Krystian Król – finalista
- Adam Nieżurawski (Klub Naukowy „Feniks" w Warszawie)
  - Bartłomiej Wrona – finalista z wyróżnieniem
- Rafał Nowak (Wydział Matematyki i Informatyki Uniwersytetu Wrocławskiego)
  - Anadi Agrawal – laureat III miejsca
  - Łukasz Wróblewski – finalista
- Irena Oleander (IV Liceum Ogólnokształcące im. Mikołaja Kopernika w Rzeszowie)
  - Łukasz Kondraciuk – laureat II miejsca
- Małgorzata Piekarska (VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich w Bydgoszczy)
  - Michał Tepper – laureat II miejsca
- Mirosław Pietrzycki (I Liceum Ogólnokształcące im. Stanisława Staszica w Lublinie)
  - Piotr Grabowski – laureat II miejsca
  - Piotr Borowski – laureat III miejsca
- Andrzej Piotrowski (Zespół Szkół Ogólnokształcących w Krośnie)
  - Grzegorz Eugeniusz Uriasz – finalista z wyróżnieniem
- Karol Pokorski (Google Zurich)
  - Jarosław Kwiecień – laureat I miejsca
  - Anadi Agrawal – laureat III miejsca
  - Julia Majkowska – finalistka z wyróżnieniem
  - Mateusz Orda – finalista z wyróżnieniem
  - Tomasz Ponitka – finalista z wyróżnieniem
  - Bartosz Rudzki – finalista z wyróżnieniem
  - Paweł Jasiak – finalista
  - Sebastian Książczyk – finalista
- Adam Polak (doktorant w Katedrze Algorytmiki Uniwersytetu Jagiellońskiego)
  - Krzysztof Potępa – laureat III miejsca
  - Jakub Buziewicz – finalista
  - Rafał Kaszuba – finalista
  - Leopold Kozioł – finalista
  - Maciej Nemś – finalista
  - Bartosz Podkanowicz – finalista
  - Marcin Serwin – finalista
  - Krzysztof Ziobro – finalista

- Agnieszka Samulska (VIII Liceum Ogólnokształcące im. Króla Władysława IV w Warszawie)
  - Krzysztof Piesiewicz – finalista z wyróżnieniem
  - Michał Balcerzak – finalista
- Radosław Serafin (Wydział Matematyki i Informatyki Uniwersytetu Wrocławskiego)
  - Mateusz Orda – finalista z wyróżnieniem
- Paweł Seta (Centralny Ośrodek Informatyki w Warszawie)
  - Konrad Staniszewski – laureat III miejsca
- Piotr Smulewicz
  - Stanisław Szcześniak – laureat II miejsca
- Marek Sokołowski (student Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego)
  - Mateusz Radecki – laureat I miejsca
  - Tomasz Kościuszko – laureat II miejsca
  - Jan Tabaszewski – laureat II miejsca
- Marek Sommer (student Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego)
  - Tomasz Kościuszko – laureat II miejsca
  - Stanisław Szcześniak – laureat II miejsca
- Hanna Stachera (XIV Liceum Ogólnokształcące im. Stanisława Staszica i XVIII Liceum Ogólnokształcące im. Jana Zamoyskiego w Warszawie)
  - Tomasz Nowak – laureat III miejsca
  - Konstanty Subbotko – finalista z wyróżnieniem
  - Jan Omeljaniuk – finalista
- Szymon Stankiewicz (ERICPOL w Krakowie)
  - Mateusz Radecki – laureat I miejsca
  - Rafał Łyżwa – laureat III miejsca
  - Konrad Staniszewski – laureat III miejsca
  - Krystian Król – finalista
- Ryszard Szubartowski (III Liceum Ogólnokształcące im. Marynarki Wojennej RP w Gdyni, Stowarzyszenie Talent)
  - Paweł Burzyński – laureat I miejsca
  - Juliusz Pham – laureat I miejsca
  - Piotr Kowalewski – laureat II miejsca
  - Mateusz Gienieczko – laureat III miejsca
  - Maciej Nadolski – laureat III miejsca
  - Kacper Walentynowicz – laureat III miejsca
  - Marek Żochowski – laureat III miejsca
  - Tomasz Kanas – finalista z wyróżnieniem
  - Mateusz Maciej Masłowski – finalista z wyróżnieniem
  - Paweł Sawicki – finalista
  - Michał Szostek – finalista

- Karol Szymański (LXIV Liceum Ogólnokształcące im. Stanisława Witkiewicza w Warszawie)
  - Konstanty Subbotko – finalista z wyróżnieniem
- Joanna Śmigielska (XIV Liceum Ogólnokształcące im. Stanisława Staszica w Warszawie)
  - Tomasz Kościuszko – laureat II miejsca
  - Stanisław Szcześniak – laureat II miejsca
  - Jan Tabaszewski – laureat II miejsca
  - Jakub Bartmiński – laureat III miejsca
  - Igor Dolecki – laureat III miejsca
  - Tomasz Grześkiewicz – laureat III miejsca
  - Jan Lebioda – laureat III miejsca
  - Jan Olkowski – laureat III miejsca
  - Stanisław Strzelecki – laureat III miejsca
  - Iwona Kotlarska – finalistka z wyróżnieniem
  - Konstanty Subbotko – finalista z wyróżnieniem
- Krzysztof Wabik (Liceum Ogólnokształcące w Zespole Szkół Ogólnokształcących im. Edwarda Szylki w Ożarowie)
  - Kamil Ćwintal – laureat III miejsca
- Justyna Wilińska (VIII Liceum Ogólnokształcące im. Króla Władysława IV w Warszawie)
  - Michał Balcerzak – finalista
- Robert Zieliński (III Liceum Ogólnokształcące im. Adama Mickiewicza w Tarnowie)
  - Krzysztof Pióro – finalista

WYNIKI ZAWODÓW MIĘDZYNARODOWYCH

W dniach 12-19 sierpnia 2016 r. w Kazaniu w Rosji odbyły się zawody XXVIII Międzynarodowej Olimpiady Informatycznej. W gronie 308 zawodników z 80 krajów nasi zawodnicy osiągnęli następujące wyniki:

- złote medale zdobyli Jarosław Kwiecień, absolwent Liceum Ogólnokształcącego nr XIV im. Polonii Belgijskiej we Wrocławiu (*ex-aequo* 8 miejsce w całym rankingu), oraz Mateusz Radecki, absolwent VI Liceum Ogólnokształcącego im. Jana Kochanowskiego w Radomiu,
- srebro wywalczył Juliusz Pham, uczeń klasy pierwszej III Liceum Ogólnokształcącego im. Marynarki Wojennej RP w Gdyni,
- brązowy medal zdobył Paweł Burzyński, uczeń klasy drugiej III Liceum Ogólnokształcącego im. Marynarki Wojennej RP w Gdyni.

Był to już trzeci złoty medal zdobyty przez Jarosława Kwietnia na Międzynarodowej Olimpiadzie Informatycznej, dzięki czemu jako czwarty Polak trafił do pierwszej dziesiątki najlepszych zawodników w historii zawodów IOI. Pozostali reprezentanci Polski występowali na zawodach IOI po raz pierwszy. Najlepszy wynik w zawodach osiągnął chiński zawodnik Ce Jin.

Łącznie we wszystkich zawodach IOI Polska zdobyła już 101 medali i zajmuje czwarte miejsce w klasyfikacji medalowej krajów, za Chinami, Rosją i USA.

Opiekunami drużyny byli dr Jakub Radoszewski oraz Bartosz Kostka. Pracami Międzynarodowego Komitetu Naukowego podczas Międzynarodowej Olimpiady Informatycznej kierował dr Jakub Łącki.

W dniach 18-23 lipca 2016 roku w miejscowości Piatra-Neamţ w Rumunii odbyła się Olimpiada Informatyczna Krajów Europy Środkowej CEOI'2016. W Olimpiadzie tej wzięli udział reprezentanci z następujących krajów: Bułgaria, Chorwacja, Czechy, Gruzja, Mołdawia, Niemcy, Polska, Rumunia, Słowacja, Słowenia, Szwajcaria i Węgry. Każdy kraj reprezentowało czterech uczniów, laureatów krajowych olimpiad informatycznych.

W XXIII Olimpiadzie Informatycznej Krajów Europy Środkowej przyznano 5 złotych, 9 srebrnych i 14 brązowych medali. Znakomicie zaprezentowali się polscy uczniowie, zdobywając 1 złoty, 1 srebrny i 2 brązowe medale:

- złoty medal dla Polski wywalczył Mateusz Radecki, absolwent VI Liceum Ogólnokształcącego im. Jana Kochanowskiego w Radomiu,
- srebrny medal zdobył Jarosław Kwiecień, absolwent Liceum Ogólnokształcącego nr XIV im. Polonii Belgijskiej we Wrocławiu,
- brązowe medale zdobyli: Juliusz Pham, uczeń klasy pierwszej III Liceum Ogólnokształcącego im. Marynarki Wojennej RP w Gdyni, oraz Bartłomiej Karasek, absolwent Zespołu Szkół nr 1 w Grodzisku Mazowieckim.

Opiekunami drużyny byli Kamil Dębowski oraz Dominik Klemba.

W dniach 11-15 maja 2016 roku w Helsinkach miała miejsce XXII Bałtycka Olimpiada Informatyczna. Olimpiada jest adresowana do utalentowanych informatycznie uczniów z Danii, Estonii, Finlandii, Litwy, Łotwy, Niemiec, Norwegii, Polski i Szwecji. Każdy kraj reprezentowało 6 uczniów, laureatów krajowych olimpiad informatycznych.

W XXII Bałtyckiej Olimpiadzie Informatycznej przyznano 5 złotych medali, 10 medali srebrnych oraz 11 medali brązowych. Znakomicie zaprezentowali się polscy uczniowie, zdobywając 3 medale złote, 2 medale srebrne i 1 medal brązowy.

Absolutnym zwycięzcą XXII Bałtyckiej Olimpiady Informatycznej został Mariusz Trela, uczeń klasy pierwszej V Liceum Ogólnokształcącego im. Augusta Witkowskiego w Krakowie. Złote medale dla Polski wywalczyli także:

- Juliusz Pham, uczeń klasy pierwszej III Liceum Ogólnokształcącego im. Marynarki Wojennej RP w Gdyni oraz
- Franciszek Budrowski, uczeń klasy drugiej I Liceum Ogólnokształcącego im. Adama Mickiewicza w Białymstoku.

Na zawodach w Helsinkach srebrnymi medalistami zostali:

- Paweł Burzyński, uczeń klasy drugiej III Liceum Ogólnokształcącego im. Marynarki Wojennej RP w Gdyni oraz
- Stanisław Szcześniak, uczeń klasy pierwszej XIV Liceum Ogólnokształcącego im. Stanisława Staszica w Warszawie.

Brąz do Polski przywiózł

- Piotr Kowalewski, uczeń klasy pierwszej III Liceum Ogólnokształcącego im. Marynarki Wojennej RP w Gdyni.

Opiekunami drużyny byli Bartosz Kostka oraz Karol Kaszuba.

MATERIAŁY OLIMPIJSKIE

Podobnie jak w ubiegłych latach, w przygotowaniu jest publikacja zawierająca pełną informację o XXIII 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, 26 sierpnia 2016 roku

Regulamin Ogólnopolskiej Olimpiady Informatycznej

Olimpiada Informatyczna, zwana dalej Olimpiadą, jest olimpiadą przedmiotową powołaną przez Instytut Informatyki Uniwersytetu Wrocławskiego 10 grudnia 1993 roku. Olimpiada działa zgodnie z Rozporządzeniem Ministra Edukacji Narodowej i Sportu z 29 stycznia 2002 roku w sprawie organizacji oraz sposobu przeprowadzenia konkursów, turniejów i olimpiad z późniejszymi zmianami (Dz. U. 2002, nr 13, poz. 125).

Cele Olimpiady:

- stworzenie motywacji dla zainteresowania nauczycieli i uczniów informatyką,
- rozszerzanie współdziałania nauczycieli akademickich z nauczycielami szkół w kształceniu młodzieży uzdolnionej,
- stymulowanie aktywności poznawczej młodzieży informatycznie uzdolnionej,
- kształtowanie umiejętności samodzielnego zdobywania i rozszerzania wiedzy informatycznej,
- stwarzanie młodzieży możliwości szlachetnego współzawodnictwa w rozwijaniu swoich uzdolnień, a nauczycielom – warunków twórczej pracy z młodzieżą,
- wyłanianie reprezentacji Rzeczypospolitej Polskiej na Międzynarodową Olimpiadę Informatyczną i inne międzynarodowe zawody informatyczne.

Cele Olimpiady są osiągane poprzez:

- organizację olimpiady przedmiotowej z informatyki dla uczniów szkół ponadgimnazjalnych,
- organizowanie corocznych obozów naukowych dla najlepszych uczestników olimpiady,
- przygotowywanie i publikowanie materiałów edukacyjnych dla uczniów zainteresowanych udziałem w olimpiadach i ich nauczycieli.

Rozdział I – Olimpiada i jej organizator

§1 PRAWA I OBOWIĄZKI ORGANIZATORA

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

(2) W organizacji Olimpiady Organizator współdziała z Wydziałem Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego, Instytutem Informatyki Uniwersytetu Wrocławskiego, Katedrą Algorytmiki Uniwersytetu Jagiellońskiego, Wydziałem Matematyki i Informatyki Uniwersytetu im. Mikołaja Kopernika w Toruniu, Instytutem Informatyki Wydziału Automatyki, Elektroniki i Informatyki Politechniki Śląskiej w Gliwicach, Wydziałem Informatyki Politechniki Poznańskiej, Ośrodkiem Edukacji Informatycznej i Zastosowań Komputerów, a także z innymi środowiskami akademickimi, zawodowymi i oświatowymi działającymi w sprawach edukacji informatycznej.

(3) Zadaniem Organizatora jest:

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

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

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

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

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

§2 STRUKTURA ORGANIZACYJNA OLIMPIADY

(1) **Struktura organizacyjna**

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

(2) **Komitet Główny**

1. Komitet Główny jest odpowiedzialny za poziom merytoryczny i organizację zawodów. Komitet składa corocznie Organizatorowi sprawozdanie z przeprowadzonych zawodów.
2. Komitet wybiera ze swego grona Prezydium. Prezydium podejmuje decyzje w nagłych sprawach pomiędzy posiedzeniami Komitetu. W skład Prezydium wchodzą: przewodniczący, dwóch wiceprzewodniczących, sekretarz naukowy, kierownik Jury, kierownik techniczny i koordynator Olimpiady.
3. Komitet dokonuje zmian w swoim składzie za zgodą Organizatora.
4. Komitet powołuje i rozwiązuje komitety okręgowe Olimpiady.
5. Komitet:
   - opracowuje szczegółowe zasady organizacji zawodów, które są ogłaszane razem z treścią zadań zawodów I stopnia Olimpiady,
   - wybiera zadania na wszystkie zawody Olimpiady,
   - udziela wyjaśnień w sprawach dotyczących Olimpiady,
   - zatwierdza listy rankingowe oraz listy laureatów i wyróżnionych uczestników,
   - przyznaje uprawnienia i nagrody rzeczowe wyróżniającym się uczestnikom Olimpiady,
   - ustala skład reprezentacji na Międzynarodową Olimpiadę Informatyczną i inne międzynarodowe zawody informatyczne.
6. Decyzje Komitetu zapadają zwykłą większością głosów uprawnionych przy udziale przynajmniej połowy członków Komitetu. W przypadku równej liczby głosów decyduje głos przewodniczącego obrad.
7. Posiedzenia Komitetu, na których ustala się treści zadań Olimpiady, są tajne. Przewodniczący obrad może zarządzić tajność obrad także w innych uzasadnionych przypadkach.
8. W jawnych posiedzeniach Komitetu mogą brać udział przedstawiciele organizacji wspierających, jako obserwatorzy z głosem doradczym.

9. Do organizacji zawodów II stopnia w miejscowościach, których nie obejmuje żaden komitet okręgowy, Komitet powołuje komisję zawodów co najmniej miesiąc przed terminem rozpoczęcia zawodów.
10. Decyzje Komitetu we wszystkich sprawach dotyczących przebiegu i wyników zawodów są ostateczne.
11. Komitet dysponuje funduszem Olimpiady za zgodą Organizatora i za pośrednictwem koordynatora Olimpiady.
12. Komitet przyjmuje plan finansowy Olimpiady na przyszły rok na ostatnim posiedzeniu w roku poprzedzającym.
13. Komitet przyjmuje sprawozdanie finansowe z przebiegu Olimpiady na ostatnim posiedzeniu w roku, na dzień 30 listopada.
14. Komitet ma siedzibę w Ośrodku Edukacji Informatycznej i Zastosowań Komputerów w Warszawie. Ośrodek wspiera Komitet we wszystkich działaniach organizacyjnych zgodnie z Deklaracją z 8 grudnia 1993 roku.
15. Pracami Komitetu kieruje przewodniczący, a w jego zastępstwie lub z jego upoważnienia jeden z wiceprzewodniczących.
16. Przewodniczący:
    - czuwa nad całokształtem prac Komitetu,
    - zwołuje posiedzenia Komitetu,
    - przewodniczy tym posiedzeniom,
    - reprezentuje Komitet na zewnątrz,
    - rozpatruje odwołania uczestników Olimpiady osobiście lub za pośrednictwem kierownika Jury,
    - czuwa nad prawidłowością wydatków związanych z organizacją i przeprowadzeniem Olimpiady oraz zgodnością działalności Komitetu z przepisami.
17. Komitet Główny przyjął następujący tryb opracowywania zadań olimpijskich:
    - Autor zgłasza propozycję zadania, które powinno być oryginalne i nieznane, do sekretarza naukowego Olimpiady.
    - Zgłoszona propozycja zadania jest opiniowana, redagowana, opracowywana wraz z zestawem testów, a opracowania podlegają niezależnej weryfikacji. Zadanie, które uzyska negatywną opinię, może zostać odrzucone lub skierowane do ponownego opracowania.
    - Wyboru zestawu zadań na zawody dokonuje Komitet Główny, spośród zadań, które zostały opracowane i uzyskały pozytywną opinię.
    - Wszyscy uczestniczący w procesie przygotowania zadania są zobowiązani do zachowania tajemnicy do czasu jego wykorzystania w zawodach lub ostatecznego odrzucenia.
18. Kierownik Jury w porozumieniu z przewodniczącym powołuje i odwołuje członków Jury Olimpiady, które jest odpowiedzialne za opracowanie i sprawdzanie zadań.

19. Kierownik techniczny odpowiada za stronę techniczną przeprowadzenia zawodów.

(3) **Komitety okręgowe**

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

Rozdział II – Organizacja Olimpiady

§3 UCZESTNICY OLIMPIADY

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

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

(3) By wziąć udział w Olimpiadzie, Uczestnik powinien zarejestrować się w Systemie Internetowym Olimpiady, zwanym dalej SIO, o adresie `http://sio2.mimuw.edu.pl`.

(4) Uczestnicy zobowiązani są do:

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

(5) Uczestnik ma prawo do:

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

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

§4 ORGANIZACJA ZAWODÓW

(1) Zawody Olimpiady mają charakter indywidualny.

(2) **Przebieg zawodów**

1. Zawody są organizowane przez Komitet Główny przy wsparciu komitetów okręgowych.
2. Zawody są trójstopniowe.
3. **Zawody I stopnia**
   1. Zawody I stopnia mają charakter otwarty i polegają na samodzielnym rozwiązywaniu przez uczestnika zadań ustalonych dla tych zawodów oraz przekazaniu rozwiązań w podanym terminie; sposób przekazania określony jest w zasadach organizacji zawodów danej edycji Olimpiady.
   2. Zawody I stopnia są przeprowadzane zdalnie przez Internet.
   3. Liczbę uczestników kwalifikowanych do zawodów II stopnia ustala Komitet Główny i podaje ją w zasadach organizacji zawodów. Komitet Główny kwalifikuje do zawodów II stopnia odpowiednią liczbę uczestników, których rozwiązania zadań I stopnia zostaną ocenione najwyżej, spośród uczestników, którzy uzyskali co najmniej 50% punktów możliwych do zdobycia w zawodach I stopnia.
   4. Komitet Główny może zmienić podaną w zasadach liczbę uczestników zakwalifikowanych do zawodów II stopnia co najwyżej o 30%. Komitet Główny ma prawo zakwalifikować do zawodów II stopnia uczestników, którzy zdobyli mniej niż 50% ogólnej liczby punktów, w kolejności zgodnej z listą rankingową, jeśli uzna, że poziom zakwalifikowanych jest wystarczająco wysoki.
4. **Zawody II stopnia**
   1. Zawody II stopnia są organizowane przez komitety okręgowe Olimpiady lub komisje zawodów powołane przez Komitet Główny i koordynowane przez Komitet Główny.
   2. Zawody II stopnia polegają na samodzielnym rozwiązywaniu zadań przygotowanych centralnie przez Komitet Główny. Zawody te odbywają się w ciągu dwóch sesji, przeprowadzanych w różnych dniach, w warunkach kontrolowanej samodzielności.

3. Rozwiązywanie zadań w zawodach II stopnia jest poprzedzone jednodniową sesją próbną umożliwiającą zapoznanie się uczestników z warunkami organizacyjnymi i technicznymi Olimpiady.
4. W czasie trwania zawodów nie można korzystać z żadnych książek ani innych pomocy takich jak: dyski, kalkulatory, notatki itp. Nie wolno mieć w tym czasie telefonu komórkowego ani innych własnych urządzeń elektronicznych.
5. Każdy uczestnik zawodów II stopnia musi mieć ze sobą legitymację szkolną.
6. Do zawodów III stopnia zostanie zakwalifikowanych 80 uczestników, których rozwiązania zadań II stopnia zostaną ocenione najwyżej, spośród uczestników, którzy uzyskali co najmniej 50% punktów możliwych do zdobycia w zawodach II stopnia.
7. Komitet Główny może zmienić liczbę uczestników zakwalifikowanych do zawodów III stopnia co najwyżej o 30%. Komitet Główny ma prawo zakwalifikować do zawodów III stopnia uczestników zawodów II stopnia, którzy zdobyli mniej niż 50% ogólnej liczby punktów, w kolejności zgodnej z listą rankingową, jeśli uzna, że poziom zakwalifikowanych jest wystarczająco wysoki.
8. Uczestnik zawodów II stopnia zakwalifikowany do zawodów III stopnia uzyskuje tytuł finalisty Olimpiady Informatycznej.

5. **Zawody III stopnia**

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

6. **Postanowienia ogólne**
   1. Rozwiązaniem zadania zawodów I, II i III stopnia są, zgodnie z treścią zadania, dane lub program. Program powinien być napisany w języku programowania i środowisku wybranym z listy języków i środowisk ustalanej przez Komitet Główny i ogłaszanej w zasadach organizacji zawodów.
   2. Rozwiązania są oceniane automatycznie. Jeśli rozwiązaniem zadania jest program, to jest on uruchamiany na testach z przygotowanego zestawu. Podstawą oceny jest zgodność sprawdzanego programu z podaną w treści zadania specyfikacją, poprawność wygenerowanego przez program wyniku, czas działania tego programu oraz ilość wymaganej przez program pamięci. Jeśli rozwiązaniem zadania jest plik z danymi, wówczas ocenia się poprawność danych. Za każde zadanie zawodnik może zdobyć maksymalnie 100 punktów, gdzie 100 jest sumą maksymalnych liczb punktów za poszczególne testy (lub dane z wynikami) dla tego zadania. Oceną rozwiązań zawodnika jest suma punktów za poszczególne zadania. Oceny rozwiązań zawodników są podstawą utworzenia listy rankingowej zawodników po zawodach każdego stopnia.
   3. Komitet Główny zastrzega sobie prawo do opublikowania rozwiązań zawodników, którzy zostali zakwalifikowani do następnego etapu, zostali wyróżnieni lub otrzymali tytuł laureata.
   4. Komitet Główny zawiadamia uczestnika oraz dyrektora jego szkoły o zakwalifikowaniu do zawodów stopnia II i III, podając jednocześnie miejsce i termin zawodów.
   5. Dane osobowe uczestników Olimpiady będą wykorzystywane wyłącznie do celów związanych z Olimpiadą i będą chronione zgodnie z ustawą z dnia 29 sierpnia 1997 r. o ochronie danych osobowych (Dz. U. Nr 133 poz. 883 z późn. zm.).
   6. Udział w zawodach pierwszego stopnia oznacza zgodę własną, a w wypadku niepełnoletniego uczestnika zgodę jego prawnych opiekunów, na przetwarzanie danych osobowych w stopniu niezbędnym do oceniania nadesłanych prac konkursowych, kwalifikacji do zawodów następnego stopnia oraz przygotowania zestawień statystycznych, a także – w wypadku zakwalifikowania do zawodów stopnia drugiego lub finału – zgodę na umieszczenie na stronie internetowej, na liście osób zakwalifikowanych do zawodów stopnia drugiego lub finału konkursu, imienia, nazwiska, klasy oraz nazwy szkoły, do której uczęszcza zawodnik.
   7. Udział w zawodach pierwszego stopnia jest jednoznaczny z akceptacją niniejszego Regulaminu.

§5 PRZEPISY SZCZEGÓŁOWE

(1) Organizator dokona wszelkich starań, aby w miarę możliwości w danych warunkach organizować zawody w taki sposób i w takich miejscach, by nie wyklu-

czały udziału osób niepełnosprawnych. W tym celu komitety okręgowe i Komitet Główny będą dążyły do organizacji zawodów w pomieszczeniach łatwo dostępnych oraz organizacji noclegu w miejscu łatwo dostępnym.

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

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

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

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

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

§6 TRYB ODWOŁAWCZY

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

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

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

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

§7 REJESTRACJA PRZEBIEGU ZAWODÓW

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

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

Rozdział III – Uprawnienia i nagrody

§8 UPRAWNIENIA I NAGRODY

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

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

(2) Uprawnienia laureatów i finalistów określa rozporządzenie Ministra Edukacji Narodowej z dnia 30 kwietnia 2007 r. w sprawie warunków i sposobu oceniania, klasyfikowania i promowania uczniów i słuchaczy oraz przeprowadzania sprawdzianów i egzaminów w szkołach publicznych (Dz. U. 2007, nr 83, poz. 562, z późn. zm.).

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

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

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

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

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

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

Rozdział IV – Olimpiada międzynarodowa

§9 UDZIAŁ W OLIMPIADZIE MIĘDZYNARODOWEJ

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

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

Rozdział V – Postanowienia końcowe

§10 POSTANOWIENIA KOŃCOWE

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

(2) Komitet Główny rozsyła drogą elektroniczną do szkół wymienionych w §3.1 oraz kuratoriów oświaty informację o rozpoczęciu danej edycji Olimpiady.

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

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

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

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

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

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

§11 FINANSOWANIE OLIMPIADY

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

Zasady organizacji zawodów XXIII Olimpiady Informatycznej w roku szkolnym 2015/2016

§1 WSTĘP

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

§2 ORGANIZACJA OLIMPIADY

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

(2) Olimpiada Informatyczna jest trójstopniowa.

(3) W Olimpiadzie Informatycznej mogą brać indywidualnie udział uczniowie wszystkich typów szkół ponadgimnazjalnych. W Olimpiadzie mogą również uczestniczyć – za zgodą Komitetu Głównego – uczniowie szkół podstawowych i gimnazjów.

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

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

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

(7) Do zawodów II stopnia zostanie zakwalifikowanych 350 uczestników, których rozwiązania zadań I stopnia zostaną ocenione najwyżej, spośród uczestników, którzy uzyskali co najmniej 50% punktów możliwych do zdobycia w zawodach I stopnia. Do zawodów III stopnia zostanie zakwalifikowanych 80 uczestników, których rozwiązania zadań II stopnia zostaną ocenione najwyżej, spośród uczestników, którzy uzyskali co najmniej 50% punktów możliwych do zdobycia w zawodach II stopnia.

(8) Komitet Główny może zmienić podane liczby zakwalifikowanych uczestników co najwyżej o 30%. Komitet Główny ma prawo zakwalifikować do zawodów wyższego stopnia uczestników zawodów niższego stopnia, którzy zdobyli mniej niż 50% ogólnej liczby punktów, w kolejności zgodnej z listą rankingową, jeśli uzna, że poziom zakwalifikowanych jest wystarczająco wysoki.

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

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

(11) Terminarz zawodów:

- **zawody I stopnia** – 19 października – 16 listopada 2015 roku
  ogłoszenie wyników w witrynie Olimpiady – 4 grudnia 2015 roku godz. 20.00
- **zawody II stopnia** – 9–11 lutego 2016 roku
  ogłoszenie wyników w witrynie Olimpiady – 19 lutego 2016 roku godz. 20.00
- **zawody III stopnia** – 12–16 kwietnia 2016 roku

§3 ROZWIĄZANIA ZADAŃ

(1) 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.

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

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

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

(4) Za każde zadanie można zdobyć od 0 do 100 punktów. Rozwiązania oceniane są automatycznie. Jeśli rozwiązaniem zadania jest program, wówczas:

1. nadesłany program jest kompilowany i uruchamiany na pewnej liczbie grup danych testowych; każda grupa składa się z jednego lub kilku testów,
2. w przypadku, gdy wykonanie programu na danym teście nie zakończy się błędem oraz zmieści się w wyznaczonym limicie czasowym i pamięciowym, zostaje sprawdzona poprawność otrzymanej odpowiedzi,
3. w przypadku poprawnej odpowiedzi test jest zaliczany,
4. za każdą grupę, w której zostały zaliczone wszystkie testy, program otrzymuje liczbę punktów zależną od liczby punktów przypisanych do danej grupy oraz od czasu działania programu.

Jeśli rozwiązaniem zadania jest plik z danymi, wówczas ocenia się poprawność danych.

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

(6) Dane testowe oraz ostateczne wyniki sprawdzania są ujawniane po zakończeniu zawodów danego stopnia.

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

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

(9) 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.

§4 ZAWODY I STOPNIA

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

- poprzez System Internetowy Olimpiady, zwany dalej SIO, o adresie `http://sio2.mimuw.edu.pl`, do 16 listopada 2015 roku do godz. 12.00 (południe). Komitet Główny nie ponosi odpowiedzialności za brak możliwości

przekazania rozwiązań przez Internet w sytuacji nadmiernego obciążenia lub awarii SIO. Odbiór przesyłki zostanie potwierdzony przez SIO zwrotnym listem elektronicznym (prosimy o zachowanie tego listu). Brak potwierdzenia może oznaczać, że rozwiązanie nie zostało poprawnie zarejestrowane. W tym przypadku zawodnik powinien przesłać swoje rozwiązanie 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 16 listopada 2015 roku (decyduje data stempla pocztowego). Uczestnik ma obowiązek zachować dowód nadania przesyłki do czasu otrzymania wyników oceny. Nawet w przypadku wysyłania rozwiązań pocztą, każdy uczestnik musi założyć sobie konto w SIO. **Zarejestrowana nazwa użytkownika musi być zawarta w przesyłce.**

(2) Uczestnik korzystający z poczty zwykłej powinien umieścić rozwiązania wybranych przez siebie zadań za pomocą specjalnego formularza w SIO. Następnie system SIO wygeneruje dokument, który należy wydrukować i wysłać na podany adres. Dopuszczalne jest także przesłanie rozwiązań (tj. plików źródłowych lub plików z danymi) pocztą na płycie CD/DVD lub pamięci USB. W przypadku braku możliwości odczytania nośnika z rozwiązaniami, nieodczytane rozwiązania nie będą brane pod uwagę.

(3) **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.

(4) 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.

(5) 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.

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

(7) Rozwiązanie każdego zadania można zgłosić w SIO co najwyżej 10 razy. Spośród tych zgłoszeń oceniane jest jedynie najpóźniejsze poprawnie kompilujące

się rozwiązanie. Po wyczerpaniu tego limitu kolejne rozwiązanie może zostać zgłoszone już tylko zwykłą pocztą.

(8) 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.

(9) W SIO znajduje się także dział *Forum* umożliwiający prowadzenie dyskusji między zawodnikami. W dziale tym niedozwolona jest dyskusja na temat metod rozwiązania zadań zawodów I stopnia i złożoności obliczeniowych rozwiązań, pod rygorem dyskwalifikacji.

(10) Poprzez SIO udostępniane są narzędzia do sprawdzania rozwiązań pod względem formalnym. Nie są one jednak dostępne w przypadku rozwiązań przesłanych za pomocą formularza do wysyłki pocztą zwykłą. Szczegóły dotyczące sposobu postępowania będą dokładnie podane w SIO.

(11) Od piątku 27 listopada 2015 roku poprzez SIO każdy zawodnik będzie mógł zapoznać się ze wstępną oceną swojej pracy.

(12) Do środy 2 grudnia 2015 roku (włącznie) poprzez SIO każdy zawodnik będzie mógł zgłaszać uwagi do wstępnej oceny swoich rozwiązań. Reklamacji nie podlega jednak dobór testów, limitów czasowych, kompilatorów i sposobu oceny.

(13) Reklamacje złożone po 2 grudnia 2015 roku nie będą rozpatrywane.

§5 ZAWODY II I III STOPNIA

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

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

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

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

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

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

§6 UPRAWNIENIA I NAGRODY

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

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

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

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

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

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

(7) Komitet Główny ustala skład reprezentacji Polski na XXVIII Międzynarodową Olimpiadę Informatyczną w 2016 roku oraz inne zawody międzynarodowe na podstawie wyników Olimpiady oraz regulaminów Międzynarodowej Olimpiady i tych zawodów.

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

(9) Wyznaczeni przez Komitet Główny reprezentanci Polski na olimpiady międzynarodowe zostaną zaproszeni do nieodpłatnego udziału w XVII Obozie Naukowo-Treningowym im. Antoniego Kreczmara, który odbędzie się w czasie wakacji 2016 roku. Do nieodpłatnego udziału w Obozie Komitet Główny może zaprosić także innych finalistów, którzy nie są w ostatniej programowo klasie swojej szkoły, w zależności od uzyskanych wyników.

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

§7 PRZEPISY KOŃCOWE

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

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

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

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

Zasady organizacji zawodów II i III stopnia XXIII Olimpiady Informatycznej

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

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

(3) Każdy uczestnik zawodów II i III stopnia musi mieć ze sobą legitymację szkolną.

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

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

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

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

(8) W przypadku stwierdzenia awarii sprzętu w czasie zawodów termin zakończenia pracy przez uczestnika zostaje przedłużony o tyle, ile trwało usunięcie awarii. Awarie sprzętu należy zgłaszać dyżurującym członkom Komisji Regulaminowej.

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

(10) Podczas każdej sesji:

1. W trakcie pierwszych 60 minut nie wolno opuszczać przydzielonej sali zawodów. Zawodnicy spóźnieni więcej niż godzinę nie będą w tym dniu dopuszczeni do zawodów.
2. W trakcie pierwszych 90 minut każdej sesji uczestnik może zadawać pytania dotyczące treści zadań, w ustalony przez Jury sposób, na które otrzymuje

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

3. W SIO umieszczane będą publiczne odpowiedzi na pytania zawodników. Odpowiedzi te mogą zawierać ważne informacje dotyczące toczących się zawodów, więc wszyscy uczestnicy zawodów proszeni są o regularne zapoznawanie się z ukazującymi się odpowiedziami.
4. Jakikolwiek inny sposób komunikowania się z członkami Jury co do treści i sposobów rozwiązywania zadań jest niedopuszczalny.
5. Komunikowanie się z innymi uczestnikami Olimpiady (np. ustnie, telefonicznie lub poprzez sieć) w czasie przeznaczonym na rozwiązywanie zadań jest zabronione pod rygorem dyskwalifikacji.
6. Każdy zawodnik ma prawo drukować wyniki swojej pracy w sposób opisany w Ustaleniach technicznych.
7. Każdy zawodnik powinien umieścić ostateczne rozwiązania zadań w SIO, za pomocą przeglądarki lub za pomocą skryptu do wysyłania rozwiązań `submit`. Skrypt `submit` działa także w przypadku awarii sieci, wówczas rozwiązanie zostaje automatycznie dostarczone do SIO, gdy komputer odzyska łączność z siecią. Tylko zgłoszone w podany sposób rozwiązania zostaną ocenione.
8. Po zgłoszeniu rozwiązania każdego z zadań SIO dokona wstępnego sprawdzenia i udostępni jego wyniki zawodnikowi. Wstępne sprawdzenie polega na uruchomieniu programu zawodnika na testach przykładowych (wyniki sprawdzenia tych testów nie liczą się do końcowej klasyfikacji). Te same testy przykładowe są używane do wstępnego sprawdzenia za pomocą skryptu do weryfikacji rozwiązań na komputerze zawodnika (skryptu „ocen").
9. Rozwiązanie każdego zadania można zgłosić co najwyżej 10 razy.
10. Podczas zawodów III stopnia zawodnicy będą mogli poznać wynik punktowy swoich zgłoszeń zaraz po tym, jak ich programy zostaną ocenione przez system. Opcja ta może nie być dostępna w przypadku dużego obciążenia systemu, w szczególności pod sam koniec zawodów.

(11) Podczas zawodów II stopnia oceniane jest jedynie najpóźniejsze poprawnie kompilujące się rozwiązanie. Podczas zawodów III stopnia wynikiem zawodnika dla każdego zadania jest maksimum z wyników punktowych jego zgłoszeń dla tego zadania.

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

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

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

(15) Od czwartku 11 lutego 2016 roku od godz. 20.00 do poniedziałku 15 lutego 2016 roku do godz. 20.00 poprzez SIO każdy zawodnik będzie mógł zapoznać się z pełną oceną swoich rozwiązań z zawodów II stopnia i zgłaszać uwagi do tej oceny. Reklamacji nie podlega jednak dobór testów, limitów czasowych, kompilatorów i sposobu oceny.

Zawody I stopnia

opracowania zadań

Michał Włodarczyk
Treść zadania, Opracowanie

Wojciech Nadara
Program

Dostępna pamięć: 256 MB. *OI, etap I, 19.10–16.11.2015*

Hydrorozgrywka

Ekipy hydrauliczne Bajtazara i Bajtoniego wygrały przetarg na doprowadzenie wody do Bajtołów Dolnych. Sieć drogowa w tym mieście składa się z n skrzyżowań połączonych m odcinkami dróg. Z każdego skrzyżowania można dojechać do każdego innego skrzyżowania za pomocą sieci dróg. Pod każdym odcinkiem drogi należy zakopać rurę wodociągową.

Aby urozmaicić sobie pracę, Bajtazar i Bajtoni postanowili zagrać w grę. Na początek ekipy obu bohaterów stają na jednym ze skrzyżowań. Przez całą grę obie ekipy będą podążać razem. Gracze wykonują ruchy na przemian, począwszy od Bajtazara. W swoim ruchu gracz wskazuje swojej ekipie odcinek drogi (pod którym jeszcze nie ma rury) wychodzący ze skrzyżowania, na którym znajdują się obie ekipy. Ekipa gracza zakopuje rurę pod tym odcinkiem drogi i następnie obie ekipy przemieszczają się do drugiego ze skrzyżowań, które łączy ten odcinek.

Gracz, który nie może wykonać ruchu, przegrywa i za karę jego ekipa musi zakopać rury pod pozostałymi odcinkami dróg. Bajtazar zastanawia się, od którego skrzyżowania może zacząć się gra, aby był w stanie wygrać niezależnie od ruchów Bajtoniego. Poprosił Cię o pomoc w ustaleniu listy takich skrzyżowań. Dodatkowo zauważył, że sieć drogowa w Bajtołach ma ciekawą własność: **wyjeżdżając ze środka dowolnego odcinka drogi, na dokładnie jeden sposób możemy zrobić „pętlę" i wrócić do punktu wyjścia, jeśli nigdy nie zawracamy i nie odwiedzamy żadnego skrzyżowania dwukrotnie**.

Wejście

Pierwszy wiersz standardowego wejścia zawiera dwie liczby całkowite n i m oddzielone pojedynczym odstępem, oznaczające liczbę skrzyżowań i liczbę odcinków dróg. Skrzyżowania numerujemy liczbami od 1 do n. Kolejne m wierszy opisuje sieć drogową: każdy z nich zawiera dwie liczby całkowite a, b ($1 \leqslant a, b \leqslant n$, $a \neq b$) oddzielone pojedynczym odstępem, oznaczające, że skrzyżowania o numerach a i b są połączone odcinkiem drogi. Możesz założyć, że żadne dwa skrzyżowania nie są połączone więcej niż jednym odcinkiem drogi.

Wyjście

Na standardowe wyjście należy wypisać dokładnie n wierszy: i-ty z nich ma zawierać liczbę 1, jeśli Bajtazar może wygrać, gdy gra zacznie się ze skrzyżowania numer i; w przeciwnym wypadku ma zawierać liczbę 2.

Przykład

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

Testy „ocen”:

`1ocen`: *$n = 9$, $m = 12$, sieć drogowa składa się z czterech „pętli”: 1–2–3–1, 1–4–5–1, 1–6–7–1 oraz 1–8–9–1.*

`2ocen`: *$n = 998$, $m = 999$, dla każdego j takiego, że $1 \leqslant j < n$, istnieje odcinek drogi pomiędzy j-tym i $(j + 1)$-wszym skrzyżowaniem; istnieją również odcinki drogi łączące skrzyżowania 1 z 499 oraz 499 z 998.*

`3ocen`: *$n = 500\,000$, $m = 500\,000$, sieć dróg tworzy jedną „pętlę”.*

Ocenianie

Zestaw testów dzieli się na następujące podzadania. Testy do każdego podzadania składają się z jednej lub większej liczby osobnych grup testów.

Podzadanie	Warunki	Liczba punktów
1	$3 \leqslant n, m \leqslant 20$	*21*
2	$3 \leqslant n, m \leqslant 1000$	*39*
3	$3 \leqslant n, m \leqslant 500\,000$	*40*

Rozwiązanie

Hydrorozgrywka to jedno z tych zadań, które trafiają na I etap Olimpiady głównie dlatego, że wymyślenie rozwiązania i ominięcie wszystkich pułapek w implementacji w ciągu zaledwie pięciu godzin wymagałoby nadludzkich umiejętności. Z drugiej strony część doświadczonych zawodników umie uporać się ze standardowymi zadaniami z I etapu w kilka dni i wielką stratą byłoby zmarnować ich chęci do spędzenia tygodnia walki z bardziej wymagającym przeciwnikiem. Autor podpisuje się pod tezą, że najlepszym sposobem treningu w takich dziedzinach jak algorytmika jest podejmowanie *nieco za trudnych* wyzwań.

Na początek zinterpretujmy grę w układanie rur w języku teorii grafów. Zaczynając od ustalonego wierzchołka v_0, gracze na przemian wybierają krawędzie tak, aby każda kolejna miała wspólny wierzchołek z poprzednią. Każda krawędź może zostać wybrana co najwyżej raz. Innymi słowy, gracze w trakcie rozgrywki budują *marszrutę*

w grafie. Krawędzie na marszrucie nie powtarzają się, jednak ten sam wierzchołek może na niej wystąpić wielokrotnie. Gracz, który nie może wykonać poprawnego ruchu, przegrywa.

Tak zdefiniowana gra jest skończona, ponieważ liczba ruchów jest ograniczona przez liczbę krawędzi w grafie. Liczba stanów tej gry jest wykładnicza ze względu na liczbę krawędzi. Stany w grze zadane są przez marszruty zaczynające się w wierzchołku v_0, natomiast każde przejście pomiędzy stanami odpowiada wydłużeniu marszruty o jedną krawędź. Łatwo zauważyć, że taka gra jest *zdeterminowana*, tzn. dla każdego stanu jeden z graczy może doprowadzić do zwycięstwa, niezależnie od tego, jakie ruchy będzie wykonywał jego rywal. Mówimy, że gracz X posiada *strategię wygrywającą*, jeśli powyższy warunek zachodzi dla X w stanie początkowym.

Najprostszy algorytm obliczający strategie wygrywające przegląda cały graf stanów gry i klasyfikuje stany na wygrywające i przegrywające. Jest to zazwyczaj mało praktyczne podejście z powodu ogromnej liczby stanów do przeanalizowania. Częściowe rozwiązanie tego problemu opiera się na obserwacji, że niektóre stany gry są równoważne. Zauważmy, że informacja o tym, które ruchy będzie można wykonać w przyszłości, jest w całości zakodowana przez zbiór dotąd wykorzystanych krawędzi oraz aktualny końcowy wierzchołek marszruty. Możemy zatem rozważać stany opisane przez podzbiór krawędzi grafu i jeden wierzchołek. Aby sprawdzić, czy pierwszy gracz ma strategię wygrywającą w grze rozpoczynającej się w wierzchołku v_0, wystarczy sprawdzić wartość obliczoną dla stanu $(\emptyset, v_0)$. Liczba stanów zostaje w ten sposób ograniczona do $O(n2^m)$ i tyle też wynosi złożoność pamięciowa naiwnego algorytmu. Rozwiązanie oparte na tym podejściu (zaimplementowane w pliku `hyds1.cpp`) działa w czasie $O(m2^m)$. Na zawodach było ono warte 21 punktów. Nie korzysta ono jednak w żaden sposób ze specjalnej struktury grafu.

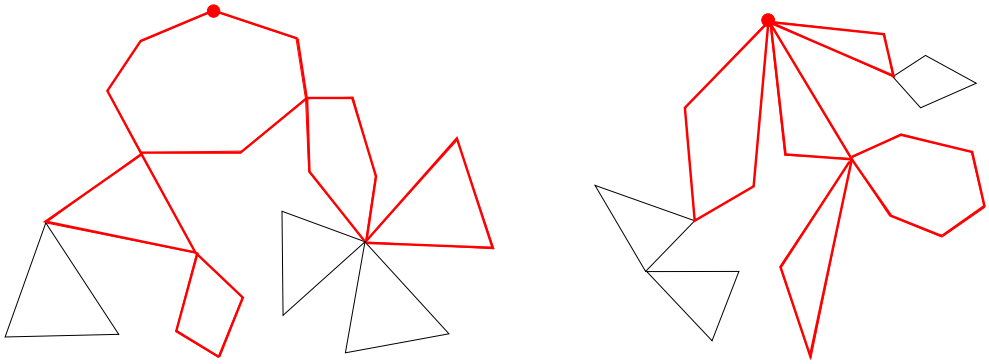
Cykle, marszruty i kaktusy

Poza gwarancją spójności grafu w treści zadania pojawia się pozornie niewiele mówiący warunek: *wyjeżdżając ze środka dowolnego odcinka drogi, na dokładnie jeden sposób możemy zrobić „pętlę" i wrócić do punktu wyjścia, jeśli nigdy nie zawracamy i nie odwiedzamy żadnego skrzyżowania dwukrotnie.* Równoważnie możemy napisać, że każda krawędź leży na dokładnie jednym *cyklu prostym*, gdzie cykl prosty to cykl, który przechodzi przez każdy wierzchołek co najwyżej jednokrotnie. Jako że często będziemy używać tego pojęcia, zamiast *cykl prosty* pisać będziemy po prostu *cykl*.

Grafy spełniające warunek z zadania będziemy nazywać *kaktusami* ze względu na podobieństwo ich graficznych reprezentacji do krzewu opuncji. Jako że rysunki potrafią czasem prowadzić do mylnych intuicji, udowodnijmy formalnie kilka własności kaktusów.

Po pierwsze, jeśli marszruta prowadzi po cyklu C, następnie opuszcza go w wierzchołku v_i, po czym wraca na C, wchodząc do wierzchołka v_j, to $v_i = v_j$.

Lemat 1. Rozważmy marszrutę $(v_i)_{i=0}^{k}$ w kaktusie. Ustalmy dowolny cykl prosty C. Jeśli $v_i \in C$, $v_{i+1} \notin C$, a v_j jest następnym po v_i wierzchołkiem należącym do C, to $v_i = v_j$.



Rys. 1: Przykłady kaktusów z wyróżnioną potencjalną marszrutą gry.

Dowód: Przypuśćmy $v_i \neq v_j$. Na podstawie cyklu C konstruujemy cykl prosty C', zastępując odcinek C pomiędzy v_i oraz v_j przez ścieżkę $v_i, v_{i+1}, \ldots, v_j$ z usuniętymi pętlami. Otrzymujemy dwa różne cykle proste C i C', które mają co najmniej jedną wspólną krawędź. Dla tej krawędzi nie jest spełniony warunek z zadania, który definiuje kaktus. ■

Lemat 2. Cykle proste kaktusa są krawędziowo rozłączne.

Dowód: Przypuśćmy, że dwa różne cykle proste C_1, C_2 posiadają wspólną krawędź. Zatem istnieje ścieżka łącząca dwa wierzchołki v_i i v_j należące do cyklu C_1, składająca się z krawędzi cyklu C_2, które nie należą do C_1. Gdyby $v_i = v_j$, to ponieważ cykl C_2 jest prosty, ścieżka ta przebiegałaby po całym cyklu C_2. Wówczas jednak C_2 nie miałby krawędzi wspólnych z C_1. Z kolei gdy $v_i \neq v_j$, otrzymujemy ścieżkę przeczącą tezie lematu 1. Zatem w obydwóch przypadkach otrzymujemy sprzeczność. ■

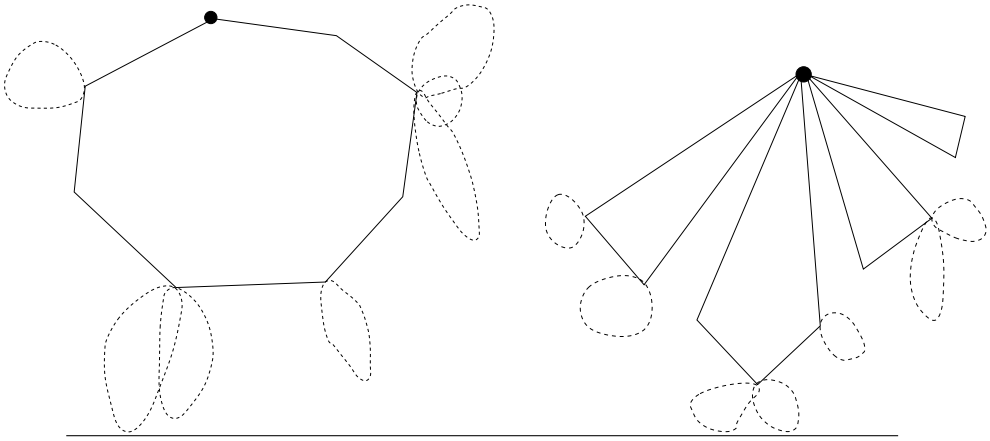
Lemat 3. Wszystkie wierzchołki kaktusa mają parzysty stopień.

Dowód: Indukcja po liczbie cykli prostych w grafie. Kaktus składający się tylko z jednego cyklu oczywiście spełnia tezę lematu. Przypuśćmy zatem, że spełniają ją kaktusy o liczbie cykli mniejszej od k, i rozważmy kaktus o k cyklach. Usuńmy z niego krawędzie dowolnego cyklu. W ten sposób stopnie niektórych wierzchołków zmniejszą się o 2, co nie wpływa na ich parzystość.

Liczba cykli przechodzących przez dowolną zachowaną krawędź na pewno nie wzrosła, a istniejący cykl nie posiadał części wspólnej z usuniętym cyklem na mocy lematu 2. Powstały graf, choć potencjalnie niespójny, nadal jest kaktusem i, z założenia indukcyjnego, wszystkie jego wierzchołki mają parzysty stopień. Zatem po przywróceniu usuniętego cyklu wszystkie wierzchołki mają nadal parzysty stopień. ■

Lemat 4. Każda rozgrywka na kaktusie kończy się w wierzchołku początkowym (patrz rys. 1).

Dowód: Oznaczmy wierzchołek początkowy przez v_0. Niech v będzie wierzchołkiem, po dojściu do którego gracz nie może wykonać ruchu. Jeśli $v \neq v_0$, to każde przejście przez wierzchołek v „zużywa” dwie krawędzie incydentne z v. Z lematu 3 wiemy, że



Rys. 2: Po lewej: kaktus prosty z zaznaczonymi przerywaną linią wewnętrznymi pękami. Po prawej: pęk składający się z czterech kaktusów prostych.

stopień v jest parzysty. Zatem tuż po wejściu do wierzchołka v mamy do wyboru nieparzystą (a więc niezerową) liczbę krawędzi incydentnych z v. ■

Lematy 1 oraz 4 charakteryzują marszruty, jakie mogą pojawić się w trakcie gry. Idąc po cyklu zawierającym v_0, gracze mogą zdecydować, czy przejść do innego cyklu. Tam rozpoczyna się wewnętrzna rozgrywka, która (z lematu 1) skończy się w tym samym punkcie. Jedyną istotną informację stanowi to, który z graczy będzie wykonywał ruch po powrocie na pierwotny cykl.

Jak się dobrać do kaktusa?

Aby móc wykorzystać specjalną strukturę grafu, musimy umieć reprezentować kaktusy w wygodny sposób; patrz rys. 2.

Definicja 1. Kaktusem prostym ukorzenionym w wierzchołku v_0 nazwiemy kaktus z wyróżnionym wierzchołkiem v_0 (korzeniem) o stopniu 2.

Definicja 2. Pękiem nazwiemy rodzinę kaktusów prostych ukorzenionych w tym samym wierzchołku v_0. Kaktusy tworzące pęk są rozłączne poza posiadaniem wspólnego korzenia. W szczególności, pęk może składać się z jednego kaktusa prostego.

Każdy kaktus prosty możemy reprezentować rekurencyjnie jako cykl przechodzący przez v_0, zawierający informacje o pękach ukorzenionych w wierzchołkach cyklu. W ten sposób otrzymujemy drzewiastą strukturę danych dla kaktusa ukorzenionego w konkretnym wierzchołku. Poniżej przedstawiamy pseudokod zmodyfikowanej procedury *DFS* (*Depth First Search*), tłumaczącej standardową reprezentację kaktusa przy pomocy list sąsiedztwa na postać rekurencyjną. Jako argumenty przekazywane są numer analizowanego i poprzednio odwiedzonego wierzchołka, zwanego *rodzicem*. W tablicy dynamicznej *stack* trzymamy stos odwiedzonych wierzchołków i kiedy trafimy do wierzchołka znajdującego się już na stosie, zapamiętujemy znaleziony cykl.

Lista *cycles*[v] zawiera pęk kaktusów ukorzenionych w v, natomiast w tablicy *belong* zapamiętujemy dla każdego wierzchołka, przez który cykl można do niego trafić, idąc od korzenia. Nie używamy tej informacji bezpośrednio w omówieniu, jednak jest ona przydatna w implementacji.

```
 1: function DFS(v, parent)
 2: begin
 3:   stack.push(v);
 4:   onStack[v] := true;
 5:   visited[v] := true;
 6:   for w ∈ neighbors[v] do begin
 7:     if onStack[w] and (w ≠ parent) then begin
 8:       C := new Cycle;
 9:       C.insert(w);
10:       i := stack.size() − 1;
11:       while stack[i] ≠ w do begin
12:         C.insert(stack[i]);
13:         belong[stack[i]] := C;
14:         i := i − 1;
15:       end
16:       cycles[w].insert(C);
17:     end
18:     if not visited[w] then
19:       DFS(w, v);
20:   end
21:   stack.pop(v);
22:   onStack[v] := false;
23: end
```

Należy zaznaczyć, że korzeń DFS-a, tzn. wierzchołek, od którego zaczęliśmy przeszukiwanie grafu, pozostanie z pustym polem w tablicy *belong* i należy rozważyć go jako przypadek szczególny. Można tego uniknąć, reprezentując graf jako kaktus prosty, jeśli rozpoczniemy przeszukiwanie grafu w wierzchołku stopnia 2 (zachęcamy Czytelnika do udowodnienia, że taki wierzchołek istnieje w każdym kaktusie) i przypiszemy go do jedynego cyklu, który z niego wyrasta.

Klasyfikacja kaktusów

W wielu teoriach matematycznych centralne miejsce zajmują twierdzenia o klasyfikacji, rozstrzygające, które obiekty możemy traktować jako równoważne, a pomiędzy którymi zachodzą istotne strukturalne różnice. Takie twierdzenia przydadzą się nam, aby uprościć opis rozwiązania wzorcowego.

Gracza rozpoczynającego rozgrywkę w interesującym nas podkaktusie nazwiemy graczem I, a jego rywala – graczem II. Pęki podzielimy na wygrywające i przegrywające, w zależności od tego, czy gracz I może zagwarantować sobie zebranie ostatniej krawędzi w grze ograniczonej do rozważanego pęku. W przypadku kaktusów prostych sytuacja jest (wbrew nazwie) bardziej skomplikowana, ponieważ niekiedy graczowi I

opłaca się zmusić rywala do wzięcia ostatniej krawędzi, aby samemu kontynuować grę w korzystnym dla siebie stanie. Dlatego wprowadzimy dwa rodzaje gier na kaktusach prostych: rodzaj A, w którym wygrywa gracz zabierający ostatnią krawędź, oraz rodzaj B, w którym taki gracz przegrywa. W zależności od istnienia strategii wygrywających w tych grach, kaktus prosty może należeć do jednego z czterech typów: **00**, **10**, **01**, **11**, gdzie kolejne bity kodują odpowiednio wynik gry A oraz B – naturalnie 1 oznacza zwycięstwo[1]. Przykładowo, typ **10** oznacza, że w grze rodzaju A gracz I ma strategię wygrywającą, a w grze rodzaju B to gracz II ma strategię wygrywającą. Zauważmy, że dla pęków rozważamy tylko grę rodzaju A i takiej gry dotyczy oryginalne pytanie z zadania.

Twierdzenie 1. *Rozważmy kaktus prosty ukorzeniony w v_0. Niech C będzie cyklem zawierającym v_0. Wówczas kaktus jest typu:*

10, *jeśli na cyklu C nie ma wygrywających pęków oraz C jest nieparzystej długości,*

01, *jeśli na cyklu C nie ma wygrywających pęków oraz C jest parzystej długości,*

11, *jeśli wychodząc z v_0 w pewną stronę, dojdziemy do wygrywającego pęku po parzystej liczbie kroków,*

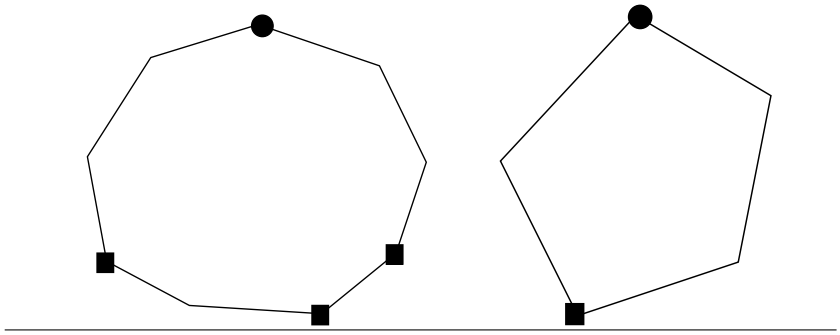
00, *jeśli wychodząc z v_0 w dowolną stronę, dojdziemy do wygrywającego pęku po nieparzystej liczbie kroków.*

Dowód: Załóżmy wpierw, że na cyklu C nie znajduje się żaden wygrywający pęk. W przypadku, gdy C jest nieparzystej długości, zaznaczmy na C krawędzie, których odległość od v_0 jest parzysta. W ten sposób zaznaczone zostaną obydwie krawędzie incydentne z v_0, a poza nimi co druga krawędź na C. Gracz I może zagwarantować sobie wzięcie wszystkich zaznaczonych krawędzi. Gracz II może potencjalnie starać się mu przeszkodzić i zamiast zebrać niezaznaczoną krawędź na C, rozpocząć wewnętrzną rozgrywkę na pęku ukorzenionym w pewnym wierzchołku v. Aby zmusić gracza I do wybrania niezaznaczonej krawędzi, gracz II musi doprowadzić do sytuacji, w której gracz I będzie miał ruch w v i wszystkie kaktusy proste wyrastające z v będą już odwiedzone. Jednak aby do tego doprowadzić, gracz II musiałby mieć strategię wygrywającą w pęku ukorzenionym w v.

Z drugiej strony, gdyby gracz I chciał zmienić kolejność ruchów i zmusić gracza II do wzięcia ostatniej krawędzi, potrzebowałby strategii wygrywającej na jednym z pęków. Zatem również gracz II jest w stanie zagwarantować, że gracz I wykona ostatni ruch na cyklu C. Analogicznie, jeżeli cykl C jest parzystej długości, gracz II potrafi zagwarantować sobie jedynie wzięcie ostatniej krawędzi, natomiast gracz I – przedostatniej. W ten sposób udało nam się sklasyfikować dwa pierwsze przypadki.

W dwóch pozostałych przypadkach w co najmniej jednym wierzchołku cyklu C ukorzeniony jest wygrywający pęk. Gracz, który jako pierwszy będzie wykonywał ruch w takim wierzchołku – oznaczmy ten wierzchołek przez v – jest w stanie zdeterminować dalszy przebieg rozgrywki na cyklu. Jako że jedna z krawędzi idących do v jest wykorzystana, o wyniku rozgrywki decyduje, który z graczy opuści wierzchołek v.

[1] W filmowym omówieniu zadania pojawiły się alternatywne definicje typów kaktusów. Kaktusy typu 01, 10 to takie, w których gracz zabierający ostatnią krawędź jest zdeterminowany, zaś typy 11, 00 odpowiadają kaktusom, w których o finale rozgrywki decyduje odpowiednio gracz I lub II. Po przemyśleniu sprawy, autor uważa, że obecna notacja jest prostsza i bardziej precyzyjna.



Rys. 3: Ilustracja do twierdzenia 1: korzeń zaznaczono kółkiem, a wierzchołki zawierające wygrywające pęki – kwadratami. Po lewej stronie kaktus typu **00**, po prawej **11**.

Jeśli ruch ten prowadzi do stanu ze strategią wygrywającą w grze A bądź B, gracz wykonujący ruch może ten ruch wykonać od razu. W przeciwnym przypadku, może rozpocząć grę na wygrywającym pęku i zagwarantować sobie wzięcie ostatniej krawędzi. Drugi gracz może próbować opuścić grę na pęku przed odwiedzeniem wszystkich wewnętrznych kaktusów, lecz w ten sposób również zbierze niechcianą krawędź.

Załóżmy, że wychodząc z v_0 w pewną stronę, można w parzystej liczbie kroków dojść do korzenia wygrywającego pęku. Oznaczmy ten korzeń przez v. Gracz I rozpocznie grę w tym właśnie kierunku i zagwarantuje sobie wzięcie krawędzi o nieparzystych indeksach, co zapewni mu ruch w wierzchołku v i zwycięstwo w obu rodzajach gry. Jeżeli zaś idąc w obu kierunkach, pierwszy wygrywający pęk znajdziemy po nieparzystej liczbie kroków, to niezależnie od początkowego ruchu, gracz II zagwarantuje sobie wzięcie krawędzi o parzystych indeksach i to on będzie miał ruch przy wygrywającym pęku. Powyższa obserwacja kończy analizę ostatnich dwóch przypadków. ■

Wynik gry na pęku zależy tylko od liczby kaktusów poszczególnych typów. Czwórkę liczb $(x_{00}, x_{10}, x_{01}, x_{11})$ opisującą, ile kaktusów typu odpowiednio **00**, **10**, **01**, **11** znajduje się w pęku, nazwiemy *konfiguracją*. Naturalnym pomysłem jest wykorzystanie programowania dynamicznego do pogrupowania konfiguracji na wygrywające i przegrywające. Aby rozstrzygnąć, jakiego typu jest dana konfiguracja, można rozważyć wszystkie ruchy gracza I, polegające na wyborze początkowego kaktusa i rodzaju wewnętrznej gry (A lub B), i przyjrzeć się wcześniej obliczonym wynikom gier dla konfiguracji z pewnym x_i pomniejszonym o 1. Strategią wygrywającą może być rozpoczęcie gry A w kaktusie gwarantującym zwycięstwo, jeśli prowadzi to do przegrywającej konfiguracji, lub podjęcie gry B, kiedy otrzymana konfiguracja zapewnia zwycięstwo. Niestety liczba konfiguracji jest rzędu $\Theta(n^4)$, co wymusza na nas znalezienie sprytniejszego sposobu klasyfikacji pęków. Takiego sposobu dostarcza poniższe kryterium.

Twierdzenie 2. *Pęk jest wygrywający wtedy i tylko wtedy, gdy zawiera kaktus prosty typu* **11** *lub liczba kaktusów typu* **10** *jest nieparzysta.*

Dowód: Indukcja po liczbie kaktusów w pęku. Konfiguracja (0,0,0,0) jest zgodnie z definicją przegrywająca, ponieważ gracz I nie może zagwarantować sobie zabrania

ostatniej krawędzi. Sytuacja jest równoważna wzięciu ostatniej krawędzi przez gracza II, ponieważ wewnętrzna gra na pęku jest zakończona i ruch należy do gracza I. Załóżmy teraz, że rozważamy konfigurację $(x_{00}, x_{10}, x_{01}, x_{11})$ i teza indukcyjna zachodzi dla wszystkich pęków składających się z mniej niż $x_{00}+x_{10}+x_{01}+x_{11}$ kaktusów.

Jeżeli pęk zawiera kaktus typu **11**, to gracz I może na nim zagrać zarówno tak, by wziąć ostatnią krawędź, jak i tak, by ostatnią krawędź wziął gracz II. W zależności od tego, czy konfiguracja $(x_{00}, x_{10}, x_{01}, x_{11}-1)$ jest wygrywająca, czy przegrywająca, wybierze on odpowiednią strategię i w ten sposób zapewni sobie zwycięstwo. Jeśli zaś $x_{11}=0$, ale x_{10} jest nieparzyste, strategią gracza I jest takie granie na kaktusie typu **10**, by wziąć ostatnią krawędź. Wówczas gracz II będzie zmuszony wykonać ruch w konfiguracji $(x_{00}, x_{10}-1, x_{01}, 0)$. Jednak zgodnie z założeniem indukcyjnym taka konfiguracja jest przegrywająca.

Pozostaje nam pokazać, że pozostałe konfiguracje są przegrywające. Przypuśćmy $x_{11}=0$ oraz $2 \mid x_{10}$. Jeśli gra rozpocznie się w kaktusie typu **10**, gracz II zadba o to, by gracz I wziął ostatnią krawędź. Prowadzi to do konfiguracji, w której x_{10} jest nieparzyste i rozpoczyna gracz II, co gwarantuje mu zwycięstwo. Jeśli zaś wybrany zostanie kaktus typu **00** lub **01**, wówczas gracz II może sprawić, że weźmie on ostatnią krawędź, przez co gracz I znajdzie się w konfiguracji $(x_{00}-1, x_{10}, x_{01}, 0)$ lub, odpowiednio, $(x_{00}, x_{10}, x_{01}-1, 0)$, która zgodnie z tezą indukcyjną jest przegrywająca. ■

Uzbrojeni w reprezentację cyklową grafu, możemy z łatwością rekurencyjnie sklasyfikować wszystkie pęki i kaktusy proste. Twierdzenia 1 i 2 pozwalają nam wykonać wszystkie obliczenia w złożoności obliczeniowej $O(m)$ i odczytać, kto posiada strategię wygrywającą w grze rozpoczynającej się w korzeniu kaktusa. To jednak nie koniec zadania – wszak w treści jesteśmy proszeni o rozstrzygnięcie wyniku gry dla wszystkich możliwych wierzchołków początkowych. Możemy obliczyć całą dekompozycję cyklową dla wszystkich możliwych wierzchołków w grafie, lecz takie rozwiązanie obarczone jest złożonością obliczeniową[2] $O(nm)$. Za zaimplementowanie tego algorytmu można było otrzymać 60 punktów.

Rozwiązanie wzorcowe

Kluczem do przyspieszenia obliczeń jest dogłębne wykorzystanie raz obliczonej dekompozycji cyklowej. Przypomnijmy, że ma ona rekurencyjną strukturę drzewiastą. Kaktus przedstawiamy jako cykl, w którego wierzchołkach zaczepione są inne kaktusy proste (być może wiele w tym samym wierzchołku). Każdy taki kaktus prosty będziemy nazywać *podkaktusem* przez analogię do poddrzewa. Za pomocą twierdzeń 1 i 2 umiemy rozstrzygnąć, jak potoczy się gra rozpoczynająca się w korzeniu v pewnego podkaktusa, o ile znamy typy podkaktusów poniżej v. Ostatecznie w ten sposób otrzymujemy wynik dla wierzchołka startowego v_0, w którym ukorzeniony jest cały kaktus.

Obliczenie wyniku gry rozpoczynającej się gdzie indziej niż w v_0 jest nieco trudniejsze, bo musimy wziąć pod uwagę kaktus, który zawiera v_0. Ustalmy wierzchołek

[2] W istocie dla każdego kaktusa zachodzi $m \leqslant \frac{3}{2}n$ (zachęcamy Czytelnika do przekonania się o tym osobiście), zatem złożoność obliczeniową tego rozwiązania można oszacować jako $O(n^2)$.

$v \neq v_0$ będący korzeniem pewnego podkaktusa. Aby rozstrzygnąć, jak potoczy się gra rozpoczynająca się w v, musimy znać typy wszystkich kaktusów ukorzenionych w v, jak również typ „nadkaktusa" v. Mówiąc formalnie, *nadkaktus* v powstaje przez usunięcie podkaktusów ukorzenionych w v (z wyjątkiem samego wierzchołka v) i następnie ukorzenienie powstałego grafu właśnie w v.

Potrzebujemy zatem wykonać dwa dodatkowe kroki: ustalić typy wszystkich nadkaktusów oraz wykorzystać te informacje do obliczenia ostatecznej odpowiedzi. Algorytm zaprezentowany został na poniższym pseudokodzie. Procedura *analyze* zostaje wywołana rekurencyjnie dla każdego podkaktusa podanego grafu. Pierwszy argument to cykl C zawierający korzeń podkaktusa (w implementacji może to być numer lub wskaźnik do obiektu). Cykl reprezentowany jest jako tablica wierzchołków, gdzie $C[0]$ to korzeń podkaktusa. Drugi argument procedury określa typ nadkaktusa (**00**, **10**, **01** lub **11**). W pierwszym wywołaniu C to cykl zaczynający się od v_0, a $overcactusType = \mathbf{00}$.

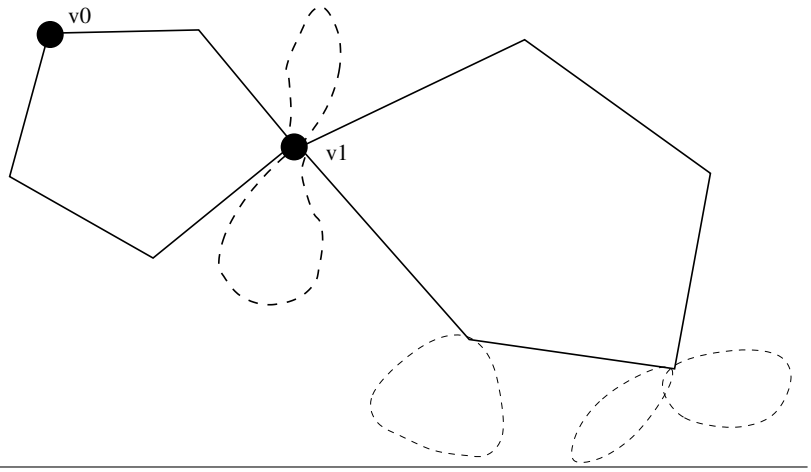
```
 1: function analyze(C, overcactusType)
 2: begin
 3:   externalGame[C[0]] := getGameResult(cycles[C[0]] \ C, overcactusType);
 4:   for i := 1 to C.size() − 1 do
 5:     externalGame[C[i]] := getGameResult(cycles[C[i]], 00);
 6:   cycleType := getCycleTypes(C, externalGame);
 7:   for i := 1 to C.size() − 1 do begin
 8:     result[C[i]] := getGameResult(cycles[C[i]], cycleType[C[i]]);
 9:     foreach C' ∈ cycles[C[i]] do
10:       analyze(C', cycleType[C[i]]);
11:   end
12: end
```

Pierwszy krok (wiersze 3-5) to wypełnienie tablicy *externalGame*, w której dla każdego wierzchołka $v \in C$ zapisujemy wynik gry rozpoczynającej się w v, przy założeniu, że z grafu usunęliśmy krawędzie cyklu C. Innymi słowy, dla każdego v rozpatrujemy jedynie kaktusy *wyrastające* z v.

Korzystamy tu z funkcji *getGameResult*, która oblicza (na podstawie twierdzenia 2), czy dany pęk jest wygrywający. Pierwszy argument funkcji to zbiór cykli stanowiących wyjściowe cykle wszystkich kaktusów w pęku poza jednym. Drugi argument to typ ostatniego kaktusa w pęku. W wierszu 5 nie potrzebujemy uwzględniać typu żadnego dodatkowego kaktusa, dlatego jako argument przekazujemy typ **00**, który nie wpływa na wynik.

W drugim kroku (wiersz 6), dla każdego $v \in C$ rozpatrujemy graf powstały przez usunięcie wszystkich kaktusów wyrastających z v. Wynik gry rozpoczynającej się w v zapisujemy w $cycleType[v]$. Za wykonanie tego kroku odpowiada funkcja *getCycleTypes* zaimplementowana na podstawie twierdzenia 1.

W ostatnim kroku łączymy wcześniej obliczone wartości, używając funkcji *getGameResult* (wiersz 8). Wyniki obliczone w tablicy *result* stanowią końcowe odpowiedzi dla wszystkich wierzchołków początkowych. Warto zaznaczyć, że wartość $result[v]$ jest obliczana podczas analizy cyklu $belong[v]$ (z wyjątkiem wyniku dla korzenia całego kaktusa, który musimy obliczyć osobno). Następnie wywołujemy pro-



Rys. 4: Cykl C widać na prawo od wierzchołka v_1. Wszystkie cykle wewnętrzne C niewyrastające z v_1 zostaną podstawione pod zmienną C' w wierszu 9. Kaktus po lewej wyrastający w stronę korzenia v_0 został narysowany ciągłą linią.

cedurę *analyze* rekurencyjnie dla wszystkich wewnętrznych cykli. Zauważmy, że typ nadkaktusa v jest równy $cycleType[v]$.

Kilku słów komentarza wymaga jeszcze funkcja *getCycleTypes*. Bezpośrednia implementacja twierdzenia 1 może działać w czasie $O(d^2)$, gdzie d równa się długości cyklu, która może być tego samego rzędu co n. Aby tego uniknąć, należy obliczyć odległość od najbliższego wygrywającego pęku w lewo oraz w prawo przy pomocy programowania dynamicznego. Przypadkiem szczególnym, na który trzeba uważać, jest cykl z tylko jednym pękiem wygrywającym, zakorzenionym w v. Patrząc z perspektywy v, na cyklu nie ma pęków wygrywających, zaś dla pozostałych wierzchołków taki kaktus będzie typu **00** lub **11**.

Ostatnia uwaga: w przedstawionych pseudokodach beztrosko przekazywaliśmy tablice jako parametry funkcji, jednak należy zadbać o to, by przekazywanymi obiektami były jedynie indeksy lub wskaźniki. W przeciwnym razie kopiowanie obiektów może prowadzić do złożoności obliczeniowej $\Omega(n^2)$. Podobnie może się skończyć przekazywanie do funkcji *getGameResult* tablicy cykli zamiast operowania na samych konfiguracjach. Poradzenie sobie z tą ostatnią pułapką pozwala na osiągnięcie złożoności liniowej i taki algorytm można znaleźć w pliku `hyd.cpp`.

Karol Pokorski
Treść zadania, Opracowanie

Karol Pokorski, Piotr Smulewicz
Program

Dostępna pamięć: 256 MB. *OI, etap I, 19.10–16.11.2015*

Korale

Bajtyna ma n korali ponumerowanych liczbami od 1 do n. Korale są parami różne. Pewne z nich są bardziej wartościowe od innych – dla każdego z korali znana jest jego wartość w bajtalarach.

Bajtyna chciałaby stworzyć naszyjnik z niektórych ze swoich korali. Jest wiele sposobów utworzenia takiego naszyjnika. Powiemy, że dwa sposoby są różne, jeśli zbiory korali użytych do ich konstrukcji są różne. Aby nieco ułatwić sobie wybór, Bajtyna postanowiła uporządkować wszystkie sposoby utworzenia naszyjnika.

Najważniejszym kryterium jest suma wartości korali w naszyjniku. Im większa suma, tym sposób powinien być późniejszy w uporządkowaniu. Jeśli zaś mamy dwa różne sposoby utworzenia naszyjnika, które mają równą sumę wartości, to porównujemy je według porządku leksykograficznego posortowanych rosnąco list numerów użytych korali[1]*.*

Dla przykładu rozważmy sytuację, w której są cztery korale warte kolejno (zgodnie z numeracją) 3, 7, 4 i 3 bajtalary. Z takich korali naszyjnik można utworzyć na 16 sposobów. Poniżej znajduje się uporządkowanie tych sposobów zgodnie z pomysłem Bajtyny.

Numer sposobu	*Wartości wybranych korali*	*Suma wartości wybranych korali*	*Numery wybranych korali*
1	*brak*	*0*	*brak*
2	*3*	*3*	*1*
3	*3*	*3*	*4*
4	*4*	*4*	*3*
5	*3 3*	*6*	*1 4*
6	*3 4*	*7*	*1 3*
7	*7*	*7*	*2*
8	*4 3*	*7*	*3 4*
9	*3 7*	*10*	*1 2*
10	*3 4 3*	*10*	*1 3 4*
11	*7 3*	*10*	*2 4*
12	*7 4*	*11*	*2 3*
13	*3 7 3*	*13*	*1 2 4*
14	*3 7 4*	*14*	*1 2 3*
15	*7 4 3*	*14*	*2 3 4*
16	*3 7 4 3*	*17*	*1 2 3 4*

Bajtyna chciałaby stworzyć naszyjnik, który ma k-ty numer w uporządkowaniu. Pomóż jej!

[1] Ciąg numerów korali $i_1, \ldots, i_p$ jest mniejszy leksykograficznie od ciągu numerów korali $j_1, \ldots, j_q$, jeśli albo pierwszy ciąg jest początkowym fragmentem drugiego (czyli $p < q$, $i_1 = j_1, \ldots, i_p = j_p$), albo na pierwszej pozycji, na której ciągi te różnią się, pierwszy ciąg ma mniejszy element niż drugi (czyli istnieje takie $u \in \{1, \ldots, \min(p, q)\}$, że $i_1 = j_1, \ldots, i_{u-1} = j_{u-1}$ oraz $i_u < j_u$).

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie dodatnie liczby całkowite n i k oddzielone pojedynczym odstępem, określające liczbę korali oraz żądany numer sposobu utworzenia naszyjnika według uporządkowania opisanego powyżej. W drugim wierszu wejścia znajduje się ciąg n dodatnich liczb całkowitych $a_1, a_2, \ldots, a_n$ pooddzielanych pojedynczymi odstępami – wartości kolejnych korali.

Możesz założyć, że Bajtyna nie pomyliła się i rzeczywiście istnieje co najmniej k różnych sposobów utworzenia jej naszyjnika.

Wyjście

W pierwszym wierszu standardowego wyjścia należy wypisać jedną liczbę całkowitą, oznaczającą sumę wartości korali w znalezionym rozwiązaniu. W drugim wierszu wyjścia należy wypisać ciąg numerów korali użytych w naszyjniku w kolejności rosnącej, rozdzielając liczby pojedynczymi odstępami.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`4 10`	`10`
`3 7 4 3`	`1 3 4`

Testy „ocen":

`1ocen`: *$n = 10$, wszystkie korale mają wartość 1,*

`2ocen`: *$n = 9$, wartości korali są kolejnymi potęgami dwójki,*

`3ocen`: *$n = 11$, jest jeden koral wart 1 bajtalar oraz 10 korali wartych 10^9 bajtalarów, zaś poprawne rozwiązanie używa wszystkich jedenastu korali.*

`4ocen`: *$n = 1\,000\,000$, $k = 10$, wartości kolejnych korali są kolejnymi liczbami od 1 do $1\,000\,000$.*

Ocenianie

Zestaw testów dzieli się na podane poniżej podzadania. Testy do każdego podzadania składają się z jednej lub większej liczby grup testów.

We wszystkich podzadaniach zachodzą warunki $n, k \leqslant 1\,000\,000$ oraz $a_i \leqslant 10^9$.

Jeśli odpowiedź dla danego testu nie jest prawidłowa, jednak pierwszy wiersz wyjścia (suma wartości korali w znalezionym rozwiązaniu) jest prawidłowy, wówczas przyznaje się połowę liczby punktów za dany test (oczywiście odpowiednio przeskalowaną w przypadku przekroczenia przez program połowy limitu czasowego). Dzieje się tak, nawet jeśli drugi wiersz wyjścia jest nieprawidłowy, nie został wypisany lub gdy wypisano więcej niż dwa wiersze wyjścia.

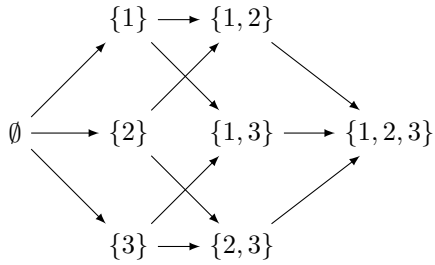
Podzadanie	Dodatkowe warunki	Liczba punktów
1	$n \leqslant 20$, $k \leqslant 500\,000$	*8*
2	$n \leqslant 60$, $k \leqslant 50\,000$	*12*
3	$n \leqslant 3\,000$, $n \cdot k \leqslant 10^6$, $a_i \leqslant 100$	*14*
4	$n \cdot k \leqslant 10^6$	*16*
5	$n \cdot k \leqslant 10^7$	*20*
6	*brak*	*30*

Rozwiązanie

Graf naszyjników

Liczba naszyjników, które można utworzyć z podanych na wejściu korali, wynosi aż 2^n, jednak zadanie dotyczy wypisania k-tego w kolejności naszyjnika. Z uwagi na stosunkowo niewielki limit na wartość liczby k (do miliona) rozsądnym podejściem wydaje się rozpatrywanie naszyjników po kolei.

Rozważmy skierowany graf, w którym wierzchołki reprezentują różne naszyjniki. Każda krawędź prowadzi do wierzchołka reprezentującego naszyjnik zawierający jeden dodatkowy koral względem naszyjnika reprezentowanego przez początek krawędzi.



Rys. 1: Graf naszyjników dla $n = 3$.

Krawędziom można przypisać wagi równe różnicy wartości naszyjników na końcach krawędzi (czyli wartości dodanego korala). Zadanie wówczas sprowadza się do znalezienia k-tego najbliższego wierzchołka od źródła (przy odpowiednim rozpatrywaniu remisów), przy czym źródłem jest wierzchołek reprezentujący pusty naszyjnik.

Algorytm Dijkstry

W tym miejscu wypada przypomnieć algorytm Dijkstry, który służy do znajdowania najkrótszych ścieżek z ustalonego źródła s do wszystkich pozostałych wierzchołków grafu $G = (V, E)$, przy założeniu nieujemnych wag krawędzi:

```
1: procedure Dijkstra(s, (V, E))
2: begin
3:     foreach v ∈ V do d[v] := ∞;
```

```
4:    d[s] := 0;
5:    C := ∅;
6:    for i := 1 to |V| do begin
7:      u := argmin{d[x] : x ∈ V \ C};
8:      foreach uv ∈ E do
9:        if d[v] > d[u] + c(u, v) then
10:         d[v] := d[u] + c(u, v);
11:     C := C ∪ {u};
12:   end
13: end
```

W każdym obiegu głównej pętli jako u przyjmuje się wierzchołek o najmniejszym oszacowaniu odległości ze źródła (najmniejszej wartości $d[u]$), który nie został jeszcze przetworzony (wierzchołki przetworzone dodawane są do zbioru C). Przetwarzanie wierzchołka sprowadza się do przeanalizowania możliwości poprawy oszacowania odległości ze źródła dla wszystkich końców krawędzi z niego wychodzących.

Poprawność algorytmu wynika z faktu, że w każdym obiegu głównej pętli oszacowanie odległości ze źródła s do przetwarzanego wierzchołka u jest w istocie już poprawnie obliczoną odległością z s do u. Dowód tego faktu można przeprowadzić indukcyjnie; patrz np. książka [6]. Dobrym ćwiczeniem na sprawdzenie zrozumienia tego dowodu jest zastanowienie się, w których miejscach dowodu korzysta się z faktu, że wagi krawędzi są nieujemne.

W standardowych implementacjach algorytmu Dijkstry zbiór $V \setminus C$ przechowywany jest na kolejce priorytetowej, implementowanej na przykład za pomocą kopca binarnego lub, w języku C++, struktury `std::priority_queue` czy struktury `std::set`. Przechowywanie zbioru w kolejce priorytetowej pozwala znajdować wierzchołek u o najmniejszym oszacowaniu odległości w czasie $O(\log |V|)$. Należy jeszcze pamiętać, że po zmianie oszacowania odległości dowolnego wierzchołka uporządkowanie zbioru nieprzetworzonych wierzchołków może się zmienić. Implementacje kolejek priorytetowych z użyciem kopca binarnego lub struktury `std::set` potrafią sobie z tym radzić z użyciem $O(\log |V|)$ porównań i zmian w pamięci, co powoduje, że algorytm ma złożoność czasową $O((|V| + |E|) \log |V|)$. Istnieją implementacje kolejek priorytetowych, które przy zmniejszeniu oszacowania odległości wierzchołka potrafią uporządkować zbiór w zamortyzowanym czasie $O(1)$[1], co poprawia złożoność algorytmu do $O(|V| \log |V| + |E|)$. Są one jednak dość skomplikowane, a w praktycznym zastosowaniu nie można zaobserwować ich asymptotycznej przewagi.

W naszym zadaniu należałoby odrobinę zmodyfikować algorytm Dijkstry, aby poprawnie rozpatrywał remisy, czyli sytuacje, w których oszacowanie odległości dla dwóch lub więcej różnych wierzchołków jest takie samo. Oczywiście należy po prostu porównać leksykograficznie naszyjniki reprezentowane przez te wierzchołki zgodnie z treścią zadania. Problemem jest jednak rozmiar grafu. Wierzchołki reprezentują bowiem naszyjniki, a tych jest (jak już wcześniej zauważyliśmy) 2^n. Grafu naszyjników nie trzeba jednak tworzyć w pamięci, a na kolejce priorytetowej wystarczy pamiętać wierzchołki, dla których została już znaleziona (niekoniecznie najkrótsza) ścieżka ze źródła, którym jest wierzchołek reprezentujący pusty naszyjnik.

[1] Chodzi między innymi o kopiec Fibonacciego ([6]).

Zastosowanie najkrótszych ścieżek do rozwiązania zadania

Siła algorytmu Dijkstry w zastosowaniu do tego zadania tkwi w tym, że przetwarza on wierzchołki grafu (naszyjniki) w kolejności takiej, której chciała Bajtyna. Algorytm zatem można przerwać po przetworzeniu k wierzchołków, a naszyjnik reprezentowany przez ów k-ty wierzchołek zwrócić na wyjście.

Łącznie liczbę operacji wykonywanych na kolejce priorytetowej będzie można oszacować przez $O(nk \log(nk))$, ponieważ przetworzenie każdego naszyjnika wprowadzi co najwyżej n zmian do kolejki priorytetowej (zmianą jest tutaj dodanie nowego naszyjnika lub poprawienie oszacowania odległości). Takie rozwiązanie powinno zaliczyć pierwsze dwa podzadania, a przy efektywnej implementacji miało szansę zaliczyć także podzadania trzecie i czwarte, co prowadziło do wyniku rzędu 20–50% punktów.

Na koniec, warto jeszcze zastanowić się nad tym, ile czasu zajmuje porównanie naszyjników. Jeśli naszyjniki mają różną wartość, a implementacja ją przechowuje, wówczas porównanie może się odbyć w czasie stałym. Jeśli jednak naszyjniki mają równą wartość, wówczas należy je porównać leksykograficznie względem numerów korali, co kosztuje czas proporcjonalny do długości krótszego z naszyjników. Teoretycznie, w najgorszym przypadku porównywane naszyjniki mają $O(n)$ korali. W naszej sytuacji możemy jednak podać dużo lepsze oszacowanie. Zauważmy, że jeśli naszyjnik ma więcej niż $\lceil \log k \rceil$ korali, to jest więcej niż $2^{\lceil \log k \rceil} - 1 \geqslant k - 1$ podzbiorów jego korali, które na pewno będą miały mniejszą wartość. A zatem k najwcześniejszych w kolejności naszyjników ma $O(\log k)$ korali i tyle czasu w najgorszym przypadku zajmuje ich porównanie.

Redukcja liczby naszyjników na kolejce priorytetowej

Powyższe rozwiązanie ma istotny problem: dla każdego przetwarzanego naszyjnika liczba zmian w kolejce priorytetowej jest rzędu $O(n)$. Wynika to stąd, że do naszyjnika, który ma c korali, kolejny koral można dołożyć na $n - c$ sposobów. Intuicja podpowiada, że tak duża liczba krawędzi grafu nie jest konieczna. Do każdego wierzchołka reprezentującego naszyjnik c-koralowy prowadzi bowiem $c!$ ścieżek, bo dla każdej permutacji korali w naszyjniku, można je dokładać w tej właśnie kolejności. Jedna ścieżka byłaby wystarczająca.

Niech $I = (i_1, i_2, \ldots, i_n)$ oznacza listę numerów korali w kolejności od najmniejszych do największych wartości (remisy rozstrzygamy dowolnie). Zmieńmy teraz definicję krawędzi w grafie naszyjników. Z wierzchołka reprezentującego naszyjnik $B = \{i_{j_1}, i_{j_2}, \ldots, i_{j_c}\}$, przy czym $j_1 < j_2 < \ldots < j_c$, będą wychodzić dwie krawędzie (o ile $j_c \neq n$):

a) do naszyjnika, w którym koral i_{j_c} zastępujemy koralem i_{j_c+1},

b) do naszyjnika z dodanym koralem i_{j_c+1}.

Tym razem źródłem przeszukiwania będzie wierzchołek reprezentujący naszyjnik $\{i_1\}$ i będziemy poszukiwali $(k-1)$-szego najbliższego wierzchołka ze źródła. Zauważmy, że teraz do wierzchołka reprezentującego naszyjnik $\{i_{j_1}, i_{j_2}, \ldots, i_{j_c}\}$ prowadzi już tylko jedna ścieżka: taka, w której najpierw koral i_1 zamieniamy sukcesywnie w koral i_{j_1},

następnie dodajemy koral i_{j_1+1}, po czym sukcesywnie zamieniamy go w i_{j_2}, itd. Trzeba będzie tylko pamiętać, aby przypadek $k = 1$ obsłużyć jako przypadek szczególny.

W nowym grafie waga krawędzi ponownie będzie równa różnicy wartości naszyjników reprezentowanych przez jej końce. Tym razem nie będzie to zawsze wartość dodanego korala, gdyż niektóre krawędzie wymieniają jeden koral na inny. Nadal jednak wszystkie krawędzie mają nieujemne wagi, bo ewentualna wymiana korala zastępuje go koralem o co najmniej takiej samej wartości.

W nowym grafie również można zastosować algorytm Dijkstry. Tym razem na kolejce priorytetowej znajdzie się tylko $O(k)$ naszyjników, gdyż przetworzenie każdego naszyjnika może dodać do kolejki co najwyżej dwa nowe. Na początku potrzebne jest posortowanie ciągu n korali według wartości. Następnie wykonujemy $O(k)$ operacji na kolejce priorytetowej, z których każda wymaga wykonania $O(\log k)$ porównań naszyjników. Jedno porównanie naszyjników kosztuje $O(\log k)$ operacji, zakładając, że korale w każdym naszyjniku pamiętamy zarówno w porządku niemalejących wartości (co służy do generowania krawędzi wychodzących), jak i w porządku numerów (co jest potrzebne właśnie do wykonywania porównań). Dodatkowo, wygenerowanie krawędzi wychodzących z danego wierzchołka zajmuje czas $O(\log k)$. Ostatecznie otrzymamy rozwiązanie działające w czasie $O(n \log n + k \log^2 k)$. Tego typu rozwiązanie otrzymywało 80–100% punktów w zależności od jakości implementacji. Przykładowy kod znajduje się w pliku `kors5.cpp`.

Szybsze generowanie naszyjników

W rozwiązaniu wzorcowym najpierw wyznaczymy listę L, w której dla każdego z k pierwszych naszyjników zapamiętamy jego wartość (v) oraz najmniejszy numer korala (y). Pokażemy dalej, że taka lista pozwoli nam odtworzyć numery wszystkich korali k-tego z kolei naszyjnika. Zysk, jaki z tego osiągniemy, będzie taki, że podczas generowania listy L nie będziemy musieli pamiętać wszystkich korali w poszczególnych naszyjnikach, dzięki czemu czas porównywania naszyjników zmniejszy się do stałego.

Przypomnijmy, że do generowania krawędzi w zmodyfikowanym grafie naszyjników wystarczy pamiętać dla danego naszyjnika numer korala o największej wadze, czyli numer najpóźniejszego korala w kolejności listy I. Oznaczmy numer tego korala w danym naszyjniku jako x. Dodatkowo, wymiana tego korala na inny (krawędź typu a)) może czasami spowodować zmianę numeru korala y. Aby sobie z tym poradzić, będziemy także pamiętać najmniejszy numer korala w naszyjniku z pominięciem tego o największej wadze (z). (Jeśli koral z nie istnieje, przyjmiemy, że $z = \infty$). Wreszcie poza numerem korala x zapamiętamy także jego pozycję p na liście I (tj. $i_p = x$). Ostatecznie dla każdego naszyjnika będziemy pamiętali tylko pięć liczb: v, x, y, z i p.

Upewnijmy się teraz, że przy przejściu krawędzią w grafie naszyjników będziemy w stanie zaktualizować przechowywane informacje. Przechodząc krawędzią typu a), koral x zastępujemy koralem $x' = i_{p+1}$ (i mamy $p' = p + 1$), co może wpłynąć na zmianę y: $y' = \min(z, x')$; natomiast z nie zmienia się ($z' = z$). Natomiast wskutek przejścia krawędzią typu b) po prostu dochodzi nam nowy koral $x' = i_{p+1}$ (i znów mamy $p' = p + 1$); to oznacza, że $z' = y$, natomiast $y' = \min(y, x')$. W obydwu przypadkach łatwo aktualizujemy wartość naszyjnika.

W kolejce priorytetowej naszyjniki porównujemy tylko według wartości v oraz, w drugiej kolejności, numeru korala y. Zauważmy, że to *nie wystarcza* do porównania naszyjników według kryterium Bajtyny, jednak pozwala w poprawny sposób wygenerować wszystkie elementy listy L jako $k-1$ pierwszych wierzchołków przetworzonych w algorytmie Dijkstry (na początek listy dodajemy pusty naszyjnik).

Przyjrzyjmy się teraz, jak za pomocą listy L możemy odzyskać wynik. Niech v_i oraz y_i oznaczają wartość i najmniejszy numer korala w i-tym naszyjniku z listy. Wiemy wówczas, że pierwszy koral k-tego naszyjnika to v_k, i możemy go wypisać. Niech m oznacza liczbę elementów listy L o wartości v_k i koralu y_k. Wówczas kolejne korale k-tego naszyjnika będą takie same jak korale m-tego naszyjnika na liście spośród tych o wartości $w = v_k - a_{y_k}$ i najmniejszym numerze korala większym niż v_k. Naszyjnik ten możemy znaleźć, cofając się na liście L do pierwszego naszyjnika o wartości w i najmniejszym koralu większym niż v_k, a następnie idąc o $m-1$ naszyjników do przodu; niech i oznacza numer tak określonego naszyjnika. Wówczas numer korala y_i tego naszyjnika to zarazem drugi w kolejności numer korala k-tego naszyjnika. Rozumowanie to powtarzamy, począwszy od naszyjnika numer i, aż do znalezienia wszystkich numerów korali k-tego naszyjnika. Całe odzyskiwanie działa w czasie $O(k)$.

Przykład 1. Oto jak wygląda lista L dla przykładu w treści zadania ($k = 10$):

i	1	2	**3**	4	5	6	7	**8**	9	**10**
v_i	0	3	**3**	4	6	7	7	**7**	10	**10**
y_i	-	1	**4**	3	1	1	2	**3**	1	**1**

Przypomnijmy, że $a_1 = 3$, $a_2 = 7$, $a_3 = 4$ i $a_4 = 3$.

Pierwszy koral szukanego, dziesiątego w kolejności naszyjnika ma numer $y_{10} = 1$. Dalej, są $m = 2$ naszyjniki na liście o wartości $v_i = v_{10} = 10$ i koralu $y_i = y_{10} = 1$. Stąd kolejny koral dziesiątego naszyjnika jest taki sam jak pierwszy koral drugiego spośród naszyjników, które mają wartość $v_i = v_{10} - a_{y_{10}} = 7$ i najmniejszy numer korala $y_i > y_{10} = 1$. Szukany koral to zatem $y_8 = 3$ z naszyjnika numer 8. Kontynuując ten proces, stwierdzamy, że jest tylko $m = 1$ naszyjnik o wartości $v_i = v_8 = 7$ i koralu $y_i = y_8 = 3$. Tak więc kolejny koral dziesiątego naszyjnika jest taki sam jak pierwszy koral jedynego naszyjnika, który ma wartość $v_i = v_8 - a_{y_8} = 3$ i najmniejszy numer korala $y_i > y_8 = 3$. Jest to zatem koral numer $y_3 = 4$ z naszyjnika numer 3.

Złożoność rozwiązania zmniejsza się do $O(n \log n + k \log k)$, gdyż każde porównanie realizowane jest w czasie stałym, a liczba operacji na kolejce priorytetowej nie zmieniła się. Implementacja znajduje się w pliku `kor.cpp` i otrzymuje oczywiście maksymalną punktację. W praktyce jednak rozwiązanie to nie jest o wiele szybsze od poprzedniego, gdyż stały czynnik ukryty w notacji asymptotycznej jest dość duży – dla danych z zadania niewiele mniejszy od czynnika logarytmicznego w poprzednim rozwiązaniu.

Rozwiązanie alternatywne

Na zadanie można spojrzeć inaczej, bez rozważania grafu naszyjników. Zamiast tego, można próbować obliczać listy k pierwszych naszyjników w kolejności wyznaczonej

przez Bajtynę zbudowanych z wykorzystaniem wyłącznie korali o numerach $i, \ldots, n$. Listy te wyznaczymy kolejno dla $i = n + 1, \ldots, 1$.

Na początku, dla pustego ciągu dostępnych korali jest tylko jeden możliwy do uzyskania naszyjnik – pusty, o wartości równej 0.

Niech $l_1, l_2, \ldots, l_{k'}$ $(k' \leqslant k)$ oznaczają kolejne naszyjniki zgodnie z uporządkowaniem Bajtyny utworzone dla korali $a_{i+1}, a_{i+2}, \ldots, a_n$. Dodamy teraz do rozważań koral o numerze i. W tym celu utworzymy drugą listę naszyjników $m_1, m_2, \ldots, m_{k'}$, w której $m_i = l_i \cup \{i\}$ (do każdego naszyjnika dodaliśmy koral i).

Teraz wystarczy złączyć listy naszyjników l oraz m w jedną, podobnie jak scala się dwie posortowane listy w jedną w algorytmie sortowania przez scalanie. Jeśli długość listy wynikowej przekroczyła k, to można bezpiecznie odrzucić jej koniec po k-tym naszyjniku, bo żaden z usuwanych naszyjników nie będzie k-ty w kolejności (skoro jest co najmniej k wcześniejszych od każdego z nich). Rozpatrywanie sufiksów ciągu korali (zamiast np. prefiksów) pomaga w utrzymywaniu poprawnej kolejności leksykograficznej ciągu korali w przypadku rozstrzygania remisów – zawsze bowiem pierwszeństwo w przypadku równej wartości naszyjników w scalanych listach będzie miał naszyjnik z listy m (z dodanym koralem i) nad naszyjnikiem z listy l (z najwcześniejszym koralem o numerze co najmniej $i + 1$).

Rozwiązanie to można zaimplementować, aby działało w czasie $O(nk)$. Powinno zaliczać wszystkie podzadania poza ostatnim i uzyskiwać około 70% punktów. Implementacja znajduje się w pliku `kors3.cpp`.

Inne rozwiązania

Rozwiązanie wykładnicze

Narzucającym się rozwiązaniem jest wygenerowanie wszystkich 2^n naszyjników. Wówczas można je posortować lub skorzystać z algorytmu selekcji[2], co prowadzi do algorytmu o złożoności $\Omega(2^n)$, który mógł zaliczyć jedynie pierwsze podzadanie.

Implementacja znajduje się w pliku `kors1.cpp`.

Rozwiązanie oparte o wydawanie reszty

Na zadanie można też spojrzeć nieco inaczej, jako na problem wydawania reszty, gdzie nominałami są wartości korali. Z użyciem algorytmu opartego o programowanie dynamiczne możliwe jest wyznaczenie dla każdej kwoty (wartości naszyjnika) liczby sposobów jej wydania (liczby różnych naszyjników o tej wartości).

Niestety, odzyskanie k-tego naszyjnika tą metodą nie jest możliwe; możliwe jest jedynie odzyskanie jego wartości.

Rozwiązanie to pozwalało uzyskać połowę punktów za trzecie podzadanie. Jest ono zaimplementowane w pliku `korb2.cpp`.

[2] Algorytm magicznych piątek lub algorytm Hoare'a [6]

Jacek Tomasiewicz Treść zadania | **Krzysztof Kiljan** Opracowanie | **Kamil Dębowski** Program

Dostępna pamięć: 256 MB. *OI, etap I, 19.10–16.11.2015*

Nadajniki

Bajtazar został nowym dyrektorem zabytkowej kopalni soli pod Bajtowem. Aby zwiększyć popularność tego obiektu wśród turystów, postanowił zainstalować w korytarzach kopalni bezprzewodowy Internet.

Kopalnia składa się z n komór połączonych $n - 1$ korytarzami. Z każdej komory można przejść do każdej innej, używając korytarzy. Bajtazar postanowił rozmieścić w komorach nadajniki wi-fi tak, by Internet był dostępny w każdym z korytarzy kopalni. Aby można było korzystać z Internetu w korytarzu łączącym komory a i b, musi być spełniony co najmniej jeden z poniższych warunków:

- *w komorze a lub w komorze b znajduje się nadajnik, lub*
- *w zbiorze komór, do których można dojść z komory a lub komory b, używając co najwyżej jednego korytarza, znajdują się co najmniej dwa nadajniki.*

Bajtazar zastanawia się teraz, jaka jest minimalna liczba nadajników wi-fi, które musi rozmieścić, aby można było korzystać z Internetu w każdym korytarzu. W każdej komorze można umieścić dowolną liczbę nadajników.

Wejście

Pierwszy wiersz standardowego wejścia zawiera dodatnią liczbę całkowitą n oznaczającą liczbę komór w kopalni. Komory numerujemy liczbami od 1 do n.

Kolejne $n - 1$ wierszy opisuje korytarze w kopalni. Każdy z nich zawiera dwie liczby całkowite a i b ($1 \leqslant a, b \leqslant n$, $a \neq b$) oddzielone pojedynczym odstępem, oznaczające, że komory o numerach a i b są połączone korytarzem.

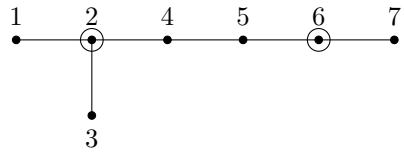
Wyjście

Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą, oznaczającą minimalną liczbę nadajników, które musi rozmieścić Bajtazar.

Przykład

Dla danych wejściowych:

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

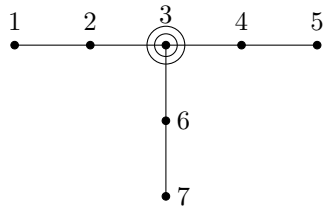


poprawnym wynikiem jest:

```
2
```

i dla danych wejściowych:

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



poprawnym wynikiem jest:

```
2
```

Wyjaśnienie do przykładów: *W pierwszym przykładzie wystarczy umieścić nadajniki w komorach o numerach 2 i 6, natomiast w drugim przykładzie wystarczy umieścić dwa nadajniki w komorze numer 3.*

Testy „ocen":

`1ocen`: $n = 16$. *Komora* i *jest połączona z komorą* $\lfloor i/2 \rfloor$ *dla* $2 \leqslant i \leqslant n$.

`2ocen`: $n = 303$. *Komora 2 jest połączona z komorami 1 oraz 3. Każda z komór 1, 2, 3 jest dodatkowo połączona z setką komór. Optymalnym rozwiązaniem jest umieszczenie dwóch nadajników w komorze 2.*

`3ocen`: $n = 200\,000$. *Komory* i *oraz* $i + 1$ *są połączone korytarzem dla* $1 \leqslant i \leqslant n - 1$.

Ocenianie

Zestaw testów dzieli się na następujące podzadania. Testy do każdego podzadania składają się z jednej lub większej liczby osobnych grup testów.

Podzadanie	Warunki	Liczba punktów
1	$n \leqslant 10$	*15*
2	$n \leqslant 500$	*20*
3	$n \leqslant 200\,000$, *do każdej komory prowadzą co najwyżej trzy korytarze*	*25*
4	$n \leqslant 200\,000$	*40*

Rozwiązanie

Wprowadzenie

W zadaniu mamy dany spójny graf nieskierowany o n wierzchołkach oraz $n-1$ krawędziach. Powszechnie przyjętą nazwą dla takich grafów jest *drzewo*. Drzewa charakteryzują się brakiem jakichkolwiek cykli oraz tym, że dla każdej pary wierzchołków istnieje dokładnie jedna ścieżka je łącząca.

Na potrzeby tego zadania, jako *zlewisko* krawędzi $e = (u, v)$ zdefiniujemy zbiór wszystkich wierzchołków sąsiadujących z u lub z v, co matematycznie można zapisać na przykład w następujący sposób: $Z(u, v) = \{w : (u, w) \in E \vee (v, w) \in E\}$, gdzie E to zbiór wszystkich krawędzi naszego drzewa.

W każdym z wierzchołków wstawiamy pewną nieujemną liczbę nadajników. Krawędź $e = (u, v)$ nazwiemy *spełnioną*, jeżeli zachodzi przynajmniej jeden z poniższych warunków:

- W wierzchołku u lub w wierzchołku v znajduje się co najmniej jeden nadajnik.
- W zlewisku krawędzi e znajdują się co najmniej dwa nadajniki.

W pierwszym przypadku mówimy, że krawędź jest spełniona *bezpośrednio*, a w drugim, że jest spełniona *pośrednio*.

Naszym zadaniem jest wyznaczenie najmniejszej liczby nadajników, których wstawienie gwarantuje, że wszystkie krawędzie w drzewie będą spełnione.

Rozwiązanie wykładnicze

Rozwiązywanie zadania zacznijmy od prostego spostrzeżenia.

Obserwacja 1. W żadnym wierzchołku nie warto umieścić więcej niż 2 nadajników.

Obserwacja ma oczywiste uzasadnienie – z warunków zadania wynika, że postawienie więcej niż dwóch nadajników nie może w żaden sposób wpłynąć na spełnialność którejkolwiek krawędzi. Pomysł ten pozwala nam na stworzenie pierwszego wolnego rozwiązania. W każdym z wierzchołków mamy trzy możliwości – możemy ustawić 0 nadajników, 1 albo 2.

Jako że mamy n wierzchołków, wszystkich możliwych rozstawień nadajników w wierzchołkach jest 3^n. Jeśli dane rozstawienie jest poprawne, to sprawdzamy, czy wymaga ono mniejszej liczby nadajników niż najlepsze znalezione przez nas dotychczas. Jeśli tak, to aktualizujemy potrzebną liczbę nadajników. Najprościej zaimplementować takie rozwiązanie jako funkcję rekurencyjną, którą przedstawiamy poniżej.

Przyjmujemy, że mamy napisane funkcje *calcCnt*() oraz *check*(). Pierwsza z nich ma za zadanie wyznaczyć liczbę nadajników używanych przez aktualne rozstawienie. Można ją napisać w złożoności $O(n)$, po prostu obliczając sumę wartości w tablicy *cnt*[] przechowującej liczby nadajników umieszczonych w poszczególnych wierzchołkach. Druga z funkcji sprawdza, czy dane rozstawienie nadajników powoduje spełnienie wszystkich krawędzi. Można to zrealizować, dla każdego wierzchołka zliczając

nadajniki umieszczone w jego sąsiadach (oznaczmy taką wartość przez $cntInNeigh[v]$), a potem dla każdej krawędzi (u, v) sprawdzając, czy $cnt[u] + cnt[v] \geqslant 1$ lub $cntInNeigh[u] + cntInNeigh[v] \geqslant 2$. Przy takim podejściu każde sprawdzenie zajmuje czas $O(n)$, tak więc otrzymujemy łączną złożoność czasową $O(3^n \cdot n)$.

```
 1: bestAns := ∞;
 2: function genRec(v)
 3: begin
 4:   for i := 0 to 2 do begin
 5:     cnt[v] := i;
 6:     if v = n then begin
 7:       if check() then
 8:         bestAns := min(bestAns, calcCnt());
 9:     end else genRec(v + 1);
10:   end
11: end
```

Funkcję wywołujemy jako $genRec(1)$. Rozwiązania podobne do powyższego można znaleźć w plikach `nads1.cpp` oraz `nads2.cpp`. Na zawodach poprawna implementacja takiego rozwiązania pozwalała na zdobycie między 10 a 15 punktów.

Rozwiązanie wzorcowe

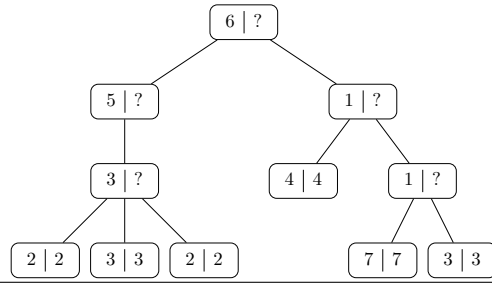
Nie jest trudno zauważyć, że przy ograniczeniach, jakie mamy dane w zadaniu ($n \leqslant 200\,000$), rozwiązania wykładnicze nie mają prawa skończyć się w żadnym sensownym czasie. Musimy więc poszukać czegoś o wiele szybszego.

Programowanie dynamiczne na drzewach

Na początku postaramy się przybliżyć, na czym polega technika programowania dynamicznego na drzewach. Jeśli Czytelnik jest z nią obeznany, może przejść do kolejnego podrozdziału, w którym będziemy próbować ją zastosować do naszego zadania. W ogólności programowanie dynamiczne polega na obliczaniu jakichś wartości dla większych egzemplarzy problemu na podstawie wyników dla mniejszych egzemplarzy. Zaczyna się od bardzo prostych przypadków, a następnie pokazuje się, jak wyniki dla mniejszych egzemplarzy złożyć w wyniki dla bardziej skomplikowanych egzemplarzy. Przykładami podstawowych problemów rozwiązywanych za pomocą programowania dynamicznego jest najdłuższy wspólny podciąg oraz (dyskretny) problem plecakowy.

Technikę tę da się również zaadaptować do wielu problemów, w których mamy do czynienia z drzewami. Wystarczy, że ukorzenimy drzewo w jednym z wierzchołków i będziemy je przechodzić od liści do korzenia. Wartość dla poddrzewa ukorzenionego w ojcu będziemy wyznaczać na podstawie wartości dla poddrzew jego dzieci. Takie podejście nazywane jest programowaniem dynamicznym *z dołu do góry*.

Dla lepszego zrozumienia, spróbujemy zobrazować to na przykładzie następującego prostego zadania: Mamy dane drzewo ukorzenione w wierzchołku numer 1. Każdy wierzchołek zawiera pewną liczbę monet v. Chcemy być w stanie w czasie $O(1)$ odpowiadać na zapytania postaci „Ile jest monet w danym poddrzewie?”.

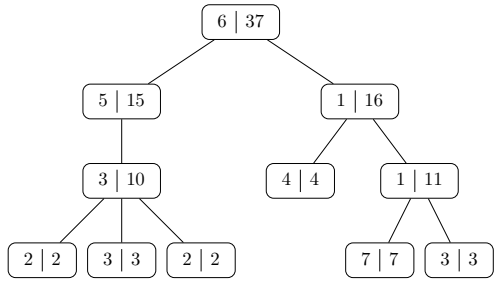


Rys. 1: Lewa wartość oznacza liczbę monet w wierzchołku, prawa – w całym poddrzewie.

W tym przypadku wnioski są oczywiste – dla każdego wierzchołka v szukana wartość jest równa jego liczbie monet $coins[v]$ powiększonej o liczbę monet w poddrzewach jego dzieci. Takie rozwiązanie możemy zrealizować za pomocą poniższego pseudokodu:

```
1:  function dfs(v)
2:  begin
3:    vis[v] := true;
4:    dp[v] := coins[v];
5:    foreach (v, child) ∈ E do
6:      if not vis[child] then begin
7:        dfs(child);
8:        merge(v, child);
9:      end
10: end
11: function merge(parent, child)
12: begin
13:   dp[parent] += dp[child];
14: end
```

Procedurę wywołujemy jako $dfs(1)$. Wartości wyznaczone za pomocą powyższego algorytmu dla drzewa z rysunku 1 można obejrzeć na rysunku 2.



Rys. 2: Drzewo z wyznaczonymi wartościami.

Stanem w programowaniu dynamicznym nazywamy egzemplarz podproblemu, dla którego obliczamy wartość. W powyższym przykładzie stan odpowiada poddrzewu oryginalnego drzewa, identyfikowanemu poprzez wierzchołek będący korzeniem poddrzewa.

Warto wspomnieć, że w programowaniu dynamicznym można wyróżnić dwa podejścia, tzw. *w tył* oraz *w przód*.

W podejściu *w tył* (częściej spotykanym) ustalamy pewien stan, którego wartość chcemy obliczyć, i wyznaczamy ją na podstawie już znanych wartości wyliczonych dla jego poprzedników, czyli stanów, od wartości których zależy jego wartość.

W podejściu *w przód* ustalamy pewien stan, którego wartość już znamy, i przeglądamy wszystkie stany, których wartości jeszcze nie znamy, a na które wpływ ma wartość owego ustalonego stanu (tzn. następników stanu).

Podejście *w tył* możemy zastosować, gdy dla stanu potrafimy wyznaczyć jego poprzedników, a podejście *w przód*, gdy dla stanu potrafimy wyznaczyć jego następników. W zależności od problemu, być może możemy zastosować oba podejścia lub tylko jedno z nich. Przykładowo, w przypadku naszego prostego zadania możliwe są oba podejścia. W powyższym pseudokodzie zaimplementowaliśmy podejście *w tył*, co wymaga założenia, że możemy efektywnie wyznaczyć wszystkie dzieci wierzchołka – a zatem należy je utrzymywać np. na liście sąsiedztwa.

Aby przećwiczyć technikę programowania dynamicznego na drzewie, można zmierzyć się na przykład z zadaniem *Farmercraft* z dnia próbnego finałów XXI Olimpiady Informatycznej [1].

Programowanie dynamiczne na drzewach w zadaniu *Nadajniki*

Do rozwiązania naszego zadania również można zastosować metodę programowania dynamicznego, jednak nie jest zupełnie oczywiste jak to zrobić. Dużym utrudnieniem może okazać się to, że nadajniki powodujące spełnienie krawędzi łączącej aktualnie przetwarzany wierzchołek z jego synem mogą znajdować się nie tylko w poddrzewie wierzchołka, ale również gdzieś wyżej. Łatwo jest też wpaść w pułapkę i wymyślić stany, których łączenie jest bardzo trudne lub wręcz niemożliwe do wykonania.

Całość zaczynamy oczywiście od ukorzenienia naszego drzewa w dowolnym wierzchołku (np. w tym o numerze 1). W rozwiązaniu wzorcowym dla każdego wierzchołka v będziemy wyliczać trójwymiarową tablicę o indeksach od 0 do 2. Dla ustalenia uwagi, będziemy ją oznaczać $t[inMe][inPar][later]$. Parametry używane w tej tablicy reprezentują odpowiednio:

- *inMe* – liczbę nadajników, które wstawiamy w wierzchołku v;
- *inPar* – liczbę nadajników, które są nam potrzebne w ojcu wierzchołka v ze względu na to, że któraś z krawędzi między wierzchołkiem v a jego synem jeszcze nie została spełniona;
- *later* – liczbę nadajników, których brakuje do pośredniego spełnienia krawędzi między wierzchołkiem v a jego ojcem, przy założeniu, że krawędź ta istnieje i nie jest spełniona bezpośrednio. Mogą być w tym celu wykorzystane nadajniki umieszczone w innych dzieciach ojca wierzchołka v lub w wierzchołku będącym dziadkiem wierzchołka v. Rozważana krawędź będzie spełniona bez względu na

parametr *later*, jeżeli w wierzchołku v lub w jego ojcu postawimy co najmniej jeden nadajnik.

Wartość $t[inMe][inPar][later]$ oznacza minimalną liczbę nadajników umieszczonych w poddrzewie wierzchołka v, które są zgodne z podanymi wartościami parametrów i spełniają wszystkie krawędzie w poddrzewach synów wierzchołka v.

Mając pewien korzeń v poddrzewa oraz wyliczone stany dla każdego z jego synów, będziemy chcieli wyliczyć jego tablicę w następujący sposób:

1. Składamy wyniki wszystkich synów v.
2. Wyliczamy tablicę dla v na podstawie scalonego wyniku dla synów, próbując umieszczenia w v każdej możliwej liczby nadajników.

Warto w tym momencie od razu zastanowić się, jakimi wartościami powinniśmy takie tablice zainicjować dla pojedynczych wierzchołków (co będzie wykorzystywane w liściach). Krawędzie od wierzchołka v do synów nie istnieją, zatem parametr *inPar* powinien być równy zeru. Jeżeli parametr *inMe* jest dodatni, to spełniamy także krawędź od wierzchołka v do jego ojca, więc w takim przypadku $later = 0$. Jeżeli jednak $inMe = 0$, to potrzebujemy dwóch nadajników, aby spełnić tę krawędź pośrednio, czyli $later = 2$. Jeżeli zatem $(inMe, inPar, later)$ przyjmuje którąś z wartości $(0, 0, 2)$, $(1, 0, 0)$, $(2, 0, 0)$, to wartość w tablicy programowania dynamicznego wynosi *inMe*. Wszystkie pozostałe stany reprezentują niedopuszczalny zbiór parametrów, zatem ustawiamy w nich bardzo dużą wartość, np. 10^9 (równoważną nieskończoności w tym zadaniu). Można dodatkowo poczynić obserwację, że jeżeli w dopuszczalnym rozstawieniu dla całego drzewa wszystkie nadajniki z konkretnego liścia przesuniemy do jego ojca, to także otrzymamy dopuszczalne rozstawienie o takim samym koszcie. Pozwala to założyć, że w liściach nie wstawiamy żadnych nadajników, co zmniejsza zbiór stanów, które musimy uwzględnić, do $(0, 0, 2)$.

Scalanie synów

Będziemy scalać dwie tablice – *t1* oraz *t2* – w jedną nową. Pierwsza z nich będzie odpowiadała wszystkim dotychczas scalonym synom, druga zaś temu, którego aktualnie będziemy dołączać. Parametry *inMe*, *inPar* oraz *later* będą otrzymywać przyrostki 1 i 2 w zależności od tablicy, z której pochodzą. Trzeba w tym momencie podkreślić, że wynikowa tablica nie jest tablicą odpowiadającą żadnemu konkretnemu poddrzewu, a zbiorowi poddrzew posiadających tego samego ojca (bez uwzględnienia samego ojca). Definicje parametrów *inMe*, *inPar* i *later* zmieniają się, ale w dość intuicyjny sposób:

- *inMe* – liczba nadajników, które wstawiliśmy w korzeniach tych poddrzew;
- *inPar* – liczba nadajników, które są nam potrzebne we wspólnym ojcu wszystkich uwzględnionych poddrzew, która zapewnia spełnienie wszystkich krawędzi pomiędzy korzeniami tych poddrzew a ich synami;
- *later* – liczba nadajników, których brakuje do spełnienia w sposób pośredni tych krawędzi między korzeniami poddrzew a ich wspólnych ojcem, które nie

są spełnione bezpośrednio. Mogą być one uzupełniane przez nadajniki z jeszcze nieuwzględnionych poddrzew, które mają tego samego ojca, lub z ojca wspólnego ojca.

Zastosujemy programowanie dynamiczne *w przód* (definicja znajduje się we wcześniejszej sekcji). Scalanie zrealizujemy, iterując po wszystkich możliwych wartościach parametrów tablic wejściowych i poprawiając, jeśli to możliwe, odpowiednie komórki tablicy wynikowej *res* (której wartości początkowo ustawiamy na nieskończoność).

Wyznaczanie parametrów *newInMe*, *newInPar* oraz *newLater* tablicy wynikowej przebiega następująco:

- *newInMe* jest równe $inMe1 + inMe2$.
- *newInPar* jest równe maksimum spośród *inPar1* oraz *inPar2*. Uzasadnienie jest jasne – oba warunki są spełnione wtedy i tylko wtedy, kiedy większy z nich jest spełniony.
- *newLater* równa się maksimum z wartości $later1 - inMe2$ i $later2 - inMe1$. Wartości te wynikają z tego, że nadajniki z korzenia sąsiedniego poddrzewa (brat aktualnego wierzchołka) mogą pomóc spełnić krawędź, której dotyczy *later*.

Jeśli wykonujemy jakieś operacje dodawania lub odejmowania na parametrach, to zawsze na koniec musimy je obciąć do przedziału $[0, 2]$ poprzez wzięcie minimów bądź maksimów z odpowiednich wartości. Zostało to zilustrowane w poniższym pseudokodzie.

```
1:  function merge(t1, t2)
2:  begin
3:    res := tablica wypełniona nieskończonościami;
4:    for inMe1 := 0 to 2 do
5:      for inPar1 := 0 to 2 do
6:        for later1 := 0 to 2 do
7:          for inMe2 := 0 to 2 do
8:            for inPar2 := 0 to 2 do
9:              for later2 := 0 to 2 do begin
10:               newInMe := min(2, inMe1 + inMe2);
11:               newInPar := max(inPar1, inPar2);
12:               newLater := max(0, max(later1 − inMe2, later2 − inMe1));
13:               res[newInMe][newInPar][newLater] :=
14:                 min(res[newInMe][newInPar][newLater],
15:                     t1[inMe1][inPar1][later1] + t2[inMe2][inPar2][later2]);
16:             end
17:   return res;
18: end
```

Obliczanie stanu ojca

Po scaleniu tablic t wszystkich poddrzew w jedną musimy ją jeszcze przełożyć na wynik dla samego korzenia v. Indeksy *newInMe*, *newInPar* oraz *newLater* będą określać wartości parametrów dla wierzchołka v. Będziemy poprawiać wartości w odpowiednich komórkach tablicy wynikowej *res*, iterując po wszystkich możliwych indeksach *inMe*, *inPar*, *later* zdefiniowanych jak w poprzedniej sekcji oraz po parametrze *newInMe*.

Zwróćmy uwagę na kilka faktów:

- Z definicji *inPar* wynika, że *newInMe* musi być co najmniej tak duże jak *inPar*.
- Jeśli $newInMe = 0$, to *newInPar* będzie równe staremu *later* – nadajniki, które wcześniej musieliśmy wstawić gdzieś wyżej, teraz musimy już wstawić w ojcu wierzchołka v.
- Jeśli $newInMe = 0$, to *newLater* będzie równe $2 - inMe$, czyli liczbie nadajników, których potrzebujemy do spełnienia krawędzi od wierzchołka v do jego ojca minus te, które otrzymaliśmy już z synów wierzchołka v.
- Jeśli *newInMe* jest większe od zera, to *newInPar* oraz *newLater* są równe zeru. Faktycznie, jeśli w wierzchołku v mamy jakiś nadajnik, to automatycznie wszystkie interesujące nas krawędzie stają się spełnione.

Po tych obserwacjach możemy już napisać pseudokod zmieniający tablicę sumy poddrzew na tablicę stanu przedstawiającego poddrzewo ukorzenione w wierzchołku v:

```
1:  function createParent(t)
2:  begin
3:    res := tablica wypełniona nieskończonościami;
4:    for newInMe := inPar to 2 do
5:      for inMe := 0 to 2 do
6:        for inPar := 0 to 2 do
7:          for later := 0 to 2 do begin
8:            if newInMe = 0 then begin
9:              newInPar := later;
10:             newLater := 2 - inMe;
11:           end else begin
12:             newInPar := 0;
13:             newLater := 0;
14:           end
15:           res[newInMe][newInPar][newLater] :=
16:             min(res[newInMe][newInPar][newLater],
17:                 newInMe + t[inMe][inPar][later]);
18:       end
19:   return res;
20: end
```

Dostosowanie funkcji *dfs* z sekcji o programowaniu dynamicznym do warunków naszego zadania pozostawiamy już jako ćwiczenie dla Czytelnika.

Wynik

Ostatnim krokiem, który musimy wykonać po przetworzeniu całego drzewa, jest odczytanie wyniku. Jak pamiętamy, parametr *inPar* oznacza, ile jeszcze potrzeba nadajników w ojcu wierzchołka v. W przypadku korzenia całego drzewa, ze względu na brak możliwości umieszczenia jakichkolwiek nadajników wyżej, musi być on równy zeru. Parametr *later* odnosił się do nadajników, które są wymagane, aby spełnić warunek dla krawędzi ponad wierzchołkiem v. Jednak dla korzenia całego drzewa takiej krawędzi nie ma, zatem wartość tego parametru w korzeniu jest dla nas nieistotna. W związku z tym, wybieramy stan o minimalnym wyniku spośród stanów $res[inMe][0][later]$, gdzie *res* jest tablicą wyników dla korzenia, a *inMe* i *later* przyjmują dowolne całkowite wartości z przedziału $[0, 2]$.

Całe rozwiązanie wykonuje liniową liczbę scaleń poddrzew i w każdym sprawdza $3^3 \cdot 3^3 = 729$ możliwości. Daje nam to asymptotycznie złożoność $O(n)$. Trzeba jednak być świadomym, że jest to rozwiązanie z wyjątkowo dużą stałą, zatem będzie znacznie wolniejsze od większości algorytmów o złożonościach liniowych dla podobnych rozmiarów danych.

Rozwiązanie opisane powyżej było wystarczająco dobre, żeby uzyskać maksymalną punktację. Jego kod można znaleźć w plikach `nad.cpp` oraz `nad2.cpp`.

Trochę szybciej

Zawodnicy, którzy chcieli przyspieszyć swój program, mogli to zrobić poprzez zmniejszenie stałego czynnika w czasie działania programu. Jedna z optymalizacji, które można zastosować, aby przyspieszyć najbardziej czasochłonny proces scalania poddrzew, jest taka, że jeżeli dla pojedynczego poddrzewa (ale już nie ich zbioru) zachodzi $inMe2 > 0$, to pozostałe parametry (*inPar2* oraz *later2*) są zerowe. Druga z optymalizacji polega na tym, że jeżeli $inPar2 > 0$, to spełniamy automatycznie wszystkie krawędzie od korzeni poddrzew do wspólnego ojca, zatem można założyć, że $later2 = 0$. Daje to rozwiązanie działające oczywiście cały czas w złożoności $O(n)$, jednak w każdym scaleniu zamiast 729 rozważamy jedynie $27 \cdot 7 = 189$ przypadków. Takie rozwiązania na zawodach nie były odróżniane od powyższego i również dostawały 100 punktów. Implementację takiego podejścia można znaleźć w pliku `nad1.cpp`.

Podzadanie 3: każda komora ma maksymalnie trzech sąsiadów

W zadaniu znalazło się podzadanie, w którym dodatkowo było nałożone ograniczenie górne na liczbę sąsiadów danego wierzchołka. W rzeczywistości oznacza ono, że jeśli dobrze ukorzenimy drzewo (w wierzchołku o maksymalnie dwóch sąsiadach), to otrzymamy drzewo, w którym każdy wierzchołek będzie miał co najwyżej dwóch synów.

Dzięki temu jesteśmy w stanie scalać wszystkie poddrzewa naraz i nie jest nam potrzebna obserwacja (być może trochę nieintuicyjna), że można je dołączać jedno po drugim.

Takie rozwiązanie przechodziło całe trzecie podzadanie. Można je obejrzeć w pliku `nadb2.cpp`.

Marek Sokołowski
Treść zadania, Opracowanie

Paweł Parys
Program

Dostępna pamięć: 256/64 MB.

OI, etap I, 19.10–16.11.2015

Nim z utrudnieniem

Ulubioną rozrywką Alicji i Bajtazara jest gra Nim. Do gry potrzebne są żetony, podzielone na kilka stosów. Dwaj gracze na przemian zabierają żetony ze stosów – ten, na którego przypada kolej, wybiera dowolny stos i usuwa z niego dowolną dodatnią liczbę żetonów. Gracz, który nie może wykonać ruchu, przegrywa[1].

Alicja zaproponowała Bajtazarowi kolejną partyjkę Nima. Aby jednak tym razem uczynić grę ciekawszą, gracze umówili się między sobą na dodatkowe warunki. Żetony, których było m, Alicja podzieliła na n stosów o licznościach $a_1, a_2, \ldots, a_n$. Zanim rozpocznie się rozgrywka, Bajtazar może wskazać niektóre spośród stosów, które zostaną natychmiast usunięte z gry. Liczba usuniętych stosów musi być jednak podzielna przez pewną ustaloną liczbę d, a ponadto Bajtazar nie może usunąć wszystkich stosów. Potem rozgrywka będzie toczyć się już normalnie, a rozpocznie ją Alicja.

Niech k oznacza liczbę sposobów, na które Bajtazar może wskazać stosy do usunięcia tak, aby mieć pewność, że wygra partię niezależnie od posunięć Alicji. Twoim zadaniem jest podanie reszty z dzielenia k przez $10^9 + 7$.

Wejście

Pierwszy wiersz standardowego wejścia zawiera dwie dodatnie liczby całkowite n i d oddzielone pojedynczym odstępem, oznaczające odpowiednio liczbę stosów i ograniczenie „podzielnościowe" zabieranych stosów.

Drugi wiersz opisuje stosy i zawiera ciąg n dodatnich liczb całkowitych $a_1, a_2, \ldots, a_n$ pooddzielanych pojedynczymi odstępami, gdzie a_i oznacza liczbę żetonów na i-tym stosie.

Wyjście

Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą, równą liczbie sposobów (modulo $10^9 + 7$), na które Bajtazar może usunąć stosy tak, aby później na pewno zwyciężyć.

Przykład

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

Wyjaśnienie do przykładu: *Bajtazar może zabrać 2 lub 4 stosy. Wygra tylko wtedy, gdy zabierze stosy o licznościach 1 i 4 (może to zrobić na dwa sposoby).*

[1] W Internecie łatwo znaleźć więcej informacji na temat gry Nim, a w szczególności opis strategii wygrywającej w tej grze.

Testy „ocen”:

`1ocen`: *$n = 9$, $d = 2$, wynikiem jest 0,*

`2ocen`: *$n = 12$, $d = 4$,*

`3ocen`: *$n = 30$, $d = 10$, wszystkie stosy mają po 30 żetonów,*

`4ocen`: *$n = 500\,000$, $d = 2$, wszystkie stosy mają wysokość 1.*

Ocenianie

Zestaw testów dzieli się na podane poniżej podzadania. Testy do każdego podzadania składają się z jednej lub większej liczby osobnych grup testów.

We wszystkich podzadaniach zachodzą warunki $n \leqslant 500\,000$, $d \leqslant 10$, $a_i \leqslant 1\,000\,000$. Ponadto sumaryczna liczba żetonów $m = a_1 + a_2 + \ldots + a_n$ jest nie większa niż 10 000 000. Zwróć uwagę, że limit pamięci jest różny dla różnych podzadań.

Podzadanie	Dodatkowe warunki	Limit pamięci	Liczba punktów
1	$n \leqslant 20$, $a_1, \ldots, a_n \leqslant 1000$	*256 MB*	*10*
2	$n \leqslant 10\,000$, $a_1 \ldots, a_n \leqslant 1000$	*256 MB*	*18*
3	$d \leqslant 2$	*256 MB*	*25*
4	*brak*	*256 MB*	*27*
5	*brak*	*64 MB*	*20*

Rozwiązanie

W zadaniu prosiliśmy Uczestników o pomoc Bajtazarowi w wygraniu niewielkiej modyfikacji gry Nim. Aby więc móc rozwiązać ten problem, powinniśmy najpierw poznać kilka informacji na temat tej dość znanej gry matematycznej.

Teoria gier dla początkujących

Najpierw potrzebujemy definicji funkcji alternatywy wykluczającej (`xor`); dalej będziemy oznaczać ją $\oplus$. Aby wyznaczyć $a \oplus b$, czyli wartość `xor` liczb całkowitych a oraz b, zapisujemy obie liczby w systemie binarnym, a następnie dodajemy je pisemnie – z tą różnicą, że nie wykonujemy przeniesień. Poniższy przykład opisuje wykonanie operacji `xor` dla liczb $9 = 1001_{(2)}$ oraz $19 = 10011_{(2)}$:

$$\begin{array}{cccccccc} & & & 1 & 0 & 0 & 1 & \\ \oplus & 1 & 0 & 0 & 1 & 1 & & \\ \hline & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & = 26_{(10)} & \end{array}$$

Odpowiada to oczywiście wstawieniu jedynki w danej kolumnie wyniku, gdy znajdowało się w niej nieparzyście wiele jedynek. W przeciwnym przypadku wstawiamy zero.

Funkcję tę można uogólnić na więcej liczb całkowitych – dokładnie w ten sam sposób definiujemy `xor` wielu liczb $a_1 \oplus a_2 \oplus \ldots \oplus a_n$. Łatwo zauważyć, że działanie $\oplus$ jest łączne oraz przemienne, zatem nie ma znaczenia, w jakiej kolejności obliczamy `xor` wielu liczb.

Operacja $a \oplus b$ jest wbudowana w dostępne języki programowania: w C/C++ jest to `a ^ b`, zaś w Pascalu `a xor b`.

Kto wygra Nima?

Okazuje się, że istnieje bardzo prosty sposób na sprawdzenie, kto wygra rozgrywkę Nima:

Twierdzenie 1. *Niech $a_1, a_2, \ldots, a_n$ będą liczbami żetonów na kolejnych stosach w grze Nim. Gracz rozpoczynający grę może wygrać przy optymalnej grze przeciwnika wtedy i tylko wtedy, gdy $a_1 \oplus a_2 \oplus \ldots \oplus a_n \neq 0$.*

Dowód tego twierdzenia można znaleźć w Internecie, jak i w opracowaniu zadania *Kamyki* z XVI Olimpiady Informatycznej [1].

Jako że w naszym problemie Bajtazar jest drugim graczem, dąży on do tego, by na początku właściwej rozgrywki `xor` wszystkich wysokości stosów wynosił 0. Tak więc zadanie sprowadza się do tego, by policzyć liczbę sposobów usunięcia pewnej liczby stosów z rozgrywki tak, aby:

- liczba usuniętych stosów była podzielna przez d ($d \leqslant 10$),
- pozostał co najmniej jeden stos,
- `xor` wysokości wszystkich stosów wynosił 0.

Rozwiązanie przeglądające wszystkie możliwe podzbiory usuniętych stosów i sprawdzające, czy wszystkie trzy warunki zachodzą, zostało zaimplementowane w pliku `nims12.cpp`. Działa w czasie $O(2^n)$ i przechodzi testy w pierwszym podzadaniu.

Programowanie dynamiczne

Znacznie lepsze rozwiązanie można uzyskać, stosując metodę programowania dynamicznego. Zauważmy, że drugi warunek z poprzedniego podrozdziału można łatwo usunąć, ponieważ wpływa on na wynik tylko wtedy, gdy łączna liczba stosów n jest podzielna przez d. Jeśli zignorujemy drugi warunek, otrzymamy wówczas wynik o 1 za duży (wliczymy bowiem do wyniku sytuację, w której na początku usuniemy wszystkie stosy; jest ona wygrywająca dla Bajtazara). Możemy więc wymazać drugi warunek, a potem na samym końcu w razie potrzeby odjąć od wyniku 1.

Teraz próbujmy dodawać stosy po jednym – dla przykładu, od lewej do prawej. W każdym momencie decydujemy, czy pozostawić stos na planszy, czy go usunąć. Zauważmy, że w każdym momencie wystarczy nam pamiętać jedynie resztę z dzielenia przez d liczby usuniętych stosów oraz `xor` pozostawionych do tej pory stosów.

Mamy więc już szkielet programowania dynamicznego. Przez $dp[i][r][x]$ oznaczymy liczbę sposobów usunięcia spośród pierwszych i stosów podzbioru stosów o liczności

dającej resztę r z dzielenia przez d w taki sposób, że pozostałe z tych i stosów mają `xor` równy x. Jeśli oznaczymy przez A najmniejszą potęgę dwójki przekraczającą największą z wysokości stosów, to zachodzi $0 \leqslant i \leqslant n$, $0 \leqslant r < d$, $0 \leqslant x < A$. Aby nie musieć operować na dużych liczbach, w zadaniu jesteśmy proszeni jedynie o podanie reszty z dzielenia wyniku przez $M = 10^9 + 7$. Wobec tego w tablicy dp wystarczy pamiętać jedynie wartości modulo M.

Dla zerowej liczby stosów ($i = 0$) wszystko jest proste – musi zajść $r = 0$ (usuwamy zero stosów) oraz $x = 0$ (`xor` zerowej liczby stosów to 0). Tak więc $dp[0][0][0] = 1$ oraz $dp[0][\star][\star] = 0$ dla pozostałych wartości w tablicy.

Przypuśćmy, że rozważyliśmy do tej pory $i-1$ stosów i dokładamy i-ty, o wysokości a_i. Chcemy obliczyć $dp[i][r][x]$. Mamy dwie możliwości:

1. Zachowujemy i-ty stos. Liczba usuniętych stosów w porównaniu do poprzedniego stanu pozostaje niezmieniona. Natomiast poprzedni `xor` wysokości pozostawionych stosów musiał wynosić $x \oplus a_i$ (teraz dokładamy do poprzedniego `xor`-a wartość a_i i musi wyjść x; mamy zaś $(x \oplus a_i) \oplus a_i = x \oplus (a_i \oplus a_i) = x$). Do wyniku dodajemy liczbę sposobów dojścia do poprzedniego stanu równą $dp[i-1][r][x \oplus a_i]$.

2. Odrzucamy i-ty stos. Liczba usuniętych stosów krok wcześniej była o jeden mniejsza, za to `xor` się nie zmienił. Dodajemy więc do wyniku wartość $dp[i-1][(r-1) \bmod d][x]$. Zakładamy tożsamość $(-1) \bmod d = d-1$, która jest prawdziwa w matematyce, ale nie zachodzi w większości języków programowania.

Ostatecznie więc

$$dp[i][r][x] = (dp[i-1][r][x \oplus a_i] + dp[i-1][(r-1) \bmod d][x]) \bmod M.$$

Odpowiedź odczytujemy jako $dp[n][0][0]$ (rozpatrzyliśmy n stosów, wyrzuciliśmy liczbę stosów podzielną przez d, pozostałe stosy mają `xor` równy 0). W razie potrzeby odejmujemy od wyniku 1.

Złożoność czasowa i pamięciowa takiego rozwiązania wynosi $O(ndA)$. Niestety, proste obliczenia prowadzą do wniosku, że tablica dp zajmie 390 MB pamięci w drugim podzadaniu. Można jednak mocno ograniczyć zużycie pamięci na jeden z dwóch poniższych sposobów:

1. Zauważmy, że $dp[i][\star][\star]$ zależy tylko od $dp[i-1][\star][\star]$. Możemy więc utrzymywać w pamięci tylko dwie ostatnie podtablice: $dp[i][\star][\star]$ i $dp[i-1][\star][\star]$. Po obliczeniu nowej podtablicy możemy nadpisać poprzednią, ponieważ nie będzie ona nam już do niczego potrzebna.

   Metodę tę można zaimplementować, zmniejszając pierwszy wymiar do 2 elementów i utrzymując w nim resztę z dzielenia wartości i przez 2. Wtedy zamiast dokonywać przejścia $dp[i-1][\star][\star] \to dp[i][\star][\star]$, zerujemy podtablicę $dp[i \bmod 2][\star][\star]$ i wykonujemy przejście $dp[(i-1) \bmod 2][\star][\star] \to dp[i \bmod 2][\star][\star]$. Wynik znajdziemy w pozycji $dp[n \bmod 2][0][0]$.

2. Druga obserwacja jest bardziej wnikliwa. Na początek przypomnijmy, że $(x \oplus a_i) \oplus a_i = x$. Wartości $dp[i][\star][x]$ oraz $dp[i][\star][x \oplus a_i]$ zależą tylko od

$dp[i-1][\star][x]$ i $dp[i-1][\star][x \oplus a_i]$. Możemy więc wykonywać obliczenia niezależnie dla rozłącznych par $(x, x \oplus a_i)$, każdorazowo tworząc dwuwymiarową tablicę dp' o wymiarach $d \times 2$ i następie nadpisując fragment oryginalnej tablicy. W ten sposób możemy całkowicie zignorować pierwszy wymiar tablicy (i).

Obie metody ograniczają zapotrzebowanie na pamięć do $O(dA)$. Implementacja drugiego sposobu znajduje się w pliku `nims3.cpp`. Wystarcza do zdobycia punktów za dwa pierwsze podzadania.

Rozwiązanie wzorcowe

W rozwiązaniu wzorcowym skorzystamy w końcu z ograniczenia $m \leqslant 10^7$ na sumę wysokości stosów $a_1 + a_2 + \ldots + a_n$. Wykorzystamy następujący fakt:

Fakt 1. *Niech* $a_1 \leqslant a_2 \leqslant \ldots \leqslant a_i$. *Wtedy* $a_1 \oplus a_2 \oplus \ldots \oplus a_i < 2a_i$.

Dowód: Wybierzmy potęgę dwójki 2^t taką, że $2^t \leqslant a_i < 2^{t+1}$. Ze względu na to, że wszystkie `xor`-owane liczby są mniejsze niż potęga dwójki 2^{t+1} (gdyż w szczególności nie przekraczają a_i), to ich `xor` też jest mniejszy niż 2^{t+1}. Jednak $2a_i \geqslant 2^{t+1}$. To kończy dowód. ■

Będziemy postępować następująco: posortujmy wszystkie wysokości stosów. Teraz $a_1 \leqslant a_2 \leqslant \ldots \leqslant a_n$. W momencie, gdy liczymy $dp[i][\star][\star]$, znajdujemy potęgę dwójki 2^t taką, że $2^t \leqslant a_i < 2^{t+1}$, a następnie liczymy wszystkie wartości $dp[i][r][x]$ tylko dla $0 \leqslant x < 2^{t+1}$. Nie jesteśmy w stanie uzyskać $x \geqslant 2^{t+1}$ za pomocą dotychczas rozważonych stosów $a_1, \ldots, a_i$, więc dla większych x wartości dp będą równe 0.

Czas obliczenia $dp[i][\star][\star]$ nie przekroczy $2da_i$, ponieważ na podstawie powyższego faktu liczymy maksymalnie $2a_i$ podtablic $dp[i][\star][x]$, zaś czas obliczenia pojedynczej to $O(d)$. Łączny czas działania programowania dynamicznego możemy więc oszacować z góry przez

$$d \cdot (2a_1 + 2a_2 + \ldots + 2a_n) = 2dm = O(dm).$$

Do tego dochodzi czas sortowania wysokości stosów ($O(n \log n)$) oraz zaniedbywalny czas wyznaczania odpowiednich potęg dwójki 2^{t+1}.

Złożoność obliczeniowa jest już w zupełności wystarczająca do zaliczenia wszystkich testów, musimy jednak znów poradzić sobie z problemem ograniczonej pamięci. Możemy zaprząc do tego zadania te same sposoby, które omawialiśmy przy poprzednim rozwiązaniu.

1. Utrzymujemy dwie ostatnie podtablice $dp[i-1][\star][\star]$ oraz $dp[i][\star][\star]$. Jako że pojedyncza liczba 32-bitowa zajmuje 4 bajty, to zużycie pamięci wynosi $2 \cdot \max_d \cdot \max_A \cdot 4\,\mathrm{B} = 2 \cdot 10 \cdot 2^{20} \cdot 4\,\mathrm{B} = 80\,\mathrm{MB}$. Pozwala to na zaliczenie podzadań 1–4 z podwyższonym limitem pamięci. Takie rozwiązanie zaimplementowano w plikach `nims2.cpp`, `nims4.cpp` oraz `nims14.cpp`.

2. Usuwamy pierwszy wymiar tablicy i liczymy $dp[\star][x], dp[\star][x \oplus a_i]$ niezależnie dla każdej pary $(x, x \oplus a_i)$ na podstawie poprzednich wartości tablicy. Zużycie pamięci jest oczywiście dwukrotnie niższe (40 MB). Pozwala to już na zdobycie pełnej punktacji. Przykładowe implementacje znajdują się w plikach `nim.cpp`, `nim1.pas` oraz `nim15.cpp`.

Inne rozwiązanie – przypadek $d \leqslant 2$

Istnieje dość prosty sposób na rozwiązanie trzeciego podzadania. Zauważmy najpierw, że przypadek $d = 2$ można prosto sprowadzić do przypadku $d = 1$, do każdej liczby dodając dodatkowy bit (2^{20}) zawsze równy 1. Wtedy `xor` podzbioru liczb jest równy 0 tylko wtedy, gdy w oryginalnych wysokościach stosów `xor` był równy 0 oraz liczba stosów była parzysta (ten warunek jest wymuszany przez dodatkowy bit). Tak samo jak poprzednio, odejmujemy 1 od wyniku tylko w przypadku, gdy d jest dzielnikiem n. Pozostało więc rozwiązać przypadek $d = 1$.

Udowodnijmy następujące fakty:

Fakt 2. *Niech S będzie zbiorem liczb, które można uzyskać za pomocą* `xor`*-a pewnej (być może zerowej) liczby wysokości stosów. Jeśli $x, y \in S$, to też $x \oplus y \in S$.*

Dowód: Niech $X = \{x_1, \ldots, x_p\}$ będzie podzbiorem stosów wykorzystanych do uzyskania `xor` równego x, zaś $Y = \{y_1, \ldots, y_q\}$ – `xor` równego y. Wtedy oczywiście $x_1 \oplus \ldots \oplus x_p \oplus y_1 \oplus \ldots \oplus y_q = x \oplus y$. Jeśli pewien stos a_i znajduje się zarówno w X, jak i w Y, to w powyższym równaniu wystąpi dwukrotnie i można go pominąć, gdyż $a_i \oplus a_i = 0$. Rozważmy zbiór stosów $Z = \{x : x \text{ jest doiładnie w jednym ze zbiorów } X, Y\}$. Wtedy `xor` wysokości stosów ze zbioru Z jest równy dokładnie $x \oplus y$. ■

Fakt 3. *Każdy z elementów S można uzyskać na tyle samo sposobów.*

Dowód: Prosta indukcja po liczbie stosów i. Na początku $i = 0$ i baza jest oczywista (S jest jednoelementowy, tj. $S = \{0\}$). Weźmy teraz zbiór S uzyskany dla stosów $a_1, \ldots, a_{i-1}$, zaś z niech będzie liczbą sposobów na uzyskanie każdego spośród elementów S. Dołóżmy stos a_i. W opisie dp mamy

$$dp[i][0][x] = dp[i-1][0][x] + dp[i-1][0][x \oplus a_i].$$

1. Jeśli $a_i \in S$, to nie może zajść jednocześnie $x \in S$, $x \oplus a_i \notin S$ (natychmiastowy wniosek z faktu 2) – podobnie nie zajdzie jednocześnie $x \notin S$, $x \oplus a_i \in S$. Tak więc albo oba składniki będą równe z, albo oba będą równe 0. Wobec tego $dp[i][0][\star] \in \{0, 2z\}$.

2. Jeśli $a_i \notin S$, to nie może być jednocześnie $x, x \oplus a_i \in S$ (wtedy $a_i = x \oplus (x \oplus a_i) \in S$). Stąd maksymalnie jeden składnik jest równy z. Tak więc $dp[i][0][\star] \in \{0, z\}$.

W obu przypadkach wszystkie niezerowe wartości dp są równe. ■

Z powyższych faktów i ich dowodów wynika bardzo prosty algorytm – będziemy utrzymywać zbiór S niezerowych pozycji w dp oraz wartość tych pozycji z modulo M. Początkowo $S = \{0\}$, $z = 1$. Dla każdego nowego stosu a_i:

1. jeśli $a_i \in S$, to z rośnie dwukrotnie;

2. jeśli $a_i \notin S$, to do S dodajemy wszystkie wysokości stosów postaci $x \oplus a_i$ dla $x \in S$ (widzimy ze wzoru na dp, że tylko one staną się niezerowe).

Wynikiem jest ostatecznie $z \bmod M$ (minus jeden w razie potrzeby). Rozwiązanie bazujące na podobnych pomysłach zostało zaimplementowane w pliku `nimb11.cpp`. Przechodzi ono wszystkie testy w trzecim podzadaniu.

Miłośnikom matematyki podpowiemy, że S posiada strukturę przestrzeni liniowej nad ciałem rzędu 2 ($1 \oplus 1 = 0$), rozpinanej przez zbiór wysokości stosów. Przedstawiony wyżej algorytm można interpretować jako metodę eliminacji Gaussa na odpowiednio skonstruowanej macierzy. Każdą liczbę a_i zapisujemy w postaci binarnej: $a_i = a_{i,0} \cdot 2^0 + a_{i,1} \cdot 2^1 + \ldots + a_{i,20} \cdot 2^{20}$. Konstruujemy macierz binarną A (z dodawaniem bitów takim, jak w operacji `xor`):

$$A = \begin{pmatrix} a_{1,0} & a_{1,1} & \ldots & a_{1,20} \\ a_{2,0} & a_{2,1} & \ldots & a_{2,20} \\ a_{3,0} & a_{3,1} & \ldots & a_{3,20} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,0} & a_{n,1} & \ldots & a_{n,20} \end{pmatrix}.$$

Okazuje się, że wynikiem jest $2^{n-\operatorname{rk} A}$, gdzie $\operatorname{rk} A$ jest tak zwanym *rzędem* macierzy A, który można łatwo wyznaczyć metodą eliminacji Gaussa.

Rozwiązania błędne

- Uczestnicy mogli zapomnieć o odjęciu 1 w programowaniu dynamicznym w przypadku $d \mid n$. Przykład takiego rozwiązania jest w pliku `nimb5.cpp`. Rozwiązanie nie otrzymuje żadnych punktów, ale błąd jest wyłapywany przez dostępne publicznie testy `ocen`.

- Program zapominający o tym, iż trzeba podać resztę z dzielenia wyniku przez $10^9 + 7$, znajduje się w pliku `nimb6.cpp`. Przechodzi pierwsze podzadanie.

- Bardzo trudnym do wychwycenia przypadkiem (w szczególności podczas testowania z wygenerowanymi losowo testami) była sytuacja, w której musimy odjąć 1 od wyniku podzielnego przez $10^9 + 7$. Reszta przed odjęciem jedynki wynosiła 0. Rozwiązanie wypisujące w tym przypadku -1 zamiast 10^9+6 można znaleźć w pliku `nimb7.cpp`. Nie przechodzi pojedynczego testu w podzadaniu 2.

Jacek Tomasiewicz
Treść zadania, Opracowanie

Bartosz Kostka
Program

Dostępna pamięć: 256 MB.

OI, etap I, 19.10–16.11.2015

Park wodny

Park wodny Aquabajt bierze udział w konkursie na największy basen. Teren parku, na którym zlokalizowane są baseny, ma kształt kwadratu o boku długości n i jest podzielony na n^2 segmentów, z których każdy jest kwadratem o boku długości 1. Każdy z segmentów może być basenikiem albo alejką między basenikami. Baseniki połączone bezpośrednio ze sobą (czyli będące segmentami stykającymi się bokami) tworzą większe baseny. Obecnie w parku wodnym każdy basen ma kształt prostokąta.

Dyrekcja Aquabajtu postanowiła zwiększyć swoje szanse na wygraną w konkursie, przebudowując park. Ze względu na ograniczony czas i fundusze zdecydowano o przekształceniu co najwyżej dwóch segmentów z alejkami na baseniki. Pomóż władzom parku uzyskać basen złożony z maksymalnej liczby baseników. Zakładamy, że po przebudowie największy basen nie musi być już prostokątem.

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną dodatnią liczbę całkowitą n oznaczającą wielkość parku wodnego.

W następnych n wierszach znajduje się dwuwymiarowa mapa parku: każdy z tych wierszy zawiera słowo złożone z n liter. Litera `A` *oznacza segment z alejką, natomiast litera* `B` *oznacza basenik. Możesz założyć, że w opisie znajduje się co najmniej jedna litera* `B`.

Wyjście

Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą oznaczającą wielkość największego basenu, jaki można uzyskać.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*

`5`
`BBBAB`
`BBBAB`
`AAAAA`
`BBABA`
`BBAAB`
	`14`

Testy „ocen”:

`1ocen`: $n = 10$, *tylko jeden basen na planszy,*

`2ocen`: $n = 10$, *cała plansza pokryta basenem,*

`3ocen`: $n = 1000$, *szachownica.*

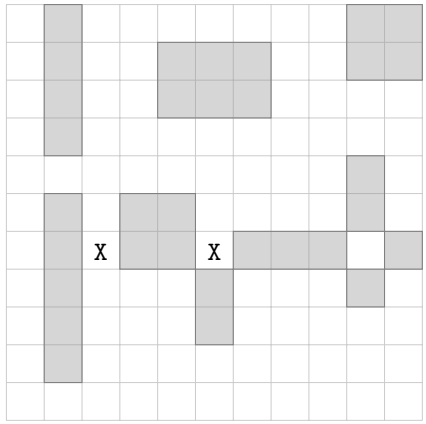
Ocenianie

Zestaw testów dzieli się na następujące podzadania. Testy do każdego podzadania składają się z jednej lub większej liczby osobnych grup testów.

Podzadanie	Warunki	Liczba punktów
1	$n \leqslant 10$	*11*
2	$n \leqslant 50$, *liczba basenów na początku nie przekroczy 80*	*11*
3	$n \leqslant 60$	*22*
4	$n \leqslant 1000$, *na początku każdy basen jest prostokątem* 1×1	*22*
5	$n \leqslant 1000$	*34*

Rozwiązanie

W zadaniu mamy daną mapę podzieloną na n^2 segmentów. Każdy z segmentów może być basenikiem albo alejką między basenikami. Baseniki połączone bezpośrednio ze sobą tworzą większe baseny, które są początkowo prostokątami (zaznaczone na rysunku kolorem szarym). Chcielibyśmy zamienić dwie alejki na baseniki, tak aby utworzyć jak największy basen (który nie musi być już prostokątem).



W optymalnym rozwiązaniu dla powyższego przykładu zbudujemy dwa baseniki oznaczone znakiem `X`, otrzymując jeden basen złożony z $5+1+4+1+2+3 = 16$ baseników.

Rozwiązanie brutalne $O(n^6)$

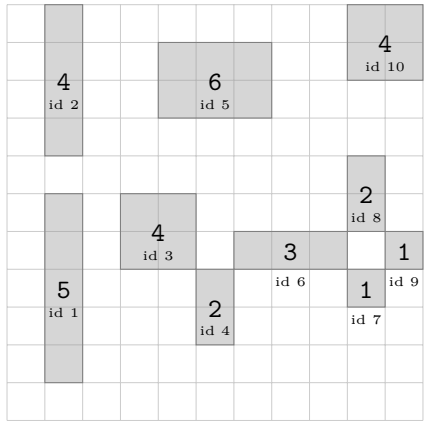
Sprawdzamy każdą możliwą parę segmentów i wybieramy tę, której zamiana na baseniki utworzy największy basen. Aby znaleźć największy basen po takiej zamianie, możemy wyobrazić sobie graf, w którym wierzchołkami są segmenty zawierające baseniki, a krawędzie łączą wierzchołki odpowiadające sąsiednim segmentom, a następnie

użyć algorytmu przeszukiwania grafu w głąb (DFS). Jako że wszystkich par segmentów jest $O(n^4)$, natomiast znalezienie największego basenu zajmuje czas liniowy względem liczby segmentów, złożonością czasową tego rozwiązania jest $O(n^6)$. Warto zwrócić uwagę na przypadek szczególny (patrz test `2ocen`), w którym cały park jest od początku pokryty basenem, więc nie ma żadnego segmentu niebędącego basenikiem (natomiast nie jest możliwe, żeby początkowo tylko jeden segment nie był basenikiem).

Rozwiązanie to zaimplementowane jest w pliku `pars1.cpp`. Za poprawne zaprogramowanie takiego rozwiązania na zawodach można było uzyskać około 10% punktów.

Rozwiązanie wolne $O(n^4)$

Na początku obliczamy wielkość wszystkich basenów w czasie $O(n^2)$, zapisując dla każdego basenu jego identyfikator i rozmiar. Można to zrobić na kilka sposobów, np. używając opisanego wcześniej grafu.



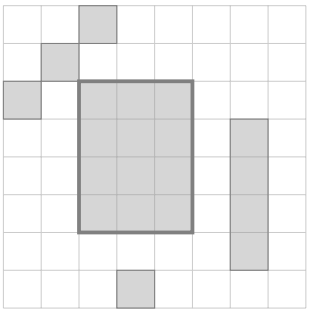
W praktyce informacje te najwygodniej zapisać w każdym segmencie basenu. Następnie rozpatrujemy każdą możliwą parę segmentów i sprawdzamy, które baseny zostaną połączone, gdy te segmenty zmienimy w baseniki. Zauważmy, że każdy segment sąsiaduje z co najwyżej czterema basenami, więc dodanie dwóch segmentów powoduje połączenie tylko stałej liczby basenów. Jako że znamy identyfikatory basenów i ich rozmiary, to w czasie stałym obliczamy całkowity rozmiar powstałego basenu.

Implementacja takiego rozwiązania znajduje się w plikach `pars3.cpp` i `pars9.cpp`. Rozwiązanie tego typu otrzymywało na zawodach około 40% punktów.

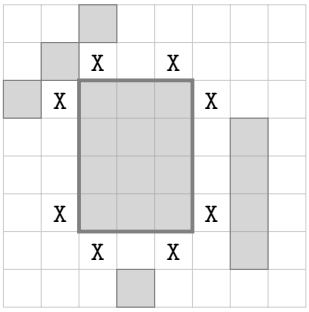
Rozwiązanie wzorcowe $O(n^2)$

Załóżmy na początek, że zmieniana para segmentów sąsiaduje ze sobą. Możemy rozważyć wszystkie takie pary, ponieważ będzie ich tylko $O(n^2)$. Dla każdej pary w czasie stałym sprawdzamy jak wyżej, które baseny zostaną połączone, i zapamiętujemy najlepszy wynik.

Dalej zakładamy, że zmieniane segmenty nie będą ze sobą sąsiadować. Zauważmy, że w tym przypadku w optymalnym rozwiązaniu segmenty te będą sąsiadować z jednym, wybranym basenem. Każdy taki basen będziemy rozpatrywali oddzielnie. Przykładowo, na rysunku poniżej rozważmy basen zaznaczony pogrubioną linią.



Znajdźmy wszystkie segmenty, za pomocą których możemy rozbudować wybrany basen. Liczba takich segmentów będzie liniowa względem obwodu rozpatrywanego basenu. Prawie wszystkie segmenty łączą się z co najwyżej jednym innym basenem. Jedyne segmenty, które mogą się łączyć z dwoma (lub nawet trzema) basenami, znajdują się na rogach. Dokładniej, są po dwa takie segmenty na każdym z rogów, co daje łącznie maksymalnie osiem różnych segmentów (oznaczonych na rysunku znakiem `X`).



Niech k oznacza obwód rozważanego basenu. Wszystkie narożne segmenty rozpatrujemy ze wszystkimi innymi segmentami (również tymi, które nie są na rogach), co daje maksymalnie $8k$ par segmentów. Natomiast z pozostałych segmentów (które nie są na rogach) wybieramy te dwa, które łączą się z największymi różnymi basenami – można je znaleźć w czasie $O(k)$.

Jako że suma obwodów wszystkich basenów nie przekroczy $O(n^2)$, otrzymujemy rozwiązanie wzorcowe, działające w czasie kwadratowym. Przykładową implementację można znaleźć w plikach `par.cpp` i `par2.cpp`.

Zawody II stopnia

opracowania zadań

Jakub Radoszewski — Treść zadania, Opracowanie
Marek Sommer — Program

Dostępna pamięć: 1024 MB. *OI, etap II, dzień próbny, 9.02.2016*

Świąteczny łańcuch

Każdego roku na święta Bożego Narodzenia Bajtazar dekoruje swój dom łańcuchem złożonym z różnokolorowych lampek. Tym razem Bajtazar zamierza samemu dobrać kolory lampek, które będą wchodziły w skład łańcucha. Bajtazar ma w głowie pewne wymagania estetyczne, które streszczają się w tym, że pewne fragmenty łańcucha powinny mieć identyczny układ lampek jak inne. Ponadto żona Bajtazara poprosiła go, aby tegoroczny łańcuch był jak najbardziej urozmaicony, co Bajtazar rozumie tak, że powinno w nim być jak najwięcej różnych kolorów lampek. Pomóż naszemu bohaterowi stwierdzić, ile kolorów lampek będzie musiał kupić.

Wejście

Pierwszy wiersz standardowego wejścia zawiera dwie liczby całkowite n oraz m ($n \geqslant 2$, $m \geqslant 1$) oddzielone pojedynczym odstępem, określające liczbę lampek w planowanym łańcuchu i liczbę wymagań estetycznych Bajtazara. Zakładamy, że kolejne lampki łańcucha będą ponumerowane od 1 do n. Każdy z m kolejnych wierszy opisuje jedno z wymagań za pomocą trzech liczb całkowitych a_i, b_i i l_i ($1 \leqslant a_i, b_i, l_i$; $a_i \neq b_i$; $a_i, b_i \leqslant n - l_i + 1$) oddzielonych pojedynczymi odstępami. Taki opis oznacza, że fragmenty łańcucha złożone z lampek o numerach $\{a_i, \ldots, a_i + l_i - 1\}$ oraz $\{b_i, \ldots, b_i + l_i - 1\}$ powinny być jednakowe. Innymi słowy, lampki o numerach a_i oraz b_i powinny mieć taki sam kolor, podobnie lampki o numerach $a_i + 1$ oraz $b_i + 1$, i tak dalej aż do lampek o numerach $a_i + l_i - 1$ i $b_i + l_i - 1$.

Wyjście

Twój program powinien wypisać na standardowe wyjście jedną dodatnią liczbę całkowitą k oznaczającą maksymalną liczbę różnych kolorów lampek, jakie mogą wystąpić w łańcuchu spełniającym wymagania estetyczne opisane na wejściu.

Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
3
```

natomiast dla danych wejściowych:

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

poprawnym wynikiem jest:

```
1
```

Wyjaśnienie do pierwszego przykładu: *Niech* `a`, `b` *i* `c` *oznaczają trzy różne kolory lampek. Przykładowy łańcuch spełniający wymagania Bajtazara i jego żony to* `abacbababa`.

Testy „ocen”:

`1ocen`: $n = 2000$, $m = 2$; *Bajtazar wymaga, aby fragmenty* $\{1, \ldots, 1000\}$ *i* $\{1001, \ldots, 2000\}$ *były równe oraz aby fragmenty* $\{1, \ldots, 500\}$ *i* $\{501, \ldots, 1000\}$ *były równe; w łańcuchu może wystąpić maksymalnie 500 kolorów lampek.*

`2ocen`: $n = 500\,000$, $m = 499\,900$; *i-te wymaganie jest postaci* $a_i = i$, $b_i = i + 100$, $l_i = 1$; *w łańcuchu może wystąpić maksymalnie 100 kolorów lampek.*

`3ocen`: $n = 80\,000$, $m = 79\,995$, *i-te wymaganie jest postaci* $a_i = i$, $b_i = i + 2$, $l_i = 4$; *w łańcuchu mogą wystąpić maksymalnie dwa kolory lampek.*

`4ocen`: $n = 500\,000$, $m = 250\,000$, *i-te wymaganie jest postaci* $a_i = 1$, $b_i = i + 1$, $l_i = i$; *łańcuch może składać się jedynie z lampek o tym samym kolorze.*

Ocenianie

Zestaw testów dzieli się na podzadania spełniające poniższe warunki. Testy do każdego podzadania składają się z jednej lub większej liczby osobnych grup testów.

Podzadanie	Warunki	Liczba punktów
1	$n, m \leqslant 2000$	*30*
2	$n, m \leqslant 500\,000$, *wszystkie liczby* l_i *są równe 1*	*20*
3	$n, m \leqslant 80\,000$	*30*
4	$n, m \leqslant 500\,000$	*20*

Rozwiązanie

W zadaniu występuje ciąg n *lampek* ponumerowanych od 1 do n. Mamy danych m *wymagań* określających równość kolorów lampek w pewnych fragmentach tego ciągu. Naszym celem jest dobrać kolory wszystkich lampek w ciągu tak, aby spełnione były wszystkie wymagania i ponadto aby liczba użytych kolorów była jak największa. Pula kolorów jest nieograniczona. Wystarczy nam podać liczbę wykorzystanych kolorów.

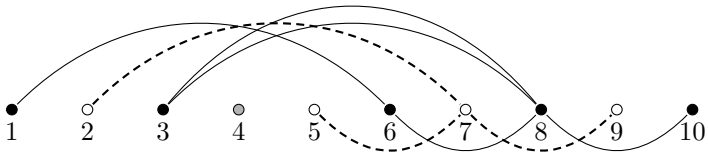
Pierwsze rozwiązanie: graf lampek

Spróbujmy najpierw zaproponować jakiekolwiek rozwiązanie zadania, nie bacząc na to, na ile będzie ono efektywne.

Zauważmy, że każde wymaganie polega na wymuszeniu równości kolorów pewnych par lampek. Dokładniej, wymaganie postaci (a, b, l) oznacza, że lampki o numerach $a + i$ oraz $b + i$, dla każdego $i = 0, \ldots, l - 1$, mają taki sam kolor. Jako że nie interesuje nas na razie efektywność rozwiązania, możemy na starcie rozbić każde takie wymaganie na l pojedynczych wymagań, postaci $(a + i, b + i, 1)$ dla $i = 0, \ldots, l - 1$.

W tym momencie przydatna okazuje się interpretacja grafowa. Skonstruujmy multigraf nieskierowany $G = (V, E)$ o zbiorze wierzchołków V i multizbiorze krawędzi E, w którym wierzchołki odpowiadają lampkom ($V = \{1, \ldots, n\}$), a krawędzie odpowiadają wymaganiom (dla każdego wymagania $(u, v, 1)$ w multizbiorze E umieszczamy

krawędź uv). Przypomnijmy, że multigraf to po prostu graf, w którym dopuszczamy powtórzenia krawędzi. Multigraf G nazwiemy *grafem lampek*.



Rys. 1: Graf lampek G skonstruowany dla pierwszego przykładu z treści zadania. Ma on trzy spójne składowe: pierwsza z nich zawiera wierzchołki $\{1, 3, 6, 8, 10\}$, druga wierzchołki $\{2, 5, 7, 9\}$, a trzecia tylko jeden wierzchołek 4.

Jeśli wierzchołki odpowiadające dwóm lampkom są połączone ścieżką w grafie lampek, to lampki te muszą mieć ten sam kolor. Podzielmy więc graf lampek na spójne składowe. Lampki odpowiadające wierzchołkom z tej samej spójnej składowej muszą mieć ten sam kolor, natomiast lampki odpowiadające wierzchołkom z różnych spójnych składowych mogą mieć różne kolory. Ponieważ zależy nam na wykorzystaniu jak największej liczby kolorów, najbardziej opłaca nam się każdej spójnej składowej przypisać inny kolor lampek. Wynikiem będzie więc liczba spójnych składowych w grafie lampek. Na rysunku 1 zobrazowaliśmy to na przykładzie z treści zadania.

Graf lampek ma $|V| = n$ wierzchołków oraz $|E| = O(nm)$ krawędzi. Spójne składowe w (multi)grafie możemy wyznaczyć w czasie $O(|V| + |E|)$ za pomocą przeszukiwania w głąb (DFS) lub – co w praktyce jest równie szybkie – w czasie $O((|V| + |E|) \log^* |V|)$ z wykorzystaniem struktury danych dla zbiorów rozłącznych (Find-Union)[1]. W ten sposób otrzymujemy rozwiązanie o złożoności czasowej $O(nm)$ (lub $O(nm \log^* n)$), które przechodzi pierwsze podzadanie. Można zauważyć, że rozwiązanie to jest wystarczające również dla drugiego podzadania, jako że warunek $l_i = 1$ gwarantuje, że graf lampek ma tylko m krawędzi.

Lepsze rozwiązanie: graf wymagań

Do zaliczenia kolejnego podzadania wystarczało rozwiązanie dzielące wymagania na fragmenty długości $\Theta(\sqrt{n})$. Można zaproponować kilka takich algorytmów; omówimy tu jeden z nich, który potem będziemy mogli jeszcze usprawnić.

Niech $p = \lfloor \sqrt{n} \rfloor$. *Długością* wymagania (a, b, l) nazwijmy liczbę l. Jeśli wszystkie wymagania mają długość mniejszą niż p, to możemy każde z nich przekształcić na wymagania długości 1 tak jak w poprzednim rozwiązaniu. Jeśli natomiast jakieś wymaganie ma długość $l \geqslant p$, to możemy je podzielić na pewną liczbę wymagań o długości p oraz jedno wymaganie o długości $l \bmod p < p$; tych pierwszych będzie co najwyżej $\frac{n}{p}$, a to ostatnie możemy rozbić na mniej niż p wymagań o długości 1. W ten sposób otrzymujemy łącznie co najwyżej:

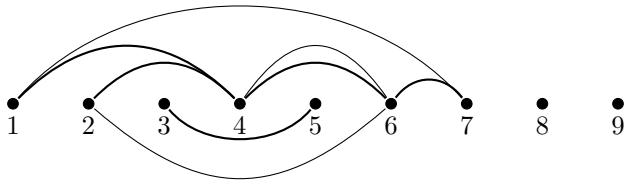
(a) mp wymagań o długości 1 oraz

[1] Więcej o obu sposobach podziału grafu na spójne składowe można przeczytać np. w książce [6].

(b) $m \cdot \frac{n}{p}$ wymagań o długości p.

Z wymaganiami typu (a) poradzimy sobie dokładnie tak jak w poprzednim rozwiązaniu. Skoncentrujmy się teraz na wymaganiach typu (b). Okazuje się, że jeśli jest ich dostatecznie dużo, to muszą one być redundantne, tzn. możemy je wówczas zastąpić niewielką liczbą równoważnych wymagań. W tym celu wprowadzimy pomocniczy multigraf $G' = (V', E')$, który nazwiemy *grafem wymagań*. Graf wymagań ma n wierzchołków: $V' = \{1, \ldots, n\}$, a krawędź $ab \in E'$ odpowiada wymaganiu (a, b, p). Mamy więc $|E'| = O(\frac{nm}{p}) = O(m\sqrt{n})$.

Jeśli dwa wierzchołki $a, b \in V'$ są połączone krawędzią w grafie wymagań, to fragmenty ciągu lampek o długości p zaczynające się na pozycjach a i b są równe. A zatem taka sama własność zachodzi także wtedy, gdy dwa wierzchołki są połączone ścieżką w grafie wymagań, czyli jeśli znajdują się w tej samej spójnej składowej. To oznacza, że do reprezentacji wszystkich wymagań znajdujących się w grafie G' wystarczy wziąć wymagania z dowolnego lasu rozpinającego grafu G'; patrz rys. 2. Taki las rozpinający ma oczywiście co najwyżej $n - 1$ krawędzi, a można go znaleźć chociażby za pomocą wspomnianego już algorytmu DFS w czasie $O(|V'| + |E'|) = O(n + m\sqrt{n})$.



Rys. 2: Graf wymagań G' odpowiadający wymaganiom: $(1, 4, 3)$, $(1, 7, 3)$, $(2, 4, 3)$, $(2, 6, 3)$, $(3, 5, 3)$, $(4, 6, 3)$ (dwukrotnie), $(6, 7, 3)$. Pogrubieniem zaznaczono las rozpinający tego grafu (wierzchołki izolowane możemy pominąć).

W ten sposób uzyskujemy co najwyżej $n - 1$ wymagań o długości p. Wszystkie te wymagania wraz z wymaganiami typu (a) rozbijamy na wymagania o długości 1, otrzymując graf lampek rozmiaru $O((n + m)p) = O((n + m)\sqrt{n})$. Następnie stosujemy do tego grafu poprzednie rozwiązanie. Całość – przetwarzanie grafu wymagań, a następnie grafu lampek – działa w czasie $O((n + m)\sqrt{n})$.

Rozwiązanie wzorcowe

Często bywa, że podział ciągu na fragmenty o długości $\sqrt{n}$ można zastąpić odpowiednią strukturą fragmentów o długościach będących potęgami dwójki, co pozwala zredukować w złożoności czasowej czynnik $\sqrt{n}$ do czynnika $\log n$. Tak samo możemy zrobić także i w tym zadaniu. Kluczowe spostrzeżenie jest takie, że metoda redukcji liczby wymagań za pomocą grafu wymagań działa tak samo dobrze dla dowolnej, ustalonej długości wymagania.

W tym rozwiązaniu będziemy konstruować grafy wymagań dla długości wymagań będących malejącymi potęgami dwójki: 2^k dla $k = \lfloor \log n \rfloor, \ldots, 0$. Przez W_k oznaczymy

zbiór wymagań, jakie pozostały nam do rozważenia przed krokiem k. Dla każdego k spełniony będzie niezmiennik, że wszystkie wymagania ze zbioru W_k są nie dłuższe niż 2^{k+1}. Zauważmy, że dla $k = \lfloor \log n \rfloor$ niezmiennik ten jest spełniony w sposób trywialny.

W kroku odpowiadającym danemu k wydzielimy wszystkie wymagania o długościach co najmniej 2^k. Spośród nich, wymagania dłuższe niż 2^k przekształcimy na pary wymagań o długości 2^k według wzoru:

$$(a, b, l) \quad \rightarrow \quad (a, b, 2^k),\ (a + l - 2^k, b + l - 2^k, 2^k).$$

Następnie z wszystkich wymagań o długości 2^k zbudujemy graf wymagań G'_k. Tak jak w poprzednim rozwiązaniu, zbiór krawędzi grafu G'_k możemy zastąpić lasem rozpinającym, zawierającym co najwyżej $n-1$ krawędzi. W ten sposób przekształcamy zbiór W_k w zbiór W_{k-1}, w którym wszystkie wymagania są długości co najwyżej 2^k. Kontynuujemy to postępowanie dla kolejnych k, a ostateczny wynik uzyskujemy jako liczbę spójnych składowych grafu G'_0.

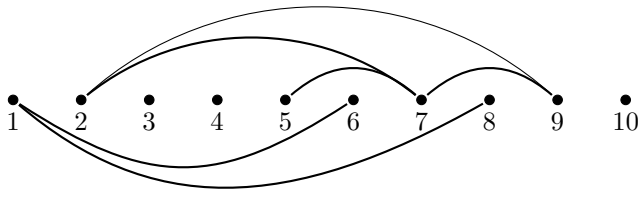
Z powyższej konstrukcji wynika, że w każdym zbiorze W_k jest co najwyżej $n+m-1$ wymagań, z czego co najwyżej $n-1$ wymagań o długości 2^{k+1} oraz co najwyżej m wymagań krótszych. Stąd rozmiar każdego z grafów G'_k to $O(n+m)$. Całe rozwiązanie działa więc w czasie $O((n+m)\log n)$ i w pamięci $O(n+m)$ (nie musimy przechowywać grafów i zapytań dla wcześniej rozważonych k).

Przykład 1. Rozważmy $n = 10$ i zbiór wymagań z pierwszego przykładu z treści zadania z dodanym wymaganiem $(1, 8, 3)$ (które nie zmienia ostatecznego wyniku).

- $k = 3$, $W_3 = \{(1,6,3), (5,7,4), (3,8,1), (1,8,3)\}$.
  Brak wymagań o długości co najmniej 8.
- $k = 2$, $W_2 = \{(1,6,3), (5,7,4), (3,8,1), (1,8,3)\}$.
  Jest tylko jedno wymaganie o długości co najmniej 4. Redukcja za pomocą grafu G'_2 nie przynosi żadnych zmian.
- $k = 1$, $W_1 = \{(1,6,3), (5,7,4), (3,8,1), (1,8,3)\}$.
  Rozbijamy wymagania o długości co najmniej 2:

  $$(1,6,3) \rightarrow (1,6,2),\ (2,7,2), \quad (5,7,4) \rightarrow (5,7,2),\ (7,9,2),$$
  $$(1,8,3) \rightarrow (1,8,2),\ (2,9,2).$$

  Konstruujemy graf G'_1. W lesie rozpinającym nie znajdzie się krawędź $(2, 9)$.

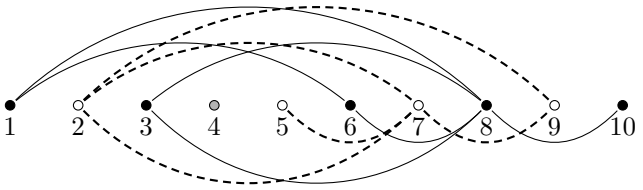


Rozmiar zbioru wymagań wzrasta, ale nie może przekroczyć $n + m - 1$.

- $k = 0$, $W_0 = \{(1,6,2),(1,8,2),(2,7,2),(5,7,2),(7,9,2),(3,8,1)\}$.
  Rozbijamy wymagania ze zbioru W_0 na wymagania o długości 1:

$$(1,6,2) \to (1,6,1),\ (2,7,1), \quad (1,8,2) \to (1,8,1),\ (2,9,1),$$
$$(2,7,2) \to (2,7,1),\ (3,8,1), \quad (5,7,2) \to (5,7,1),\ (6,8,1),$$
$$(7,9,2) \to (7,9,1),\ (8,10,1).$$

  Konstruujemy graf G_0 i dzielimy go na spójne składowe:



Posłowie: Układ równań na słowie

Aby osadzić to zadanie w szerszym kontekście, warto pokusić się o jego interpretację w dziedzinie algorytmów tekstowych. Wówczas jego treść można sformułować równoważnie tak: o pewnym nieznanym słowie (tj. ciągu symboli) długości n wiemy, że określone pary podsłów tego słowa są równe. Chcemy odtworzyć szukane słowo, a jeśli jest wiele możliwości, wyznaczyć taką, w której występuje najwięcej różnych symboli.

Zorientowanemu Czytelnikowi taka interpretacja pozwoli zauważyć pewne związki między tym zadaniem (i jego pierwszym rozwiązaniem) a zadaniem *Równanie na słowach* z V Olimpiady Informatycznej [1]. To jednak nie wszystko. Przykładowo, rozważmy inny problem, w którym mamy odtworzyć nieznane słowo, znając jedynie zbiór długości jego prefikso-sufiksów[2]. Aby go rozwiązać, możemy zastosować nasz algorytm, zadając mu na wejściu wymagania odpowiadające poszczególnym prefikso-sufiksom. Algorytm skonstruuje słowo, które ma te wszystkie długości prefikso-sufiksów. Na koniec musimy jeszcze sprawdzić, czy skonstruowane słowo nie zawiera innej długości prefikso-sufiksów. To jednak możemy łatwo uczynić w czasie $O(n)$, wyznaczając dla tego słowa funkcję prefiksową P i obliczając $P[n], P[P[n]], \ldots$ (patrz książka [6]). Jeśli słowo okaże się nie mieć żadnych innych prefikso-sufiksów, to mamy wynik. W przeciwnym razie zaś możemy od razu stwierdzić, że słowo o żądanym zbiorze prefikso-sufiksów *nie istnieje*, gdyż nasz algorytm konstruuje „najbardziej różnorodne" słowo spełniające podany zbiór równości (czyli jeśli jakieś dwie litery słowa spełniającego zestaw równości mogą być różne, to nasz algorytm rzeczywiście wykorzysta w tym miejscu różne litery), więc jeśli jakieś dwa jego podsłowa są równe, to muszą być także równe w dowolnym słowie spełniającym zadane wymagania.

Podobnie możemy zastosować nasz algorytm do odtworzenia słowa (nad największym alfabetem symboli) o zadanej funkcji prefiksowej, funkcji PREF czy np. o zadanych długościach maksymalnych palindromów. Więcej zastosowań tego algorytmu oraz jego sprytniejszą wersję działającą w czasie $O(n+m)$ można znaleźć w pracy [24].

[2] *Prefikso-sufiksem* słowa nazywamy początkowy fragment słowa równy jego końcowemu fragmentowi tej samej długości.

Marek Cygan
Treść zadania, Opracowanie

Bartosz Kostka
Program

Dostępna pamięć: 512 MB. *OI, etap II, dzień pierwszy, 10.02.2016*

Drogi zmiennokierunkowe

Bajtazar zastanawia się nad przeprowadzką do Bajtowa i chce wynająć tam mieszkanie. Bajtów jest pięknym miastem o licznych zaletach, choć niestety nie należy do nich komunikacja. W mieście jest n skrzyżowań połączonych mniej lub bardziej chaotyczną siecią m dróg. Drogi są bardzo wąskie, więc z przyczyn obiektywnych wszystkie są jednokierunkowe. Jakiś czas temu miejscy specjaliści od komunikacji wpadli na pomysłowe rozwiązanie, które bez konieczności poszerzania dróg umożliwia poruszanie się po nich w różnych kierunkach. A mianowicie, codziennie na wszystkich ulicach zmienia się kierunek poruszania. Innymi słowy, w dni nieparzyste ruch odbywa się zgodnie z oryginalnym skierowaniem ulic, natomiast w dni parzyste ruch na wszystkich ulicach odbywa się w przeciwnych kierunkach.

Bajtazar chce wynająć mieszkanie w takim miejscu, z którego będzie mógł wszędzie łatwo dojechać. Konkretnie, interesuje go mieszkanie przy takim skrzyżowaniu, z którego da się dojechać do każdego innego skrzyżowania **w ciągu jednego dnia** *– w przypadku niektórych skrzyżowań może to być tylko nieparzysty dzień, a w przypadku innych tylko parzysty. Drogą powrotną nie trzeba się przejmować, Bajtazar może wrócić do siebie następnego dnia.*

Mając daną sieć drogową w Bajtowie, wyznacz wszystkie skrzyżowania, które spełniają wymagania Bajtazara.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie liczby całkowite n i m ($n \geqslant 2$, $m \geqslant 1$) oddzielone pojedynczym odstępem, oznaczające liczbę skrzyżowań i liczbę dróg w Bajtowie. Skrzyżowania numerujemy od 1 do n. W kolejnych m wierszach zawarte są opisy dróg: i-ty z tych wierszy zawiera dwie liczby całkowite a_i i b_i ($1 \leqslant a_i, b_i \leqslant n$, $a_i \neq b_i$) oddzielone pojedynczym odstępem, oznaczające, że istnieje jednokierunkowa droga ze skrzyżowania o numerze a_i do skrzyżowania o numerze b_i (tzn. w dni nieparzyste można przejechać tą drogą z a_i do b_i, natomiast w dni parzyste można nią przejechać z b_i do a_i). Każda uporządkowana para (a_i, b_i) wystąpi na wejściu co najwyżej raz.

Wyjście

W pierwszym wierszu standardowego wyjścia należy zapisać jedną liczbę całkowitą k oznaczającą liczbę skrzyżowań spełniających wymagania Bajtazara. W drugim wierszu należy zapisać rosnący ciąg k liczb pooddzielanych pojedynczymi odstępami, oznaczających numery tych skrzyżowań. Jeśli $k = 0$, drugi wiersz powinien pozostać pusty (tj. program może wypisać pusty wiersz albo po prostu go nie wypisywać).

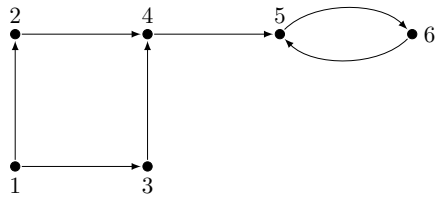
Przykład

Dla danych wejściowych:

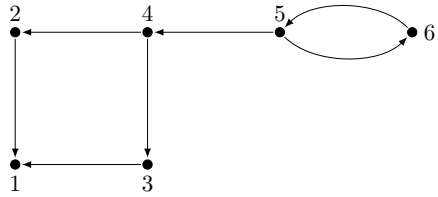
```
6 7
1 2
1 3
2 4
3 4
4 5
5 6
6 5
```

poprawnym wynikiem jest:

```
4
1 4 5 6
```



Sieć dróg w dni nieparzyste.



Sieć dróg w dni parzyste.

Wyjaśnienie do przykładu: *Ze skrzyżowania numer 1 można dojechać do wszystkich innych skrzyżowań w dni nieparzyste. Ze skrzyżowań numer 5 i 6 można dojechać do wszystkich innych skrzyżowań w dni parzyste. Ze skrzyżowania numer 4 do skrzyżowań numer 5 i 6 można dojechać w dni nieparzyste, a do skrzyżowań numer 1, 2 i 3 – w dni parzyste.*

Testy „ocen":

`1ocen`: $n = 10$, $m = 9$; *„ścieżka", w której co druga droga jest skierowana w lewo, a co druga w prawo. Żadne skrzyżowanie nie spełnia wymagań Bajtazara.*

`2ocen`: $n = 100\,000$, $m = 100\,000$, *w dni nieparzyste można przejechać bezpośrednią drogą ze skrzyżowania numer 1 do każdego innego skrzyżowania. Dodatkowo, w dni nieparzyste można przejechać bezpośrednią drogą ze skrzyżowania numer n do skrzyżowania numer 1. Tylko skrzyżowania numer 1 i n spełniają wymagania Bajtazara.*

`3ocen`: $n = 500\,000$, $m = 499\,999$, *„ścieżka"; wszystkie skrzyżowania spełniają wymagania Bajtazara.*

Ocenianie

Zestaw testów dzieli się na podzadania spełniające poniższe warunki. Testy do każdego podzadania składają się z jednej lub większej liczby osobnych grup testów.

Podzadanie	Warunki	Liczba punktów
1	$n, m \leqslant 5000$	*28*
2	$n \leqslant 300\,000$, $m \leqslant 1\,000\,000$; *ze wszystkich skrzyżowań spełniających wymagania Bajtazara można dojechać do każdego innego w dzień nieparzysty*	*29*
3	$n \leqslant 500\,000$, $m \leqslant 1\,000\,000$	*43*

Rozwiązanie

Sieć dróg w Bajtowie możemy przedstawić jako graf skierowany G. Każdy wierzchołek odpowiada skrzyżowaniu, natomiast skierowana krawędź reprezentuje drogę jednokierunkową. Skrzyżowanie x znajduje się w kręgu zainteresowań Bajtazara, jeśli dla każdego innego skrzyżowania y w ciągu jednego dnia da się dojechać ze skrzyżowania x do skrzyżowania y. Innymi słowy, skoro w kolejnych dniach drogi (krawędzie grafu) zmieniają swoją orientację, dla każdego skrzyżowania y musi być spełniony *co najmniej* jeden z warunków:

- w grafie G istnieje ścieżka z wierzchołka x do wierzchołka y,
- w grafie G istnieje ścieżka z wierzchołka y do wierzchołka x.

W problemach dotyczących ścieżek w grafach skierowanych bardzo często przydaje się pojęcie silnie spójnych składowych. Podobnie jest w przypadku naszego zadania. Przypomnijmy, że dwa wierzchołki x i y znajdują się w jednej silnie spójnej składowej wtedy i tylko wtedy, gdy istnieje zarówno ścieżka z wierzchołka x do wierzchołka y, jak i ścieżka z wierzchołka y do wierzchołka x.

Zauważmy, że w przypadku pojedynczej silnie spójnej składowej albo wszystkie jej wierzchołki spełniają wymagania postawione przez Bajtazara, albo też żaden wierzchołek ich nie spełnia. Możemy zatem ściągnąć każdą silnie spójną składową podanego grafu do wierzchołka, tym samym otrzymując *graf silnie spójnych składowych*. Ten krok możemy wykonać w czasie liniowym od rozmiaru grafu za pomocą standardowych metod (więcej o silnie spójnych składowych można znaleźć np. w książkach [4, 6]).

Graf silnie spójnych składowych z natury jest acykliczny, dlatego dla uproszczenia w dalszych rozważaniach będziemy zakładać, że rozważany graf nie ma cykli. Przyjmijmy dodatkowo, że wierzchołki grafu są podane w porządku topologicznym $(v_1, \ldots, v_n)$, tj. każda krawędź prowadzi od wierzchołka o mniejszym indeksie do wierzchołka o większym indeksie. Wygodnie będzie nam wyobrazić sobie, że wierzchołki są uszeregowane na prostej, a krawędzie skierowane są wyłącznie w prawą stronę (rys. 1). Na tej podstawie możemy wywnioskować, że aby wierzchołek v_i spełniał wymagania Bajtazara, muszą być spełnione dwa następujące warunki:

1. dla każdego $j < i$ z wierzchołka v_j istnieje ścieżka do wierzchołka v_i,
2. dla każdego $j > i$ z wierzchołka v_i istnieje ścieżka do wierzchołka v_j.

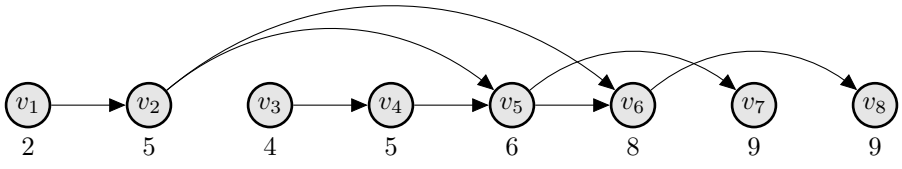
Ze względu na symetrię, jeśli będziemy umieli wyznaczyć wszystkie wierzchołki spełniające pierwszy warunek, to będziemy także umieli wyznaczyć wszystkie wierzchołki spełniające drugi warunek, gdyż drugi warunek jest równoważny pierwszemu w odwrotnie zorientowanym grafie. Aby rozwiązać zadanie wystarczy zatem wyznaczyć wszystkie wierzchołki v_i, do których da się dojść z każdego wierzchołka na lewo od v_i (tj. z każdego wierzchołka o mniejszym indeksie). Wierzchołki takie będziemy nazywać wierzchołkami *dobrymi*, natomiast pozostałe – wierzchołkami *złymi*.

Rozważmy wierzchołek v_i. Jeśli z wierzchołka v_i nie wychodzi krawędź prowadząca do wierzchołka v_{i+1}, to z całą pewnością wierzchołek v_{i+1} jest zły. Aby uogólnić

tę prostą obserwację zdefiniujmy funkcję $r(i)$, której wartość to najmniejszy indeks wierzchołka, do którego prowadzi bezpośrednia krawędź z v_i. Formalnie,

$$r(i) = \min_{(v_i,v_j)\in A} j\,,$$

gdzie A to zbiór krawędzi naszego skierowanego grafu acyklicznego. W przypadku, gdy z wierzchołka v_i nie wychodzi żadna krawędź, ustalamy $r(i) = n + 1$.



Rys. 1: Ilustracja wartości $r(i)$.

Ustalmy pewien wierzchołek v_j. Jeśli istnieje wierzchołek v_i, dla którego $i < j < r(i)$, to wierzchołek v_j jest zły – nie jest on osiągalny z wierzchołka v_i, gdyż wszystkie krawędzie wychodzące z v_i „przeskakują" v_j. Okazuje się, że ten warunek pozwala wykryć *wszystkie* złe wierzchołki.

Lemat 1. Jeśli wierzchołek v_j jest zły, to istnieje wierzchołek v_i, taki że $i < j < r(i)$.

Dowód: Skoro wierzchołek v_j jest zły, istnieje pewien wierzchołek v_a, taki że $a < j$ i z v_a nie da się dojść do v_j. Naszym celem jest znalezienie wierzchołka v_i spełniającego $i < j$ oraz $r(i) > j$. Wykonajmy następujący spacer w grafie, rozpoczynając w wierzchołku v_a. Dopóki znajdujemy się w wierzchołku na lewo od v_j, wykonujemy przejście z aktualnego wierzchołka v_a do $v_{r(a)}$, czyli do najbliższego wierzchołka, do którego możemy wykonać ruch. Spacer może zakończyć się z jednego z dwóch powodów: (i) z pewnego wierzchołka v_i na lewo od v_j przeszliśmy na prawo od v_j, czyli $i < j < r(i)$, lub też (ii) z pewnego wierzchołka v_i, gdzie $i < j$, nie wychodzi żadna krawędź, co oznacza, że $r(i) = n + 1$. W obydwóch przypadkach otrzymujemy $i < j < r(i)$. ■

Powyższy lemat oraz poprzedzająca go obserwacja prowadzą nas do rozwiązania.

Na początku obliczamy graf silnie spójnych składowych i sortujemy jego wierzchołki w porządku topologicznym. Obydwa te kroki możemy wykonać w czasie $O(n+m)$. Następnie w czasie $O(n+m)$ obliczamy wartości $r(i)$, po czym znajdujemy wierzchołki, których indeksy należą do sumy przedziałów $(i, r(i))$. Ten ostatni krok zrealizować możemy w czasie $O(n)$, przeglądając przedziały dla rosnących wartości i i pamiętając największą dotychczas napotkaną wartość $r(i)$. Tym samym znajdziemy wszystkie złe wierzchołki w grafie silnie spójnych składowych, co pozwoli nam zidentyfikować złe wierzchołki w oryginalnym grafie. Jeśli powtórzymy wszystkie kroki dla grafu z odwróconą orientacją krawędzi, otrzymamy rozwiązanie problemu Bajtazara.

Tomasz Syposz	Jakub Radoszewski	Aleksander Łukasiewicz
Treść zadania	Opracowanie	Program

Dostępna pamięć: 32 MB. *OI, etap II, dzień pierwszy, 10.02.2016*

Zająknięcia

Bitek zapadł ostatnio na dziwną chorobę: strasznie się jąka, a przy tym jedyne słowa, które wypowiada, to liczby. Jego starszy brat, Bajtek, zauważył jednak dziwną powtarzalność w zająknięciach Bitka. Podejrzewa, że Bitek tak naprawdę udaje, żeby nie chodzić do szkoły i móc więcej grać na komputerze. Bajtek nie może przez to uczyć się programowania i jest z tego powodu bardzo smutny. Postanowił więc zdemaskować młodszego brata i liczy, że w nagrodę będzie miał tyle czasu na programowanie, ile dusza zapragnie.

Opiszmy formalnie podejrzenia Bajtka. Załóżmy, że mamy dany ciąg liczb A.

- **Podciągiem** *A nazywamy ciąg powstały przez wyrzucenie z A dowolnych wyrazów, np. $1, 1, 7, 5$ jest podciągiem ciągu $1, 3, 1, 7, 6, 6, 5, 5$.*
- **Zająknięciem** *A nazywamy podciąg A, który składa się z ustawionych po kolei par takich samych wyrazów, np. $1, 1, 1, 1, 3, 3$ jest zająknięciem ciągu $1, 2, 1, 2, 1, 2, 1, 3, 3$.*

Mając dane dwie wypowiedzi Bitka jako ciągi liczb, pomóż Bajtkowi stwierdzić, jaka jest długość najdłuższego zająknięcia, które występuje w każdym z tych ciągów, a nagroda Cię nie ominie.

Wejście

Pierwszy wiersz standardowego wejścia zawiera dwie liczby całkowite n oraz m ($n, m \geqslant 2$) oddzielone pojedynczym odstępem, oznaczające długości ciągów A i B, które reprezentują wypowiedzi Bitka. W drugim wierszu wejścia znajduje się n liczb całkowitych $a_1, a_2, \ldots, a_n$ oddzielonych pojedynczymi odstępami, czyli kolejne wyrazy ciągu A ($1 \leqslant a_i \leqslant 10^9$). W trzecim wierszu wejścia znajduje się m liczb całkowitych $b_1, b_2, \ldots, b_m$ oddzielonych pojedynczymi odstępami, czyli kolejne wyrazy ciągu B ($1 \leqslant b_i \leqslant 10^9$).

Wyjście

Twój program powinien wypisać na standardowe wyjście jedną nieujemną liczbę całkowitą oznaczającą długość najdłuższego wspólnego zająknięcia ciągów A i B. Jeśli ciągi nie mają żadnego wspólnego zająknięcia, poprawnym wynikiem jest 0.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*

`7 9`
`1 2 2 3 1 1 1`
`2 4 2 3 1 2 4 1 1`
	`4`

Wyjaśnienie do przykładu: *Szukanym ciągiem jest 2, 2, 1, 1.*

Testy „ocen”:

`1ocen`: *$n = 5$, $m = 4$, wszystkie liczby to 42,*

`2ocen`: *$n = 9$, $m = 13$, ciągi to słowa* `OLIMPIADA` *i* `INFORMATYCZNA` *zapisane w kodzie ASCII,*

`3ocen`: *$n = 15\,000$, $m = 15\,000$, ciąg A składa się z par rosnących liczb $(1, 1, 2, 2, 3, 3, \ldots, 7500, 7500)$, natomiast B powstał w wyniku odwrócenia A,*

`4ocen`: *$n = 10\,000$, $m = 5000$, oba ciągi składają się z par naprzemiennych liczb 13 oraz 37 $(13, 37, 13, 37, \ldots)$.*

Ocenianie

Zestaw testów dzieli się na podzadania spełniające poniższe warunki. Testy do każdego podzadania składają się z jednej lub większej liczby osobnych grup testów.

Podzadanie	Warunki	Liczba punktów
1	$n, m \leqslant 2000$	*30*
2	$n, m \leqslant 15\,000$ *i każda liczba w każdym ciągu występuje co najwyżej dwa razy*	*28*
3	$n, m \leqslant 15\,000$	*42*

Rozwiązanie

W zadaniu dane są dwa ciągi liczb $A = (a_1, \ldots, a_n)$ i $B = (b_1, \ldots, b_m)$. W opisie rozwiązania zamiast o wartościach elementów ciągów wygodniej nam będzie mówić o ich kolorach, tak więc np. a_i oznaczać będzie kolor i-tego elementu ciągu A. Naszym celem jest znaleźć najdłuższe wspólne zająknięcie ciągów A i B, czyli najdłuższy ciąg kolorów złożony z parami powtarzających się elementów, który jest podciągiem każdego z ciągów A i B. Co istotne, w odpowiedzi wystarczy podać długość najdłuższego wspólnego zająknięcia.

Najdłuższy wspólny podciąg

Nasze zadanie ewidentnie ma związek z problemem znajdowania najdłuższego wspólnego podciągu dwóch ciągów. Rozważania zacznijmy więc od przypomnienia klasycznego rozwiązania tego problemu za pomocą programowania dynamicznego (patrz np. książka [6]). Wyznacza się w nim dwuwymiarową tablicę NWP rozmiaru $(n+1)\times(m+1)$, taką że $NWP[i, j]$ oznacza długość najdłuższego wspólnego podciągu ciągów $a_1, \ldots, a_i$ oraz $b_1, \ldots, b_j$. Szukanym wynikiem jest oczywiście $NWP[n, m]$. Mamy następującą zależność rekurencyjną:

$$NWP[i,j] = \begin{cases} 0 & \text{jeżeli } i = 0 \text{ lub } j = 0 \\ NWP[i-1, j-1] + 1 & \text{jeżeli } a_i = b_j \\ \max(NWP[i-1, j], NWP[i, j-1]) & \text{w przeciwnym przypadku.} \end{cases}$$

Z zależności tej wynika następujący algorytm o złożoności $O(nm)$, w którym wypełniamy kolejne pola tablicy $NWP[i,j]$ dla $i = 0,\ldots,n$ oraz $j = 0,\ldots,m$.

```
1:  procedure ObliczNWP
2:  begin
3:    for j := 0 to m do NWP[0,j] := 0;
4:    for i := 1 to n do begin
5:      NWP[i,0] := 0;
6:      for j := 1 to m do
7:        if a[i] = b[j] then
8:          NWP[i,j] := NWP[i-1,j-1] + 1
9:        else
10:         NWP[i,j] := max(NWP[i-1,j], NWP[i,j-1]);
11:   end
12:   return NWP[n,m];
13: end
```

Warto dodać, że choć złożoność pamięciowa powyższego algorytmu to także $O(nm)$, to można ją łatwo zredukować do $O(n+m)$. Wystarczy mianowicie pamiętać tylko dwa ostatnie wiersze tablicy, tj. $NWP[i-1,\star]$ oraz $NWP[i,\star]$. W tym celu można po prostu we wszystkich odwołaniach do pól tablicy NWP w powyższym pseudokodzie na pierwszej współrzędnej brać resztę z dzielenia przez 2.

Pierwsze rozwiązanie

Nasze pierwsze rozwiązanie będzie naśladować opisaną wyżej metodę. Niech $NWZ[i,j]$ oznacza długość najdłuższego wspólnego zająknięcia ciągów $a_1,\ldots,a_i$ oraz $b_1,\ldots,b_j$.

Jeśli do wyznaczenia tablicy $NWZ[i,j]$ chcielibyśmy zastosować metodę z powyższego pseudokodu, to przypadki brzegowe oraz przypadek $a_i \neq b_j$ pozostaną bez zmian. Natomiast w sytuacji, gdy $a_i = b_j$, powinniśmy do zająknięcia dołożyć jeszcze jeden element koloru a_i. Odpowiada to wybraniu pary elementów: $a_{i'} = a_i$ dla $i' < i$ oraz $b_{j'} = b_j$ dla $j' < j$, co zwiększa długość zająknięcia o dwa elementy. Może się też okazać, że w tym przypadku któryś z szukanych elementów $a_{i'}$ oraz $b_{j'}$ nie istnieje lub znajduje się bardzo wcześnie w ciągu; wówczas lepiej jest wybrać, podobnie jak w przypadku $a_i \neq b_j$, większą z wartości NWZ dla krótszych fragmentów ciągów.

Sprecyzujmy, że jako indeks i' – jeśli istnieje – najlepiej wybrać indeks wskazujący najbliższy wcześniejszy element o tym samym kolorze co a_i. Rzeczywiście, gdyby w najdłuższym wspólnym zająknięciu w ciągu A występowała jako para kolejnych jednokolorowych elementów para $a_{i'}$ oraz a_i, a istniałby indeks i'' taki że $i' < i'' < i$ oraz $a_{i''} = a_i$, to moglibyśmy równie dobrze zamiast $a_{i'}$ wziąć do zająknięcia element $a_{i''}$. Podobnie rzecz ma się w przypadku ciągu B.

Dla danego indeksu $i \in \{1,\ldots,n\}$ przez $prev_A[i]$ oznaczmy indeks najbliższego wcześniejszego elementu o kolorze a_i w ciągu A. Jeśli element $prev_A[i]$ nie istnieje, przyjmujemy, że $prev_A[i] = 0$. Wprowadźmy też analogiczne oznaczenie $prev_B[j]$ dla elementu b_j w ciągu B. Pozwala nam to zapisać następujący pseudokod wyznaczania tablicy $NWZ[i,j]$.

```
 1: procedure ObliczNWZ
 2: begin
 3:   for j := 0 to m do NWZ[0, j] := 0;
 4:   for i := 1 to n do begin
 5:     NWZ[i, 0] := 0;
 6:     for j := 1 to m do begin
 7:       if a[i] = b[j] and prev_A[i] > 0 and prev_B[j] > 0 then
 8:         NWZ[i, j] := NWZ[prev_A[i] − 1, prev_B[j] − 1] + 2
 9:       else
10:         NWZ[i, j] := 0;
11:       NWZ[i, j] := max(NWZ[i, j], NWZ[i − 1, j], NWZ[i, j − 1]);
12:     end
13:   end
14:   return NWZ[n, m];
15: end
```

Algorytm ten ma złożoność czasową i pamięciową $O(nm)$, pod warunkiem, że będziemy mieli do dyspozycji tablice $prev_A[i]$ oraz $prev_B[j]$. Tablice te można wyznaczyć zupełnie siłowo w czasie $O(n^2 + m^2)$, co było wystarczające w tym zadaniu.

Całe rozwiązanie ma zatem złożoność czasową $O((n + m)^2)$ i pamięciową $O(nm)$. Przykładowe implementacje można znaleźć w plikach `zajb2.cpp`, `zajb3.cpp` i `zajb7.pas`. Tego typu rozwiązania przechodziły pierwsze podzadanie, natomiast w pozostałych podzadaniach przekraczały limit pamięciowy. Rzeczywiście, dla maksymalnych wartości n i m z zadania (tj. 15 000) tablica NWZ musiałaby mieć $(15\,000)^2 = 225\,000\,000$ komórek, co nie ma możliwości zmieścić się w 32 MB pamięci.

Dodajmy jeszcze, że implementację rozwiązania o złożoności czasowej $O(n^2m^2)$, które nie zapamiętuje wartości tablic $prev_A[i]$ oraz $prev_B[j]$, tylko każdorazowo sprawdza wszystkich kandydatów na elementy $a_{i'}$ i $b_{j'}$, można znaleźć w pliku `zajb4.cpp`.

Rozwiązanie wzorcowe

W naszym zadaniu nie jest niestety tak łatwo zmniejszyć złożoność pamięciową rozwiązania jak w przypadku problemu najdłuższego wspólnego podciągu. Moglibyśmy zastosować sztuczkę z traktowaniem pierwszej współrzędnej modulo 2, gdyby nie konieczność odwoływania się do wartości $NWZ[prev_A[i] - 1, prev_B[j] - 1]$, która teoretycznie może znajdować się w zupełnie dowolnym miejscu tablicy NWZ.

Przyjrzyjmy się jednak dokładniej, które komórki tablicy występują w tych kłopotliwych odwołaniach w poszczególnych momentach obliczeń. Gdy w zewnętrznej pętli rozpatrujemy konkretny indeks i, wartość $prev_A[i]$ jest oczywiście ustalona i wskazuje na wcześniejszy element tego samego koloru co i; niech będzie to kolor c. W tym obrocie pętli interesują nas tylko indeksy $prev_B[j]$ dla j takich, że $b_j = c$. Elementy o tych indeksach w ciągu B mają także kolor c.

Gdy w algorytmie przechodzimy do kolejnych indeksów i, takich że $a_i \neq c$, to interesujące nas indeksy $prev_B[j]$ są zatem zupełnie inne. Natomiast kiedy napotkamy pierwszy indeks $i' > i$, taki że $a_{i'} = c$, będziemy mieli $prev_A[i'] = i$, a interesujące nas wartości $prev_B[j]$ będą znów odpowiadały elementom koloru c.

Wprowadźmy do rozwiązania pomocniczą tablicę jednowymiarową *memo* indeksowaną parametrem j. Zauważmy, że gdybyśmy podczas rozpatrywania indeksu i zapamiętali, jako $memo[j]$, wartości $NWZ[i-1, j-1]$ dla wszystkich indeksów j takich że $b_j = c$, to wówczas, rozpatrując indeks i', moglibyśmy jako $NWZ[prev_A[i']-1, prev_B[j]-1]$ wziąć dokładnie wartość $memo[prev_B[j]]$. Mamy wówczas gwarancję, że obliczenia dla indeksów pomiędzy i a i' nie nadpiszą pól $memo[j]$ dla indeksów j, na których w ciągu B znajdują się elementy koloru c.

Ostatecznie musimy wprowadzić w pseudokodzie stosunkowo niewielkie zmiany.

```
 1: procedure ObliczNWZ2
 2: begin
 3:   for j := 0 to m do NWZ[0, j] := memo[j] := 0;
 4:   for i := 1 to n do begin
 5:     NWZ[i mod 2, 0] := 0;
 6:     for j := 1 to m do begin
 7:       if a[i] = b[j] and prev_A[i] > 0 and prev_B[j] > 0 then
 8:         NWZ[i mod 2, j] := memo[prev_B[j]] + 2
 9:       else
10:         NWZ[i mod 2, j] := 0;
11:       NWZ[i mod 2, j] := max(NWZ[i mod 2, j], NWZ[(i - 1) mod 2, j],
12:                              NWZ[i mod 2, j - 1]);
13:     end
14:     for j := 1 to m do
15:       if a[i] = b[j] then
16:         memo[j] := NWZ[(i - 1) mod 2, j - 1];
17:   end
18:   return NWZ[n mod 2, m];
19: end
```

Otrzymane rozwiązanie ma ewidentnie złożoność pamięciową $O(n + m)$, a jego złożoność czasowa nie uległa zmianie. Implementację tego typu rozwiązania można znaleźć w plikach `zaj.cpp`, `zaj3.pas`, `zaj4.cpp` i `zaj6.cpp`.

Dodatkowe optymalizacje

W naszych rozwiązaniach tablice $prev_A$ i $prev_B$ wyznaczaliśmy, odpowiednio, w czasie $O(n^2)$ i $O(m^2)$. Programujący w języku C++ mogli obliczyć je efektywniej np. z użyciem kontenera `map`. Faktycznie, przeglądając ciąg A za pomocą indeksu i, dla każdego koloru wystarczy pamiętać w strukturze danych ostatnio napotkany indeks elementu tego koloru. Wówczas $prev_A[i]$ wyznaczamy jako indeks zapamiętany w strukturze danych pod kolorem a_i, a odtąd w strukturze pamiętamy dla tego koloru indeks i. Złożoność pojedynczej operacji na kontenerze `map` to $O(\log n)$, więc cały proces zajmuje czas $O(n \log n)$. Tak samo w czasie $O(m \log m)$ można wyznaczyć elementy tablicy $prev_B[j]$. W identycznej złożoności tablice te można wypełnić także bez użycia wspomnianego kontenera – w przypadku tablicy $prev_A[i]$ wystarczy przejrzeć wszystkie pary postaci (a_i, i), posortowawszy je niemalejąco po współrzędnych.

Warto też wspomnieć o pewnym prostym usprawnieniu, które można było zastosować na początku każdego z omawianych rozwiązań. Otóż jeśli elementy jakiegoś koloru występują w którymś z ciągów A, B mniej niż dwukrotnie, to możemy usunąć wszystkie elementy tego koloru z obydwu ciągów. Optymalizację tę łatwo przeprowadzić w czasie $O((n+m)\log(n+m))$. Choć nie zmniejsza ona pesymistycznej złożoności czasowej rozwiązania, w przypadku wielu typów testów pozwala istotnie zmniejszyć długość ciągów.

Rozwiązanie drugiego podzadania

W drugim podzadaniu mieliśmy gwarancję, że w obu ciągach każdy z kolorów występuje co najwyżej dwukrotnie. Przy tym założeniu zadanie można rozwiązać w inny sposób, stosując metodę programowania dynamicznego z liniową liczbą stanów.

Dla każdego koloru elementu, który w każdym z ciągów występuje dwukrotnie (pozostałe kolory możemy w ogóle odrzucić na podstawie opisanej powyżej optymalizacji), znajdujemy indeks i późniejszego wystąpienia elementu tego koloru w ciągu A i zapamiętujemy dla niego indeks $odp[i]$ późniejszego wystąpienia elementu tego koloru w ciągu B. Pozostałe pola tablicy odp inicjujemy zerami. Dla każdego indeksu i w ciągu A wyznaczymy, w tablicy $NWZ'[i]$, długość najdłuższego wspólnego zająknięcia ciągów A i B, które kończy się w ciągu A elementem a_i. Widzimy, że $NWZ'[i] > 0$ tylko dla indeksów i takich że $odp[i] > 0$.

Aby obliczyć $NWZ'[i]$, wystarczy przejrzeć wszystkie wcześniejsze pozycje j i sprawdzić, czy kończące się na nich najdłuższe wspólne zająknięcia (jeśli istnieją) można przedłużyć o elementy znajdujące się pod indeksami $prev_A[i]$, i w ciągu A oraz te pod indeksami $prev_B[odp[i]]$, $odp[i]$ w ciągu B. W ten sposób uzyskujemy poniższy pseudokod.

```
1:  procedure ObliczNWZ'
2:  begin
3:    for i := 0 to n do begin
4:      NWZ'[i] := 0;
5:      if odp[i] > 0 then
6:        for j := 0 to prev_A[i] - 1 do
7:          if odp[j] < prev_B[odp[i]] then
8:            NWZ'[i] := max(NWZ'[i], NWZ'[j] + 2);
9:    end
10:   return max{NWZ'[1], ..., NWZ'[n]};
11: end
```

Indeksy $odp[i]$ można wyznaczyć siłowo; nie będziemy się na ten temat szczegółowo rozpisywać. Otrzymane rozwiązanie ma złożoność czasową $O((n+m)^2)$ i pamięciową $O(n+m)$. Przykładową implementację można znaleźć w pliku `zajb1.cpp`.

Dodajmy na koniec, że podzadanie 2 można także rozwiązać efektywniej, bo w czasie $O((n+m)\log(n+m))$. Rozwiązanie to wykorzystuje drzewo przedziałowe i jest podobne do rozwiązania problemu najdłuższego wspólnego podciągu w przypadku, gdy każdy kolor występuje w każdym z ciągów co najwyżej raz. Dopracowanie jego szczegółów pozostawiamy Czytelnikowi.

Tomasz Idziaszek
Treść zadania, Opracowanie

Jakub Radoszewski, Piotr Smulewicz
Program

Dostępna pamięć: 512 MB. *OI, etap II, dzień drugi, 11.02.2016*

Arkanoid

Arkanoid *jest grą komputerową, w której za pomocą ruchomej paletki odbija się poruszającą się po planszy piłeczkę. Piłeczka ta zbija znajdujące się na planszy klocki, a celem gry jest zbicie ich wszystkich. Ci, którzy grali w tę grę, wiedzą, jak frustrujące i czasochłonne może być zbicie kilku ostatnich klocków. Warto mieć zatem program, który dla początkowego ustawienia planszy obliczy czas potrzebny na wygranie gry. Na potrzeby tego zadania zakładamy dla uproszczenia, że gracz gra bezbłędnie, tzn. że zawsze odbije piłeczkę i uczyni to środkiem paletki.*

Plansza ma długość m i wysokość n, przy czym m jest nieparzyste, a m i n są względnie pierwsze[1]*. Wprowadzamy na niej prostokątny układ współrzędnych: lewy dolny róg planszy ma współrzędne $(0, 0)$, a prawy górny współrzędne (m, n). Dla uproszczenia zakładamy, że piłeczka ma pomijalny rozmiar, a paletka pomijalną grubość. Paletka porusza się po prostej $y = 0$, natomiast początkowo piłeczka znajduje się w punkcie $(\frac{m}{2}, 0)$ i jej początkowy wektor prędkości to $(-\frac{1}{2}, \frac{1}{2})$.*

W momencie, w którym piłeczka dotknie paletki, brzegu planszy lub dowolnego klocka na planszy, odbija się idealnie sprężyście. Dodatkowo, dotknięty klocek zostaje zbity i znika z planszy. Po ilu jednostkach czasu wszystkie klocki zostaną zbite?

Wejście

W pierwszym wierszu standardowego wejścia znajdują się trzy liczby całkowite m, n i k ($m, n, k \geqslant 1$, $k \leqslant nm - 1$) oddzielone pojedynczymi odstępami, oznaczające wymiary planszy oraz początkową liczbę klocków na planszy. W kolejnych k wierszach znajdują się opisy klocków: i-ty z tych wierszy zawiera dwie liczby całkowite x_i i y_i ($1 \leqslant x_i \leqslant m$, $1 \leqslant y_i \leqslant n$) oddzielone pojedynczym odstępem, oznaczające, że na planszy znajduje się klocek, który jest prostokątem o przeciwległych wierzchołkach $(x_i - 1, y_i - 1)$ oraz (x_i, y_i). Możesz założyć, że na polu opisanym przez $x_i = \frac{m+1}{2}$, $y_i = 1$ nie znajduje się żaden klocek.

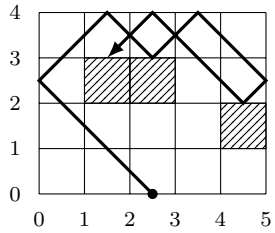
Wyjście

W jedynym wierszu standardowego wyjścia należy wypisać jedną liczbę całkowitą oznaczającą liczbę jednostek czasu, po których wszystkie klocki na planszy zostaną zbite.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
5 4 3 2 3 5 2 3 3	22

[1] Dwie liczby całkowite dodatnie są względnie pierwsze, jeśli ich największym wspólnym dzielnikiem jest 1.



Testy „ocen”:

`1ocen`: $m = 5$, $n = 4$, $k = 2$, *całkiem duży wynik,*

`2ocen`: $m = 11$, $n = 10$, *klocki tworzą szachownicę niedotykającą brzegów planszy,*

`3ocen`: $m = 99\,999$, $n = 100\,000$, *klocki na polach* $(\frac{m-1}{2}, 2)$, $(\frac{m-5}{2}, 2)$, $(\frac{m-9}{2}, 2), \ldots$,

`4ocen`: $m = 99\,999$, $n = 100\,000$, *jeden klocek na polu* $(1, 1)$, *duży wynik.*

Ocenianie

Zestaw testów dzieli się na podzadania spełniające poniższe warunki. Testy do każdego podzadania składają się z jednej lub większej liczby osobnych grup testów.

Podzadanie	Warunki	Liczba punktów
1	$m, n \leqslant 100$, $k \leqslant 1000$	*25*
2	$n, m \leqslant 100\,000$, $k \leqslant 50$	*25*
3	$m, n, k \leqslant 100\,000$, *żaden klocek nie styka się bokiem z innymi klockami ani z brzegiem planszy (klocki mogą się stykać wierzchołkami)*	*25*
4	$m, n, k \leqslant 100\,000$	*25*

Rozwiązanie

Zadanie *Arkanoid* jest dość dobrym przykładem, jak można, wychodząc od prostego rozwiązania, aczkolwiek działającego wolno, stopniowo je poprawiać, aż do uzyskania rozwiązania efektywnego czasowo. W związku z tym w poniższym opracowaniu całkiem dokładnie prześledzimy proces konstrukcji rozwiązania. Zaczniemy od zaimplementowania symulacji procesu opisanego w zadaniu, a następnie będziemy dokładać kolejne obserwacje. Ponieważ dużo trudności w tym zadaniu było natury implementacyjnej, pozwolimy sobie na sporą ilość pseudokodu w opisie.

Symulujemy ruch piłeczki w najprostszy sposób

Wygodniej nam będzie operować na liczbach całkowitych, więc na początek wszystkie współrzędne pomnożymy przez 2. Po tej operacji lewy dolny róg planszy ma współrzędne $(0, 0)$, prawy górny współrzędne $(2m, 2n)$, a piłeczka zaczyna w punkcie o współrzędnych $(m, 0)$ i porusza się z początkowym wektorem prędkości $(-1, 1)$.

Piłeczka odbija się idealnie sprężyście, zatem jej tor jest łamaną, której każdy odcinek jest nachylony pod kątem 45° do obu brzegów planszy, a wektor prędkości jest równy $(\pm 1, \pm 1)$. Współrzędne (x, y) punktów całkowitych, w których może znaleźć się piłeczka, są takie, że jedna z liczb x, y jest parzysta, a druga nieparzysta. Istotnie: zaczynamy z punktu $(m, 0)$ dla nieparzystego m i w każdym ruchu każda z współrzędnych zmienia się o jeden (w górę lub w dół), co zmienia jej parzystość. Co więcej, wszystkie punkty odbicia piłeczki mają współrzędne całkowite.

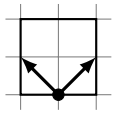
Najprostsze rozwiązanie polega na bezpośredniej symulacji ruchu piłeczki w kolejnych jednostkach czasu. Na początek spróbujmy zrobić to dla pustej planszy.

Tor piłeczki na planszy jest zdeterminowany przez jej położenie (x, y) oraz wektor prędkości (dx, dy). W dalszej części opisu czwórkę (x, y, dx, dy) będziemy nazywać *pozycją* piłeczki. W każdej jednostce czasu położenie piłeczki zostaje uaktualnione o wektor prędkości i zmienia się na $(nx, ny) = (x+dx, y+dy)$. Po tej operacji musimy sprawdzić, czy piłeczka odbije się od ściany (brzegu planszy). Jeśli po pojedynczym kroku dotyka ona pionowego brzegu planszy (czyli nx jest równe 0 lub $2m$), to jej składowa dx prędkości zmienia się na przeciwną. Analogicznie, gdy piłeczka dotyka poziomego brzegu (dla ny równego 0 lub $2n$), jej składowa dy się zmienia. Zauważmy, że zawsze zmienia się co najwyżej jedna składowa, jako że piłeczka nigdy nie znajdzie się w rogu planszy (gdyż obie współrzędne każdego z rogów są parzyste). Na zmiennej *czas* będziemy przechowywać liczbę jednostek czasu, które upłynęły od początku ruchu piłeczki. Poniższa funkcja Ruszaj implementuje ruch piłeczki w pojedynczej jednostce czasu.

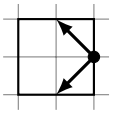
```
1:  function Ruszaj
2:  begin
3:    x := x + dx;
4:    y := y + dy;
5:    czas := czas + 1;
6:    if x = 0 or x = 2m then { odbicie od pionowego brzegu }
7:      dx := −dx
8:    else if y = 0 or y = 2n then { odbicie od poziomego brzegu }
9:      dy := −dy;
10: end
```

Teraz dodajmy obsługę klocków. Zakładamy tak jak w treści zadania, że plansza jest podzielona na $m \times n$ pól, a niektóre z nich mogą zawierać klocki. Jeśli po pojedynczym kroku piłeczka nie dotyka brzegu planszy, to musimy sprawdzić, czy dotyka boku któregoś z klocków. Innymi słowy, czy piłeczka w trakcie następnego ruchu wjechałaby na pole, na którym znajduje się klocek. Pole to będziemy reprezentować przez współrzędne (sx, sy) jego środka.

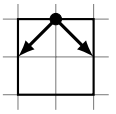
Każde pole ma 4 punkty, w których piłeczka może dotknąć jego brzegu (co, uwzględniając kierunki, daje 8 możliwych pozycji; patrz też rys. 1). Punkty, w których



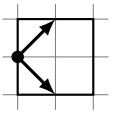
$(sx, sy-1, 1, 1)$
$(sx, sy-1, -1, 1)$



$(sx+1, sy, -1, 1)$
$(sx+1, sy, -1, -1)$



$(sx, sy+1, -1, -1)$
$(sx, sy+1, 1, -1)$



$(sx-1, sy, 1, -1)$
$(sx-1, sy, 1, 1)$

Rys. 1: Osiem pozycji, w których może znaleźć się piłeczka wjeżdżająca na pole o środku (sx, sy).

piłeczka dotyka pionowego boku, mają współrzędną x parzystą, a punkty, w których dotyka poziomego boku pola, mają współrzędną x nieparzystą. Funkcja SrodekPola wyznacza środek pola, którego dotyka piłeczka o położeniu w punkcie (x, y) i wektorze prędkości (dx, dy) skierowanym do wewnątrz tego pola:

```
1:  function SrodekPola(x, y, dx, dy)
2:  begin
3:    if x mod 2 = 0 then begin { piłeczka dotyka pionowego boku pola }
4:      sx := x + dx;
5:      sy := y;
6:    end else begin { piłeczka dotyka poziomego boku pola }
7:      sx := x;
8:      sy := y + dy;
9:    end
10:   return (sx, sy);
11: end
```

Na podstawie powyższych rozważań możemy też napisać ogólną funkcję Odbij, która uaktualnia prędkość piłeczki przy odbiciu zarówno od ścian, jak i klocków:

```
1: function Odbij
2: begin
3:   if x mod 2 = 0 then { pionowy bok }
4:     dx := −dx
5:   else { poziomy bok }
6:     dy := −dy;
7: end
```

Jeśli piłeczka właśnie ma odbić się od ściany, to funkcja SrodekPola zwróci współrzędne punktu leżącego poza planszą. Poniższa funkcja sprawdza, czy mamy do czynienia z takim przypadkiem:

```
1: function Sciana(x, y, dx, dy)
2: begin
3:   (sx, sy) := SrodekPola(x, y, dx, dy);
4:   return sx < 0 or sx > 2m or sy < 0 or sy > 2n;
5: end
```

Aby móc szybko sprawdzać, gdzie na planszy są klocki, użyjemy najprostszej metody i będziemy stan całej planszy trzymać w dwuwymiarowej tablicy. Przyjmujemy, że jeśli na polu o środku (sx, sy) znajduje się klocek, to $klocek[sx, sy] =$ **true**:

```
1: function InicjujPlansze
2: begin
3:   for sx := 1 to 2m − 1 step 2 do
4:     for sy := 1 to 2n − 1 step 2 do
5:       klocek[sx, sy] := false;
6:   for i := 1 to k do
7:     klocek[2x_i − 1, 2y_i − 1] := true;
8: end
```

Jesteśmy już gotowi do napisania funkcji Ruszaj, uwzględniającej klocki:

```
1: function Ruszaj
2: begin
3:   x := x + dx;
4:   y := y + dy;
5:   czas := czas + 1;
6:   (sx, sy) := SrodekPola(x, y, dx, dy);
7:   if Sciana(x, y, dx, dy) then { odbicie od ściany }
8:     Odbij
9:   else if klocek[sx, sy] then begin { odbicie od klocka }
10:    klocek[sx, sy] := false;
11:    k := k − 1;
12:    Odbij;
13:  end
14: end
```

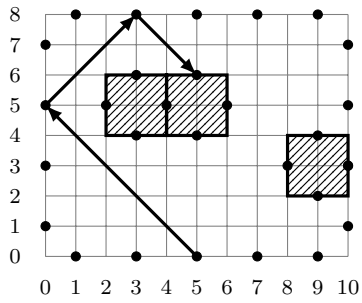
Główna pętla programu symulującego ruch piłeczki jest bardzo prosta. Inicjujemy $czas := 0$ oraz $(x, y, dx, dy) := (m, 0, -1, 1)$, a następnie wywołujemy funkcję Ruszaj dopóty, dopóki liczba niezbitych klocków k jest dodatnia.

Przeanalizujmy czas działania tego algorytmu. Inicjowanie planszy kosztuje czas $O(mn + k)$. Przy niesprzyjającym układzie klocków, potencjalnie możemy odwiedzić większość planszy, by zbić kolejny klocek, więc liczbę jednostek czasu pomiędzy kolejnymi odbiciami od klocków oszacujemy przez $O(mn)$. Zatem cały algorytm działa w czasie $O(mnk)$. Rozwiązanie to zostało zapisane w pliku `arks1.cpp`; przechodzi ono pierwsze podzadanie.

Przyspieszamy ruch piłeczki bez odbić

Zauważmy, że jeśli na dużej planszy jest mało klocków, to często wielokrotnie będziemy wywoływali funkcję Ruszaj bez wywoływania funkcji Odbij. Spróbujmy przyspieszyć to tak, aby funkcja Ruszaj poruszała piłeczką aż do następnego miejsca, w którym nastąpi odbicie (od ściany lub od klocka).

Przypomnijmy, że pozycja piłeczki jest czwórką (x, y, dx, dy). O zbiorze pozycji, które może przyjąć piłeczka, możemy też myśleć jak o zbiorze wierzchołków pewnego



Rys. 2: Plansza z zaznaczonymi wierzchołkami grafu: każdy pogrubiony punkt (x, y) oznacza cztery wierzchołki-pozycje $(x, y, \pm 1, \pm 1)$. Trzy zaznaczone krawędzie to $graf[5, 0, -1, 1] = (0, 5, 5)$, $graf[0, 5, 1, 1] = (3, 8, 3)$ i $graf[3, 8, 1, -1] = (5, 6, 2)$.

grafu. Na potrzeby poprzedniego algorytmu dwa wierzchołki-pozycje połączone były (skierowaną) krawędzią, jeśli piłeczka mogła przejść z jednej pozycji do drugiej w pojedynczej jednostce czasu. Taki graf był duży, bo zawierał aż $O(mn)$ wierzchołków i krawędzi. Ponieważ ruch piłeczki jest zdeterminowany przez jej pozycję, to z każdego wierzchołka wychodzi co najwyżej jedna krawędź. (Piszemy „co najwyżej jedna", a nie „dokładnie jedna", gdyż z niektórych pozycji kontynuowanie ruchu piłeczki nie jest możliwe; są to pozycje odbić, w których pozycja piłeczki zmienia się za sprawą funkcji Odbij).

Teraz zbudujemy ważony graf, w którym dwa wierzchołki-pozycje połączone będą krawędzią o wadze ℓ, jeśli piłeczka może przejść z jednej pozycji do drugiej w ℓ jednostkach czasu, nie wykonując przy tym żadnego odbicia. Interesować nas będą jedynie wierzchołki-pozycje, w których może dojść do odbicia, czyli które leżą na brzegach planszy i na bokach klocków. Dzięki temu nasz graf będzie miał jedynie $O(m+n+k)$ wierzchołków i tyle samo krawędzi (rys. 2).

Dla ustalenia notacji przyjmijmy, że $graf[x, y, dx, dy] = (nx, ny, \ell)$ oznacza, że piłeczka startująca z pozycji (x, y, dx, dy) musi przebyć ℓ kroków, żeby wykonać następne odbicie; będzie ono na pozycji (nx, ny, dx, dy).

Symulacja ruchu piłeczki do następnego odbicia może wyglądać następująco:

```
 1: function Ruszaj
 2: begin
 3:   (x, y, ℓ) := graf[x, y, dx, dy];
 4:   czas := czas + ℓ;
 5:   if Sciana(x, y, dx, dy) then { odbicie od ściany }
 6:     Odbij
 7:   else begin { odbicie od klocka }
 8:     UsunKlocek(x, y, dx, dy);
 9:     k := k − 1;
10:     Odbij;
11:   end
12: end
```

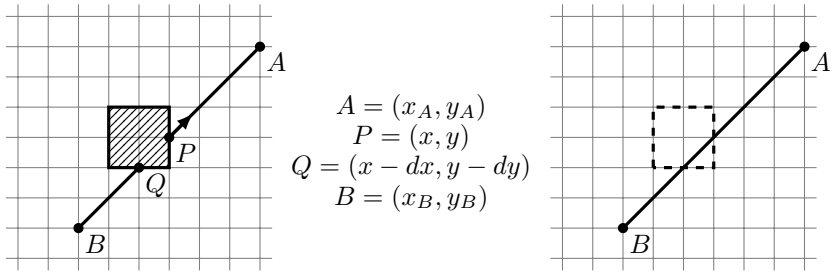
Przejdźmy teraz do kwestii budowania grafu, czyli wyznaczenia wartości *graf*. Załóżmy, że chcemy obliczyć wartość dla wierzchołka (x, y, dx, dy). Poruszamy się zatem z punktu (x, y) w kierunku (dx, dy), aż nie natrafimy na taką pozycję (nx, ny, dx, dy), za którą jest już ściana lub pole z klockiem. Liczba wykonanych ruchów to oczywiście wartość bezwzględna różnicy $nx - x$. Zauważmy, że skoro wierzchołek (x, y, dx, dy) też leżał na brzegu planszy lub przy klocku (bo tylko takie nas interesują), to wyznaczyliśmy jednocześnie wartość dla wierzchołka $(nx, ny, -dx, -dy)$, bo wystarczy rozważyć ruch piłeczki do tyłu.

```
1:  function DodajKrawedz(x, y, dx, dy)
2:  begin
3:    nx := x;
4:    ny := y;
5:    while not Sciana(nx, ny, dx, dy) and
6:              not klocek[SrodekPola(nx, ny, dx, dy)] do begin
7:      nx := nx + dx;
8:      ny := ny + dy;
9:    end
10:   ℓ := abs(nx − x);
11:   graf[x, y, dx, dy] := (nx, ny, ℓ);
12:   graf[nx, ny, −dx, −dy] := (x, y, ℓ);
13: end
```

Aby zbudować cały graf, należy wywołać powyższą funkcję dla wszystkich wierzchołków. Z tego co powiedzieliśmy o symetrii, wystarczy to zrobić dla punktów, z których idziemy przekątną w górę.

```
1:  function InicjujGraf
2:  begin
3:    for x := 1 to 2m − 1 step 2 do begin
4:      DodajKrawedz(x, 0, 1, 1);
5:      DodajKrawedz(x, 0, −1, 1);
6:    end
7:    for y := 1 to 2n − 1 step 2 do begin
8:      DodajKrawedz(0, y, 1, 1);
9:      DodajKrawedz(2m, y, −1, 1);
10:   end
11:   for i := 1 to k do begin
12:     sx := 2x_i − 1;
13:     sy := 2y_i − 1;
14:     DodajKrawedz(sx, sy + 1, 1, 1);
15:     DodajKrawedz(sx, sy + 1, −1, 1);
16:     DodajKrawedz(sx − 1, sy, −1, 1);
17:     DodajKrawedz(sx + 1, sy, 1, 1);
18:   end
19: end
```

Zauważmy, że w sytuacji, gdy jakiś klocek dotyka bokiem ściany lub dwa klocki sąsiadują bokiem, w naszym grafie będą istnieć krawędzie o wadze 0. Co prawda



Rys. 3: Scalanie pary krawędzi przechodzących przez usunięty klocek.

nie używamy ich w funkcji Ruszaj, ale przy usuwaniu klocków będziemy korzystali z faktu, że te krawędzie istnieją. Często dość łatwo przeoczyć tego rodzaju przypadki szczególne; z tego też powodu w testach w jednym z podzadań mieliśmy gwarancję, że takie przypadki nie występują.

Uaktualnianie grafu po usunięciu klocka

Po każdym usunięciu klocka struktura naszego grafu przestaje opisywać sytuację na planszy. Musimy zatem odpowiednio zaktualizować graf. Dla ustalonego klocka mamy osiem pozycji na jego brzegu, które były wierzchołkami, natomiast po usunięciu klocka już nimi nie będą.

Zobaczmy to na przykładzie: ustalmy pewien klocek oraz pozycję (x, y, dx, dy) na jego brzegu; oznaczmy $P = (x, y)$ (rys. 3). Niech $graf[x, y, dx, dy] = (x_A, y_A, \ell_A)$, zatem kolejne odbicie nastąpi na pozycji (x_A, y_A, dx, dy) po ℓ_A jednostkach czasu. Dopóki klocek leży na planszy, to oczywiście $graf[x_A, y_A, -dx, -dy] = (x, y, \ell_A)$. Jeśli jednak klocek zostanie usunięty, to startując z wierzchołka $(x_A, y_A, -dx, -dy)$, piłeczka nie zatrzyma się w wierzchołku $(x, y, -dx, -dy)$, ale przeleci dalej do wierzchołka $(x - dx, y - dy, -dx, -dy)$, a ponieważ jest to wierzchołek, który znajdował się na zbitym klocku, wobec tego piłeczka poruszy się z niego aż do $graf[x - dx, y - dy, -dx, -dy] = (x_B, y_B, \ell_B)$. Wierzchołki $(x, y, -dx, -dy)$ oraz $(x - dx, y - dy, -dx, -dy)$ mogą zostać usunięte z grafu (bo w nich już nie będzie odbić), natomiast wierzchołkom $(x_A, y_A, -dx, -dy)$ oraz (x_B, y_B, dx, dy) należy uaktualnić wychodzące z nich krawędzie, tak jak w poniższej funkcji.

```
1: function ScalKrawedz(x, y, dx, dy)
2: begin
3:     (x_A, y_A, ℓ_A) := graf[x, y, dx, dy];
4:     (x_B, y_B, ℓ_B) := graf[x − dx, y − dy, −dx, −dy];
5:     ℓ := ℓ_A + 1 + ℓ_B;
6:     graf[x_A, y_A, −dx, −dy] := (x_B, y_B, ℓ);
7:     graf[x_B, y_B, dx, dy] := (x_A, y_A, ℓ);
8: end
```

Przez każdy klocek przechodzą cztery pary krawędzi, które muszą być scalone, wobec tego przy usuwaniu klocka należy uwzględnić je wszystkie:

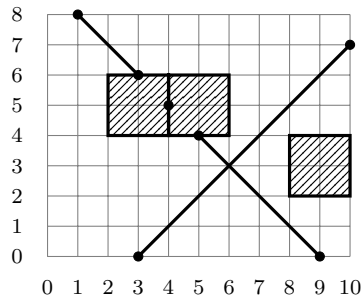
```
1: function UsunKlocek(x, y, dx, dy)
2: begin
3:     (sx, sy) := SrodekPola(x, y, dx, dy);
4:     klocek[sx, sy] := false;
5:     ScalKrawedz(sx + 1, sy, 1, 1);
6:     ScalKrawedz(sx, sy + 1, −1, 1);
7:     ScalKrawedz(sx − 1, sy, −1, −1);
8:     ScalKrawedz(sx, sy − 1, 1, −1);
9: end
```

W powyższym kodzie jest jeszcze jedna subtelność, na którą należy zwrócić uwagę. Powiedzieliśmy, że wierzchołki na brzegach klocka można usunąć z grafu, ale jest jeden wyjątek. Bowiem tuż po usunięciu klocka piłeczka znajduje się w jednym z wierzchołków, które były na brzegu klocka, więc potrzebujemy pamiętać jego wartość *graf*, gdyż będziemy z niej korzystać w następnym wywołaniu funkcji Ruszaj. Jednak to nie jest problem, gdyż nie musimy fizycznie usuwać tej wartości z tablicy.

Choć nowy graf jest dużo mniejszy, to nadal tworzymy go, przechodząc przez całą planszę, co zabiera czas $O(mn + k)$. Dużo szybsze jest jednak wyznaczanie kolejnego klocka do zbicia – jako że nadal potencjalnie możemy się odbić od wielu ścian, czas działania programu pomiędzy kolejnymi odbiciami od klocków oszacujemy przez $O(m+n)$. Zatem cały algorytm działa w czasie $O(mn+(m+n)k)$. Jego kod jest w pliku `arks2.cpp`. Pomimo poprawy czasu działania, program wciąż zalicza tylko pierwsze podzadanie. Widać, że faza, którą musimy przyspieszyć, to tworzenie grafu.

Przyspieszamy tworzenie grafu

Wyznaczenie zbioru wszystkich wierzchołków grafu jest proste – problem leży w szybkim wyznaczeniu krawędzi pomiędzy nimi. Zauważmy, że każda krawędź leży na prostej nachylonej pod kątem 45°, zatem jeśli dwa wierzchołki o współrzędnych (x_1, y_1),



Rys. 4: Plansza i pogrubione dwie przykładowe przekątne: dla $x - y = 3$ zawierająca punkty $(3, 0)$ i $(10, 7)$ oraz dla $x + y = 9$ zawierająca punkty $(1, 8)$, $(3, 6)$, $(4, 5)$, $(4, 5)$, $(5, 4)$ i $(9, 0)$. Utworzone z nich krawędzie to $graf[3, 0, 1, 1] = (10, 7, 7)$, $graf[1, 8, 1, -1] = (3, 6, 2)$, $graf[4, 5, 1, -1] = (4, 5, 0)$, $graf[5, 4, 1, -1] = (9, 0, 4)$ oraz ich symetryczne odpowiedniki.

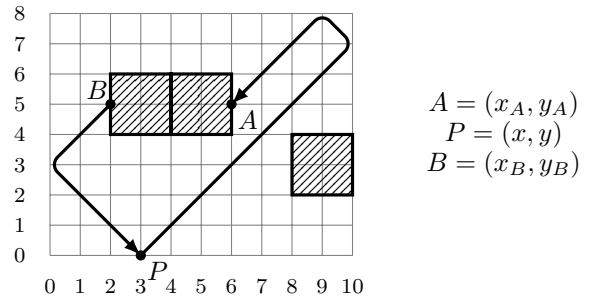
(x_2, y_2) są połączone krawędzią, to $x_1 - y_1 = x_2 - y_2$ (dla prostych nachylonych w prawo) lub $x_1 + y_1 = x_2 + y_2$ (dla prostych nachylonych w lewo). Ustalmy zatem pewną prostą, np. przechodzącą przez punkty (x, y), dla których $x + y = \text{const}$. Aby wyznaczyć krawędzie dla wierzchołków na tej prostej, wystarczy posortować je w kolejności występowania na prostej (czyli innymi słowy w kolejności współrzędnych x), a następnie połączyć krawędzią kolejne pary wierzchołków (bo pierwszy z nich będzie na dolnym lub lewym brzegu planszy, kolejne pary na bokach klocków, a ostatni na górnym lub prawym brzegu planszy). Przykład jest na rys. 4.

Zatem stworzenie grafu wymaga posortowania $O(m + n + k)$ par liczb, co możemy zrobić w czasie $O((m + n + k) \log(m + n))$. Warto też nadmienić, że już nie potrzebujemy kosztownej tablicy *klocek*; położenia klocków są jedynie wykorzystywane przy generowaniu wierzchołków. Zatem cały algorytm będzie działał w czasie $O((m + n + k) \log(m + n) + (m + n)k)$. Przykładowa implementacja jest w pliku `arks3.cpp`. Przechodzi ona dwa podzadania i uzyskuje połowę punktów za zadanie.

Przyspieszamy odbicia od brzegów planszy

Teraz w czasie stałym wyznaczamy położenie piłeczki w momencie kolejnego odbicia. Ale czasem możemy długo odbijać się od brzegów planszy, aby dostać się do klocka. Naturalna jest więc próba modyfikacji algorytmu, żeby w czasie stałym znajdować kolejne odbicie od klocka.

Rozważmy pewien wierzchołek naszego grafu, który odpowiada pozycji (x, y, dx, dy) na brzegu planszy. Istnieje dokładnie jedna krawędź wchodząca do tej pozycji oraz jedna krawędź wychodząca z pozycji otrzymanej po zastosowaniu funkcji Odbij (rys. 5). Zatem za każdym razem, gdy piłeczka będzie przechodzić przez tę pozycję, będzie to robić tymi dwiema krawędziami. Nic nie stoi zatem na przeszkodzie, abyśmy skleili te dwie krawędzie w jedną (oczywiście o wadze będącej sumą ich wag). Jeśli tak zrobimy dla wszystkich pozycji na brzegu planszy (poza pozycją początkową), to uzyskamy graf o $O(k)$ wierzchołkach i tylu krawędziach. Co



Rys. 5: Kompresja krawędzi przy brzegu planszy. Do pozycji $(3, 0, 1, -1)$ wchodzi krawędź z $(2, 5, -1, -1, 5)$, a z pozycji $(3, 0, 1, 1)$ wychodzi krawędź do $(6, 5, -1, -1, 11)$. Po kompresji będziemy mieli $graf[2, 5, -1, -1] = (6, 5, -1, -1, 16)$.

więcej, jeśli będziemy umieli poprawiać ten graf po usunięciu klocka, to znajdziemy rozwiązanie zadania po przejściu zaledwie k krawędzi.

Praktyczna realizacja tego pomysłu wymaga jednak pewnej uwagi. Po pierwsze, musimy dokonać drobnej zmiany w definicji grafu: ponieważ teraz pojedyncza krawędź może opisywać trasę piłeczki, która zawiera odbicia od ścian, nie mamy gwarancji, że wypadkowa pozycja piłeczki będzie miała ten sam kierunek. Zatem musimy go pamiętać w strukturze: $graf[x, y, dx, dy] = (nx, ny, ndx, ndy, \ell)$ będzie oznaczać, że piłeczka startująca z pozycji (x, y, dx, dy) przebędzie ℓ kroków, aby wykonać następne odbicie od klocka na pozycji (nx, ny, ndx, ndy). Stosowne zmiany w wartościach *graf* należy wprowadzić w funkcji generującej początkowy graf oraz w funkcji ScalKrawedz.

Poniższa funkcja do kompresji krawędzi jest bardzo podobna do funkcji ScalKrawedz:

```
1:  function KompresujKrawedz(x, y, dx, dy)
2:  begin
3:    (x_A, y_A, dx_A, dy_A, ℓ_A) := graf[x, y, dx, dy];
4:    (x_B, y_B, dx_B, dy_B, ℓ_B) := graf[x, y, −dy, dx];
5:    ℓ := ℓ_A + ℓ_B;
6:    graf[x_A, y_A, −dx_A, −dy_A] := (x_B, y_B, dx_B, dy_B, ℓ);
7:    graf[x_B, y_B, −dx_B, −dy_B] := (x_A, y_A, dx_A, dy_A, ℓ);
8:  end

9:  function KompresujGraf
10: begin
11:   for x := 1 to 2m − 1 step 2 do begin
12:     if x ≠ m then
13:       KompresujKrawedz(x, 0, 1, 1);
14:     KompresujKrawedz(x, 2n, −1, −1);
15:   end
16:   for y := 1 to 2n − 1 step 2 do begin
17:     KompresujKrawedz(0, y, 1, −1);
18:     KompresujKrawedz(2m, y, −1, 1);
19:   end
20: end
```

Niestety, dość łatwo przegapić jeszcze jedną zmianę, którą należy wprowadzić do funkcji UsunKlocek. Do tej pory cztery pary krawędzi przecinające klocek mogliśmy scalać niezależnie od siebie, gdyż na pewno były rozłączne. Teraz jednak, gdy w grafie nie ma wierzchołków na brzegach planszy, może wystąpić krawędź, która ma początek na jednym boku klocka, a koniec na innym (lub nawet tym samym) boku tego samego klocka. Wobec tego może się zdarzyć tak, że następny ruch będzie się odbywał po scalonej krawędzi, która przecina aktualnie usuwany klocek.

To rodzi potencjalny problem z usuwanym wierzchołkiem, którego będziemy używać w kolejnym wywołaniu funkcji Ruszaj. Musimy zagwarantować, że wartość *graf* dla tego wierzchołka nie zostanie zmieniona po scaleniu jego krawędzi. Na szczęście istnieje prosty sposób na poradzenie sobie z tym kłopotem: wystarczy że para krawędzi przecinająca klocek, do której należy ten wierzchołek, będzie uaktualniana *jako ostatnia*. Po szczegóły odsyłamy do implementacji.

Procedura zmniejszania grafu poprzez usuwanie wierzchołków na brzegach planszy i kompresję przechodzących przez nie krawędzi działa w czasie $O(m+n)$ i jest zdominowana przez procedurę tworzenia grafu w czasie $O((m+n+k)\log(m+n))$. Z kolei wyznaczanie odbić od kolejnych klocków działa w końcu w czasie stałym, zatem cała symulacja wykona się w czasie $O(k)$. Daje to ostateczną złożoność $O((m+n+k)\log(m+n))$. Takie rozwiązanie zapisano w pliku `ark.cpp` i zdobywa ono komplet punktów. Dla pełności zauważmy, że stosując sortowanie przez zliczanie, można jeszcze zmniejszyć tę złożoność do $O(m+n+k)$.

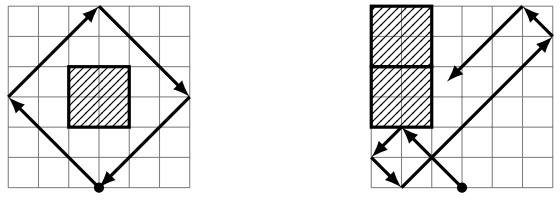
Dlaczego piłeczka zbije wszystkie klocki?

Pozostała do rozważenia jeszcze jedna kwestia. Otóż, do tej pory przyjmowaliśmy założenie, że piłce uda się zbić wszystkie klocki, tzn. że algorytm w pewnym momencie się zatrzyma. Nie jest jednak zupełnie oczywiste, dlaczego tak się stanie. Dla przykładu, dla planszy rozmiarów 3×3 z jednym klockiem na środku, piłeczka nigdy nie zbije tego klocka. Mogą też zdarzyć się układy klocków, w których jedynie część klocków zostanie zbita (rys. 6).

Jednakże wymiary tej planszy są niezgodne z założeniem z treści zadania, które mówi, że wymiary te muszą być liczbami względnie pierwszymi. W istocie dla planszy zgodnych z tym założeniem, piłeczka zawsze zbije wszystkie klocki. W tym rozdziale poczynimy pewne obserwacje, które zbliżą nas do odpowiedzi na pytanie, dlaczego tak musi być, ale do pełnego dowodu będziemy potrzebowali pewnego faktu, który udowodnimy na końcu opisu.

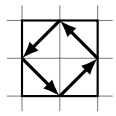
Rozważmy ruch piłeczki po pustej planszy. Piłeczka może poruszać się w nieskończoność, ale ponieważ plansza zawiera skończoną liczbę pozycji, to w pewnym momencie piłeczka drugi raz pojawi się w uprzednio odwiedzonej pozycji i dalej jej ruch będzie się powtarzał w cyklu.

Przeanalizujmy, które z pozycji (x, y, dx, dy) mogą się pojawić na trasie piłeczki. Ustalmy w tym celu pewne pole o środku (sx, sy) i załóżmy, że piłeczka dotyka boku tego pola od wewnętrznej strony. Z tego, co powiedzieliśmy już wcześniej, parzystości liczb x i y są różne, zatem są cztery możliwe położenia piłeczki na bokach tego pola: $(sx \pm 1, sy)$ oraz $(sx, sy \pm 1)$. Uwzględniając możliwe wartości dx i dy, otrzymujemy osiem pozycji piłeczki w obrębie pola. W zależności od ich skierowania podzielmy te pozycje na cztery lewoskrętne i cztery prawoskrętne (rys. 7).

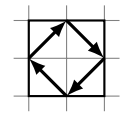


Rys. 6: Przykłady plansz 3×3, dla których piłeczka nie zbije wszystkich klocków.

$(sx, sy-1, 1, 1)$
$(sx+1, sy, -1, 1)$
$(sx, sy+1, -1, -1)$
$(sx-1, sy, 1, -1)$



$(sx, sy-1, -1, 1)$
$(sx+1, sy, -1, -1)$
$(sx, sy+1, 1, -1)$
$(sx-1, sy, 1, 1)$



Rys. 7: Lewo- i prawoskrętne pozycje dla pola (sx, sy).

Zauważmy, że jeśli po pojedynczym kroku z danej pozycji następuje odbicie (zatem pole sąsiaduje ze ścianą lub z klockiem), to nowa pozycja również znajduje się na tym samym polu, a jej skrętność zostaje zachowana. Z kolei, gdy odbicie nie następuje, czyli piłeczka przechodzi na pole sąsiadujące bokiem, to nowa pozycja (na nowym polu) ma przeciwną skrętność.

Pomalujmy teraz pola planszy w szachownicę (na czarno i na biało, przy czym każde dwa pola mające wspólny bok są pomalowane przeciwnymi kolorami). Załóżmy, że pole, z którego startuje piłeczka, jest białe. Wtedy widać, że pozycje piłeczki na każdym polu białym muszą być prawoskrętne, natomiast pozycje piłeczki na każdym polu czarnym muszą być lewoskrętne.

Wynika z tego ograniczenie górne na liczbę pozycji, które może przyjąć piłeczka podczas swojego ruchu: mamy mn pól, na każdym są cztery możliwe pozycje o odpowiedniej skrętności, zatem w sumie mamy co najwyżej $4mn$ pozycji.

Dla zupełnie dowolnych plansz nie wszystkie z tych $4mn$ pozycji będą mogły być osiągnięte. Dla lewej planszy z rys. 6 spośród 36 pozycji piłeczka przechodzi przez 12, wracając do pola startowego, natomiast dla prawej planszy po zbiciu klocka po dwóch ruchach piłeczka wpada w cykl składający się z 12 innych pozycji, nie powracając już nigdy do pozycji startowej.

W ogólności jest tak, że zbiór możliwych pozycji piłeczki rozpada się na kilka niezależnych cykli (między którymi możliwe są przeskoki w momencie odbicia piłeczki od klocków). Jednakże (jak udowodnimy nieco później), gdy liczby m i n są względnie pierwsze, to cykl jest zawsze jeden. Innymi słowy, w tym przypadku piłeczka na pustej planszy odwiedzi wszystkie $4mn$ pozycji, po czym wróci do punktu startowego. Z kolei w przypadku odbicia od klocka, jej pozycja zmienia się, ale nadal na którąś z pozycji z tego cyklu.

Alternatywny sposób znajdowania kolejnego odbicia od klocka

Obserwację, że wszystkie możliwe pozycje piłeczki leżą na jednym cyklu, wykorzystamy przy alternatywnym podejściu do rozwiązania naszego zadania. Pomysł jest taki, że każdej możliwej pozycji piłeczki przyporządkujemy unikalny numer, oznaczający liczbę jednostek czasu, po których pierwszy raz ta pozycja pojawia się na cyklu. Oznaczmy ten numer przez $\mathrm{Numer}(x, y, dx, dy)$.

Przykładowo dla planszy rozmiarów 5×4 (rys. 2) mamy $\mathrm{Numer}(5, 0, -1, 1) = 0$, $\mathrm{Numer}(4, 1, -1, 1) = 1$, $\mathrm{Numer}(0, 5, 1, 1) = 5$, $\mathrm{Numer}(7, 4, 1, -1) = 12$ i ostatecznie $\mathrm{Numer}(6, 1, -1, -1) = 79$.

Przyda nam się też funkcja odwrotna, która na podstawie numeru poda nam pozycję, tzn. $\mathrm{Pozycja}(t) = (x, y, dx, dy)$ jeśli $\mathrm{Numer}(x, y, dx, dy) = t$.

Najprostsza implementacja funkcji Numer i Pozycja będzie bezpośrednio korzystać z tablic *numery* i *pozycje*, które wyznaczymy, jednokrotnie przechodząc piłeczką po wszystkich pozycjach planszy:

```
1:  function InicjujNumery
2:  begin
3:    (x, y, dx, dy) := (m, 0, −1, 1);
4:    for t := 0 to 4·m·n − 1 do begin
5:      numery[x, y, dx, dy] := t;
6:      pozycje[t] := (x, y, dx, dy);
7:      x := x + dx;
8:      y := y + dy;
9:      if Sciana(x, y, dx, dy) then
10:       Odbij;
11:   end
12: end
```

Najbardziej nas będą interesować numery tych pozycji, które znajdują się na bokach klocków (czyli opisanych na rys. 7). Niech T oznacza zbiór tych numerów dla wszystkich klocków (zatem zbiór ten ma $4k$ elementów).

Będziemy znajdować kolejne odbicia od klocków. Załóżmy, że piłeczka znajduje się na pewnej pozycji (x, y, dx, dy), czyli że jest w odległości $t = \text{Numer}(x, y, dx, dy)$ od początku cyklu. Będzie teraz odwiedzać pozycje o kolejnych numerach, aż znajdzie pierwszy numer większy od t, należący do zbioru T (czyli w którym jest odbicie od klocka); oznaczmy ten numer przez t'. Jeśli zbiór T będziemy reprezentowali jako strukturę `set` z biblioteki standardowej C++, to taki element łatwo możemy znaleźć w czasie $O(\log k)$. Ponadto należy uwzględnić przypadek, gdy w T nie ma żadnego elementu większego od t. To oznacza, że piłeczka przejdzie przez pozycje początkową i cykl zacznie się od nowa; wtedy za t' przyjmujemy najmniejszy element zbioru T.

Na podstawie pozycji piłeczki Pozycja(t') wyznaczamy klocek, od którego nastąpi odbicie, i je wykonujemy. W końcu usuwamy ze zbioru T wszystkie numery, które odpowiadały temu klockowi.

```
1:  function Ruszaj
2:  begin
3:    t := Numer(x, y, dx, dy);
4:    if w zbiorze T istnieje element większy od t then begin
5:      t' := najmniejszy element z T większy od t;
6:      czas := czas + t' − t;
7:    end else begin
8:      t' := najmniejszy element z T;
9:      czas := czas + 4·m·n − (t − t');
10:   end
11:   (x, y, dx, dy) := Pozycja(t');
12:   UsunKlocek(x, y, dx, dy); { usuwa z T numery odpowiadające klockowi }
13:   k := k − 1;
14:   Odbij;
15: end
```

Czas działania tego algorytmu to $O(mn+k\log k)$, przy czym $O(mn)$ to koszt obliczeń wstępnych służących wyznaczaniu tablic *numery* i *pozycje*, natomiast $O(k\log k)$ to koszt symulacji ruchu piłeczki. Algorytm jest zapisany w pliku `arks4.cpp` i zalicza pierwsze podzadanie.

Przyspieszamy generowanie numerów

Dotychczas funkcje Numer i Pozycja były bardzo proste, ale kosztem tego, że podczas obliczeń wstępnych generowaliśmy numery po kolei dla wszystkich możliwych pozycji, pomimo tego, że podczas właściwej fazy algorytmu potrzebowaliśmy jedynie $O(k)$ pozycji, odpowiadających miejscom odbić od klocków.

Pomysł na przyspieszenie jest następujący: podczas obliczeń wstępnych przejdziemy cały cykl, ale będziemy zapamiętywać jedynie wartości funkcji Numer dla pozycji, w których piłeczka odbija się od ściany. Dzięki temu będziemy mogli przeskakiwać całe przekątne naraz.

W tym celu potrzebujemy funkcji, która dla danej pozycji piłeczki (x, y, dx, dy) obliczy, po ilu jednostkach czasu piłeczka odbije się od ściany. Można to zrobić następująco. Załóżmy, że chcemy wyznaczyć moment odbicia od prawego brzegu planszy. Kandydatem będzie taka wartość ℓ, dla której piłeczka dotknie prostej $x = 2m$:

$$x + \ell \cdot dx = 2m, \qquad \text{czyli} \qquad \ell = (2m - x)/dx.$$

Oczywiście, kandydat ten jest poprawny, jeśli $\ell \geqslant 0$ oraz nie ma żadnego mniejszego. Ponadto, w przypadku, gdy dostaniemy $\ell = 0$, musimy sprawdzić, czy piłeczka rzeczywiście porusza się w kierunku ściany (a nie oddala się od niej).

```
1: function IdzDoSciany(x, y, dx, dy)
2: begin
3:   minℓ := ∞;
4:   foreach ℓ in {−x/dx, −y/dy, (2m − x)/dx, (2n − y)/dy} do
5:     if ℓ ⩾ 0 and Sciana(x + ℓ · dx, y + ℓ · dy, dx, dy) then
6:       minℓ := min(minℓ, ℓ);
7:   return minℓ;
8: end
```

Mając taką funkcję, możemy wyznaczyć kolejność, w jakiej piłeczka będzie odwiedzać przekątne na cyklu. Dla każdej z tych przekątnych zapamiętujemy pierwszą pozycję na przekątnej oraz numer tej pozycji (służą do tego tablice *przekpoz* i *przeknum*). Ponadto dla pozycji, które są początkami przekątnych, zapamiętujemy numery tych przekątnych w tablicy *przek*.

```
1:  function InicjujNumery
2:  begin
3:    (x, y, dx, dy) := (m, 0, −1, 1);
4:    t := 0;
5:    for i := 0 to 2(m + n) − 1 do begin
6:      przek[x, y, dx, dy] := i;
7:      przekpoz[i] := (x, y, dx, dy);
8:      przeknum[i] := t;
9:      ℓ := IdzDoSciany(x, y, dx, dy);
10:     t := t + ℓ;
11:     x := x + ℓ · dx;
12:     y := y + ℓ · dy;
13:     Odbij;
14:   end
15: end
```

Teraz napisanie funkcji Numer jest już nietrudne. Na początek znajdujemy początek przekątnej, na której jest pozycja (x, y, dx, dy). W tym celu z punktu (x, y) cofamy się w kierunku $(-dx, -dy)$ do pierwszego odbicia ze ścianą. Wynik to suma numeru przypisanego początkowi przekątnej i liczby kroków, o które musieliśmy się cofnąć.

```
1: function Numer(x, y, dx, dy)
2: begin
3:   ℓ := IdzDoSciany(x, y, −dx, −dy);
4:   i := przek[x − ℓ · dx, y − ℓ · dy, dx, dy];
5:   return przeknum[i] + ℓ;
6: end
```

Funkcja odwrotna jest równie prosta. Znajdujemy wyszukiwaniem binarnym przekątną o największym numerze początku nie większym od t (niech to będzie i-ta przekątna o numerze początku t'). Łatwo to zrobić przy pomocy funkcji `upper_bound` z biblioteki standardowej C++ wywołanej na tablicy *przeknum*. Idziemy tą przekątną $t - t'$ kroków:

```
1: function Pozycja(t);
2: begin
3:   i := największa liczba, że przeknum[i] ⩽ t;
4:   (x, y, dx, dy) := przekpoz[i];
5:   ℓ := t − przeknum[i];
6:   return (x + ℓ · dx, y + ℓ · dy, dx, dy);
7: end
```

Ponieważ przekątnych jest $2(m+n)$, więc obliczenia wstępne zajmą czas $O(m+n)$. Funkcja Pozycja działa w czasie $O(\log(m + n))$ z uwagi na wyszukiwanie binarne. Ostatecznie algorytm zadziała w czasie $O(m + n + k \log(m + n + k))$. Jest zapisany w pliku `ark1.cpp`. Wystarczył do zaliczenia kompletu testów.

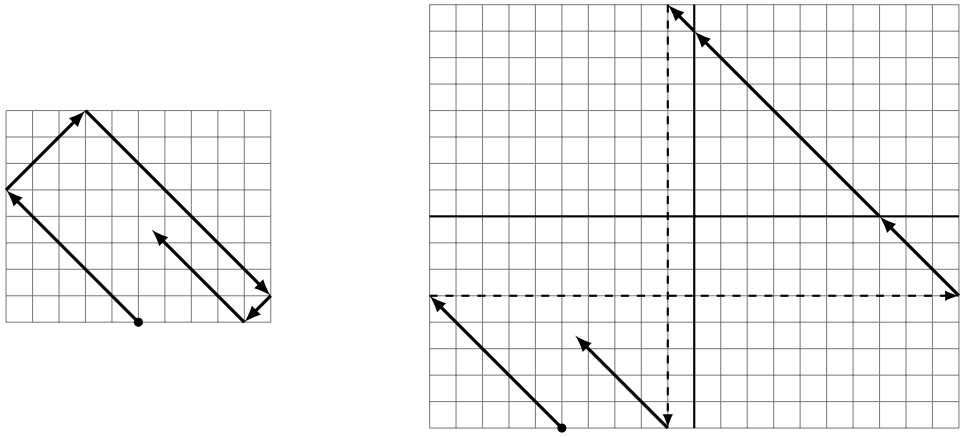
Jeszcze szybsza funkcja do numerów i dowód istnienia dokładnie jednego cyklu

Okazuje się, że istnieje jeszcze szybsze rozwiązanie zadania, które działa podliniowo od rozmiarów planszy. Wykorzystuje się w nim trik implementacyjny, który ma zastosowanie w przypadku wielu zadań, w których mamy piłeczkę odbijającą się od ścian prostokątnej planszy. Jedną z trudności w tego typu zadaniach jest konieczność rozważania różnych przypadków w zależności od kierunku poruszania się piłeczki. Trik jest następujący: zamiast odbijać piłeczkę, odbijamy planszę i zakładamy, że piłeczka porusza się zawsze w jednym kierunku. A konkretniej: zamiast planszy $2m \times 2n$ rozważamy czterokrotnie większą planszę $4m \times 4n$, na której piłeczka porusza się jak po torusie (tzn. dochodząc do brzegu planszy, pojawia się z przeciwległego brzegu; rys. 8).

Ruchowi piłeczki na oryginalnej planszy w kierunku $(-1, 1)$ odpowiada ruch piłeczki w dolnej lewej ćwiartce torusa. Po dotarciu do lewej ściany na oryginalnej planszy następuje odbicie i zmiana kierunku na $(1, 1)$, natomiast na torusie piłeczka przenosi się na prawą ścianę i kontynuuje ruch w kierunku $(-1, 1)$ w dolnej prawej ćwiartce. Analogicznie ruchom w kierunkach $(1, -1)$ i $(-1, -1)$ na oryginalnej planszy odpowiadają ruchy w górnej prawej i górnej lewej ćwiartce.

Poniższy wzór ustala odpowiedniość między pozycjami (x, y, dx, dy) na oryginalnej planszy a położeniami (X, Y) na torusie. Nietrudno napisać wzór, który będzie przeliczał współrzędne w drugą stronę.

$$(X, Y) = \begin{cases} (x, y) & \text{dla } (dx, dy) = (-1, 1), \\ (4m - x, y) & \text{dla } (dx, dy) = (1, 1), \\ (4m - x, 4n - y) & \text{dla } (dx, dy) = (1, -1), \\ (x, 4n - y) & \text{dla } (dx, dy) = (-1, -1). \end{cases}$$



Rys. 8: Oryginalna plansza i czterokrotnie większa plansza, na której piłeczka porusza się jak po torusie.

Ponieważ na torusie ruch odbywa się cały czas w jednym kierunku, a jego brzegi są „zawinięte", możemy w łatwy sposób opisać położenie piłeczki (X, Y) w dowolnej chwili t przy użyciu układu równań modularnych. A mianowicie:

$$X = (m - t) \bmod 4m, \qquad Y = t \bmod 4n. \tag{1}$$

Zauważmy, że z tego bardzo łatwo wynika sposób obliczania funkcji Pozycja(t). Wystarczy wyznaczyć położenie (X, Y) z równania (1) i zobaczyć, jaka pozycja odpowiada mu na oryginalnej planszy.

Wyznaczenie funkcji Numer(x, y, dx, dy) jest już trudniejsze, bo wymaga (po zamianie na współrzędne na torusie) rozwiązania układu kongruencji

$$t \equiv m - X \pmod{4m}, \qquad t \equiv Y \pmod{4n}. \tag{2}$$

W tym celu wykorzystamy Chińskie Twierdzenie o Resztach. Musimy jeszcze przezwyciężyć drobny kłopot polegający na tym, że twierdzenie to działa dla układów o względnie pierwszych modułach, a w naszym przypadku liczby $4m$ i $4n$ nie są względnie pierwsze. Jednakże w założeniach zadania jest, że liczby m i n są względnie pierwsze oraz m jest nieparzysta, zatem względnie pierwsze są liczby m i $4n$. Możemy zatem najpierw rozwiązać układ kongruencji

$$t' \equiv m - X \pmod{m}, \qquad t' \equiv Y \pmod{4n}, \tag{3}$$

w którym wyznaczymy t', takie że $t' \equiv t \pmod{4mn}$. Zatem t' jest też rozwiązaniem oryginalnego układu (2).

Układ kongruencji (3) możemy rozwiązać w czasie stałym, jeśli znamy dwie liczby całkowite α i β spełniające równanie

$$\alpha \cdot m + \beta \cdot 4n = 1.$$

Ponieważ m i $4n$ są względnie pierwsze, to liczby te istnieją i możemy je wyznaczyć w czasie $O(\log(m+n))$ za pomocą rozszerzonego algorytmu Euklidesa. Rozwiązaniem układu kongruencji (3) jest wtedy

$$t' \equiv (m - X \bmod m) \cdot \beta \cdot 4n + Y \cdot \alpha \cdot m \pmod{4mn}.$$

Zatem cały algorytm będzie działał w czasie $O(\log(m+n) + k \log k)$. Jego implementacja znajduje się w pliku `ark2.cpp`.

Pozostała nam do wykonania ostatnia obserwacja: z Chińskiego Twierdzenia o Resztach wiemy, że w układzie kongruencji (3) różnym wartościom $0 \leqslant t' < m \cdot 4n$ odpowiadają różne pary (X, Y). Pociąga to za sobą różnowartościowość rozwiązań układu (2) dla $0 \leqslant t < 4mn$. Wynika z tego, że każda z $4mn$ początkowych pozycji piłeczki będzie różna. To kończy dowód twierdzenia, że wszystkie $4mn$ pozycje piłeczki leżą na jednym cyklu.

Na koniec jeszcze jedna uwaga natury implementacyjnej. Otóż trik z odbijaniem planszy zamiast piłeczki można było zrobić na samym początku i przez całe rozwiązanie operować na współrzędnych (X, Y) i ich numerach. Niektóre z powyższych rozwiązań mogą stać się dzięki temu prostsze w implementacji (np. wszystkie przekątne będą miały ten sam kierunek), ale trzeba uważać na nowe komplikacje (np. funkcja Odbij będzie musiała „teleportować" piłeczkę na odpowiednią ćwiartkę torusa).

Tomasz Idziaszek	Tomasz Idziaszek, Paweł Parys	Karol Pokorski
Treść zadania	Opracowanie	Program

Dostępna pamięć: 128 MB. *OI, etap II, dzień drugi, 11.02.2016*

Wcale nie Nim

Bajtoni i jego młodszy braciszek Bajtuś często grają w grę Nim. Bajtoni objaśnił braciszkowi, jaka jest strategia wygrywająca w tej grze, ale Bajtuś jeszcze nie radzi sobie z jej stosowaniem, i często przegrywa. Z tego powodu co rusz proponuje zmiany w regułach gry, mając nadzieję, że rozgrywka będzie łatwiejsza.

Właśnie zaproponował nową wersję: mamy n par stosów, przy czym stosy z i-tej pary zawierają początkowo po a_i kamieni. Gracze wykonują ruchy na przemian. Bajtuś w swoim ruchu zabiera niezerową liczbę kamieni z dowolnego wybranego przez siebie stosu. Z kolei Bajtoni w swoim ruchu przekłada niezerową liczbę kamieni pomiędzy stosami w wybranej przez niego parze. Bajtuś wykonuje ruch jako pierwszy. Przegrywa ten, kto nie może wykonać już żadnego ruchu.

Bajtoni od razu zauważył, że przy takich zasadach nie ma żadnych szans na zwycięstwo, ale nie chcąc robić przykrości braciszkowi, zgodził się zagrać. Postawił sobie jednak za punkt honoru jak najdłużej odwlekać nieuchronną przegraną. Pomóż mu i napisz program, który stwierdzi, jak długo może potrwać rozgrywka, jeśli obaj bracia grają optymalnie (Bajtuś dąży do zwycięstwa w najmniejszej liczbie ruchów, zaś Bajtoni dąży do maksymalnego wydłużenia gry).

Wejście

W pierwszym wierszu standardowego wejścia znajduje się dodatnia liczba całkowita n oznaczająca liczbę par stosów. W drugim wierszu znajduje się ciąg n dodatnich liczb całkowitych $a_1, a_2, \ldots, a_n$ pooddzielanych pojedynczymi odstępami, oznaczających liczności kolejnych par stosów.

Wyjście

W jedynym wierszu standardowego wyjścia należy zapisać jedną liczbę całkowitą oznaczającą liczbę ruchów, po których nastąpi koniec gry, jeśli obaj bracia grają optymalnie.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
2 1 2	7

Wyjaśnienie do przykładu: *Optymalna rozgrywka może wyglądać następująco:*

1 1 2 2 → 1 1 2 0 → 1 1 1 1 → 1 1 1 0 → 1 1 0 1 → 1 1 0 0 → 2 0 0 0 → 0 0 0 0

Testy „ocen”:

`1ocen`: $n = 1$, $a_1 = 100$, *wynik 15*,

`2ocen`: $n = 5$, *wszystkie stosy po 2 kamienie, wynik 21*,

`3ocen`: $n = 3$, $a_1 = 10^7$, $a_2 = 10^8$, $a_3 = 10^9$, *wynik 163*,

`4ocen`: $n = 3000$, $a_i = i$, *wynik 65 197*,

`5ocen`: $n = 100\,000$, *wszystkie stosy po 1 kamieniu, wynik 200 001*.

Ocenianie

Zestaw testów dzieli się na podzadania spełniające następujące warunki. Testy do każdego podzadania składają się z jednej lub większej liczby osobnych grup testów. We wszystkich podzadaniach zachodzi $a_i \leqslant 1\,000\,000\,000$.

Podzadanie	Warunki	Liczba punktów
1	$n = 1$	*10*
2	*suma wartości* $a_i \leqslant 10$	*10*
3	$n \leqslant 3$	*20*
4	$n \leqslant 3\,000$	*20*
5	$n \leqslant 500\,000$	*40*

Rozwiązanie

W zadaniu mamy do czynienia z grą dla dwóch graczy, a naszym celem jest ustalenie, jaka jest optymalna strategia gry dla każdego z nich. Jak zobaczymy później, samo zaprogramowanie wyliczania liczby ruchów, które zostaną wykonane przez graczy przy zastosowaniu tych strategii, będzie już proste.

Na początku opiszemy, mniej lub bardziej intuicyjny, sposób dochodzenia do rozwiązania; formalny dowód poprawności zaprezentowanych strategii zostanie przedstawiony w dalszej części.

Jak grać w grę?

Dla łatwiejszego rozróżniania graczy, nazwijmy gracza pierwszego A, a drugiego B. Gracz A chce jak najszybciej zakończyć grę, zatem chce minimalizować liczbę kamieni na stosach. Dość łatwo więc zgodzić się z następującą obserwacją:

Obserwacja 1. *Gdy gracz A ustali, z której pary stosów zabierze kamienie w następnym ruchu, to najbardziej opłaca mu się zabrać wszystkie kamienie z większego ze stosów z pary.*

Skoro wiemy już, że ruchy A powodują opróżnienie większego ze stosów z pary, to celem B powinno być zmaksymalizowanie liczby kamieni na mniejszym ze stosów w parze. Czynimy więc kolejną obserwację:

Obserwacja 2. *Gdy gracz B ustali, że w następnym ruchu będzie przekładał kamienie w pewnej parze stosów zawierającej a i b kamieni ($a + b = k$), to najbardziej opłaca mu się:*

- *wyrównać zawartość stosów do $(\frac{k}{2}, \frac{k}{2})$, gdy k jest parzyste i $a \neq b$,*
- *lekko odejść od równowagi, do $(\frac{k}{2} - 1, \frac{k}{2} + 1)$, gdy k jest parzyste i $a = b = \frac{k}{2}$,*
- *„prawie" wyrównać do $(\frac{k-1}{2}, \frac{k+1}{2})$, gdy k jest nieparzyste.*

Początkowo dla wszystkich par stosów łączna ilość kamieni na obu stosach z pary jest parzysta i kamienie te są równo rozłożone. Zatem nie ma powodu, aby B psuł na nich równowagę, bo może tylko stracić. Natomiast gdy A usuwa wszystkie kamienie z jakiegoś stosu, to B zapewne będzie chciał wyrównać układ na parze, do której należał ten stos, bo inaczej A w następnym ruchu zdejmie od razu wszystkie kamienie z drugiego stosu, zamiast robić to w wielu ruchach. Możemy więc przyjąć za prawdziwą (bez ścisłego dowodu) następującą obserwację:

Obserwacja 3. *O ile w parze stosów, z której ostatnio usuwał gracz A, są jeszcze jakieś kamienie, to graczowi B najbardziej opłaca się ruszać w tej parze.*

Zatem jeśli para, z której ostatnio usuwał gracz A, jest postaci $(k, 0)$ dla $k \neq 0$, to B wyrówna do $(\frac{k}{2}, \frac{k}{2})$ dla k parzystego lub „prawie" wyrówna do $(\frac{k-1}{2}, \frac{k+1}{2})$ dla k nieparzystego.

W przypadku, gdy A usunie ostatni kamień w parze stosów, gracz B nie może wykonać swojego ruchu w tej parze i musi popsuć równowagę w jakiejś innej parze. Jak ma wybrać tę parę, opiszemy nieco dalej.

Z powyższych obserwacji wynika też, że przed ruchem gracza A wszystkie pary stosów będą jednego z trzech rodzajów wymienionych w obserwacji 2. Istotnie, na początku gry wszystkie pary są wyrównane, a potem po każdym ruchu A, który psuje którąś parę, gracz B poprawia tę parę, a po każdym ruchu A opróżniającym parę, gracz B lekko odchodzi od równowagi w innej parze (jeśli wszystkie są wyrównane) lub robi ruch zamieniający kolejność stosów w pewnej „niewyrównanej" parze.

Przyjrzyjmy się bliżej, jakie ruchy A ma do dyspozycji. Załóżmy, że przed ruchem gracza A w parze stosów znajduje się $a + b = k$ kamieni. Zapiszmy w systemie dwójkowym łączną liczbę kamieni k i zobaczmy, jak zapis ten zmieni się po ruchu gracza A. Otóż, jeśli k było parzyste i była równowaga lub jeśli k było nieparzyste i była „prawie"-równowaga, to ruch A usuwa ostatnią cyfrę z liczby k, np.:

$$k = 44 = 101100_2 \quad \rightarrow \quad \tfrac{k}{2} = 22 = 10110_2,$$
$$k = 45 = 101101_2 \quad \rightarrow \quad \tfrac{k-1}{2} = 22 = 10110_2.$$

Natomiast jeśli k było parzyste i równowaga była zaburzona, to ruch A zamienia k na $\frac{k}{2} - 1$, czyli najpierw usuwa ostatnią cyfrę, po czym ostatnią jedynkę w zapisie zamienia na zero, a zera za nią zamienia na jedynki:

$$k = 44 = 101\underline{10}0_2 \quad \rightarrow \quad \tfrac{k}{2} - 1 = 21 = 101\underline{01}_2.$$

Nazwijmy ten pierwszy rodzaj ruchu gracza A ruchem *zwykłym*, a ten drugi ruchem *ulepszonym*. Z tego, co powiedzieliśmy o tym, jak wyglądają stosy przed ruchem

gracza A, wynika, że są to jedyne rodzaje ruchów, które może wykonać A. Ponadto, jeśli ruchy te nie czyszczą pary stosów, to następny ruch gracza B na tej parze przywróci na niej równowagę lub „prawie"-równowagę.

Zastanówmy się teraz, ile ruchów będzie potrzebował A, żeby wyczyścić ustaloną parę stosów, na której znajduje się $a+b=k$ kamieni. Jeśli A będzie wykonywał same ruchy zwykłe, to będzie musiał wykonać tyle ruchów, ile cyfr ma zapis dwójkowy liczby k. Oznaczmy tę wartość przez $\ell(k) = \lfloor \log_2 k \rfloor + 1$.

Przeanalizujmy, czy używanie ruchów ulepszonych umożliwia graczowi A szybsze wyczyszczenie pary stosów (pomińmy na razie fakt, że wykonanie ruchu ulepszonego nie zawsze jest możliwe). Zauważmy, że dla większości liczb k ruch ulepszony również powoduje usunięcie jednej cyfry z zapisu dwójkowego. Wyjątkiem jest na przykład $k = 2 = 10_2$, gdzie pojedynczy ruch ulepszony powoduje opróżnienie pary stosów, czym skraca zapis dwójkowy o *dwie* cyfry.

Wynika z tego, że dla dowolnej liczby kamieni k, która w zapisie binarnym zaczyna się od jedynki i zera, możliwe jest wyczyszczenie stosów, używając $\ell(k) - 1$ ruchów. Najpierw wykonujemy $\ell(k) - 2$ ruchów zwykłych, doprowadzając do 10_2, a potem jeden ruch ulepszony.

W przypadku liczb k, których zapis binarny zaczyna się od większej liczby jedynek, jest nieco trudniej. Dla przykładu dla $k = 6 = 110_2$ wykonanie ruchu zwykłego prowadzi do liczby nieparzystej $3 = 11_2$, co uniemożliwia użycie ruchu ulepszonego. W tym przypadku lepiej jest od razu zacząć od ruchu ulepszonego, który prowadzi do 10_2, a następnie wykonać drugi ruch ulepszony. W ogólności, jeśli zapis dwójkowy liczby k zaczyna się od j jedynek, po których następuje zero, a potem $\ell(k) - 1 - j$ dowolnych cyfr, to A może wyczyścić taką parę, wykonując najpierw $\ell(k)-1-j$ ruchów zwykłych, a następnie j ruchów ulepszonych; zatem w sumie wykonując $\ell(k) - 1$ ruchów.

Zatem A, chcąc opróżnić ustaloną parę stosów, mógłby poprawić swój wynik o jeden ruch, jeśli tylko mógłby zapewnić sobie wykonanie tylu ruchów ulepszonych, ile jest jedynek na początku zapisu dwójkowego liczby kamieni w tej parze. Nietrudna analiza pokazuje, że lepiej się nie da (nie można poprawić wyniku o więcej niż jeden ruch, a aby poprawić o jeden ruch, nie wystarczy wykonanie mniejszej liczby ruchów ulepszonych).

Musimy jednak uwzględnić fakt, że wykonanie przez A na parze stosów ruchu ulepszonego możliwe jest tylko, gdy para ta spełnia następujące dwa warunki:

- liczba kamieni w parze k w zapisie binarnym składa się z pewnej liczby jedynek, po której następuje zero, czyli $k = 2^m - 2$ dla $m \geqslant 2$; oznaczmy zbiór wszystkich takich liczb przez M;
- para stosów ma zaburzoną równowagę przez gracza B.

Gracz A łatwo może zapewnić, że zaburzanie równowagi przez gracza B będzie odbywać się tylko na takich parach stosów, dla których liczba kamieni jest już w zbiorze M. Wystarczy, że początkowo, wykonując tylko ruchy zwykłe (nie czyszcząc w międzyczasie żadnej pary stosów do końca) dla każdej pary stosów usunie z zapisu dwójkowego liczby kamieni wszystkie cyfry występujące za pierwszym zerem z lewej strony. Bez straty ogólności możemy zatem założyć, że liczba kamieni dla *każdej* pary jest w zbiorze M.

Przy powyższych założeniach odnośnie strategii graczy A i B, pozostaje im już niewielki wybór. Otóż gracz A może (musi) wybrać, dla której pary stosów rezygnuje z poprawiania wyniku o jeden i czyści tę parę ruchami zwykłymi. Gdy to zrobi, gracz B może (musi) wybrać, dla której pary stosów zaburzy równowagę, czyli pozwoli graczowi A na wykonanie ruchu ulepszonego, który kasuje *jedną* jedynkę z zapisu dwójkowego liczby kamieni. Jeśli to była akurat ostatnia jedynka, to kolejka powraca do gracza B, który znowu musi wybrać, dla której pary stosów zaburzy równowagę. Jeśli nie była to ostatnia jedynka, to znowu musi wybierać gracz A.

Do skompletowania strategii potrzebna jest jeszcze jedna obserwacja:

Obserwacja 4. *Załóżmy, że wszystkie liczby kamieni na parach stosów są ze zbioru M i jest na nich równowaga. Jeśli jest ruch gracza A, to wybiera on najliczniejszą parę i wykonuje na niej ruchy aż do opróżnienia. Jeśli jest ruch gracza B, to wybiera on najliczniejszą parę i zaburza na niej równowagę.*

Aby ściśle uzasadnić tę obserwację (jak również poprzednie), zredukujemy naszą grę (nazwijmy ją G) do pewnej nowej gry (nazwijmy ją H). W grze H będzie n stosów, które dla odmiany zawierają zapałki. Jeśli w grze G początkowa liczba kamieni na i-tej parze stosów w zapisie dwójkowym zaczyna się od j jedynek, to na i-tym stosie w grze H znajduje się początkowo j zapałek. Gracze ruszają się na zmianę, zaczyna gracz A. Gracz A może opróżnić dowolny niepusty stos. Gracz B może natomiast usunąć dowolną dodatnią liczbę zapałek z dowolnie wybranych stosów, pod warunkiem że:

- co najmniej jedna usuwana zapałka nie była ostatnią zapałką na stosie, lub
- ruch opróżnia wszystkie stosy.

Gra kończy się, gdy wszystkie stosy są puste. Celem gracza A jest wykonanie jak najmniejszej liczby ruchów (czyli celem B jest zapewnienie, że A będzie ruszał się jak najwięcej razy). Dotychczasowe rozumowanie sugeruje, że prawdziwy powinien być następujący lemat:

Lemat 1. *Niech s będzie łączna długością (liczbą cyfr) zapisów dwójkowych liczb a_i. Przy optymalnej grze obu graczy w obu grach, liczba ruchów gracza A w G jest o s większa od liczby ruchów gracza A w grze H.*

Zanim udowodnimy ten lemat, powiedzmy, jak rozwiązać grę H. Po pierwsze, oczywiste jest, że gracz B będzie usuwał tylko jedną zapałkę na raz (za wyjątkiem ostatniego ruchu, kiedy to musi opróżnić wszystkie stosy mające po jednej zapałce); możliwość usuwania jeszcze jakichś dodatkowych zapałek nic mu nie daje, skoro chce usunąć ich jak najmniej, zostawiając jak najwięcej graczowi A. Dalej, dość łatwo przekonać się, że graczowi A zawsze najbardziej opłaca się opróżnienie największego stosu (bo gdy zostaną małe stosy, to B, czyszczący stosy po jednej zapałce, zdąży ich więcej opróżnić, zastępując w tym gracza A, który chce minimalizować liczbę swoich ruchów). Jeśli to wiemy, to widzimy także, że graczowi B również najbardziej opłaca się zabranie zapałki z najliczniejszego stosu (bo ten stos i tak zostanie szybko opróżniony przez A, natomiast zapałki z niższych stosów gracz B może usuwać później).

Rozwiązanie zadania wygląda więc następująco. Najpierw konstruujemy grę H (czyli liczymy długości liczb a_i w zapisie dwójkowym oraz liczbę jedynek na początku

każdego zapisu). Następnie gramy w H, wykonując opisane powyżej optymalne strategie obu graczy; liczymy przy tym, ile ruchów wykona gracz A. Możemy po prostu grać ruch po ruchu, bo ruchów jest niewiele (w zasadzie równie dobrze można wykonywać ruchy od razu w G, symulując optymalne strategie obu graczy). Do wyznaczania najliczniejszego stosu w H możemy skorzystać z kolejki priorytetowej (za każdym razem wyjmujemy maksimum i wstawiamy wynik ruchu).

Ponieważ w grze H gracz A wykona co najwyżej n ruchów, a każdy ruch gracza wymaga czasu $O(\log n)$ na kolejce priorytetowej, to powyższe rozwiązanie ma złożoność czasową $O(n \log n)$.

Wszędzie tu pisaliśmy o liczbie ruchów gracza A, natomiast w zadaniu mamy wypisać łączną liczbę ruchów obu graczy. Łatwo jednak zauważyć, że między tymi wielkościami zachodzi następująca zależność: jeśli gracz A ruszał się x razy, to obaj gracze ruszali się $2x - 1$ razy.

Opisane powyżej rozwiązanie zostało zaimplementowane w pliku `wca.cpp`. Alternatywna implementacja w plikach `wca1.cpp`, `wca2.pas` nie używa kolejki priorytetowej. Zamiast tego dla każdego j pamięta, ile liczb zaczyna się od j jedynek (różnych j jest mało, bo co najwyżej $\log_2 a_i$). W takiej tablicy łatwo znajdujemy, jaka jest maksymalna liczba jedynek w jakiejś liczbie; możemy też tę liczbę usunąć, czy zmniejszyć jej liczbę jedynek o jeden.

Odpowiedniość między grami G i H

W tej sekcji udowodnimy lemat 1, czyli formalnie pokażemy odpowiedniość między oryginalną grą G, a grą H.

Lemat 1 (**implikacja w jedną stronę**). *Jeśli A może zakończyć grę H, wykonując x ruchów, to może zakończyć grę G, wykonując $x + s$ ruchów.*

Dowód: Zagrajmy równocześnie w obie gry. Ruchy gracza A w G będziemy konstruować, patrząc na to, co dzieje się w H, a ruchy gracza B w H będziemy konstruować, patrząc na to, co dzieje się w grze G. Gracz A, grając w G oznacza sobie pary stosów jedną z liter X, Z. Początkowo wszystkie stosy oznaczone są literą X.

Będziemy utrzymywać następujący niezmiennik:

- Jeśli i-ta para stosów w G jest oznaczona literą X, to zapis dwójkowy łącznej liczby kamieni na stosach z tej pary zawiera zero, a liczba jedynek na początku tego zapisu jest równa liczbie zapałek na i-tym stosie w H.
- Jeśli i-ta para stosów w G jest oznaczona literą Z, to i-ty stos w H jest pusty.

Widzimy, że niezmiennik jest spełniony, gdy obie gry są w sytuacji początkowej (w szczególności łączna liczba kamieni na obu stosach w parze jest parzysta, czyli jej zapis dwójkowy zawiera zero).

Symulacja przebiega następująco:

1. Na samym początku pozwalamy graczowi A ruszyć się w H; usuwa on wszystkie zapałki ze stosu numer i, a my oznaczamy i-tą parę stosów w G literą Z. Niezmiennik pozostaje spełniony: napisaliśmy Z, a w H stos jest pusty.

2. Niech k_i to liczba kamieni na i-tej parze stosów w G (dla każdego i). Gracz A rusza się w G w następujący sposób:

   (a) Znajduje (o ile istnieje) takie i, że i-ta para jest oznaczona przez X oraz $k_i \notin M \cup \{0\}$ i usuwa z i-tej pary $\lceil \frac{k_i}{2} \rceil$ kamieni (na większym ze stosów jest co najmniej tyle). Ponieważ zapis dwójkowy k_i zawiera zero i $k_i \notin M$ (czyli ostatnia cyfra nie jest jedynym zerem), to po usunięciu ostatniej cyfry nadal będzie jakieś zero, a liczba jedynek na początku nie zmienia się; zatem niezmiennik pozostaje prawdziwy.

   (b) Jeśli nie było i jak wyżej, znajduje (o ile istnieje) takie i, że i-ta para jest oznaczona przez X, $k_i \in M$ oraz kamienie nie są rozłożone po równo na stosach; usuwa z i-tej pary $\frac{k_i}{2} + 1$ kamieni (na większym ze stosów jest co najmniej tyle). Jednocześnie w H gracz B usuwa jedną zapałkę z i-tego stosu (ruchy gracza B w H opisujemy zapałka po zapałce, wiele takich ruchów składa się na jeden pełny ruch). Opisany ruch A w G powoduje usunięcie jednej jedynki w zapisie dwójkowym liczby kamieni w i-tej parze stosów, czyli liczba jedynek na początku tego zapisu spada o 1. Jednocześnie usunęliśmy jedną zapałkę z i-tego stosu w H, więc niezmiennik pozostaje zachowany (a niezmiennik przed ruchem zapewnia, że rzeczywiście na i-tym stosie w H była jakaś zapałka i B może ją usunąć).

   (c) Jeśli nie było i jak wyżej, znajduje (o ile istnieje) takie i, że i-ta para jest oznaczona przez Z oraz $k_i > 0$ i usuwa z i-tej pary $\lceil \frac{k_i}{2} \rceil$ kamieni (na większym ze stosów jest co najmniej tyle). Niezmiennik jest zachowany w trywialny sposób.

   (d) Jeśli nie było i jak wyżej, to kończymy ruch gracza B w H i pozwalamy ruszyć się graczowi A (udowodnimy później, że rzeczywiście uzyskaliśmy poprawny ruch gracza B oraz że gra H jeszcze się nie skończyła); on usuwa wszystkie zapałki ze stosu numer i, a my oznaczamy i-tą parę stosów literą Z. Niezmiennik pozostaje spełniony: napisaliśmy Z, a w H stos jest pusty. Zwróćmy uwagę, że i-ta para stosów była wcześniej oznaczona literą X (gdyby była oznaczona przez Z, to i-ty stos w H byłby pusty) oraz jest tam jakiś kamień (liczba jedynek na początku zapisu dwójkowego liczby kamieni była równa liczbie zapałek na i-tym stosie, która była niezerowa). Następnie próbujemy ponownie wykonać punkt (c), w którym teraz już na pewno szukanie i zakończy się sukcesem.

3. Jeśli wszystkie stosy w G są puste, to gra się kończy. W tej sytuacji niezmiennik zapewnia, że również w H wszystkie stosy są puste.

4. Pozwalamy ruszyć się graczowi B w G i wracamy do punktu 2. Zauważmy, że niezależnie od jego ruchu, niezmiennik nadal jest spełniony.

Musimy teraz udowodnić, że w punkcie 2(d) możemy rzeczywiście zakończyć w H ruch gracza B i czekać na ruch gracza A. Ruch gracza B trwał od ostatniego momentu, gdy jakąś niepustą parę stosów oznaczyliśmy przez Z (w punkcie 1 lub 2(d)); zatem na początku tego ruchu istniała niepusta para oznaczona przez Z (dokładnie jedna, ale to dla nas bez znaczenia), a teraz nie istnieje (skoro jej znalezienie w 2(c) się nie

powiodło i przeszliśmy do 2(d)). Spójrzmy na moment, gdy ostatni kamień z tej pary stosów został usunięty przez A. Stało się to w punkcie 2(c), czyli nie udało się wówczas znalezienie i ani w punkcie 2(a), ani w punkcie 2(b). Oznacza to, że dla każdej pary stosów i oznaczonej przez X zachodziło $k_i \in M \cup \{0\}$ i na obu stosach tej pary było po tyle samo kamieni. Po rozważanym ruchu gracza A z punktu 2(c), usuwającym ostatni kamień z pary stosów oznaczonej Z, gracz B musi wykonać jakiś ruch i jedyne, co może zrobić, to zaburzyć równowagę w pewnej parze stosów i oznaczonej X, dla której $k_i \in M \cup \{0\}$. W kolejnym kroku gracz A na pewno zrobi ruch według punktu 2(b), dla tego i, zdejmując graczem B jedną zapałkę w grze H. Jeśli było $k_i > 2$, to zapałka usunięta w H nie była ostatnią na stosie, co zapewnia poprawność ruchu gracza B w H. Jeśli jednak $k_i = 2$, to A usuwa oba kamienie z tego stosu i B musi ruszyć się gdzie indziej, czyli znowu jedyne co może zrobić to zaburzyć równowagę w pewnej parze stosów i oznaczonej X, dla której $k_i \in M \cup \{0\}$; sytuacja się powtarza. Jedyny sposób wyjścia z tej pętli to albo przypadek $k_i > 2$, albo koniec gry. Skoro jednak ileś ruchów później doszliśmy do punktu 2(d), to musieliśmy przejść przez przypadek $k_i > 2$, czyli zapewnić poprawność ruchu gracza B w H. To kończy dowód poprawności ruchu gracza B w H. Musimy jeszcze zobaczyć, że H jeszcze się nie skończyła i jest sens czekać w niej na ruch gracza A. Skoro jednak G się nie skończyła, a jednocześnie nie ma tam niepustej pary stosów oznaczonej przez Z, to jest niepusta para stosów oznaczona przez X; niezmiennik zapewnia, że odpowiadający jej stos w H także jest niepusty.

Pozostaje porównać liczbę ruchów gracza A w obu grach. Niech s_i będzie liczbą cyfr w zapisie dwójkowym liczby a_i. Widzimy, że dla każdego i zachodzą następujące własności:

1. Jeśli na koniec i-ta para stosów jest oznaczona przez X, to aby ją opróżnić A wykonał s_i ruchów. Istotnie, początkowo na tej parze stosów mamy $2a_i$ kamieni (zapis dwójkowy długości $s_i + 1$, na pewno zawiera jakieś zero). Dopóki liczba kamieni nie jest w M, każdy ruch A dotyczący tej pary (z punktu 2(a)) usuwa ostatnią cyfrę z zapisu dwójkowego, skracając go o jeden. W pewnym momencie na pewno liczba kamieni zacznie być w M (gdy usuniemy wszystkie cyfry za pierwszym zerem). Wówczas każdy ruch A (z punktu 2(b)) usuwa jedną jedynkę z zapisu dwójkowego, w szczególności ostatni ruch przechodzi od $2 = 10_2$ kamieni od razu do 0 kamieni, skracając zapis dwójkowy o dwie cyfry. Ruchów A musiało więc być rzeczywiście s_i.

2. Jeśli na koniec i-ta para stosów jest oznaczona przez Z, to aby ją opróżnić A wykonał $s_i + 1$ ruchów. Istotnie, w tej sytuacji początek gry przebiega jak poprzednio, czyli każdy ruch A skraca zapis dwójkowy łącznej liczby kamieni o jedną cyfrę. W pewnym momencie (gdy jest jeszcze niepusta) para zostaje oznaczona przez Z. Później każdy ruch A na tej parze (z punktu 2(c)) powoduje usunięcie ostatniej cyfry z zapisu dwójkowego łącznej liczby kamieni, czyli skrócenie tego zapisu o jedną cyfrę. Skrócenie liczby $(s_i + 1)$-cyfrowej do 0-cyfrowej zajmuje więc rzeczywiście $s_i + 1$ ruchów.

Jednocześnie widzimy, że oznaczenie pary stosów przez Z występuje dokładnie wtedy, gdy gracz A rusza się w H. Zatem jeśli A w grze H wykonał x ruchów, to w G wykonał $x + s$ ruchów. ■

Lemat 1 (implikacja w drugą stronę). *Jeśli A może zakończyć grę G, wykonując $x + s$ ruchów, to może zakończyć grę H, wykonując x ruchów.*

Dowód: Znowu gramy równocześnie w obie gry. Ruchy gracza A w H będziemy konstruować, patrząc na to, co dzieje się w G, a ruchy gracza B w G będziemy konstruować, patrząc na to, co dzieje się w H. Z każdą parą stosów w grze G skojarzymy dwie liczby: r_i oraz z_i. Liczba r_i początkowo będzie równa długości zapisu dwójkowego liczby $2a_i$ (czyli łącznej liczby kamieni na i-tej parze stosów w G), natomiast z_i będzie równa liczbie jedynek na początku $2a_i$ (czyli liczbie zapałek na i-tym stosie w H). Dla dowolnych liczb naturalnych r, z zdefiniujmy:

$$m(r,z) = \begin{cases} \lfloor (2^z - 1) \cdot 2^{r-z} \rfloor & \text{gdy } z > 0, \\ \lceil 2^{r-1} - 1 \rceil & \text{gdy } z = 0. \end{cases}$$

Zatem gdy $z > 0$, liczba ta w zapisie dwójkowym jest długości r i zaczyna się od $\min(r, z)$ jedynek, po których są same zera. Jeśli zaś $z = 0$, to jest to po prostu $\max(0, r - 1)$ jedynek. Początkowe wyrazy tego ciągu to:

$$\begin{array}{lllll} m(0,0) = 0, & m(1,0) = 0, & m(2,0) = 1, & m(3,0) = 3, & m(4,0) = 7, \\ m(0,1) = 0, & m(1,1) = 1, & m(2,1) = 2, & m(3,1) = 4, & m(4,1) = 8, \\ m(0,2) = 0, & m(1,2) = 1, & m(2,2) = 3, & m(3,2) = 6, & m(4,2) = 12. \end{array}$$

Zauważmy, że zawsze $\lfloor \frac{m(r+1,z)}{2} \rfloor = m(r,z)$ (zmniejszenie r o jeden powoduje podzielenie liczby $m(r,z)$ przez dwa, z zaokrągleniem w dół).

Przed każdym ruchem gracza A w G prawdziwy będzie następujący niezmiennik (dla każdego i):

1. na i-tej parze stosów w G jest co najmniej $m(r_i, z_i)$ kamieni, przy czym na każdym ze stosów w parze co najmniej $\lfloor \frac{m(r_i,z_i)}{2} \rfloor$ kamieni,
2. na i-tym stosie w H jest co najwyżej z_i zapałek,
3. jeśli $r_i = 0$ to $z_i \leqslant 1$, oraz
4. $z_i = 0$ wtedy i tylko wtedy, gdy gra H już się skończyła.

Widzimy, że niezmiennik jest spełniony, gdy obie gry są w sytuacji początkowej (w szczególności zarówno $m(r_i, z_i)$ jak i liczba kamieni na i-tej parze stosów jest długości r_i i zaczyna się od z_i jedynek, przy czym w $m(r_i, z_i)$ po tych jedynkach są same zera, co daje pierwszy punkt).

Podczas symulacji powtarzamy następujące operacje:

1. Czekamy w G na ruch gracza A, który usuwa kamienie z pary stosów numer i. Może to zaburzyć pierwszy punkt niezmiennika, natomiast ciągle wiemy, że na jednym ze stosów w tej parze (na tym, którego A nie ruszał) jest co najmniej $\lfloor \frac{m(r_i,z_i)}{2} \rfloor$ kamieni.
2. Jeśli na którymś ze stosów w i-tej parze pozostało co najmniej $m(r_i, z_i)$ kamieni, to nic nie robimy, a jeśli nie (wtedy w szczególności na pewno jest

$r_i > 0$), to zmniejszamy r_i o 1. Po takim zmniejszeniu już na pewno na którymś ze stosów w i-tej parze pozostało co najmniej $m(r_i, z_i)$ kamieni, bo $\lfloor \frac{m(r_i+1,z_i)}{2} \rfloor = m(r_i, z_i)$; w szczególności punkt 1 niezmiennika jest już prawdziwy. Zauważmy, że może to zaburzyć trzeci punkt niezmiennika, ale tylko, gdy oba stosy stały się puste (jeśli któryś stos był niepusty, a r_i było 1, to byśmy nie zmniejszyli r_i, bo $m(1, z_i) \leqslant 1$).

3. Załóżmy, że po ruchu gracza A na parze stosów numer i pozostały jeszcze jakieś kamienie. Wówczas gracz B odpowiada na tym samym stosie. Jeśli $m(r_i, z_i) = 0$, to po prostu bierze dowolny kamień i przekłada; niezmiennik w trywialny sposób pozostaje prawdziwy. Jeśli $m(r_i, z_i) \geqslant 1$, to gracz B przekłada $\lceil \frac{m(r_i,z_i)}{2} \rceil$ kamieni (to jest $\geqslant 1$) ze stosu, na którym jest co najmniej $m(r_i, z_i)$ kamieni, na drugi stos, zapewniając prawdziwość niezmiennika. W tym przypadku to już koniec rozważań, wracamy do punktu 1, czekając na kolejny ruch gracza A.

4. Teraz rozważamy już tylko sytuację, że po ruchu gracza A na parze stosów numer i nie ma już kamieni. Skoro na którymś ze stosów w tej parze jest co najmniej $m(r_i, z_i)$ kamieni, to $m(r_i, z_i) = 0$ (czyli jeśli $z_i = 0$ to $r_i \leqslant 1$, a jeśli $z_i > 0$, to $r_i = 0$). Mamy trzy podprzypadki:

   (a) Być może gra H już się skończyła. Wtedy także punkt 3 niezmiennika jest prawdziwy (bo $z_i = 0$). Jeśli także gra G już się skończyła, to po prostu kończymy symulację. Załóżmy, że G jeszcze trwa i wybierzmy dowolną niepustą parę stosów j. Gracz B przekłada tam jeden kamień ze stosu nie mniejszego na nie większy (czyli z większego na mniejszy lub między równymi). Niezmiennik zapewnia, że $z_j = 0$, czyli $m(r_j, z_j)$ jest nieparzyste lub jest zerem. Jeśli $m(r_j, z_j) = 0$, to niezmiennik oczywiście pozostaje spełniony. Jeśli $m(r_j, z_j) > 0$, to z niezmiennika wiemy, że na większym ze stosów musiało być co najmniej $\lfloor \frac{m(r_j,z_j)}{2} \rfloor + 1$ kamieni, po ruchu na każdym stosie pozostaje co najmniej $\lfloor \frac{m(r_j,z_j)}{2} \rfloor$ kamieni. Wracamy do punktu 1.

   (b) Przypuśćmy jednak teraz, że H jeszcze się nie skończyła. Wówczas $z_i > 0$, czyli $r_i = 0$. Gracz A wykonuje ruch w H: jeśli i-ty stos jest niepusty, to opróżnia i-ty, a jeśli pusty, to opróżnia dowolny niepusty stos. Swobodnie (tzn. nie martwiąc się o prawdziwość punktu 2 niezmiennika) możemy teraz zmniejszyć z_i do 1, przywracając prawdziwość niezmiennika (w szczególności punktu 3). Jeśli ten ruch nie kończy gry H, to gracz B odpowiada, usuwając pewną liczbę zapałek. Jeśli nie kończy przy tym gry H, to z definicji gry H gracz B usuwa przynajmniej jedną zapałkę z jakiegoś stosu j zawierającego co najmniej 2 zapałki. Załóżmy, że rzeczywiście gra H nadal się nie skończyła. Wtedy $z_j \geqslant 2$ (z punktu 2 niezmiennika) oraz $r_j \geqslant 1$ (z punktu 3 niezmiennika), co daje $m(r_j, z_j) \geqslant 1$. Zmniejszamy z_j o jeden; ponieważ usunęliśmy zapałkę, to nadal zapałek na j-tym stosie w H będzie nie więcej niż z_j (niezmiennik pozostaje prawdziwy). Jeśli $m(r_j, z_j)$ jest nieparzyste, to na j-tej parze stosów gracz B po prostu przekłada jeden kamień z nie mniejszego stosu na nie większy i niezmiennik (w szczególności punkt 1) pozostanie prawdziwy. Jeśli zaś $m(r_j, z_j)$ jest parzyste, to jest ściśle mniejsze niż $m(r_j, z_j + 1)$ (wartość przed zmniejszeniem z_j), więc

także gracz B po prostu przekłada jeden kamień z nie mniejszego stosu na nie większy i niezmiennik pozostanie prawdziwy. Wracamy do punktu 1.

(c) Pozostaje do rozważenia sytuacja, gdy podczas wykonywania punktu 4(b) gra H się skończyła (albo od razu po wykonaniu w niej ruchu gracza A, albo dopiero po ruchu gracza B). Wtedy zmieniamy wszystkie z_k na 0, co zapewnia poprawność punktu 4 niezmiennika dla każdego k. Jednocześnie pozostałe punkty niezmiennika pozostają prawdziwe. Następnie wracamy do punktu 4(a).

Nasza symulacja przebiega aż do momentu, gdy gra G się skończy. Widzimy, że może to nastąpić jedynie w punkcie 4(a), czyli już po tym, gdy gra H się skończyła. Pozostaje porównać liczbę ruchów gracza A w obu grach. Niech s_i będzie liczbą cyfr w zapisie dwójkowym liczby a_i. Początkowo r_i jest równe s_i+1. Liczbę r_i zmniejszamy jedynie w punkcie 2, po tym jak A zrobi ruch w G. Na koniec musi być $m(r_i, z_i) = 0$ (punkt 1 niezmiennika), czyli $r_i \leqslant 1$. Zmniejszenie wszystkich r_i do 1 wymaga więc co najmniej $\sum_i s_i = s$ ruchów A w G. Zobaczmy też, że A rusza się w H jedynie w punkcie 4(b), czyli wtedy, gdy $r_i = 0$; zatem gracz A, aby ruszyć się w H, musi także zrobić dodatkowy ruch w G. Czyli jeśli w H wykonał x ruchów, to w G musiał wykonać przynajmniej $x + s$ ruchów. ■

Optymalne strategie w grze H

Pozostaje nam udowodnienie, jak wyglądają optymalne strategie w grze H. Zaczniemy od gracza A, ale wcześniej udowodnimy lemat pomocniczy.

Lemat 2. *Rozważmy dwie sytuacje C i D w grze H, przy czym D powstaje z C przez usunięcie pewnych zapałek z pewnych stosów. Załóżmy, że z sytuacji C gracz A może zakończyć grę, wykonując co najwyżej x ruchów. Wówczas także z sytuacji D gracz A może zakończyć grę, wykonując co najwyżej x ruchów.*

Dowód: Jeśli w D jest koniec gry, to teza zachodzi w sposób trywialny.

Przyjrzyjmy się najpierw przypadkowi, że w C i D jest kolej gracza B. Zakładając, że w D jest ruch gracza B, który wymusza więcej niż x ruchów gracza A, rozważmy ten właśnie ruch gracza B. Prowadzi on do sytuacji D', z której A musi wykonać w najgorszym razie więcej niż x ruchów. Ale z C również istnieje ruch gracza B, który prowadzi do D': najpierw usuwamy pewne zapałki sprowadzając sytuację do D, a potem ruszamy się jak w D. To jest sprzeczne z założeniem, że z C gracz A może zakończyć grę, wykonując co najwyżej x ruchów.

Rozważmy teraz przypadek, że w C i D jest kolej gracza A. Weźmy ruch gracza A z sytuacji C, który zapewnia zakończenie gry w co najwyżej x ruchach. Prowadzi on do sytuacji C', powstającej z C przez opróżnienie stosu o pewnym numerze i; z C' gracz A może zakończyć grę, wykonując co najwyżej $x-1$ ruchów. Jeśli w D stos i-ty jest niepusty, to rozważamy ruch gracza A opróżniający stos i-ty; jeśli zaś jest pusty, to opróżniamy dowolny niepusty stos. W obu tych przypadkach sytuacja D' powstała po tym ruchu powstaje z C' poprzez usunięcie pewnych zapałek z pewnych stosów. Z przypadku pierwszego dostajemy, że z D' gracz A może zakończyć grę, wykonując

co najwyżej $x-1$ ruchów. Zatem z D gracz A może zakończyć grę, wykonując co najwyżej x ruchów. ■

Lemat 3. *Rozważmy sytuację w grze H, z której A może zakończyć grę, wykonując co najwyżej x ruchów. Wówczas, jeśli A będzie cały czas opróżniał stos zawierający najwięcej zapałek, to także zakończy grę, wykonując co najwyżej x ruchów.*

Dowód: Dowód jest przez indukcję po x. Jeśli jest akurat kolej na gracza B, to wykonujemy jego ruch, sprowadzając do sytuacji, w której jest kolej gracza A. Załóżmy, że jest kolej gracza A. Jeśli w optymalnej strategii gracz A także opróżnia stos zawierający najwięcej zapałek (strategie różnią się później), to korzystamy z założenia indukcyjnego dla $x-1$. Załóżmy, że optymalna strategia opróżnia stos numer i, prowadząc do sytuacji C, natomiast opróżnienie stosu numer j, mającego najwięcej zapałek, prowadzi do sytuacji D. Zamieniając numerami stosy numer i i j w D, możemy założyć, że D różni się od C tylko tym, że w C na j-tym stosie jest więcej zapałek niż w D. Zatem C i D podpadają pod lemat 2, który mówi, że z D gracz A może zakończyć grę, wykonując co najwyżej $x-1$ ruchów (bo z C gracz A może zakończyć grę, wykonując co najwyżej $x-1$ ruchów). ■

Następnie udowodnimy, że także gracz B powinien zawsze zabierać jedną zapałkę ze stosu zawierającego najwięcej zapałek (chyba że na każdym stosie jest już tylko co najwyżej jedna zapałka, to wtedy B musi usunąć wszystkie). Dowód jest podzielony na trzy lematy.

Lemat 4. *Rozważmy sytuację w grze H, z której jest ruch gracza B, przy czym istnieje stos zawierający więcej niż jedną zapałkę. Załóżmy, że B może wymusić, że A wykona więcej niż x ruchów do końca gry. Wówczas istnieje taki ruch B, który usuwa tylko jedną zapałkę i po którym nadal B może wymusić, że A wykona więcej niż x ruchów do końca gry.*

Dowód: Rozważmy ruch gracza B w strategii, która wymusza, że A wykona więcej niż x ruchów do końca gry; prowadzi on do pewnej sytuacji D. Ponieważ istnieje stos zawierający więcej niż jedną zapałkę, to ruch ten usuwa przynajmniej jedną nieostatnią zapałkę (definicja gry H). Rozważmy także alternatywny ruch gracza B, który usuwa tylko tę jedną zapałkę; prowadzi on do pewnej sytuacji C. Widzimy, że D powstaje z C przez usunięcie pewnych zapałek z pewnych stosów (lub $D=C$). Z lematu 2 wynika, że jeśli z C gracz A mógłby zakończyć grę, wykonując co najwyżej x ruchów, to mógłby także z D, co kończy dowód. ■

Lemat 5. *Rozważmy dwie sytuacje C i D w grze H, przy czym D powstaje z C poprzez przeniesienie jednej zapałki z pewnego stosu i zawierającego $z_i \geqslant 2$ zapałek na pewien stos j zawierający $z_j \geqslant z_i$ zapałek. Załóżmy, że z sytuacji C gracz A może zakończyć grę, wykonując co najwyżej x ruchów. Wówczas także z sytuacji D gracz A może zakończyć grę, wykonując co najwyżej x ruchów.*

Dowód: Indukcja po x. Rozważmy najpierw przypadek, że w C i D jest kolej gracza A. Z lematu 3 wiemy, że A może zapewnić sobie zakończenie gry z C w co najwyżej x ruchach, usuwając największy stos; oznaczmy go przez k. Jeśli $z_j > z_i$, to na pewno

$k \neq i$; również jeśli $z_j = z_i$, to możemy założyć, że $k \neq i$ (nie ma znaczenia, czy usuwamy stos i-ty, czy równoliczny z nim j-ty). Rozważmy także ruch gracza A z D polegający na opróżnieniu stosu k. Niech C' i D' to sytuacje po tych ruchach. Jeśli $k = j$, to D' powstaje z C' przez usunięcie jednej zapałki ze stosu i; wystarczy więc skorzystać z lematu 2. Jeśli zaś $k \neq j$ (oraz ciągle $k \neq i$), to nadal D' powstaje z C' poprzez przeniesienie jednej zapałki ze stosu i zawierającego $z_i \geqslant 2$ zapałek na stos j zawierający $z_j \geqslant z_i$ zapałek. Wystarczy teraz skorzystać z założenia indukcyjnego (gracz A z C' może zakończyć grę, wykonując co najwyżej $x - 1$ ruchów).

Przyjrzyjmy się teraz przypadkowi, że w C i D jest kolej gracza B. Zakładając, że z D jest ruch gracza B, który wymusza więcej niż x ruchów gracza A, rozważmy ten właśnie ruch gracza B. Dzięki lematowi 4 (na j-tym stosie w D mamy co najmniej 3 zapałki, więc założenia są spełnione) możemy założyć, że ruch ten polega na usunięciu jednej, nieostatniej, zapałki z jakiegoś stosu k. Wynikową sytuację oznaczmy D'. Jeśli $k = j$, to z C gracz B usuwa zapałkę ze stosu i (nieostatnią, bo $z_i \geqslant 2$), sprowadzając również C do D'. W przeciwnym przypadku z C gracz B usuwa zapałkę ze stosu k. Wynikowa sytuacja C' również ma tę własność, że D' powstaje z C' poprzez przeniesienie jednej zapałki ze stosu i zawierającego $z'_i \geqslant 2$ zapałek na stos j zawierający $z_j \geqslant z'_i$ zapałek. Faktycznie mamy $z'_i \geqslant 2$, bo jeśli $k \neq i$ to $z'_i = z_i$, a jeśli $k = i$, to na i-tym stosie w D musiały być przynajmniej 2 zapałki (bo usuwaliśmy nieostatnią), czyli w C co najmniej 3, czyli w C' znowu co najmniej 2. Możemy więc użyć pierwszego przypadku dla C' i D'. ■

Lemat 6. *Rozważmy sytuację w grze H, z której jest ruch gracza B. Załóżmy, że B może wymusić, że A wykona więcej niż x ruchów do końca gry. Wówczas B, aby to zrobić, może zacząć od usunięcia jednej zapałki z największego stosu (chyba że wszystkie stosy są rozmiaru co najwyżej* 1*; wtedy musi usunąć wszystkie zapałki).*

Dowód: Jeśli wszystkie stosy są rozmiaru co najwyżej 1, to gracz B nie ma wyboru. Załóżmy więc, że jednak istnieje jakiś stos o liczności co najmniej 2. Rozważmy ruch gracza B w strategii, która wymusza, że A wykona więcej niż x ruchów do końca gry; prowadzi on do pewnej sytuacji D. Dzięki lematowi 4 możemy założyć, że ruch ten polega na usunięciu jednej zapałki z pewnego stosu i. Rozważmy także alternatywny ruch B, który usuwa jedną zapałkę z największego stosu; niech jego numer to j, a sytuacja po tym ruchu to C. Jeśli stosy i i j są tak samo liczne, to nie ma co dowodzić. Jeśli zaś stos i jest mniejszy, to D powstaje z C poprzez przeniesienie jednej zapałki ze stosu i zawierającego $z_i \geqslant 2$ zapałek na pewien stos j zawierający $z_j \geqslant z_i$ zapałek. Mamy przy tym $z_i \geqslant 2$, bo skoro tworząc D usunęliśmy z i-tego stosu nieostatnią zapałkę, to w C muszą tam być co najmniej dwie zapałki. Z kolei $z_j \geqslant z_i$ wynika z tego, że w początkowej sytuacji stos j był najliczniejszy, a stos i mniej liczny od niego. Z lematu 5 wynika więc, że jeśli z C gracz A mógłby zakończyć grę, wykonując co najwyżej x ruchów, to mógłby także z D, co kończy dowód. ■

Rozwiązania częściowe

Dla bardzo małych danych wejściowych możemy napisać rozwiązanie, korzystając z ogólnych metod teorii gier (algorytm min-max), bez analizowania optymalnej strategii konkretnej gry. Wystarczy, że każda rozgrywka jest skończona (każdy ruch gracza A zmniejsza łączną liczbę kamieni na stosach o co najmniej 1, zaś ruchy gracza B nie zmieniają tej liczby). W pliku `wcas1.cpp` zaimplementowano tę metodę; rozwiązanie przechodzi drugie podzadanie.

W przypadku pierwszego podzadania (tylko jedna para stosów), wystarczy jedynie odkryć Obserwacje 1 i 2. Odpowiednie rozwiązanie jest zaimplementowane w pliku `wcab1.cpp`.

Z kolei w pliku `wcas2.cpp` zaimplementowane jest rozwiązanie, które korzysta z obserwacji, co najbardziej opłaca się graczom, gdy już wybiorą stos; wie także, który stos powinien wybierać gracz B. Dla gracza A bada wszystkie możliwości wyboru stosu. Rozwiązanie przechodzi pierwsze trzy podzadania.

Zawody III stopnia

opracowania zadań

Michał Włodarczyk
Treść zadania, Opracowanie

Bartosz Kostka
Program

Dostępna pamięć: 128 MB. *OI, etap III, dzień próbny, 12.04.2016*

Równoważne programy

Bajtazar dostał nowy komputer i uczy się go programować. Program składa się z ciągu instrukcji. Jest k różnych rodzajów instrukcji, które dla uproszczenia oznaczamy liczbami od 1 do k. Niektóre pary instrukcji mają tę własność, że jeśli występują w programie bezpośrednio obok siebie (w dowolnej kolejności), to zamieniając je miejscami, nie zmienia się działania programu (czyli uzyskuje się program **równoważny***). Pozostałe pary instrukcji nie mają tej własności i nazywamy je parami* **nieprzemiennymi***. Bajtazar napisał dwa programy o długości n instrukcji każdy i zastanawia się, czy są one równoważne. Pomóż mu!*

Wejście

W pierwszym wierszu standardowego wejścia znajdują się trzy liczby całkowite n, k oraz m pooddzielane pojedynczymi odstępami, oznaczające odpowiednio długość programów, liczbę różnych instrukcji komputera oraz liczbę par instrukcji nieprzemiennych.

Kolejne m wierszy zawiera opis tych par: każdy z tych wierszy zawiera dwie liczby całkowite a i b ($1 \leqslant a < b \leqslant k$) oddzielone pojedynczym odstępem, oznaczające, że para instrukcji o numerach a i b jest nieprzemienna. Możesz założyć, że każda para wystąpi w tym opisie co najwyżej raz.

Kolejne dwa wiersze przedstawiają opisy dwóch programów. Każdy z tych wierszy zawiera ciąg n liczb całkowitych $c_1, c_2, \ldots, c_n$ ($1 \leqslant c_i \leqslant k$) pooddzielanych pojedynczymi odstępami, oznaczających numery kolejnych instrukcji programu.

Wyjście

W jedynym wierszu standardowego wyjścia należy wypisać jedno słowo `TAK` *lub* `NIE` *w zależności od tego, czy podane na wejściu programy są równoważne.*

Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
TAK
```

natomiast dla danych wejściowych:

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

poprawnym wynikiem jest:

```
NIE
```

Wyjaśnienie do pierwszego przykładu: *W pierwszym programie można zamienić instrukcje na pozycjach 2 i 3, a następnie instrukcję na pozycji 5 z instrukcjami na pozycjach 4 i 3. W ten sposób uzyska się drugi program.*

Testy „ocen":

`1ocen`: $n = 50$, $k = 50$, $m = 1$; *programy to* $(1, 2, \ldots, 49, 50)$ *oraz* $(50, 49, \ldots, 2, 1)$; *odpowiedź* `NIE`.

`2ocen`: $n = 99\,999$, $k = 3$, $m = 1$; *instrukcje nieprzemienne to 1 i 2, a programy to* $(1, 2, 3, 1, 2, 3, \ldots, 1, 2, 3)$ *oraz* $(3, 1, 2, 3, 1, 2, \ldots, 3, 1, 2)$; *odpowiedź* `TAK`.

`3ocen`: $n = 100\,000$, $k = 1000$, $m = 50\,000$; *programy to* $(13, 13, 13, \ldots, 13)$ *oraz* $(37, 37, 37, \ldots, 37)$; *odpowiedź to oczywiście* `NIE`.

Ocenianie

Zestaw testów dzieli się na następujące podzadania. Testy do każdego podzadania składają się z jednej lub większej liczby osobnych grup testów. We wszystkich testach zachodzą warunki $1 \leqslant n \leqslant 100\,000$, $1 \leqslant k \leqslant 1000$, $0 \leqslant m \leqslant 50\,000$.

Podzadanie	Warunki	Liczba punktów
1	$n \leqslant 5$	*5*
2	$k \leqslant 2$	*5*
3	$n \leqslant 1000$	*25*
4	*brak dodatkowych warunków*	*65*

Rozwiązanie

Zadanie polega na rozstrzygnięciu, czy z jednego z danych programów da się otrzymać drugi za pomocą ciągu zamian sąsiednich instrukcji. Przeszkodą jest lista par instrukcji, których nie wolno ze sobą zamieniać. Jeśli istnieje ciąg zamian przekształcający jeden program w drugi, to programy nazywamy *równoważnymi*. Taka nazwa została wybrana nieprzypadkowo. Pojęcie *relacji równoważności* występuje powszechnie w matematyce i jest uogólnieniem relacji opisanej w zadaniu. Wprawdzie znajomość definicji relacji równoważności nie pomaga w żaden szczególny sposób w wymyśleniu efektywnego algorytmu, ale dostarcza języka pomocnego przy opisie rozwiązań.

Definicja 1. Relację $\approx$ nazywamy relacją równoważności, jeżeli jest ona:

1. **zwrotna**, czyli $x \approx x$ dla każdego x,
2. **symetryczna**, czyli $x \approx y$ zachodzi wtedy i tylko wtedy, gdy $y \approx x$,
3. **przechodnia**, czyli $x \approx y$ w połączeniu z $y \approx z$ implikuje $x \approx z$.

Przykładami relacji równoważności są: równość (znak $\approx$ zastępujemy wtedy znakiem $=$), posiadanie takiej samej reszty z dzielenia przez ustalony dzielnik (relacja równoważności określona na liczbach naturalnych) albo podobieństwo figur na płaszczyźnie. Zauważmy też, że opisana w treści zadania równoważność jest relacją równoważności na zbiorze programów. Jeśli oznaczymy ją przez $\approx$, a pod x, y, z podstawimy dowolne programy, to wszystkie trzy warunki powyższej definicji będą spełnione.

Przykładem relacji zwrotnej i symetrycznej, lecz niekoniecznie przechodniej, jest relacja przemienności instrukcji z zadania. Istotnie, może się okazać, że instrukcja p jest w relacji z instrukcjami q i r i może być zamieniona miejscami z każdą z nich, ale kolejność instrukcji q i r ma znaczenie. Za przykład relacji, która nie jest zwrotna ani symetryczna, lecz jest przechodnia, może posłużyć relacja mniejszości liczb ($<$).

Sortowanie z przeszkodami

W rozwiązaniu skorzystamy z przechodniości relacji równoważności i przekształcimy oba programy do prostszych równoważnych postaci, które będziemy umieli łatwo porównać. Gdyby wszystkie pary instrukcji były przemienne, wystarczyłoby posortować numery instrukcji w każdym programie[1] i sprawdzić, czy otrzymaliśmy takie same ciągi. Okazuje się, że podobny pomysł może zadziałać w ogólniejszym przypadku. Niech x' oznacza najmniejszy w porządku leksykograficznym program równoważny z x. Analogicznie dla programu y definiujemy y'. Jeśli $x \approx y$ (programy x i y są równoważne), to z przechodniości relacji równoważności mamy $x' \approx y'$, co z definicji daje $x' = y'$. Z drugiej strony, jeżeli $x' = y'$, to z przechodniości relacji wnioskujemy, że $x \approx y$.

Przykład 1. Zakładając, że nieprzemienne są pary instrukcji 1, 2 oraz 1, 3, następujące programy są równoważne:

$$x = 1,\ 4,\ 2,\ 3,\ 2,\ 1,\ 4,\ 2,\ 1,\ 4 \quad \text{oraz} \quad y = 4,\ 1,\ 4,\ 3,\ 2,\ 2,\ 1,\ 2,\ 1,\ 4.$$

Najmniejszy leksykograficznie program równoważny x oraz y to:

$$x' = y' = 1,\ 2,\ 2,\ 3,\ 1,\ 2,\ 1,\ 4,\ 4,\ 4.$$

Zanim zaczniemy konstruować algorytm, zauważmy, że mając dane dwa numery instrukcji, umiemy w czasie stałym rozstrzygnąć, czy odpowiadające im instrukcje są przemienne. Możemy chociażby trzymać wszystkie nieprzemienne pary w tablicy haszującej. Nie musimy jednak posuwać się do takich optymalizacji – jako że numery instrukcji należą do przedziału $[1, k]$ gdzie $k \leqslant 1000$, bez problemu zmieścimy w pamięci zwykłą tablicę dwuwymiarową indeksowaną numerami instrukcji.

Jak znaleźć najmniejszy leksykograficznie program równoważny z x? Najprościej zacząć od sprawdzenia, czy na pierwszą pozycję można wstawić instrukcję o numerze 1. Aby było to możliwe, taka instrukcja musi występować w programie i wszystkie instrukcje przed nią muszą być z nią przemienne. Jeśli nie jest to możliwe, sprawdzamy instrukcję o numerze 2 i tak dalej. Gdy znajdziemy pierwszą pasującą instrukcję,

[1] Wykorzystując algorytm sortowania przez zliczanie, można to wykonać w czasie $O(n + k)$.

usuwamy ją z programu i powtarzamy proces dla pozycji 2. Dla każdej z n pozycji musimy sprawdzić potencjalnie k kandydatów. Sprawdzenie jednego kandydata wymaga czasu $O(n)$. Prowadzi to do złożoności obliczeniowej algorytmu $O(n^2k)$, co jest dalece niesatysfakcjonujące.

Aby usprawnić algorytm, zawęźmy zbiór kandydatów na pierwszą instrukcję programu. Mogą to być jedynie te instrukcje, których nie poprzedzają żadne nieprzemienne z nimi. Możemy wyznaczyć taki zbiór w czasie $O(n^2)$. Po wyborze najmniejszej wartości na pierwszą pozycję, chcielibyśmy szybko uaktualnić zbiór kandydatów w celu wyłonienia najlepszej instrukcji na pozycję 2 i kontynuować ten proces, aż skonstruujemy x'.

Niech $S^x[i]$ oznacza liczbę instrukcji leżących przed pozycją i w programie x nieprzemiennych z instrukcją $x[i]$. Początkowo kandydaci znajdują się na pozycjach spełniających $S^x[i] = 0$. Kiedy wybierzemy instrukcję na początek programu (niech pochodzi ona z pozycji i_0), przestaje ona blokować dokładnie te instrukcje na pozycjach $i > i_0$, które nie są z nią przemienne, więc możemy uaktualnić dla nich wartości $S^x[i]$. Tym razem pozycje o $S^x[i]$ równym 0 będą zawierać kandydatów do przesunięcia na pozycję 2 i tak dalej.

Algorytm można najszybciej zrozumieć przez analizę poniższego pseudokodu. Gotowy kod znajduje się w pliku `rows3.cpp`.

```
1:  begin
2:    for i := 1 to n do begin
3:      for j := 1 to i − 1 do
4:        if not commute(x[i], x[j]) then
5:          S^x[i] := S^x[i] + 1;
6:      if S^x[i] = 0 then
7:        candidates_x.insert(i);
8:    end
9:    for pos := 1 to n do begin
10:     i_0 := candidates_x.getMin();
11:     candidates_x.popMin();
12:     x'[pos] := x[i_0];
13:     for i := i_0 + 1 to n do
14:       if not commute(x[i_0], x[i]) then begin
15:         S^x[i] := S^x[i] − 1;
16:         if S^x[i] = 0 then
17:           candidates_x.insert(i);
18:       end
19:   end
20: end
```

W pseudokodzie zakładamy, że tablica S^x jest początkowo wyzerowana oraz że dysponujemy funkcją *commute* rozstrzygającą, czy dane instrukcje są przemienne. Ponadto korzystamy ze struktury danych $candidates_x$ obsługującej następujące operacje:

- *insert*(i) : dodaje element i do struktury,
- *getMin*() : zwraca element i o najmniejszej wartości $x[i]$,

- *popMin*() : usuwa element zwracany przez *getMin*().

Najprostszą strukturą danych implementującą powyższe operacje jest lista. Możemy dodać do niej nowy element w czasie stałym, a wyszukiwanie najlepszego kandydata zajmuje czas liniowy. Zazwyczaj na zawodach algorytmicznych lepiej sprawdza się *kolejka priorytetowa*, pozwalająca wykonać wszystkie operacje w czasie co najwyżej logarytmicznym. Zauważmy jednak, że liczba wywołań funkcji *commute* jest rzędu $\Theta(n^2)$, a operacje na strukturze danych wywoływane są jedynie $O(n)$ razy (każdy element zostaje dodany co najwyżej raz). Zatem niezależnie od wyboru struktury danych otrzymujemy rozwiązanie działające w złożoności obliczeniowej $\Theta(n^2)$.

Przedstawiony algorytm można zoptymalizować tak, aby wykonywał jedynie $O(nk)$ operacji, przy wykorzystaniu faktu, że dla każdego rodzaju instrukcji warto rozważać jedynie jej najwcześniejsze wystąpienie w programie. Takie rozwiązanie wymaga jednak większej staranności w doborze struktur danych; jego opis pomijamy. Zamiast tego w następnej sekcji przedstawiamy odmienne rozwiązanie o takiej samej złożoności obliczeniowej, które posiada elegancki dowód poprawności i jest proste w implementacji. Niemniej jednak zachęcamy Czytelnika do próby wymyślenia szybszego sposobu obliczania x'.

Potęga niezmienników

Rozwiązanie wzorcowe wykorzystuje ciekawą własność relacji równoważności, jaką jest występowanie *niezmienników*.

Definicja 2. Dla relacji $\approx$ określonej na zbiorze X niezmiennikiem nazywamy funkcję $f : X \to Y$, spełniającą warunek $x \approx y \Rightarrow f(x) = f(y)$. Jeśli implikacja zachodzi w obie strony, to niezmiennik nazwiemy *silnym*.

Przykładami niezmienników są liczba kątów dla relacji podobieństwa wielokątów albo naiwnie posortowany ciąg instrukcji dla relacji z zadania. Łatwo znaleźć przykłady na to, że żaden z nich nie jest silny. Silnymi niezmiennikami są za to reszta z dzielenia przez p dla relacji przystawania modulo p tudzież uporządkowany ciąg kątów wewnętrznych dla relacji podobieństwa wielokątów. Analizowane w poprzedniej sekcji przyporządkowanie najmniejszego leksykograficznie równoważnego programu z definicji stanowi silny niezmiennik. Niezmiennik, którego chcemy użyć w rozwiązaniu wzorcowym, wymaga bardziej zaawansowanej konstrukcji.

Definicja 3. Dla programu x oraz numeru instrukcji p konstruujemy ciąg x_p następująco:

(1) zastępujemy każde wystąpienie p literą X,

(2) dopisujemy X na początku i na końcu programu x,

(3) pomiędzy każdy parą kolejnych liter X liczymy instrukcje nieprzemienne z p,

(4) tworzymy ciąg x_p, zapisując kolejno wartości obliczone w punkcie (3).

Twierdzenie 1. *Rodzina ciągów $(x_p)_{p=1}^k$ stanowi silny niezmiennik równoważności programów. Innymi słowy, programy x, y są równoważne wtedy i tylko wtedy, gdy dla każdego p zachodzi $x_p = y_p$.*

W powyższym twierdzeniu dwa ciągi uznajemy za równe, jeśli mają tyle samo elementów i elementy o tych samych indeksach są równe.

Dowód: Implikacja $\Rightarrow$ (wykazanie, że przyporządkowanie jest niezmiennikiem). Jeżeli x i y są równoważne, to istnieje sekwencja zamian sąsiednich instrukcji, które są przemienne, przeprowadzająca pierwszy program na drugi. Wystarczy wobec tego zauważyć, że zamiana przemiennych instrukcji nie może zmodyfikować żadnej wartości w żadnym ciągu x_p.

Implikacja $\Leftarrow$. Indukcja po liczbie instrukcji w programie. Programy x, y o długości 1 są równoważne wtedy i tylko wtedy, gdy składają się z tej samej instrukcji q. Dla tejże instrukcji zachodzi $x_q = y_q = (0, 0)$, dla pozostałych mamy zaś $x_p = y_p = (0)$.

Przypuśćmy teraz, że $n > 1$ oraz dla każdego p zachodzi $x_p = y_p$. Niech q będzie pierwszą instrukcją w programie x. Oznacza to, że pierwsza wartość w ciągu x_q (a zarazem y_q) to 0. W takim razie w programie y wszystkie instrukcje znajdujące się przed pierwszym wystąpieniem instrukcji q są z nią przemienne. Niech y' oznacza program otrzymany z y przez przesunięcie pierwszej instrukcji q na początek programu. Oczywiście $y \approx y'$.

Niech x'', y'' oznaczają programy otrzymane z x, y' przez usunięcie początkowej instrukcji q. Zauważmy, że ciąg x''_q różni się od x_q jedynie brakiem początkowego 0 i jest tożsamy z y''_q. Jeśli $p \neq q$ i p jest nieprzemienne z q, to ciągi x''_p, y''_p można otrzymać przez odjęcie 1 od pierwszego wyrazu w ciągach x_p, y'_p. Jeśli zaś p jest przemienne z q, $x''_p = x_p$ i $y''_p = y'_p$. Wobec tego dla każdego p zachodzi $x''_p = y''_p$ i z założenia indukcyjnego wnioskujemy, że $x'' \approx y''$. Dopisanie tej samej instrukcji na początku programu zachowuje równoważność ciągów, zatem $x \approx y'$. Ostatecznie korzystamy z przechodniości, aby otrzymać $x \approx y$. ■

Przykład 2. Dla programu $x = 1,\ 4,\ 2,\ 3,\ 2,\ 1,\ 4,\ 2,\ 1,\ 4$ z przykładu 2 (nieprzemienne są pary instrukcji 1, 2 oraz 1, 3) niezmiennikiem jest:

$$x_1 = (0, 3, 1, 0), \quad x_2 = (1, 0, 1, 1), \quad x_3 = (1, 2), \quad x_4 = (0, 0, 0, 0).$$

Przykładowo, wyznaczając x_1, zapisujemy następujący ciąg:

$$x = \mathrm{X},\ \mathrm{X},\ \cdot,\ \mathrm{N},\ \mathrm{N},\ \mathrm{N},\ \mathrm{X},\ \cdot,\ \mathrm{N},\ \mathrm{X},\ \cdot,\ \mathrm{X},$$

gdzie $\cdot$ oznacza instrukcję przemienną z 1, a N instrukcję nieprzemienną z 1.

Taki sam niezmiennik ma oczywiście program y z przykładu 2.

Rozwiązanie wzorcowe zaimplementowane w pliku `row.cpp` konstruuje wszystkie ciągi x_p, y_p, po czym je porównuje. Wymaga to przeiterowania po obu programach dla każdej wartości $1 \leqslant p \leqslant k$. Jako że umiemy sprawdzić w czasie stałym, czy dwie instrukcje są nieprzemienne, złożoność obliczeniowa algorytmu wynosi $O(nk)$.

Wojciech Nadara
Treść zadania, Opracowanie

Michał Glapa, Marek Sokołowski
Program

Dostępna pamięć: 256 MB.

OI, etap III, dzień pierwszy, 13.04.2016

Posłaniec

Bajtazar po długim okresie swojego panowania w królestwie Bajtocji stwierdził, że to zajęcie bardzo go wyczerpało, i ustąpił z tronu. Jednak przywykł on do życia w wyższych sferach i chciałby pozostać na bieżąco z najważniejszymi wiadomościami dotyczącymi królestwa i dworu. Dlatego postanowił zostać królewskim posłańcem.

Już pierwszego dnia na nowym stanowisku zlecono mu dostarczenie pilnej wiadomości pomiędzy dwoma miastami królestwa. Stwierdził on jednak, że w trakcie pracy zrobi sobie małą wycieczkę krajoznawczą i niekoniecznie pojedzie najkrótszą możliwą trasą. Oczywiście nie może dopuścić, aby nowy król się o tym dowiedział – w końcu posłaniec powinien dostarczać wiadomości tak szybko, jak to tylko możliwe!

Wszystkie połączenia drogowe w Bajtocji są jednokierunkowe. Bajtazar zna doskonale całe królestwo i wie, między którymi miastami istnieją połączenia drogowe. Zadeklarował on liczbę połączeń, których chciałby użyć, przejeżdżając pomiędzy miastami, i planuje on przejechać pomiędzy nimi dowolną ścieżką wymagającą dokładnie tylu połączeń (nie zważając na to, ilu połączeń tak naprawdę wymaga taka podróż). W trakcie swojej podróży Bajtazar nie może jednak pojawić się w początkowym ani końcowym mieście więcej niż raz, gdyż wtedy wzbudziłby podejrzenia u królewskich oficjeli. Może on jednak wielokrotnie pojawiać się w innych miastach, jak też wielokrotnie korzystać z tych samych połączeń drogowych.

Pomóż naszemu bohaterowi i napisz program, który obliczy dla niego, na ile sposobów może on zrealizować swoją wycieczkę krajoznawczą. Innymi słowy, program ma wyznaczyć liczbę różnych ścieżek o zadanej długości pomiędzy dwoma wybranymi miastami królestwa (przy czym w mieście początkowym i końcowym można pojawić się tylko raz). Ponieważ wynik zapytania może być całkiem duży, wystarczy, że program poda resztę z dzielenia wyniku przez pewną wybraną przez Bajtazara liczbę.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się trzy liczby całkowite n, m i z ($n \geqslant 2$, $0 \leqslant m \leqslant n(n-1)$, $2 \leqslant z \leqslant 1\,000\,000\,000$) pooddzielane pojedynczymi odstępami, oznaczające odpowiednio: liczbę miast w Bajtocji, liczbę jednokierunkowych połączeń między nimi oraz liczbę wybraną przez Bajtazara. Miasta numerujemy liczbami od 1 do n.

Dalej następuje m wierszy; każdy zawiera parę liczb całkowitych a, b ($1 \leqslant a, b \leqslant n$, $a \neq b$) oddzielonych pojedynczym odstępem, opisującą jednokierunkowe połączenie z miasta o numerze a do miasta o numerze b. Żadne połączenie nie jest podane na wejściu wielokrotnie.

W kolejnym wierszu wejścia znajduje się dodatnia liczba całkowita q oznaczająca liczbę zapytań Bajtazara. W każdym z następnych q wierszy znajduje się opis jednego zapytania. Opis taki składa się z trzech liczb całkowitych u_i, v_i i d_i ($1 \leqslant u_i, v_i \leqslant n$, $u_i \neq v_i$, $1 \leqslant d_i \leqslant 50$) pooddzielanych pojedynczymi odstępami, oznaczających, że Bajtazar ma przejechać z miasta o numerze u_i do miasta o numerze v_i, używając dokładnie d_i połączeń.

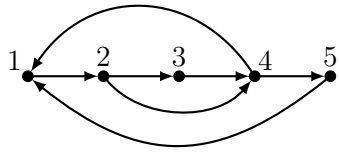
Wyjście

Na standardowe wyjście należy wypisać dokładnie q wierszy. W i-tym wierszu należy wypisać resztę z dzielenia przez z liczby ścieżek z i-tego zapytania.

Przykład

Dla danych wejściowych:

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



poprawnym wynikiem jest:

```
2
1
```

Wyjaśnienie do przykładu: *Dla pierwszego zapytania mamy dwie możliwe ścieżki: $2 \to 3 \to 4 \to 1$ oraz $2 \to 4 \to 5 \to 1$; dla drugiego zapytania tylko jedną: $5 \to 1 \to 2 \to 4 \to 1 \to 2 \to 3$.*

Testy „ocen”:

`1ocen`: *$n = 6$, $q = 10$, pięć miast połączonych ze sobą bezpośrednio w obie strony; szóste miasto niepołączone z żadnym innym;*

`2ocen`: *$n = 20$, $q = 100$, miasta w Bajtocji położone są na okręgu; każde dwa sąsiednie miasta na tym okręgu są połączone ze sobą bezpośrednio w obie strony;*

`3ocen`: *$n = 100$, $q = 500\,000$, mapa Bajtocji ma kształt trójzębu.*

Ocenianie

Zestaw testów dzieli się na następujące podzadania. Testy do każdego podzadania składają się z jednej lub większej liczby osobnych grup testów.

Podzadanie	Warunki	Liczba punktów
1	$n \leqslant 20$, $q \leqslant 100$	*12*
2	$n \leqslant 100$, $m \leqslant 500$, $q \leqslant 100$	*20*
3	$n \leqslant 100$, $q \leqslant 10\,000$	*38*
4	$n \leqslant 100$, $q \leqslant 500\,000$	*30*

Rozwiązanie

Sytuację opisaną w treści zadania można w naturalny sposób przedstawić za pomocą grafu skierowanego. Miasta Bajtocji odpowiadają wierzchołkom grafu, a połączenia – krawędziom. Zbiory wierzchołków i krawędzi grafu oznaczymy jako V oraz E (mamy $|V| = n$, $|E| = m$).

W opisie rozwiązania będziemy używali dwóch sposobów reprezentacji grafu. Pierwszy z nich to macierz sąsiedztwa, czyli tablica M rozmiaru $n \times n$, taka że $M[a][b] = 1$, jeżeli istnieje krawędź z wierzchołka a do wierzchołka b, natomiast w przeciwnym przypadku $M[a][b] = 0$. Drugi natomiast to listy sąsiedztwa, czyli w naszym przypadku tablica n list określających zbiory krawędzi wchodzących do poszczególnych wierzchołków grafu.

Początkowe spostrzeżenia

W opisie rozwiązania będziemy wielokrotnie posługiwali się wartościami typu „liczba sposobów dojścia od wierzchołka a do wierzchołka b za pomocą dokładnie k krawędzi". Oznaczmy taką wartość przez $ways(a, b, k)$. Zauważmy, że $ways(a, b, 0)$ jest równe 0 dla $a \neq b$ i 1 dla $a = b$. Ponadto mamy $ways(a, b, 1) = M[a][b]$. Zastanówmy się, jak można efektywnie obliczać wartości $ways(a, b, k)$ dla $k > 1$. Załóżmy, że chcemy przejść od wierzchołka a do wierzchołka b w k krokach. Wówczas któryś wierzchołek c odwiedzimy bezpośrednio przed wierzchołkiem b. Rozpatrując wszystkie takie wierzchołki c, że istnieje krawędź z c do b, jesteśmy w stanie stwierdzić, że

$$ways(a, b, k) = \sum_{cb \in E} ways(a, c, k - 1). \tag{1}$$

Wzór ten pozwala obliczyć wszystkie wartości $ways(a, b, k)$ dla $a, b \in V$, $0 \leqslant k \leqslant K$ za pomocą programowania dynamicznego, w kolejności rosnących wartości k. Mamy $O(n^2K)$ stanów. Aby oszacować liczbę przejść, zauważmy, że dla ustalonych a oraz k we wzorze rozważamy wszystkie wierzchołki b i dla każdego z nich wszystkie krawędzie wchodzące – dla ustalonych a i k mamy więc m przejść. Tak więc obliczenia wykonamy za pomocą list sąsiedztwa w złożoności $O(nmK)$.

Wyrażając (1) w terminach macierzy sąsiedztwa, możemy także napisać

$$ways(a, b, k) = \sum_{c \in V} ways(a, c, k - 1) \cdot M[c][b]. \tag{2}$$

W tym przypadku suma jest po wszystkich wierzchołkach c, a nie tylko po poprzednikach wierzchołka b, a fakt, czy istnieje krawędź od c do b, wyrażamy za pomocą domnożenia składnika przez $M[c][b]$. Można stąd wysnuć wniosek, że $ways(a, b, k) = M^k[a][b]$, gdzie M^k to macierz sąsiedztwa podniesiona do k-tej potęgi. Warto znać ten fakt, jednak w takiej konkretnie postaci nie będzie on nam dalej potrzebny.

Od tego momentu będziemy zakładać, że potrzebne nam wartości funkcji *ways* zostały obliczone na początku algorytmu.

Rozwiązanie brutalne

Jedno z możliwych rozwiązań zadania polega na użyciu programowania dynamicznego do każdego z zapytań oddzielnie. Ustalmy jedno z nich, z parametrami u, v, d. Będziemy obliczać wartości $dp[w][k]$ oznaczające, na ile sposobów można dojść od wierzchołka u do wierzchołka w za pomocą k krawędzi, nie odwiedzając w trakcie drogi ani u, ani v. Robimy to tak samo, jakbyśmy obliczali $ways(u, w, k)$, z jedyną różnicą, że pomijamy $w = u$ dla wszystkich wartości $k > 0$ oraz $w = v$ dla wszystkich wartości $k < d$. Takie rozwiązanie odpowiada na konkretne zapytanie w złożoności $O(md)$. Jeżeli zatem przez D oznaczymy największą wartość d_i występującą w zapytaniach, to możemy stwierdzić, że rozwiązanie działa w złożoności $O(mqD)$. Rozwiązanie o takiej złożoności wystarczało do przejścia pierwszych dwóch podzadań.

Rozwiązanie wzorcowe

Wprowadźmy najpierw kilka terminów związanych z grafami. Ścieżki w grafie, w których krawędzie i wierzchołki mogą się powtarzać (czyli takie, jakie interesują nas w tym zadaniu), nazywamy *marszrutami*. Ponadto *wnętrzem marszruty* $v_1v_2 \ldots v_{k-1}v_k$ nazwijmy zbiór wierzchołków $\{v_2, \ldots, v_{k-1}\}$. Będziemy mówili, że marszruta ma parametry u, v, d, jeżeli zaczyna się ona w wierzchołku u, kończy w wierzchołku v oraz przechodzi przez d krawędzi. Powiemy także, że zapytanie ma parametry u, v, d, jeżeli jesteśmy w nim proszeni o wyznaczenie liczby marszrut z takimi parametrami, które w swoim wnętrzu nie zawierają u ani v.

Główny pomysł rozwiązania wzorcowego jest dość ogólną metodą liczenia *dobrych obiektów* jako różnicy liczby *wszystkich obiektów* oraz *złych obiektów*. Ustalmy konkretne zapytanie z parametrami u, v, d. *Dobrymi obiektami* są w naszym przypadku marszruty z u do v o długości d odwiedzające u i v tylko na początku i końcu (odpowiednio). *Wszystkimi obiektami* będą marszruty z u do v o długości d, a *złymi obiektami* będą marszruty z u do v o długości d, których wnętrze zawiera u lub v. Niech zatem $good(a, b, k)$ i $bad(a, b, k)$ oznaczają odpowiednio liczbę dobrych i złych marszrut o długości k zaczynających się w a i kończących w b. Będziemy rozważać tylko $a, b \in \{u, v\}$ (przy czym potencjalnie może zachodzić $a = b$ lub $a = v$, $b = u$).

Złym wystąpieniem wierzchołka na marszrucie o parametrach a, b, k nazwijmy każde wystąpienie u lub v w jej wnętrzu. Przyjmijmy, że marszruta od a do b długości k jest zła. Rozważmy ostatnie złe wystąpienie wierzchołka na tej marszrucie. Załóżmy, że było nim odwiedzenie wierzchołka $w \in \{u, v\}$ po l krokach. Taka zła marszruta na odcinku swoich pierwszych l kroków mogła być dowolną marszrutą o parametrach a, w, l, natomiast na kolejnym odcinku o długości $k - l$ była dobrą marszrutą o parametrach $w, b, k - l$. Zatem sumarycznie takich złych marszrut jest $ways(a, w, l) \cdot good(w, b, k - l)$. Ostatnie złe wystąpienie mogło być wystąpieniem zarówno wierzchołka u jak i v oraz l mogło przyjąć jakąkolwiek wartość od 1 do $k - 1$. Stąd wniosek, że

$$\begin{aligned} bad(a, b, k) = {} & (ways(a, u, 1) \cdot good(u, b, k - 1) + \ldots + ways(a, u, k - 1) \cdot good(u, b, 1)) \\ & + (ways(a, v, 1) \cdot good(v, b, k - 1) + \ldots + ways(a, v, k - 1) \cdot good(v, b, 1)). \end{aligned}$$

Natomiast $good(a, b, k) = ways(a, b, k) - bad(a, b, k)$. Dzięki tym wzorom, dla zapytania o parametrach u, v, d możemy napisać krótki pseudokod realizujący programowanie dynamiczne liczące wszystkie interesujące nas wartości $good(a, b, k)$, gdzie $a, b \in \{u, v\}$, $1 \leqslant k \leqslant d$.

```
1: for k := 1 to d do
2:   foreach a in {u, v} do
3:     foreach b in {u, v} do
4:       good[a][b][k] := ways(a, b, k);
5:       foreach last in {u, v} do
6:         for l := 1 to k − 1 do
7:           good[a][b][k] := good[a][b][k] − ways(a, last, l) · good[last][b][k − l];
```

Interesująca nas odpowiedź będzie obliczona w polu $good[u][v][d]$.

Na całe rozwiązanie składa się preprocessing tablicujący wszystkie wartości funkcji $ways(a, b, k)$ dla $k \leqslant D$ w złożoności $O(nmD)$ oraz odpowiadanie na zapytania w złożoności $O(qd^2)$, co daje rozwiązanie w złożoności $O(nmD + qD^2)$.

Wojciech Nadara
Treść zadania, Opracowanie

Paweł Czerwiński
Program

Dostępna pamięć: 128 MB. *OI, etap III, dzień pierwszy, 13.04.2016*

Pracowity Jaś

Jaś miał niedawno urodziny. Jak w każdym szanującym się zadaniu algorytmicznym, Jaś nie dostał w prezencie ani zabawek, ani gier, ani komputera, a jedynie długie tablice wypełnione liczbami, drzewa, mapy różnych krajów z drogami prowadzącymi tunelami i estakadami oraz długie taśmy z wypisanymi 1048576 początkowymi literami słów Fibonacciego i Thuego-Morse'a. Najbardziej z tych wszystkich prezentów spodobała mu się tablica z wypisaną permutacją[1] *pierwszych n liczb naturalnych. Zaczął się zastanawiać, jaka jest poprzednia permutacja w porządku leksykograficznym*[2]*. Po chwili udało mu się ją wymyślić i zapragnął zapisać ją na tej samej tablicy. Jaś potrafi w jednym kroku zamienić miejscami jedynie dwie z liczb zapisanych na tablicy (gdyby operował na większej liczbie liczb naraz, to by się pogubił). Jest jednak na tyle mądry, że przekształcił początkową permutację w poprzednią leksykograficznie, wykonując minimalną liczbę takich zamian. Gdy to zrobił, wpadł w permutacyjny szał i zaczął powtarzać tę operację w kółko, zapisując na tablicy kolejne wcześniejsze permutacje w porządku leksykograficznym!*

Niestety, po pewnym czasie Jaś będzie musiał przerwać swoją zabawę, gdyż dojdzie do permutacji $1, 2, \ldots, n$*, która jest najmniejsza w porządku leksykograficznym. Przyjaciele Jasia trochę się naśmiewają z jego monotonnej zabawy, jednak wiedzą, że nie mają szans go z niej wyrwać. Chcieliby się więc dowiedzieć, kiedy Jaś skończy. Pomóż kolegom Jasia i powiedz im, ile potrwa jego zabawa, jeśli każda zamiana zajmuje mu dokładnie sekundę. Jako że ta rozrywka może być dość długa (Jaś nie bez powodu ma przydomek Pracowity), wystarczy im reszta z dzielenia tej liczby przez* $10^9 + 7$*. W końcu* $10^9 + 7$ *sekund to na tyle długo, że są w stanie co tyle czasu sprawdzać, czy Jaś już skończył swoją zabawę.*

Wejście

W pierwszym wierszu standardowego wejścia znajduje się dodatnia liczba całkowita n oznaczająca długość permutacji, którą Jaś otrzymał na urodziny. W drugim wierszu znajduje się ciąg n parami różnych liczb naturalnych $p_1, p_2, \ldots, p_n$ $(1 \leqslant p_i \leqslant n)$ *pooddzielanych pojedynczymi odstępami, opisujący tę permutację.*

Wyjście

Twój program powinien wypisać na standardowe wyjście resztę z dzielenia przez $10^9 + 7$ *liczby zamian, które wykona Jaś, zanim jego zabawa się skończy.*

[1] Permutacja liczb od 1 do n to ciąg różnych liczb całkowitych $p_1, \ldots, p_n$ spełniających $1 \leqslant p_i \leqslant n$ (każda liczba całkowita od 1 do n występuje w takiej permutacji dokładnie raz).

[2] Permutacja $P = (p_1, \ldots, p_n)$ jest wcześniej w porządku leksykograficznym niż permutacja $Q = (q_1, \ldots, q_n)$ (co zapiszemy jako $P < Q$), jeżeli $p_j < q_j$, gdzie j jest najmniejszym takim indeksem, że $p_j \neq q_j$. Permutacja P poprzedza permutację Q w porządku leksykograficznym, jeżeli $P < Q$ oraz nie istnieje taka permutacja R, że $P < R < Q$.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`3` `3 1 2`	`6`

Wyjaśnienie do przykładu: *Na tablicy Jasia będą się po kolei pokazywały permutacje $(2,3,1)$, $(2,1,3)$, $(1,3,2)$, $(1,2,3)$. Aby je uzyskiwać, będzie musiał wykonać łącznie $2+1+2+1=6$ zamian.*

Testy „ocen":

`1ocen`: $1, 2, 3, \ldots, 10$

`2ocen`: *losowa 5-elementowa permutacja*

`3ocen`: $100, 99, 98, \ldots, 1$.

Ocenianie

Zestaw testów dzieli się na następujące podzadania. Testy do każdego podzadania składają się z jednej lub większej liczby osobnych grup testów.

Podzadanie	Warunki	Liczba punktów
1	$n \leqslant 10$	*15*
2	$n \leqslant 5000$	*37*
3	$n \leqslant 1\,000\,000$, *permutacja to* $n, n-1, \ldots, 1$	*15*
4	$n \leqslant 1\,000\,000$	*33*

Rozwiązanie

Poprzednia leksykograficznie permutacja

Pierwszym pytaniem, jakie należy sobie postawić, jest „Dla danej permutacji $P = (p_1, \ldots, p_n)$, jak wygląda jej leksykograficzny poprzednik?". Nazwijmy go Q. Wśród wszystkich permutacji mniejszych leksykograficznie od P, Q jest ma maksymalny wspólny prefiks (początkowy kawałek) z P. Na początek chcielibyśmy zidentyfikować długość tego prefiksu. Podzielmy naszą permutację P na pewien prefiks oraz sufiks (końcowy kawałek) i oznaczmy je jako A i B. Leksykograficznie mniejsza od P permutacja o prefiksie równym A istnieje wtedy i tylko wtedy, gdy ciąg B nie jest posortowany rosnąco. Stąd możemy wysnuć wniosek, że P i Q będą miały wspólny prefiks o długości $k-1$, gdzie k to największy taki indeks, że $p_k > p_{k+1}$ (jeżeli P nie jest permutacją $(1, \ldots, n)$, która nie ma leksykograficznego poprzednika, to taki indeks istnieje). Sufiks $(p_k, \ldots, p_n)$ zaczynający się od k-tej pozycji nazwijmy *ogonem* permutacji P. Permutację Q możemy opisać w następujący sposób: ma ona wspólny prefiks z P o długości $k-1$ (tzn. $q_i = p_i$ dla $i = 1, \ldots, k-1$), q_k jest największą liczbą ze zbioru $\{p_{k+1}, \ldots, p_n\}$ mniejszą od p_k oraz $(q_{k+1}, \ldots, q_n)$ jest posortowanym

malejąco ciągiem złożonym z liczb ze zbioru $\{1, \ldots, n\}$, które nie występują w zbiorze $\{q_1, \ldots, q_k\}$.

Przykładowo dla permutacji $P = (2, 4, 1, 3, 5)$ mamy $k = 2$, zatem $q_1 = p_1 = 2$, największą liczbą z liczb $\{1, 3, 5\}$, która jest mniejsza od $p_2 = 4$, jest $q_2 = 3$, a $(q_3, q_4, q_5) = (5, 4, 1)$. Zatem bezpośrednim poprzednikiem leksykograficznym permutacji P jest permutacja $Q = (2, 3, 5, 4, 1)$. W dalszej części rozwiązania, bezpośredniego leksykograficznego poprzednika permutacji P będziemy oznaczać przez $prev(P)$.

Minimalna liczba zamian

Skoro już wiemy, jak wygląda leksykograficzny poprzednik permutacji P, to zastanówmy się, jaka jest minimalna liczba zamian par liczb, która pozwoli nam przejść od P do $prev(P)$.

Przyjmijmy, że ogon P to sufiks złożony z elementów o indeksach $[k, n]$. Przypomnijmy, że k możemy wyznaczyć jako największy taki indeks, że $p_k > p_{k+1}$. Na potrzeby tej sekcji dla prostoty załóżmy, że ogon ten jest permutacją liczb od 1 do t, i niech jego pierwszy element będzie równy r (mamy $r > 1$). Wtedy ogon wygląda następująco:

$$S_1 = (r, 1, \ldots, r-2, r-1, r+1, \ldots, t),$$

a jego poprzednik to

$$S_2 = (r-1, t, t-1, \ldots, r+1, r, r-2, \ldots, 1).$$

Zauważmy, że od S_1 do S_2 możemy przejść, najpierw zamieniając miejscami liczby r oraz $r-1$, a następnie odwracając sufiks $1, \ldots, r-2, r, r+1, \ldots, t$, otrzymując żądany $t, t-1, \ldots, r+1, r, r-2, \ldots, 1$. Odwrócenie ciągu c-elementowego można wykonać za pomocą $\lfloor \frac{c}{2} \rfloor$ operacji zamiany dwóch elementów[1]. W naszym przypadku $c = t - 1$, zatem jeżeli P ma ogon długości t, to potrafimy przejść z P do $prev(P)$ za pomocą $1 + \lfloor \frac{t-1}{2} \rfloor = \lfloor \frac{t+1}{2} \rfloor$ operacji.

Udowodnimy teraz, że jest to minimalna liczba takich operacji. Jest jasne, że jeżeli chcemy z pewnego ciągu A otrzymać ciąg B za pomocą operacji zamiany par elementów i te ciągi różnią się na d pozycjach, to potrzebujemy co najmniej $\lceil \frac{d}{2} \rceil$ operacji. Przekonamy się, że takie oszacowanie wystarczy nam do udowodnienia tezy.

Zauważmy na starcie, że ciągi S_1 i S_2 różnią się na pozycjach, na których w S_1 stoją liczby r i $r-1$. Dalej rozważymy dwa przypadki ze względu na parzystość t.

1. $t = 2l + 1$:

   Jeżeli ponumerujemy pozycje obu ciągów indeksami od 1 do t, to wtedy w S_1 liczby od 1 do $r-2$ stoją na pozycjach o parzystościach różnych od swoich wartości, a liczby od $r+1$ do t stoją na pozycjach o parzystościach takich samych jak swoje wartości. W przypadku S_2 mamy do czynienia z dokładnie odwrotną sytuacją. Stąd wniosek, że S_1 i S_2 różnią się na wszystkich pozycjach, zatem na otrzymanie S_2 z S_1 potrzeba co najmniej $\lceil \frac{t}{2} \rceil = l + 1 = \lfloor \frac{t+1}{2} \rfloor$ zamian.

[1] Przypomnijmy standardowe oznaczenia matematyczne: $\lfloor x \rfloor$ oraz $\lceil x \rceil$ oznaczają odpowiednio zaokrąglenia x do najbliższej liczby całkowitej w dół i w górę.

2. $t = 2l$:

   Po usunięciu pierwszych elementów z S_1 oraz S_2, pierwszy z nich staje się ciągiem rosnącym, a drugi malejącym. Ciąg rosnący i ciąg malejący mogą mieć co najwyżej jedną pozycję wspólną. Stąd wniosek, że S_1 i S_2 różnią się na co najmniej $t-1$ pozycjach, zatem do otrzymania S_2 z S_1 potrzeba co najmniej $\lceil \frac{t-1}{2} \rceil = l = \lfloor \frac{t+1}{2} \rfloor$ zamian.

W obu przypadkach oszacowaliśmy z dołu liczbę potrzebnych zamian przez liczbę, którą otrzymujemy we wcześniej opisanej metodzie, co dowodzi tego, że przedstawiony algorytm wykonuje minimalną liczbę zamian.

W szczególności dowiedzieliśmy się, że liczba zamian potrzebnych do przekształcenia S_1 w S_2 nie zależy od r, a jedynie od długości tych ciągów t. Pozwala nam to zdefiniować funkcję $g(t) = \lfloor \frac{t+1}{2} \rfloor$, która oznacza najmniejszą potrzebną liczbę zamian do przejścia od permutacji P do $prev(P)$, jeśli ogon permutacji P ma długość t.

Obliczenie sumarycznej długości zabawy

Mając do dyspozycji wniosek z poprzedniego akapitu, jesteśmy już w stanie napisać pewne (bardzo wolne) rozwiązanie. Dopóki nie otrzymamy permutacji identycznościowej, przechodzimy do poprzedniej leksykograficznie permutacji i zwiększamy liczbę potrzebnych zamian o odpowiednią wartość. Jednak takie rozwiązanie może działać bardzo długo, konkretnie w złożoności $O(n! \cdot n)$ lub $O(n!)$, jeżeli będziemy wyznaczali długość ogonu, idąc od końca permutacji. (Dociekliwemu Czytelnikowi pozostawiamy jako nieoczywiste zadanie udowodnienie, że średnią długość ogonu możemy ograniczyć przez stałą). Takie rozwiązanie wystarcza jedynie do rozwiązania pierwszego podzadania; w ogólności potrzebujemy czegoś zdecydowanie szybszego.

Na początku zauważmy, że analogiczny problem możemy zdefiniować dla dowolnych ciągów różnych liczb. Ciąg różnych liczb o długości m, tak samo jak ciąg wszystkich liczb od 1 do m, można spermutować na $m!$ sposobów i algorytm znajdowania leksykograficznego poprzednika wygląda w tym przypadku analogicznie. Dla ciągu $B = (b_1, \ldots, b_m)$ składającego się z różnych liczb możemy zdefiniować ciąg $compr(B) = C = (c_1, \ldots, c_m)$, gdzie c_i oznacza, ile liczb w ciągu B jest równych co najwyżej b_i. Ciąg C jest permutacją liczb od 1 do m. Na przykład $compr((7,3,5)) = (3,1,2)$. Jeżeli dla dowolnego ciągu różnych liczb $D = (d_1, \ldots, d_m)$ przez $ans(D)$ oznaczymy odpowiedź dla problemu analogicznego do problemu z treści, to jest jasne, że $ans(D) = ans(compr(D))$.

Przez $f(n)$ oznaczmy $ans((n, n-1, \ldots, 1))$. Zauważmy, że zaczynając od ciągu $(n, n-1, \ldots, 1)$, w trakcie zabawy Jaś napotka permutację $(n, 1, \ldots, n-1)$, a do tego momentu wykona $f(n-1)$ zamian. Ogon tej permutacji ma długość n, zatem w następnym kroku będzie musiał wykonać $g(n)$ zamian, otrzymując permutację $(n-1, n, n-2, n-3, \ldots, 1)$. Po pewnej liczbie kroków dojdzie do permutacji $(n-1, 1, \ldots, n-3, n-2, n)$, wykonując kolejne $f(n-1)$ zamian. Ogon tej permutacji ponownie ma długość n, zatem aby przejść do $(n-2, n, n-1, n-3, \ldots, 1)$, musi wykonać kolejne $g(n)$ zamian. Kontynuując to rozumowanie, dochodzimy do wniosku, że:

$$f(n) = n \cdot f(n-1) + (n-1) \cdot g(n),$$

co pozwala nam w czasie $O(n)$ wyznaczyć reszty z dzielenia $f(1), f(2), \ldots, f(n)$ przez zadaną stałą (wiedząc, że $f(1) = 0$). Otrzymujemy zarazem rozwiązanie podzadania 3.

Rozwiązanie ogólnego problemu wcale nie jest wiele trudniejsze. Jeżeli rozwiązujemy problem dla permutacji $P = (p_1, \ldots, p_n)$, to po pewnej liczbie zamian dojdziemy do ciągu $(p_1, 1, \ldots, p_1 - 1, p_1 + 1, \ldots, n)$ i od tego momentu wykonamy $(p_1 - 1) \cdot (f(n-1) + g(n))$ zamian, aby dojść do permutacji identycznościowej. Możemy zatem stwierdzić, że:

$$\begin{aligned} ans(P) &= ans((p_2, \ldots, p_n)) + (p_1 - 1) \cdot (f(n-1) + g(n)) \\ &= ans(compr((p_2, \ldots, p_n))) + (p_1 - 1) \cdot (f(n-1) + g(n)). \end{aligned}$$

Wartość $ans(compr((p_2, \ldots, p_n))$ możemy wyznaczyć, obliczając najpierw permutację $compr((p_2, \ldots, p_n))$ i wywołując się rekurencyjnie. Obliczenie permutacji $compr((p_2, \ldots, p_n))$ możemy łatwo wykonać w czasie $O(n)$ (pamiętajmy, że zbiór liczb $\{p_2, \ldots, p_n\}$ to zbiór liczb od 1 do n oprócz p_1), co daje nam rozwiązanie w czasie $O(n^2)$. Jest to zdecydowaną poprawą w stosunku do rozwiązania $O(n!)$ i pozwala zaliczyć podzadanie 2, ale wciąż nie wystarcza na zdobycie kompletu punktów.

Zauważmy jednak, że do obliczenia liczby zamian różniących $ans(P)$ oraz $ans((p_2, \ldots, p_n))$ potrzebujemy jedynie znać wartość elementu p_1. Jeżeli rozpatrujemy problem dla ogólnych ciągów różnych elementów, to wtedy analogiczna wersja wzoru z poprzedniego akapitu przedstawia się jako $ans(P) = ans((p_2, \ldots, p_n)) + s \cdot (f(n-1) + g(n))$, gdzie s to liczba elementów ciągu P mniejszych od p_1. Przewaga takiego zapisu polega na tym, że nie musimy wykonywać kosztownej kompresji przed wywołaniem rekurencyjnym. Niestety ma on też swoją wadę, mianowicie musimy znać wartość s. Zauważmy, że jeżeli oryginalną permutacją z zadania jest $(p_1, \ldots, p_n)$, to będziemy się wywoływać rekurencyjnie jedynie dla jej sufiksów. Po chwili zastanowienia możemy dojść do wniosku, że tak naprawdę wynik naszego zadania przedstawia się jako

$$s(1) \cdot (f(n-1) + g(n)) + s(2) \cdot (f(n-2) + g(n-1)) + \ldots + s(n-1) \cdot (f(1) + g(2)),$$

gdzie $s(i)$ to liczba liczb mniejszych od p_i spośród liczb $p_{i+1}, \ldots, p_n$. Jedyne, co musimy zatem zrobić, to efektywnie obliczyć wartości $s(i)$.

Będziemy potrzebowali struktury danych wspierającej zapytania $insert(i)$ oraz $less(i)$, gdzie $insert(i)$ dodaje element i do zbioru, a $less(i)$ odpowiada na zapytanie „Ile liczb mniejszych od i dodaliśmy już do zbioru?". Co więcej, w naszym zadaniu argumenty operacji są liczbami całkowitymi z przedziału $[1, n]$. Biorąc to wszystko pod uwagę, widzimy, że otrzymany problem to standardowy przykład na użycie drzew potęgowych lub przedziałowych (typu „dodaj punkt, sumuj na przedziale"), które to już wielokrotnie występowały w zadaniach z Olimpiady Informatycznej. Za pomocą podanych typów drzew możemy obsługiwać takie zapytania w złożoności czasowej $O(\log n)$. Do struktury danych będziemy dodawali elementy od prawej do lewej. Będziemy kolejno wykonywać polecenia $less(p_n), insert(p_n), less(p_{n-1}), insert(p_{n-1}), \ldots, less(p_1), insert(p_1)$. Odpowiedź na zapytanie $less(p_i)$ jest wartością $s(i)$.

Podsumowując, otrzymujemy rozwiązanie w złożoności czasowej $O(n \log n)$, które wystarczało do uzyskania kompletu punktów.

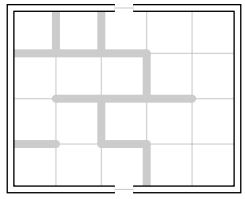
Tomasz Idziaszek Treść zadania | **Juliusz Straszyński** Opracowanie | **Marek Sommer** Program

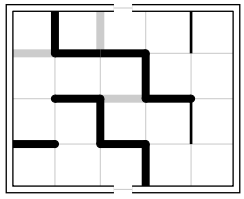
Dostępna pamięć: 128 MB. *OI, etap III, dzień pierwszy, 13.04.2016*

Żywopłot

Królewski ogrodnik Bajtazar ma za zadanie wyhodować w królewskim ogrodzie labirynt z żywopłotu. Ogród można podzielić na $m \times n$ kwadratowych pól. Otoczony jest murem, w którym na środku północnej i południowej ściany są wejścia. Na każdej krawędzi dzielącej dwa pola można zbudować kawałek żywopłotu – z cisu lub tui. Król bardziej lubi cis, więc chciałby mieć w swoim ogrodzie jak najwięcej kawałków żywopłotu z cisu. Niestety, cis wymaga lepszej gleby, więc nie wszędzie go można posadzić.

Aby żywopłot tworzył labirynt, musi spełniać dodatkowy warunek: do każdego pola musi być możliwość dojścia z obu wejść i, co więcej, tylko na jeden sposób. (Z danego pola można przejść bezpośrednio na pole sąsiadujące, jeśli na dzielącej te pola krawędzi nie znajduje się kawałek żywopłotu. Dwa sposoby dojścia uznajemy za różne, jeśli przechodzą przez różne zbiory pól.)





W lewej części powyższego rysunku przedstawiono przykładowy ogród dla $m = 4$ i $n = 5$, zawierający 31 krawędzi. Wyróżniono w nim 13 krawędzi, na których można posadzić żywopłot z cisu.

Na prawej części rysunku przedstawiono przykładowy labirynt składający się z 12 kawałków żywopłotu, z których 10 jest żywopłotem z cisu, a 2 są żywopłotem z tui. Nie istnieje labirynt zawierający więcej kawałków z cisu. Twoim zadaniem będzie napisanie programu, który pomoże Bajtazarowi w zaprojektowaniu labiryntu.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie liczby całkowite m i n oznaczające rozmiar ogrodu ($2 \leqslant m, n$ oraz n jest liczbą nieparzystą). Kolejne m wierszy zawiera po $n-1$ znaków, opisujących pionowe krawędzie (czytane rzędami, od lewej do prawej). Znak `C` *oznacza, że na danej krawędzi można posadzić żywopłot z cisu, a znak* `T` *oznacza, że można posadzić żywopłot z tui. Kolejne $m-1$ wierszy zawierające po n znaków opisuje poziome krawędzie (również czytane rzędami, od lewej do prawej).*

Wyjście

W pierwszym wierszu standardowego wyjścia należy wypisać dwie liczby całkowite: liczbę posadzonych kawałków żywopłotu tworzących labirynt oraz maksymalną liczbę kawałków żywopłotu

z cisu. W kolejnych $2m-1$ wierszach należy opisać krawędzie labiryntu (w kolejności jak na wejściu). Należy wypisać znak `Z`*, jeśli krawędź zawiera żywopłot, lub znak* `.` *(kropka) w przeciwnym wypadku.*

Jeśli istnieje wiele rozwiązań spełniających warunki króla, należy wypisać dowolne z nich.

Przykład

Dla danych wejściowych:	*jednym z poprawnych wyników jest:*
`4 5`	`12 10`
`CCTT`	`Z..Z`
`TTCT`	`..Z.`
`TCTT`	`.Z.Z`
`TTCT`	`..Z.`
`CCCTT`	`.ZZ..`
`TCCCT`	`.Z.Z.`
`CTCTT`	`Z.Z..`

Wyjaśnienie do przykładu: *Dane wejściowe opisują ogród z lewej części rysunku; wynik opisuje labirynt z prawej części rysunku.*

Testy „ocen”:

`1ocen`: *$m = 4$, $n = 3$, w każdym miejscu można posadzić cis;*

`2ocen`: *$m = 100$, $n = 99$, na pionowych krawędziach można posadzić cis, na poziomych można posadzić tuję;*

`3ocen`: *$m = 1000$, $n = 999$, w każdym miejscu można posadzić tuję.*

Ocenianie

Zestaw testów dzieli się na następujące podzadania. Testy do każdego podzadania składają się z jednej lub większej liczby osobnych grup testów.

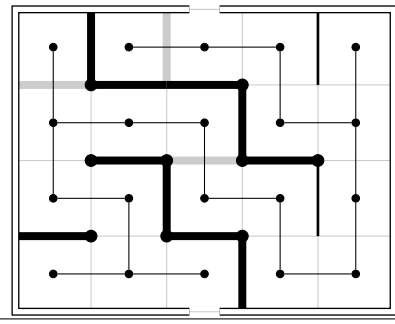
Jeśli Twój program wypisze poprawny pierwszy wiersz, a dalsza część wyjścia nie będzie poprawna, uzyska 52% punktów za dany test. W szczególności, aby uzyskać 52% punktów za test, wystarczy wypisać tylko jeden wiersz wyjścia.

Podzadanie	**Warunki**	**Liczba punktów**
1	$n \cdot m \leqslant 12$	*25*
2	$n, m \leqslant 100$	*25*
3	$n, m \leqslant 1000$	*50*

Rozwiązanie

Przypomnijmy definicję labiryntu: z każdego pola da się dojść do obu wejść na dokładnie jeden sposób. Sytuację z zadania możemy przedstawić jako graf nieskierowany, którego wierzchołki odpowiadają polom, a krawędzie łączą pary sąsiednich pól,

które *nie są* oddzielone kawałkiem żywopłotu. Graf taki jest więc labiryntem, gdy dla każdego wierzchołka istnieje dokładnie jedna ścieżka do wierzchołka oznaczonego wejściem oraz dokładnie jedna do wierzchołka oznaczonego wyjściem. Jest to tożsame z tym, że graf musi być spójny i bez cykli, bo każdy cykl w grafie spójnym oznaczałby istnienie takiego wierzchołka, który ma co najmniej dwie ścieżki do wejścia/wyjścia. Innymi słowy, graf musi być *drzewem* (rys. 1). Liczba wejść do labiryntu nie ma znaczenia – ta interpretacja byłaby prawdziwa również, gdybyśmy mieli jedno, czy też więcej niż dwa wejścia.



Rys. 1: Chcemy znaleźć takie drzewo, które połączy wszystkie pola i usunie z żywopłotu jak najmniej krawędzi z cisem.

Zatem musimy znaleźć drzewo łączące wszystkie wierzchołki o najmniejszym koszcie. Jeżeli w ostatecznym drzewie między sąsiednimi wierzchołkami jest krawędź, to nie posadzimy tam żywopłotu. W zadaniu chcemy mieć jak najwięcej kawałków żywopłotu, które są cisami. Oznacza to, że w drzewie chcemy użyć jak najmniej cisów, czyli koszt krawędzi odpowiadającej cisowi to 1, a krawędzi odpowiadającej tui to 0.

Zadanie sprowadza się więc do klasycznego problemu znajdowania minimalnego drzewa rozpinającego (ang. *MST – minimum spanning tree*). By je rozwiązać, możemy użyć znanych algorytmów znajdujących MST, np. algorytmu Kruskala [6] (`zyw2.cpp`, `zyw5.cpp`) czy też Prima [6] (`zyw3.cpp`). Dla grafu o zbiorze wierzchołków V i zbiorze krawędzi E algorytmy te działają w czasie $O((|V|+|E|)\cdot\log|V|)$. W naszym przypadku $|V| = nm$ i $|E| \leqslant 2nm$, więc ta złożoność to $O(nm\log(nm))$. Umiejętne użycie jednego z tych algorytmów pozwalało na zdobycie 100 punktów.

Nie są to optymalne rozwiązania pod względem złożoności czasowej. By je ulepszyć, należy wykorzystać strukturę grafu, a dokładnie to, że krawędzie mają dwa możliwe koszty: 0 lub 1.

Algorytm Prima zaczyna od jednowierzchołkowego drzewa i zachłannie rozszerza je o nowe wierzchołki, które aktualnie są najtańsze do dodania. Wykorzystując to, że mamy tylko dwa możliwe koszty krawędzi, możemy w kubełkach segregować krawędzie o odpowiednich kosztach, zamiast utrzymywać kosztowną kolejkę priorytetową. Dzięki takiej optymalizacji algorytm działa w czasie $O(|V|+|E|)$, czyli w naszym przypadku $O(nm)$ (`zyw12.cpp`).

W poniższym pseudokodzie zakładamy, że wszystkie krawędzie są początkowo wybrane do żywopłotu, a następnie usuwamy te z nich, które wybieramy do MST. Końce krawędzi oznaczamy jako $v1$ oraz $v2$.

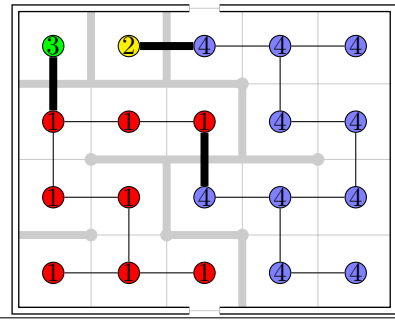
```
 1: procedure PRIM(v_0)
 2: begin
 3:   rozmiarDrzewa := 1;
 4:   dodany[v_0] := true;
 5:   krawedziePoCisach := {krawędzie z v_0 po cisach};
 6:   krawedziePoTujach := {krawędzie z v_0 po tujach};
 7:   while rozmiarDrzewa < n do begin
 8:     if not krawedziePoTujach.puste() then begin
 9:       najtanszaKrawedz := krawedziePoTujach.ostatnia();
10:       krawedziePoTujach.usunOstatnia();
11:     end else begin
12:       najtanszaKrawedz := krawedziePoCisach.ostatnia();
13:       krawedziePoCisach.usunOstatnia();
14:     end
15:     if dodany[najtanszaKrawedz.v1] then nowy := najtanszaKrawedz.v2
16:     else nowy := najtanszaKrawedz.v1;
17:     if not dodany[nowy] then begin
18:       usunZywoplot(najtanszaKrawedz);
19:       krawedziePoCisach += {krawędzie z nowego wierzchołka po cisach};
20:       krawedziePoTujach += {krawędzie z nowego wierzchołka po tujach};
21:       rozmiarDrzewa++;
22:       dodany[nowy] := true;
23:     end
24:   end
25: end
```

Algorytm Kruskala przedstawia się następująco: dopóki nie wszystko jest połączone w jedno drzewo, znajdź najtańszą krawędź, która łączy wierzchołki z niepołączonych jeszcze składowych, i dodaj ją do aktualnego minimalnego drzewa rozpinającego. Składowe są przechowywane za pomocą struktury danych do zbiorów rozłącznych, tzw. *Find-Union*: *Find* znajduje identyfikator składowej, do której należy dany wierzchołek, a *Union* łączy składowe zawierające podane dwa wierzchołki. Zamortyzowany koszt operacji *Find-Union* na zbiorze rozmiaru k to $O(\log^* k)$. Analogicznie jak w przypadku algorytmu Prima rozdzielając krawędzie według kosztów, jesteśmy w stanie sprawić, by algorytm Kruskala działał w czasie $O((|V| + |E|) \cdot \log^* |V|)$.

```
 1: procedure KRUSKAL
 2: begin
 3:   while not krawedzie.puste() do begin
 4:     najmniejsza := krawedzie.wezNajmniejsza();
 5:     krawedzie.usunNajmniejsza();
 6:     if Find(najmniejsza.v1) ≠ Find(najmniejsza.v2) then begin
 7:       usunZywoplot(najmniejsza);
 8:       Union(najmniejsza.v1, najmniejsza.v2);
 9:     end
10:   end
11: end
```

Okazuje się, że w przypadku naszego zadania także algorytm Kruskala możemy zmodyfikować, by działał w czasie liniowym (tj. $O(nm)$). Jako pierwsze w algorytmie Kruskala będą rozpatrzone krawędzie o koszcie 0. W takim razie chcemy podzielić wierzchołki grafu na takie grupy, że w obrębie jednej da się przejść między każdą parą wierzchołków wyłącznie po tujach. Można łatwo takie grupy znaleźć w czasie liniowym, np. za pomocą algorytmu DFS, startując z każdego nieodwiedzonego wierzchołka, odwiedzając wszystkie wierzchołki osiągalne po tujach i oznaczając je numerem grupy. Wierzchołki w obrębie danej grupy jesteśmy w stanie połączyć, nie usuwając z żywopłotu żadnego cisu. Natomiast by połączyć różne grupy nie mamy innego wyboru jak usunąć jakieś cisy na ich granicy (rys. 2). By połączyć k grup w drzewo, musimy postawić dokładnie $k-1$ krawędzi.

Znamy już odpowiedź, ile cisów zostanie w żywopłocie, ale teraz pytanie brzmi: jak szybko znaleźć te cisy, które zostaną usunięte? Możemy to osiągnąć, tworząc graf grup: iterujemy po każdym polu i jeżeli jego grupa to g_1, a grupa jego sąsiada to g_2 i $g_1 \neq g_2$, to g_1 i g_2 są sąsiadami w grafie grup, a krawędź między nimi odpowiada krawędzi między rozważanymi polami. W powstałym grafie znajdujemy dowolne drzewo rozpinające np. za pomocą algorytmu DFS. To rozwiązanie zostało zaimplementowane w pliku `zyw11.cpp`.



Rys. 2: Najpierw łączymy wierzchołki wewnątrz każdej z grup kosztem 0 (cienkie linie), a później łączymy grupy (grube linie).

Wojciech Rytter
Treść zadania, Opracowanie

Piotr Smulewicz
Program

Dostępna pamięć: 256 MB.

OI, etap III, dzień drugi, 14.04.2016

Klubowicze

Bajtocki Klub Dyskusyjny jest wyjątkowy pod każdym względem. Posiada on 2^n członków, z których każdy zadeklarował, jakie ma poglądy na n fundamentalnych pytań. Konkretne sformułowanie pytań nie jest istotne, wystarczy wiedzieć, że są to pytania, na które można udzielić jednej z dwóch odpowiedzi (np. „kawa czy herbata?"). Poglądy danej osoby możemy kodować za pomocą ciągu bitów, który interpretowany w systemie binarnym da liczbę całkowitą z przedziału od 0 do $2^n - 1$.

W klubie nie ma dwóch osób o jednakowych poglądach. Powiemy, że dwie osoby są **prawie zgodne**, *jeśli ich poglądy różnią się tylko na jednym pytaniu. Ponadto klubowicze to 2^{n-1} panów i 2^{n-1} pań, którzy tworzą 2^{n-1} par. Klubowicze spotykają się przy* **okrągłym** *stole. Chcemy ich tak usadzić, żeby każdy klubowicz siedział obok swojej partnerki lub swojego partnera, a obok siebie po drugiej stronie miał osobę prawie zgodną.*

Wejście

W pierwszym wierszu standardowego wejścia znajduje się liczba całkowita n oznaczająca liczbę fundamentalnych pytań. W kolejnych 2^{n-1} wierszach znajdują się opisy par klubowiczów: w i-tym z tych wierszy znajdują się dwie liczby całkowite a_i, b_i ($0 \leqslant a_i, b_i \leqslant 2^n - 1$) oddzielone pojedynczym odstępem, oznaczające, że klubowicze o zestawie poglądów opisanym liczbami a_i i b_i są parą. Każda liczba reprezentująca klubowicza pojawi się na wejściu dokładnie raz.

Wyjście

Jeśli nie istnieje usadzenie klubowiczów spełniające warunki zadania, to w jedynym wierszu standardowego wyjścia należy wypisać jedno słowo `NIE`.

Jeśli takie usadzenie istnieje, to w jedynym wierszu standardowego wyjścia należy wypisać ciąg 2^n liczb całkowitych pooddzielanych pojedynczymi odstępami, oznaczający poprawne usadzenie klubowiczów przy okrągłym stole.

Jeśli istnieje wiele poprawnych odpowiedzi, należy wypisać dowolną z nich.

Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

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

Testy „ocen”:

`1ocen`: *$n = 4$, jeśli i jest parzyste, to klubowicze o numerach i oraz $i + 1$ są w parze;*

`2ocen`: *$n = 10$, jeśli i jest nieparzyste, to klubowicze o numerach i oraz $i + 1$ są w parze; wyjątkiem jest klubowicz $2^n - 1$, który jest w parze z klubowiczem 0;*

`3ocen`: *$n = 15$, test losowy, pary na wejściu są posortowane rosnąco względem liczb a_i.*

Ocenianie

Zestaw testów dzieli się na 18 grup, z których każda warta jest 5 albo 6 punktów. W grupie numer k znajdują się wyłącznie testy z $n = k + 1$ (a zatem $2 \leqslant n \leqslant 19$).

Rozwiązanie

Opiszemy rozwiązanie w terminach grafów nieskierowanych, cykli Hamiltona i skojarzeń doskonałych. Dla przypomnienia, cykl Hamiltona to cykl przechodzący przez każdy wierzchołek grafu dokładnie raz. Z kolei skojarzenie w grafie to podzbiór krawędzi, w którym żadne dwie krawędzie nie mają wspólnego końca. Skojarzenie nazywamy *doskonałym*, gdy każdy wierzchołek grafu jest skojarzony (tzn. jest końcem pewnej krawędzi ze skojarzenia).

Rozważmy graf $\mathcal{H}_n$, którego wierzchołki stanowią liczby $0 \leqslant i < 2^n$ traktowane jako binarne ciągi n-elementowe, zaś krawędzie łączą pary liczb różniące się na jednym bicie. Taki graf nazywany jest często *hiperkostką* n-wymiarową jako uogólnienie kwadratu i sześcianu na wyższe wymiary. Natomiast niech $\mathcal{K}_n$ będzie *grafem pełnym* o tym samym zbiorze wierzchołków co $\mathcal{H}_n$, w którym każde dwa różne wierzchołki są połączone krawędzią.

Klubowicze z naszego zadania odpowiadają wierzchołkom grafów. W zadaniu mamy dany zbiór par wierzchołków (skojarzenie doskonałe grafu $\mathcal{K}_n$) i chcemy znaleźć cykl taki, że wierzchołki z każdej pary będą leżały obok siebie na cyklu oraz jeśli wierzchołki leżące obok siebie na cyklu nie są jedną z wejściowych par, to ich numery różnią się dokładnie na jednym bicie (skojarzenie doskonałe grafu $\mathcal{H}_n$).

W terminach grafowych oryginalny problem jest równoważny następującemu:

Wejście: skojarzenie doskonałe X w grafie $\mathcal{K}_n$

Wyjście: cykl Hamiltona $C = X \cup Y$ w grafie $\mathcal{K}_n$ taki, że Y jest skojarzeniem w $\mathcal{H}_n$

Inaczej mówiąc, mając doskonałe skojarzenie w grafie pełnym, chcemy je dopełnić krawędziami z hiperkostki do cyklu Hamiltona. O ile dla grafów niebędących hiperkostką może to być niewykonalne, to w trakcie konstruowania algorytmu przekonamy się, że dzięki specyficznej strukturze grafu $\mathcal{H}_n$ zawsze jesteśmy w stanie znaleźć rozwiązanie.

Głównym pomysłem jest rekurencyjne sprowadzenie problemu do obliczeń na dwóch podhiperkostkach $\mathcal{H}_{n-1}$. Na rozwiązanie składać się będzie: sprytne dołożenie pomocniczych krawędzi, obliczenie rekurencyjnie cykli w podhiperkostkach, następnie

usunięcie pomocniczych krawędzi i dołożenie krawędzi ze zbioru krawędzi łączących podhiperkostki.

Podobnie jak wierzchołki sześcianu można podzielić na dwie grupy znajdujące się na przeciwległych ścianach, tak hiperkostkę $\mathcal{H}_n$ można podzielić na dwa podgrafy izomorficzne z $\mathcal{H}_{n-1}$. W tym celu wystarczy ustalić dowolny indeks i oraz pogrupować ciągi bitowe odpowiadające wierzchołkom $\mathcal{H}_n$ w zależności od wartości i-tego bitu.

Pokażemy, że dla każdych danych wejściowych istnieje rozwiązanie. Jak się nierzadko okazuje, od dowodu niedaleka droga do algorytmu. Zaczniemy od jednej przydatnej definicji. Jeśli C jest cyklem oraz Z jest podzbiorem krawędzi C i jednocześnie skojarzeniem, to przez $Seq(C, Z)$ oznaczmy listę krawędzi Z w kolejności i skierowaniu zadanym przez C (wybieramy dowolne skierowanie cyklu). Dla przykładu jeśli $C = (1, 2, 4, 3, 6, 5, 1)$ oraz $Z = [(5, 6), (2, 4)]$, to $Seq(C, Z) = [(2, 4), (6, 5)]$.

Twierdzenie 1. *Każde skojarzenie doskonałe w grafie $\mathcal{K}_n$ można dopełnić do cyklu Hamiltona skojarzeniem z grafu $\mathcal{H}_n$.*

Dowód: Indukcja po n. Za bazę indukcyjną potraktujemy przypadek dwuwymiarowy. Wtedy graf $\mathcal{H}_n$ jest kwadratem, dla którego teza jest oczywista.

Przypuśćmy, że dla $n \geqslant 3$ dokonaliśmy podziału hiperkostki $\mathcal{H}_n$ na dwie podhiperkostki H_1 i H_2, obie izomorficzne z $\mathcal{H}_{n-1}$. Niech X będzie doskonałym skojarzeniem grafu $\mathcal{K}_n$ i podzielmy krawędzie tego skojarzenia na trzy zbiory:

$$X_1 = X \cap H_1, \qquad X_2 = X \cap H_2, \qquad Z = X - X_1 - X_2.$$

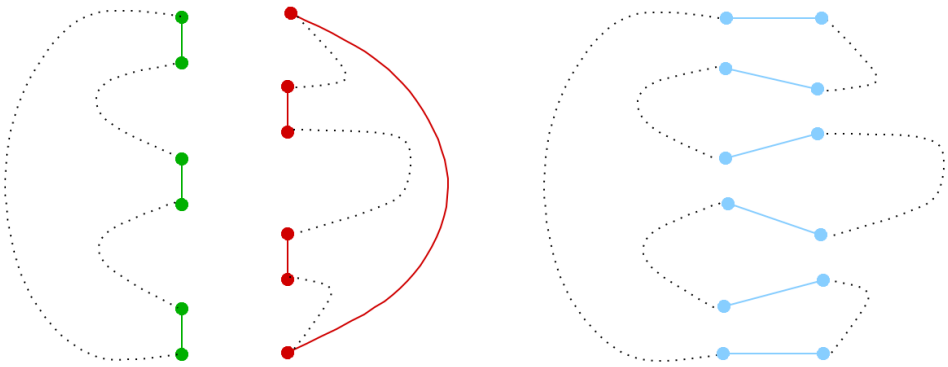
Licząc końce krawędzi w H_1 i H_2, dostajemy $2 \cdot |X_1| + |Z| = 2 \cdot |X_2| + |Z| = 2^{n-1}$, skąd wnioskujemy, że zbiór Z musi zawierać parzyście wiele krawędzi. Są to niejako łączniki pomiędzy podhiperkostkami, które zostaną użyte do połączenia dwóch cykli, które rekurencyjnie skonstruujemy w podhiperkostkach.

Niech dla $i = 1, 2$ zbiór D_i zawiera końce krawędzi ze zbioru Z, które leżą w podhiperkostce H_i. Połączmy elementy D_i w dowolny sposób w pary, uzyskując skojarzenie M_i. Wtedy $X_i \cup M_i$ jest doskonałym skojarzeniem w grafie izomorficznym z $\mathcal{K}_{n-1}$, zatem z założenia indukcyjnego wynika, że uda nam się znaleźć skojarzenie Y_i w H_i, takie że $C_i = X_i \cup M_i \cup Y_i$ będzie cyklem.

Otrzymamy tym samym dwa rozłączne cykle C_1 i C_2 w $\mathcal{H}_n$, które chcielibyśmy połączyć przy pomocy krawędzi z Z. Aby jednak to było możliwe, musimy nieco uważniej skonstruować skojarzenia M_i. Niech $\phi\colon D_1 \to D_2$ będzie przyporządkowaniem końcom krawędzi w Z leżących w H_1 ich końców leżących w H_2. Skojarzenie M_1 możemy wybrać dowolnie, ale skojarzenie M_2 będzie wyznaczone na podstawie cyklu C_1. A konkretnie: dla $Seq(C_1, M_1) = [(v_1, v_2), (v_3, v_4), \ldots, (v_{2k-1}, v_{2k})]$ budujemy cykl $C_2 = X_2 \cup M_2 \cup Y_2$, wybierając

$$M_2 = [(\phi(v_2), \phi(v_3)), (\phi(v_4), \phi(v_5)), \ldots, (\phi(v_{2k}), \phi(v_1))].$$

Teraz możemy już połączyć cykle C_1 i C_2, zastępując zbiór pomocniczych krawędzi $M_1 \cup M_2$ przez krawędzie zbioru Z. Dzięki temu uzyskujemy zbiór $C = X_1 \cup Y_1 \cup X_2 \cup Y_2 \cup Z$. Aby pokazać, że jest on cyklem, wystarczy zauważyć, że powstaje on z C_2 poprzez zastąpienie każdej krawędzi $(\phi(v_i), \phi(v_{i+1})) \in M_2$ przez ścieżkę $\phi(v_i) \to v_i \rightsquigarrow v_{i+1} \to \phi(v_{i+1})$, leżącą poza H_2. Każda taka zamiana jest



Rys. 1: Ilustracja połączenia cykli C_1 i C_2. Po lewej stronie linie ciągłe oznaczają skojarzenia M_1 i M_2. Przerywane linie oznaczają ścieżki leżące wewnątrz podhiperkostek H_1, H_2. Po prawej stronie skojarzenie $M_1 \cup M_2$ zostało zamienione na Z.

niezależna od pozostałych i żadna nie narusza spójności cyklu. Niniejsza konstrukcja została zobrazowana na rysunku 1.

W powyższym rozumowaniu łatwo przeoczyć jeden detal. Przyjęliśmy mianowicie ciche założenie, że zbiór Z jest niepusty. Jest ono istotne, bo w przeciwnym wypadku nie udałoby się nam połączyć obu cykli. Możemy jednak łatwo skonstruować taki podział $\mathcal{H}_n$ na dwie podhiperkostki, w którym to założenie jest spełnione. Wystarczy wybrać dowolną krawędź $e \in X$ i znaleźć bit, na którym jej końce się różnią, a następnie podzielić $\mathcal{H}_n$ w zależności od wartości tego bitu. W ten sposób zagwarantujemy, że co najmniej krawędź e będzie należeć do zbioru Z. ■

Dowód twierdzenia łatwo przetłumaczyć na algorytm rekurencyjny. Poniżej przedstawiamy pseudokod funkcji przyjmującej jako argumenty n, hiperkostkę wymiaru n oraz skojarzenie X, i zwracającej szukany cykl Hamiltona.

```
1:  function FindCycle(n, H, X)
2:  begin
3:    if n = 2 then
4:      return sprawdź oba skojarzenia doskonałe H i wybierz to pasujące do X;
5:    (H_1, H_2) := podział H na podhiperkostki z niepustym łącznikiem;
6:    X_1 := X ∩ H_1,  X_2 := X ∩ H_2,  Z := X − X_1 − X_2;
7:    D_1 := zbiór wierzchołków w H_1 będących końcami krawędzi z Z;
8:    M_1 := dowolne skojarzenie doskonałe w grafie pełnym na zbiorze D_1;
9:    C_1 := FindCycle(n − 1, H_1, X_1 ∪ M_1);
10:   [(v_1, v_2), (v_3, v_4), ..., (v_{2k−1}, v_{2k})] := Seq(C_1, M_1);
11:   M_2 := [(φ(v_2), φ(v_3)), (φ(v_4), φ(v_5)), ..., (φ(v_{2k}), φ(v_1))];
12:   C_2 := FindCycle(n − 1, H_2, X_2 ∪ M_2);
13:   return cykl Hamiltona złożony z (C_1 − M_1) ∪ (C_2 − M_2) ∪ Z;
14: end
```

Dla ustalonego n wszystkie zbiory występujące w funkcji FindCycle mają co najwyżej $N = 2^{n-1}$ elementów, a potrzebne operacje możemy wykonać w czasie liniowym od ich rozmiaru. Złożoność obliczeniową algorytmu można zatem opisać równaniem

$$T(N) = 2 \cdot T(N/2) + O(N),$$

które jest spełnione dla $T(N) = O(N \log N)$. Z rekurencją tego typu można się spotkać na przykład w klasycznym problemie sortowania przez scalanie (ang. *mergesort*). Jest to szczególny przypadek *Twierdzenia o rekurencji uniwersalnej* [6]. Tak więc ostatecznie złożoność rozwiązania to $O(2^{n-1} \log 2^{n-1}) = O(n2^n)$.

Istotną częścią implementacji jest interpretacja rozkładu hiperkostki w języku operacji na maskach bitowych. Kod z komentarzami objaśniającymi poszczególne operacje można znaleźć w pliku `klu2.cpp`.

Wojciech Nadara — Treść zadania, Opracowanie
Bartosz Kostka — Program

Dostępna pamięć: 256 MB. *OI, etap III, dzień drugi, 14.04.2016*

Niebanalne podróże

Bajtazar ostatnio połknął bakcyla podróżowania po Bajtocji. W kraju tym jest n miast (które dla uproszczenia numerujemy liczbami od 1 do n), a bajtocka kolej oferuje podróżnym m dwukierunkowych połączeń kolejowych pomiędzy niektórymi parami miast. Używając tych połączeń, Bajtazar może się dostać do każdego miasta w Bajtocji (być może musi się przy tym przesiadać).

Nasz bohater szczególnie upodobał sobie podróże, w których wyrusza z pewnego miasta, by na końcu do niego wrócić, ale nie odwiedzając przy tym w trakcie podróży żadnego miasta dwukrotnie ani nie używając żadnego połączenia dwukrotnie. Takie podróże nazywa **niebanalnymi**.

W trakcie kolejnej ze swoich wielu podróży Bajtazar zauważył, że każda niebanalna podróż, w którą wyruszył, używała tyle samo połączeń kolejowych. Podejrzewa on, że jest to uniwersalna własność sieci kolejowej w Bajtocji, i poprosił Ciebie o zweryfikowanie tej hipotezy. Ponadto, jeżeli hipoteza jest prawdziwa, to chciałby on także poznać liczbę różnych niebanalnych podróży, które może odbyć. Z wiadomych sobie powodów Bajtazar zadowoli się jedynie resztą z dzielenia liczby takich podróży przez $10^9 + 7$.

Podróż możemy formalnie opisać za pomocą ciągu liczb oznaczających kolejno odwiedzane miasta. Dwie podróże o tej samej długości są różne, jeśli istnieje taki indeks i, że i-te miasta z kolei odwiedzone w trakcie tych podróży są różne. Przez długość podróży rozumiemy liczbę połączeń, których ona używa.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie liczby całkowite n i m ($n \geqslant 1$, $m \geqslant 0$) oddzielone pojedynczym odstępem, oznaczające odpowiednio liczbę miast w Bajtocji oraz liczbę oferowanych połączeń. Dalej następuje m wierszy opisujących oferowane połączenia. W i-tym z tych wierszy znajdują się dwie liczby całkowite a_i i b_i ($1 \leqslant a_i, b_i \leqslant n$, $a_i \neq b_i$) oddzielone pojedynczym odstępem, oznaczające, że istnieje dwukierunkowe połączenie kolejowe umożliwiające podróż pomiędzy miastami o numerach a_i i b_i. Między każdą parą miast biegnie co najwyżej jedno bezpośrednie połączenie kolejowe.

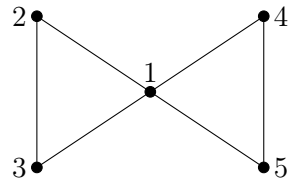
Wyjście

Jeżeli pechowo okazało się, że nie istnieje ani jedna niebanalna podróż, to na standardowe wyjście należy wypisać jedno słowo `BRAK`. *Jeżeli istnieją takie podróże, ale nie wszystkie mają tę samą długość (czyli hipoteza Bajtazara jest fałszywa), należy wypisać jedno słowo* `NIE`. *W końcu jeżeli wszystkie niebanalne podróże mają taką samą długość (czyli hipoteza jest prawdziwa), to należy wypisać jedno słowo* `TAK`, *a w kolejnym wierszu dwie liczby całkowite oddzielone pojedynczym odstępem, oznaczające długość niebanalnych podróży oraz resztę z dzielenia liczby niebanalnych podróży przez $10^9 + 7$.*

Przykład

Dla danych wejściowych:

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



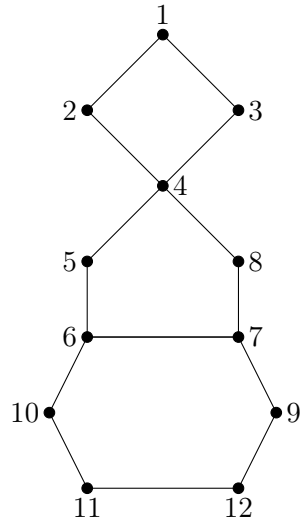
poprawnym wynikiem jest:

```
TAK
3 12
```

Wyjaśnienie do przykładu: *Wszystkie niebanalne podróże mają długość 3 i jest ich 12. Są to kolejno 1-2-3-1, 1-3-2-1, 2-1-3-2, 2-3-1-2, 3-1-2-3, 3-2-1-3, 1-4-5-1, 1-5-4-1, 4-1-5-4, 4-5-1-4, 5-1-4-5, 5-4-1-5.*

Natomiast dla danych wejściowych:

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



poprawną odpowiedzią jest:

```
NIE
```

Testy „ocen”:

`1ocen`: *$n = 500\,000$, miasta w Bajtocji leżą na ścieżce, odpowiedź to oczywiście* `BRAK`.

Ocenianie

Zestaw testów dzieli się na następujące podzadania. Testy do każdego podzadania składają się z jednej lub większej liczby osobnych grup testów.

Podzadanie	Warunki	Liczba punktów
1	$n \leqslant 18$	*20*
2	$n, m \leqslant 2000$	*40*
3	$n \leqslant 500\,000$, $m \leqslant 1\,000\,000$	*40*

Rozwiązanie

Pojęcie dwuspójności

Tłumacząc zadanie na język teorii grafów, jesteśmy proszeni o stwierdzenie, czy wszystkie *cykle proste* (tzn. takie, w których żaden wierzchołek ani krawędź się nie powtarzają) w danym grafie nieskierowanym mają taką samą długość. Jeżeli tak jest, to mamy dodatkowo wyznaczyć ich liczbę (i wypisać ją pomnożoną przez dwukrotność tej długości). W tym opracowaniu za każdym razem, gdy będziemy pisali o *cyklach*, będziemy w domyśle mieć na myśli *cykle proste*.

Przypomnijmy pojęcie *grafów dwuspójnych wierzchołkowo*. Są to takie grafy spójne, które po usunięciu dowolnego wierzchołka pozostają spójne. Równoważna definicja stwierdza, że graf dwuspójny to taki graf, w którym dla każdych dwóch krawędzi istnieje cykl prosty je zawierający.

Zbiór krawędzi dowolnego grafu można podzielić na *dwuspójne składowe*. W obrębie dwuspójnej składowej dla każdej pary krawędzi istnieje cykl je zawierający oraz ta własność nie zachodzi dla żadnej pary krawędzi z dwóch różnych składowych. W szczególności, dwuspójną składową może być także pojedyncza krawędź. Żądnym uzasadnienia istnienia takiego podziału zbioru krawędzi można podpowiedzieć, że wystarczy udowodnić, że jeżeli istnieje cykl prosty zawierający krawędzie a oraz b oraz cykl prosty zawierający krawędzie b i c, to istnieje także cykl prosty zawierający krawędzie a i c, co powinno stać się jasne po narysowaniu na kartce kilku przykładów.

Znany jest algorytm działający w złożoności $O(n + m)$ (gdzie n i m to odpowiednio liczby wierzchołków i krawędzi grafu) dzielący krawędzie grafu na dwuspójne składowe. Czytelnikom, którzy o nim nie słyszeli, polecamy się z nim zapoznać np. w książce [4] (jest on także opisany niejawnie w opracowaniu zadania *Blokada* z XV Olimpiady Informatycznej [1]).

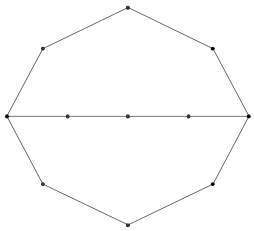
Dlaczego jednak interesujemy się podziałem grafu na dwuspójne dwuspójne? Otóż każdy cykl zawiera się w jednej takiej składowej. Jest to oczywisty wniosek z faktu, że nie istnieje cykl przechodzący przez dwie krawędzie z różnych składowych, co stwierdziliśmy powyżej. Możemy zatem oryginalne zadanie rozwiązać dla każdej dwuspójnej składowej z osobna. Jeżeli w żadnej z nich nie ma żadnego cyklu (czyli każda z nich jest pojedynczą krawędzią, a wejściowy graf jest drzewem), to wypisujemy `BRAK`. Jeżeli w którejkolwiek dwuspójnej składowej istnieją dwa cykle różnych długości, to wypisujemy `NIE`. Załóżmy natomiast, że dla każdej dwuspójnej składowej, w której istnieje cykl, wszystkie cykle są tej samej długości. Jeżeli dla pewnych dwóch dwuspójnych składowych te długości są różne, to wypisujemy `NIE`. Jeżeli jednak dla wszystkich dwuspójnych składowych ta długość jest taka sama, to wszystkie cykle w danym grafie mają tę samą długość i ich liczba jest sumą ich liczności we wszystkich dwuspójnych składowych.

Od tego momentu możemy zatem ograniczyć się do rozwiązania oryginalnego zadania dla grafów dwuspójnych wierzchołkowo, gdyż potrafimy podzielić nasz graf na dwuspójne składowe i potrafimy obliczyć całkowity wynik na podstawie wyników ze wszystkich składowych.

Algorytm dla grafów dwuspójnych

Jeżeli graf dwuspójny ma co najwyżej dwa wierzchołki, to odpowiedzią jest `BRAK`. Spróbujmy znaleźć jakieś przykłady grafów dwuspójnych, w których wszystkie cykle mają tę samą długość. Niewątpliwie takimi grafami są cykle, lecz, jak można się spodziewać, nie są to wszystkie takie grafy.

Wprowadźmy pewien typ grafu, który nazwiemy (c, l)-*cebulą*. Będziemy tak nazywać graf powstały przez połączenie dwóch wierzchołków za pomocą c ścieżek o długości l; patrz rys. 1.



Rys. 1: (3,4)-cebula

Niewątpliwie w (c, l)-cebuli wszystkie cykle są długości $2l$ i jest ich $\binom{c}{2}$. Postawimy teraz odważną hipotezę, na której oprzemy wzorcowy algorytm.

Lemat 1. Cykle oraz (c, l)-cebule (dla $c \geqslant 3$) to jedyne grafy dwuspójne wierzchołkowo, w których wszystkie cykle mają tę samą długość.

Spróbujmy zaprojektować algorytm na podstawie tego lematu. Nad jego prawdziwością zastanowimy się w następnej sekcji.

Naszym celem jest stwierdzenie, czy dany dwuspójny wierzchołkowo graf jest cyklem lub (c, l)-cebulą. Policzmy stopnie wierzchołków naszego grafu (*stopień wierzchołka* to liczba wychodzących z niego krawędzi). Jeżeli wszystkie one są równe dwa, to mamy do czynienia z cyklem (pamiętajmy, że nasz graf jest dwuspójny, zatem w szczególności także spójny). W przeciwnym przypadku, jeżeli nasz graf ma być (c, l)-cebulą, to dokładnie dwa wierzchołki powinny mieć stopień różny od dwóch. Jak jednak stwierdzić, czy graf spełniający taką własność jest (c, l)-cebulą? Nazwijmy te wierzchołki u i v. Weźmy pewną krawędź wychodzącą z u. Jeżeli zaczniemy spacerować od u w kierunku wyznaczonym przez tę krawędź, to napotkamy pewną liczbę wierzchołków o stopniu dwa, aż do momentu, w którym napotkamy albo u albo v. Gdybyśmy napotkali u, to jednak byłoby to sprzeczne z założeniem, że nasz graf jest dwuspójny, zatem napotkany wierzchołek musiał być wierzchołkiem v. Stąd wniosek,

że jeżeli dwuspójny wierzchołkowo graf ma dokładnie dwa wierzchołki o stopniu większym od dwóch, to ma on postać dwóch wierzchołków połączonych zbiorem ścieżek. Pozostaje nam stwierdzić, czy wszystkie owe ścieżki mają taką samą długość. W tym celu wykorzystamy przeszukiwanie grafu wszerz (tzw. BFS) z wierzchołka u, w którym obliczymy odległości wszystkich wierzchołków w grafie od wierzchołka u. Niech $d[w]$ oznacza odległość wierzchołka w od wierzchołka u.

Lemat 2. Graf spełniający wymienione wcześniej warunki jest cebulą wtedy i tylko wtedy, gdy $d[v]$ jest większe od wszystkich pozostałych elementów tablicy d.

Dowód: Łatwo zauważyć, że jeżeli wszystkie ścieżki mają taką samą długość równą l, to wtedy podany warunek jest prawdziwy i $d[v] = l$. Załóżmy przeciwnie, że $d[v] = a$, ale wśród ścieżek łączących u i v pewna ma długość $b > a$. Wtedy wierzchołek poprzedzający v na tej ścieżce znajduje się w odległości od u nie większej niż a (konkretniej, jeżeli nazwiemy go t, to $d[t] = \min(b-1, a+1) \geqslant a = d[v]$), co dowodzi prawdziwości lematu. ■

Jeżeli zatem $d[v]$ jest ściśle największą wartością w tablicy odległości, to nasz graf jest (c, l)-cebulą, przy czym c jest stopniem u, a $l = d[v]$. To kończy opis algorytmu dla grafów dwuspójnych. Pozostaje nam jedynie udowodnić lemat o charakteryzacji grafów dwuspójnych, w których wszystkie cykle mają taką samą długość.

Dowód charakteryzacji interesujących nas grafów

Dowód lematu 1 będzie opierał się na pojęciu *dekompozycji uchowej*. Niech G będzie grafem dwuspójnym wierzchołkowo. Jeżeli G ma co najmniej trzy wierzchołki, to zawiera on cykl. Wyróżnijmy zatem dowolny cykl w G. Twierdzimy, że cały graf można uzyskać w procesie, w którym wielokrotnie do już uzyskanej części grafu (początkowo jest to ów ustalony cykl) doklejamy ścieżkę, której część wspólna z wcześniejszą częścią grafu to jedynie początkowy i końcowy wierzchołek (w szczególności, ścieżka ta może być pojedynczą krawędzią). W przypadku grafów dwuspójnych wierzchołkowo początkowy i końcowy wierzchołek takiej ścieżki muszą być różne, jednak jeżeli pominiemy to ograniczenie, to otrzymamy analogiczną charakteryzację grafów dwuspójnych krawędziowo. Formalnie, twierdzimy, że istnieje taki ciąg grafów $G_1, \ldots, G_k$, że G_1 to dowolny cykl w G, $G_k = G$ oraz dla każdego $i = 1, \ldots, k-1$ zachodzi własność, że $E(G_{i+1}) \setminus E(G_i)$ to zbiór krawędzi tworzących ścieżkę, której część wspólna z G_i jest równa jej początkowi i końcowi (które są różne). Przyjmujemy konwencję, że do kolejnych grafów G_i należą tylko wierzchołki, z których wychodzi co najmniej jedna krawędź. Taki ciąg grafów nazywamy *dekompozycją uchową* grafu G.

Czemu taka dekompozycja istnieje? Załóżmy, że mamy do czynienia z grafem G_i powstałym w wyniku doklejenia pewnej liczby ścieżek. Jeżeli nie jest on jeszcze równy G, to istnieje jakaś krawędź grafu G niezawarta w G_i, która ma co najmniej jeden swój wierzchołek w grafie G_i. Nazwijmy ją e, a jej końce u i v, gdzie $u \in V(G_i)$. Jeżeli $v \in V(G_i)$, to G_{i+1} to graf G_i z dołożoną krawędzią e. Załóżmy zatem, że $v \notin V(G_i)$. Niech f będzie dowolną krawędzią z G_i. Skoro graf G jest dwuspójny, to istnieje cykl zawierający zarówno e jak i f. Idąc kolejnymi krawędziami tego cyklu, zaczynając od v w kierunku przeciwnym do e, musimy kiedyś dojść do jakiegoś wierzchołka z G_i –

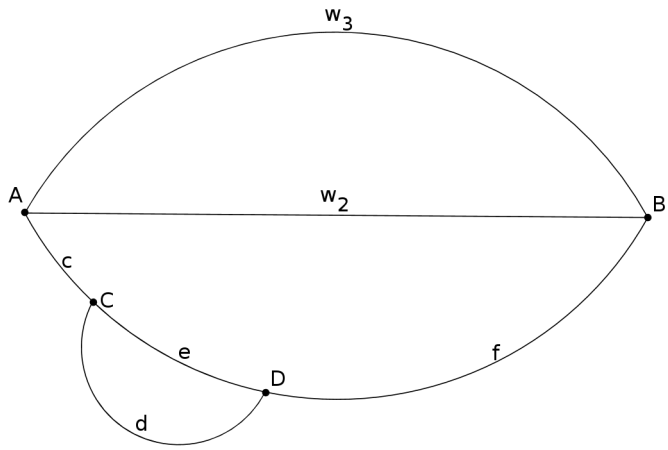
nazwijmy go w. Ten wierzchołek będzie różny od u, gdyż nasz cykl miał być cyklem prostym. Tak stworzona ścieżka – zawierająca e, a potem ścieżkę od v do w – jest ścieżką, którą dokładamy do G_i, aby otrzymać G_{i+1}. Zatem jeżeli G_i nie jest jeszcze równy G, to jesteśmy w stanie dołożyć do niego nowe „ucho". Z racji, że G_{i+1} ma więcej krawędzi niż G_i, ten proces oczywiście musi się kiedyś skończyć.

Zastanówmy się, jak możemy wykorzystać pojęcie dekompozycji uchowej do udowodnienia lematu o charakteryzacji.

Załóżmy, że graf G, w którym zachodzi własność równych długości cykli, sam nie jest cyklem. Wyróżnijmy w nim dowolny cykl G_1. Jesteśmy w stanie do niego dołożyć jakieś ucho. Graf G_2 jest zbiorem trzech ścieżek między dwoma wierzchołkami. Te dwa wierzchołki nazwijmy A i B, a ścieżki nazwijmy w_1, w_2 i w_3. Jeżeli wszystkie cykle w G mają taką samą długość, to ścieżki w_1, w_2 i w_3 mają taką samą długość. Poprzez rozważenie kilku przypadków udowodnimy, że każde kolejne ucho musi być ścieżką łączącą wierzchołki A i B o tej samej długości co trzy ścieżki w_i.

Jeżeli w jest ścieżką, to niech $|w|$ oznacza jej długość (liczoną jako liczba krawędzi). Jeżeli $p_1, \ldots, p_k$ są ścieżkami w grafie, które połączone kolejno tworzą cykl prosty, to ten cykl oznaczmy $p_1p_2 \ldots p_k$, a przez $|p_1p_2 \ldots p_k| = |p_1| + \ldots + |p_k|$ oznaczmy jego długość. Załóżmy, że nowe ucho jest ścieżką pomiędzy C i D. Rozpatrzmy kilka przypadków ze względu na to, gdzie są umiejscowione w grafie te wierzchołki.

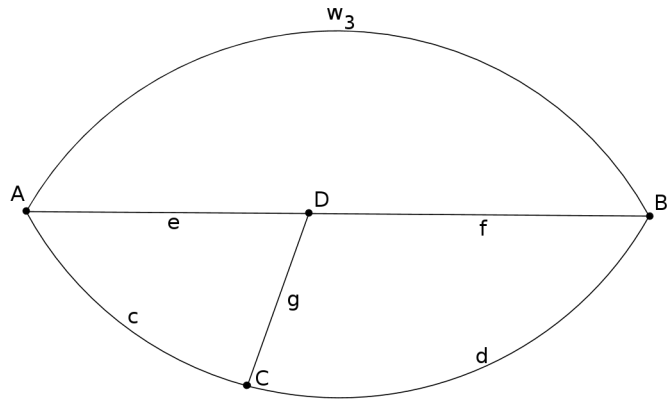
1. C i D oba leżą na jednej ścieżce łączącej A i B (dopuszczamy, że może zachodzić $C \in \{A, B\}$ lub $D \in \{A, B\}$, ale nie oba naraz)



W tym przypadku możemy zauważyć, że cykl de jest krótszy niż cykl dfw_2c, gdyż $|e| < |c| + |e| + |f| = |w_2| < |f| + |w_2| + |c|$. Przeczy to założeniu, że wszystkie cykle proste mają taką samą długość.

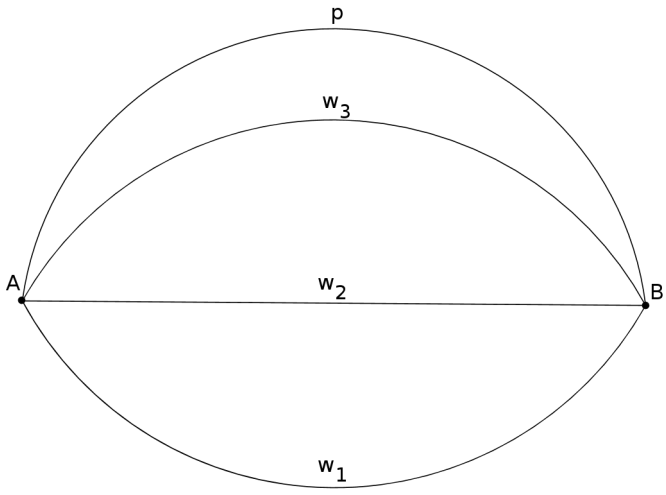
Warto wspomnieć, że na rysunku mogłyby być więcej niż trzy ścieżki łączące A oraz B, jednak istnienie takowych nie jest przeszkodą do uzyskania wspomnianej sprzeczności.

2. C i D leżą we wnętrzach różnych ścieżek łączących A i B



W tym przypadku popatrzmy na cykle $egdw_3$, $cgfw_3$, efw_3 i cdw_3. Mamy $|egdw_3| + |cgfw_3| = (|e| + |f| + |w_3|) + (|c| + |d| + |w_3|) + 2|g| > |efw_3| + |cdw_3|$, skąd wynika, że te cztery cykle nie mogą mieć takich samych długości. Ponownie otrzymujemy sprzeczność.

3. $C = A$ i $B = D$ (lub na odwrót)



Na tym obrazku nowym uchem jest ścieżka p. Jest jasne, że jeżeli wszystkie cykle mają mieć taką samą długość, to musi ona mieć taką samą długość jak każda ze ścieżek w_1, w_2, w_3.

Zaprezentowane przypadki pokrywają wszystkie możliwości. Udowodniliśmy zatem, że każde nowe ucho może być jedynie kolejną ścieżką takiej samej długości łączącą dwa wyróżnione wierzchołki. W ten sposób zakończyliśmy dowód pełnej charakteryzacji dwuspójnych grafów, w których wszystkie cykle są tej samej długości. Rzeczywiście, są to jedynie cykle oraz (c, l)-cebule.

Jacek Tomasiewicz
Treść zadania, Opracowanie

Piotr Smulewicz
Program

Dostępna pamięć: 64 MB.

OI, etap III, dzień drugi, 14.04.2016

Parada

Jak co roku na powitanie wiosny ulicami Bajtogrodu przejdzie Wielka Wiosenna Parada Bajtocka. Swoją obecnością uświetni ją sam Król Bajtazar XVI. Sieć drogowa Bajtogrodu składa się z n skrzyżowań połączonych $n-1$ dwukierunkowymi odcinkami ulic (z każdego skrzyżowania da się dojechać do każdego innego).

Dokładna trasa parady nie jest jeszcze znana, ale wiadomo, że zacznie się ona w jednym ze skrzyżowań, będzie biegła pewną liczbą odcinków ulic i zakończy się na **innym** *skrzyżowaniu. Aby nie zanudzić paradujących, trasa przechodzić będzie przez każdy odcinek ulicy co najwyżej raz.*

Z uwagi na bezpieczeństwo uczestników parady, należy zamknąć bramką wlot każdego odcinka ulicy, przez który nie przechodzi parada, a który wchodzi do skrzyżowania, przez które parada przechodzi (włączając początkowe i końcowe skrzyżowanie). Należy wyznaczyć, ile takich bramek może być potrzebnych.

Wejście

W pierwszym wierszu standardowego wejścia znajduje się liczba całkowita n ($n \geqslant 2$) oznaczająca liczbę skrzyżowań w Bajtogrodzie. Skrzyżowania numerujemy liczbami od 1 do n.

Kolejne $n-1$ wierszy opisuje sieć drogową Bajtogrodu. Każdy z nich zawiera dwie liczby całkowite a i b ($1 \leqslant a,b \leqslant n$, $a \neq b$) oddzielone pojedynczym odstępem, oznaczające, że skrzyżowania o numerach a i b są połączone dwukierunkowym odcinkiem ulicy.

Wyjście

W pierwszym i jedynym wierszu standardowego wyjścia należy wypisać jedną liczbę całkowitą, oznaczającą maksymalną liczbę bramek, które mogą być potrzebne do zabezpieczenia parady.

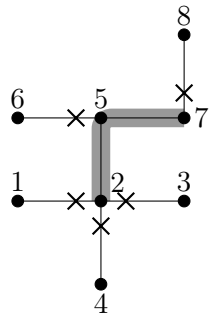
Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
5
```



Wyjaśnienie do przykładu: *Jeśli parada ruszy ze skrzyżowania 2 i zakończy się na skrzyżowaniu 7, to potrzebne będzie 5 bramek (3 do zamknięcia wlotów do skrzyżowania 2 i po jednej do zamknięcia wlotów do skrzyżowań 5 i 7).*

Testy „ocen":

`1ocen`: *$n = 20$, ścieżka;*

`2ocen`: *$n = 20$, gwiazda;*

`3ocen`: *$n = 1000$, losowy test o następującej własności: i-ty odcinek ulicy (dla $i = 1, \ldots, n-1$) łączy skrzyżowanie numer $i+1$ z jednym ze skrzyżowań o mniejszych numerach.*

Ocenianie

Zestaw testów dzieli się na następujące podzadania. Testy do każdego podzadania składają się z jednej lub większej liczby osobnych grup testów.

Podzadanie	Warunki	Liczba punktów
1	$n \leqslant 20$	*15*
2	$n \leqslant 300$	*16*
3	$n \leqslant 3000$	*22*
4	$n \leqslant 200\,000$	*47*

Rozwiązanie

W zadaniu mamy dane drzewo zawierające n węzłów. Szukamy najtrudniejszej w zabezpieczeniu trasy parady, czyli takiej ścieżki w tym drzewie, z którą połączonych jest bezpośrednio jak najwięcej innych węzłów – liczbę tych węzłów nazwiemy *trudnością* trasy.

Rozwiązanie siłowe $O(n^3)$

Sprawdzamy każdą możliwą ścieżkę i liczymy, z iloma węzłami ona sąsiaduje. Jako że wszystkich ścieżek jest $O(n^2)$, a pojedyncze sprawdzenie zajmuje czas liniowy względem długości ścieżki, złożoność czasowa tego rozwiązania to $O(n^3)$.

Rozwiązanie to zaimplementowane jest w pliku `pars4.cpp`. Za poprawne zaprogramowanie takiego rozwiązania na zawodach można było uzyskać około 30% punktów.

Rozwiązanie wolne $O(n^2)$

Sprawdzamy wszystkie możliwe początki ścieżki, ukorzeniając drzewo w każdym z węzłów. Zakładamy, że korzeń jest początkiem ścieżki, która będzie prowadziła w dół drzewa. Wykonując przeszukiwanie drzewa w głąb (DFS), w czasie stałym aktualizujemy liczbę węzłów sąsiadujących ze ścieżką prowadzącą do aktualnie odwiedzanego

węzła. W ten sposób wszystkie ścieżki o ustalonym początku rozpatrujemy w łącznym czasie $O(n)$. Jako że wszystkich początków jest $O(n)$, to złożoność czasowa tego rozwiązania to $O(n^2)$.

Implementacja takiego rozwiązania znajduje się w pliku `pars1.cpp`. Rozwiązanie tego typu otrzymywało na zawodach około 50% punktów.

Rozwiązanie wzorcowe $O(n)$

Ukorzeniamy drzewo w dowolnym węźle. Następnie, zaczynając od liści i poruszając się w górę drzewa, dla każdego węzła v wyznaczamy dwie wartości:

$h[v]$ – trudność najtrudniejszej trasy parady zaczynającej się w węźle v i prowadzącej w dół poddrzewa węzła v,

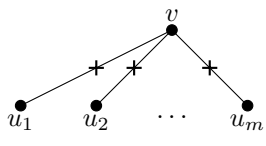
$d[v]$ – trudność najtrudniejszej trasy parady przechodzącej przez v i biegnącej w poddrzewie węzła v.

Jeśli v jest liściem, to oczywiście $h[v] = d[v] = 0$. W ogólnym przypadku, gdy węzeł v ma m synów u_1, u_2, $\ldots$, u_m, dla których obliczyliśmy już wartości $d[u_i]$ i $h[u_i]$, wartości dla węzła v obliczamy z następującej rekurencji:

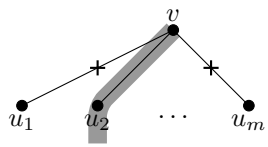
$$h[v] = \max(m, \max_{1 \leqslant i \leqslant m} (h[u_i]) + m - 1)$$

$$d[v] = \max(h[v], \max_{1 \leqslant i < j \leqslant m} (h[u_i] + h[u_j]) + m - 2)$$

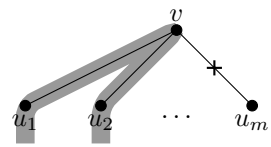
Gdy obliczamy wartość $h[v]$, bierzemy pod uwagę dwie możliwości: albo blokujemy wszystkich synów węzła v (rys. `A`) i trasa parady kończy się w węźle v, albo wybieramy najtrudniejszą trasę parady przechodzącą przez jednego z synów węzła v, blokując przy tym pozostałych synów (rys. `B`). Gdy obliczamy wartość $d[v]$, wybieramy albo trasę parady zaczynającą się w węźle v (co odpowiada wartości $h[v]$), albo połączone dwie trasy parady zaczynające się w synach węzła v (rys. `C`). Na poniższych rysunkach trasę parady zaznaczono kolorem szarym:



rys. `A`



rys. `B`



rys. `C`

Zauważmy, że ostateczny wynik to maksymalna wartość $d[v]$ powiększona o 1, jako że powinniśmy zablokować jeszcze ojca węzła v. Należy tutaj pamiętać o szczególnym przypadku, gdy węzeł v jest korzeniem (wtedy nie dodajemy jedynki).

Wyznaczenie wartości $h[v]$ i $d[v]$ możemy zaimplementować w czasie liniowym od liczby synów m. Faktycznie, wystarczy wyznaczyć maksymalną i drugą co do wielkości wartość $h[u_i]$. Ostatecznie otrzymujemy rozwiązanie działające w czasie liniowym. Przykładową implementację można znaleźć w pliku `par.cpp`.

XXVIII Międzynarodowa Olimpiada Informatyczna,

Kazań, Rosja 2016

Bartosz Kostka, Jakub Radoszewski
Tłumaczenie

Dostępna pamięć: 2 GB, czas: 2 s. *IOI 2016, dzień pierwszy, 14.08.2016*

Kolejka górska

Anna pracuje w parku rozrywki i zajmuje się budową nowej kolejki górskiej. Zaprojektowała już n specjalnych sekcji (dla wygody ponumerowanych od 0 do $n-1$), które zmieniają prędkość wagoników kolejki: są to górki, dolinki i wiele innych. Teraz musi zaproponować ostateczny kształt kolejki, używając wszystkich zaprojektowanych sekcji. W tym zadaniu dla uproszczenia zakładamy, że długość samej kolejki wynosi 0.

Dla każdego i pomiędzy 0 a $n-1$ włącznie, specjalna sekcja numer i ma następujące właściwości:

- *kiedy kolejka wjeżdża do tej sekcji, jej prędkość nie może przekraczać ustalonego limitu: prędkość wagoników musi wynosić* **co najwyżej** s_i *km/h;*
- *kiedy kolejka opuszcza sekcję, jej prędkość wynosi* **dokładnie** t_i *km/h, niezależnie od prędkości, z jaką kolejka wjechała do tej sekcji.*

W ostatecznym projekcie kolejki każdej sekcji należy użyć dokładnie raz. Ponadto pomiędzy każdymi dwiema sąsiadującymi sekcjami można wybudować tory. Anna wybiera kolejność n specjalnych sekcji, a następnie decyduje ona o długości poszczególnych torów. Długość torów jest mierzona w metrach i może być dowolną nieujemną liczbą całkowitą (w szczególności może być to 0). Każdy metr torów pomiędzy specjalnymi sekcjami spowalnia kolejkę o 1 km/h. Na początku trasy kolejka wjeżdża do pierwszej specjalnej sekcji z prędkością 1 km/h.

Ostateczny projekt musi spełniać następujące warunki:

- *w chwili wjeżdżania do specjalnej sekcji kolejka nie może przekraczać limitu prędkości;*
- *w każdym momencie trasy prędkość kolejki musi być dodatnia.*

We wszystkich podzadaniach oprócz trzeciego Twoim zadaniem jest znaleźć kolejność n specjalnych sekcji i dobrać długości torów pomiędzy nimi, tak aby całkowita długość torów była jak najmniejsza. W trzecim podzadaniu musisz jedynie sprawdzić, czy istnieje prawidłowy projekt kolejki, w którym wszystkie tory mają długość 0.

Szczegóły implementacji

Powinieneś zaimplementować następującą funkcję (metodę):

- `int64 plan_roller_coaster(int[] s, int[] t)`
  - `s`: *tablica długości n opisująca maksymalne prędkości wejścia do sekcji.*
  - `t`: *tablica długości n opisująca prędkości wyjścia z sekcji.*
  - *We wszystkich podzadaniach poza trzecim funkcja powinna zwracać minimalną sumaryczną długość wszystkich torów pomiędzy specjalnymi sekcjami. Natomiast w trzecim podzadaniu funkcja powinna zwracać 0, jeżeli istnieje poprawny projekt kolejki, w którym wszystkie tory pomiędzy sekcjami mają długość zero, natomiast dowolną dodatnią liczbę całkowitą, jeżeli taki projekt nie istnieje.*

W języku C sygnatura funkcji jest minimalnie inna:

- `int64 plan_roller_coaster(int n, int[] s, int[] t)`
  - `n`: *rozmiar tablic* `s` *oraz* `t` *(tj. liczba specjalnych sekcji),*
  - *pozostałe parametry są takie same jak powyżej.*

Przykład

```
int64 plan_roller_coaster([1, 4, 5, 6], [7, 3, 8, 6])
```

W tym przykładzie mamy cztery specjalne sekcje. Najlepszym możliwym rozwiązaniem jest wybudowanie ich w kolejności $0, 3, 1, 2$ *i połączenie ich torami o długościach, odpowiednio,* $1, 2, 0$*. Kolejka wtedy porusza się następująco:*

- *Początkowa prędkość kolejki wynosi 1 km/h.*
- *Kolejka rozpoczyna trasę, wjeżdżając do specjalnej sekcji nr 0.*
- *Kolejka opuszcza sekcję nr 0 z prędkością 7 km/h.*
- *Następnie kolejka wjeżdża na tory o długości 1 m. Po ich przejechaniu ma prędkość 6 km/h.*
- *Kolejka wjeżdża do specjalnej sekcji nr 3 z prędkością 6 km/h i opuszcza ją z tą samą prędkością.*
- *Po opuszczeniu sekcji nr 3 kolejka jedzie przez 2 m torów. Prędkość maleje do 4 km/h.*
- *Kolejka wjeżdża do specjalnej sekcji nr 1 z prędkością 4 km/h i opuszcza ją z prędkością 3 km/h.*
- *Natychmiast po opuszczeniu sekcji nr 1 kolejka wjeżdża do specjalnej sekcji nr 2.*
- *Kolejka wyjeżdża z sekcji nr 2. Ostateczna prędkość kolejki wynosi 8 km/h.*

Funkcja powinna zwrócić sumaryczną długość torów pomiędzy specjalnymi sekcjami: $1 + 2 + 0 = 3$*.*

Podzadania

We wszystkich podzadaniach zachodzi $1 \leqslant s_i \leqslant 10^9$ *oraz* $1 \leqslant t_i \leqslant 10^9$*.*

Podzadanie 1 (11 punktów): $2 \leqslant n \leqslant 8$
Podzadanie 2 (23 punkty): $2 \leqslant n \leqslant 16$
Podzadanie 3 (30 punktów): $2 \leqslant n \leqslant 200\,000$*. W tym podzadaniu Twój program musi jedynie sprawdzić, czy wynikiem jest zero, czy też nie. Jeżeli wynikiem nie jest zero, każda dodatnia liczba całkowita jest uznawana za poprawną.*
Podzadanie 4 (36 punktów): $2 \leqslant n \leqslant 200\,000$

Przykładowy program sprawdzający

Przykładowy program sprawdzający wczytuje dane w następującym formacie:

- *wiersz 1: liczba całkowita* n,
- *wiersz* $2 + i$, *dla* i *pomiędzy* 0 *i* $n - 1$*: liczby całkowite* s_i *i* t_i.

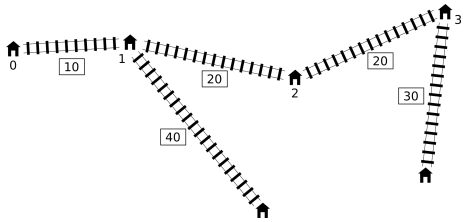
Bartosz Kostka, Jakub Radoszewski
Tłumaczenie

Dostępna pamięć: 2 GB, czas: 2 s. *IOI 2016, dzień pierwszy, 14.08.2016*

Skrót

Paweł ma zabawkową kolejkę elektryczną. Nie jest ona skomplikowana: zawiera ona jedną główną linię torów łączącą n stacji ponumerowanych kolejno od 0 do $n-1$ wzdłuż tej linii. Odległość między stacjami i oraz $i+1$ wynosi l_i centymetrów ($0 \leqslant i < n-1$).

Oprócz głównej linii, w kolejce mogą występować także poboczne linie. Każda poboczna linia łączy stację leżącą na głównej linii z pewną nową stacją, nieleżącą na głównej linii. (Te nowe stacje nie są numerowane). Z każdej stacji głównej linii może wychodzić co najwyżej jedna poboczna linia. Długość pobocznej linii, która zaczyna się na stacji i, wynosi d_i centymetrów. Jeżeli na stacji i nie zaczyna się żadna poboczna linia, to przyjmujemy, że $d_i = 0$.



Paweł chce dodać jeden skrót: ekspresową linię łączącą dwie różne (być może sąsiadujące ze sobą) stacje **głównej linii**. *Ta ekspresowa linia będzie miała długość dokładnie c centymetrów, niezależnie od tego, które dwie stacje połączy.*

Każdy odcinek torów, łącznie z nowo utworzoną ekspresową linią, kolejka może pokonywać w obie strony. **Odległość** *pomiędzy dwiema stacjami rozumiemy jako długość najkrótszej drogi, która łączy te stacje poprzez sieć torów. Z kolei* **średnicą** *całej sieci torów jest największa wśród odległości wszystkich par stacji. Innymi słowy, jest to najmniejsza liczba t, taka że odległość pomiędzy każdymi dwiema stacjami wynosi co najwyżej t centymetrów.*

Paweł chce wybudować ekspresową linię tak, aby średnica powstałej sieci torów była minimalna.

Szczegóły implementacji

Powinieneś zaimplementować jedną funkcję:

- `int64 find_shortcut(int n, int[] l, int[] d, int c)`
  - `n`: *liczba stacji na głównej linii,*
  - `l`: *odległości pomiędzy stacjami na głównej linii (tablica rozmiaru $n-1$),*
  - `d`: *długości pobocznych linii (tablica rozmiaru n),*
  - `c`: *długość nowej, ekspresowej linii.*
  - *Funkcja powinna zwracać najmniejszą możliwą średnicę sieci torów po dodaniu ekspresowej linii.*

Szczegóły implementacji w Twoim języku programowania znajdują się w dostarczonych plikach z szablonami.

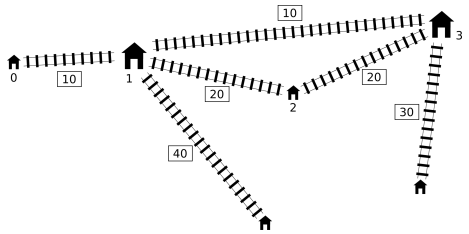
Przykłady

Przykład 1

Dla sieci torów pokazanej powyżej program sprawdzający wykonuje następujące wywołanie:

```
find_shortcut(4, [10, 20, 20], [0, 40, 0, 30], 10)
```

Optymalnym rozwiązaniem jest wybudowanie ekspresowej linii pomiędzy stacjami 1 oraz 3, jak pokazano poniżej.



Średnica nowej sieci torów wynosi 80 centymetrów, zatem funkcja powinna zwrócić wynik 80.

Przykład 2

Program sprawdzający wykonuje następujące wywołanie:

```
find_shortcut(9, [10, 10, 10, 10, 10, 10, 10, 10],
       [20, 0, 30, 0, 0, 40, 0, 40, 0], 30)
```

Optymalne rozwiązanie polega na połączeniu stacji 2 oraz 7. Średnica sieci kolejowej wynosi wtedy 110.

Przykład 3

Program sprawdzający wykonuje następujące wywołanie:

```
find_shortcut(4, [2, 2, 2], [1, 10, 10, 1], 1)
```

Połączenie stacji 1 oraz 2 jest optymalne i zmniejsza średnicę do 21.

Przykład 4

Program sprawdzający wykonuje następujące wywołanie:

```
find_shortcut(3, [1, 1], [1, 1, 1], 3)
```

Połączenie żadnych dwóch stacji ekspresową linią o długości 3 nie poprawi początkowej średnicy sieci torów, wynoszącej 4.

Podzadania

We wszystkich podzadaniach zachodzi $2 \leqslant n \leqslant 1\,000\,000$, $1 \leqslant l_i \leqslant 10^9$, $0 \leqslant d_i \leqslant 10^9$, $1 \leqslant c \leqslant 10^9$.

Podzadanie 1 (9 punktów): $2 \leqslant n \leqslant 10$
Podzadanie 2 (14 punktów): $2 \leqslant n \leqslant 100$
Podzadanie 3 (8 punktów): $2 \leqslant n \leqslant 250$
Podzadanie 4 (7 punktów): $2 \leqslant n \leqslant 500$
Podzadanie 5 (33 punkty): $2 \leqslant n \leqslant 3000$
Podzadanie 6 (22 punkty): $2 \leqslant n \leqslant 100\,000$
Podzadanie 7 (4 punkty): $2 \leqslant n \leqslant 300\,000$
Podzadanie 8 (3 punkty): $2 \leqslant n \leqslant 1\,000\,000$

Przykładowy program sprawdzający

Przykładowy program sprawdzający wczytuje dane w następującym formacie:

- *wiersz 1: liczby całkowite* n *i* c,
- *wiersz 2: liczby całkowite* $l_0, l_1, \ldots, l_{n-2}$,
- *wiersz 3: liczby całkowite* $d_0, d_1, \ldots, d_{n-1}$.

Bartosz Kostka, Jakub Radoszewski
Tłumaczenie

Dostępna pamięć: 2 GB, czas: 1 s. *IOI 2016, dzień pierwszy, 14.08.2016*

Wykrywacz cząsteczek

Firma, w której pracuje Petr, skonstruowała maszynę do wykrywania cząsteczek. Każda cząsteczka ma masę wyrażającą się dodatnią liczbą całkowitą. Maszyna ma określony **zakres pomiarowy** *$[l, u]$, przy czym l i u są dodatnimi liczbami całkowitymi. Maszyna może wykryć zbiór cząsteczek wtedy i tylko wtedy, gdy zbiór ten zawiera podzbiór, którego łączna masa należy do zakresu pomiarowego maszyny.*

Formalnie, rozważmy n cząsteczek o masach $w_0, \ldots, w_{n-1}$. Proces wykrywania kończy się powodzeniem, jeśli istnieje zbiór parami różnych indeksów $I = \{i_1, \ldots, i_m\}$ taki że $l \leqslant w_{i_1} + \ldots + w_{i_m} \leqslant u$.

Konstrukcja maszyny gwarantuje, że różnica między u i l jest nie mniejsza niż różnica mas najcięższej i najlżejszej cząsteczki. Formalnie, $u - l \geqslant w_{max} - w_{min}$, gdzie $w_{max} = max(w_0, \ldots, w_{n-1})$ i $w_{min} = min(w_0, \ldots, w_{n-1})$.

Twoim zadaniem jest napisanie programu, który albo wyznaczy jakikolwiek podzbiór zbioru cząsteczek, którego łączna masa należy do zakresu pomiarowego maszyny, albo stwierdzi, że taki podzbiór nie istnieje.

Szczegóły implementacji

Powinieneś napisać jedną funkcję (metodę):

- `int[] solve(int l, int u, int[] w)`
  - `l`, `u`: *końce zakresu pomiarowego,*
  - `w`: *masy cząsteczek.*
  - *Jeśli żądany podzbiór istnieje, funkcja powinna zwrócić tablicę indeksów cząsteczek, które tworzą dowolny taki podzbiór. Jeśli jest więcej niż jedna poprawna odpowiedź, wynikiem funkcji może być dowolna z nich.*
  - *Jeśli żądany podzbiór nie istnieje, funkcja powinna zwrócić pustą tablicę.*

W języku C sygnatura funkcji jest minimalnie inna:

- `int solve(int l, int u, int[] w, int n, int[] result)`
  - `n`: *liczba elementów tablicy* `w` *(tj. liczba cząsteczek),*
  - *pozostałe parametry są takie same jak powyżej.*
  - *Zamiast zwracać tablicę opisującą m indeksów (jak powyżej), funkcja powinna zapisać te indeksy do pierwszych m komórek tablicy* `result` *i zwrócić m.*
  - *Jeśli żądany podzbiór nie istnieje, funkcja nie powinna niczego zapisywać do tablicy* `result` *i powinna zwrócić 0.*

Twój program może zapisać indeksy do zwracanej tablicy (lub do tablicy `result` *w przypadku języka C) w dowolnej kolejności.*

Szczegóły implementacji w Twoim języku programowania znajdują się w dostarczonych plikach z szablonami.

Przykłady

Przykład 1

```
solve(15, 17, [6, 8, 8, 7])
```

W tym przykładzie mamy cztery cząsteczki o masach 6, 8, 8 i 7. Maszyna potrafi wykrywać podzbiory cząsteczek o łącznej masie między 15 a 17 włącznie. Zauważ, że $17 - 15 \geqslant 8 - 6$. *Łączna masa cząsteczek 1 i 3 to* $w_1 + w_3 = 8 + 7 = 15$, *tak więc funkcja może zwrócić* `[1, 3]`. *Inne poprawne odpowiedzi to* `[1, 2]` ($w_1 + w_2 = 8 + 8 = 16$) *i* `[2, 3]` ($w_2 + w_3 = 8 + 7 = 15$).

Przykład 2

```
solve(14, 15, [5, 5, 6, 6])
```

W tym przykładzie mamy cztery cząsteczki o masach 5, 5, 6 i 6 i szukamy podzbioru o łącznej masie między 14 a 15 włącznie. Znów, zauważ że $15 - 14 \geqslant 6 - 5$. *W tym przypadku nie ma żadnego podzbioru cząsteczek o łącznej masie między 14 a 15, więc wynikiem funkcji powinna być pusta tablica.*

Przykład 3

```
solve(10, 20, [15, 17, 16, 18])
```

W tym przykładzie mamy cztery cząsteczki o masach 15, 17, 16 i 18 i szukamy podzbioru o łącznej masie między 10 a 20 włącznie. Znów, zauważ że $20 - 10 \geqslant 18 - 15$. *Każdy podzbiór jednoelementowy ma łączną masę między 10 a 20, tak więc możliwe poprawne wyniki to:* `[0]`, `[1]`, `[2]` *i* `[3]`.

Podzadania

Podzadanie 1 (9 punktów): $1 \leqslant n \leqslant 100$, $1 \leqslant w_i \leqslant 100$, $1 \leqslant u, l \leqslant 1000$, *wszystkie* w_i *są równe.*
Podzadanie 2 (10 punktów): $1 \leqslant n \leqslant 100$, $1 \leqslant w_i, u, l \leqslant 1000$, $max(w_0, \ldots, w_{n-1}) - min(w_0, \ldots, w_{n-1}) \leqslant 1$
Podzadanie 3 (12 punktów): $1 \leqslant n \leqslant 100$, $1 \leqslant w_i, u, l \leqslant 1000$
Podzadanie 4 (15 punktów): $1 \leqslant n \leqslant 10\,000$, $1 \leqslant w_i, u, l \leqslant 10\,000$
Podzadanie 5 (23 punkty): $1 \leqslant n \leqslant 10\,000$, $1 \leqslant w_i, u, l \leqslant 500\,000$
Podzadanie 6 (31 punktów): $1 \leqslant n \leqslant 200\,000$, $1 \leqslant w_i, u, l < 2^{31}$

Przykładowy program sprawdzający

Przykładowy program sprawdzający wczytuje dane w następującym formacie:

- *wiersz 1: liczby całkowite* n, l, u.
- *wiersz 2:* n *liczb całkowitych:* $w_0, \ldots, w_{n-1}$.

Bartosz Kostka, Jakub Radoszewski
Tłumaczenie

Dostępna pamięć: 2 GB, czas: 2 s. *IOI 2016, dzień drugi, 16.08.2016*

Obcy

Na jednej z odległych planet nasz satelita wykrył ślady cywilizacji. Na Ziemię dotarło już pierwsze, niskiej rozdzielczości zdjęcie kwadratowego obszaru tej planety. Naszym ekspertom udało się zidentyfikować na zdjęciu n interesujących punktów, gdzie mogą znajdować się ślady życia. Punkty te są ponumerowane od 0 do $n-1$. Kolejnym krokiem będzie wykonanie wysokiej rozdzielczości zdjęć zawierających wszystkie te n punktów.

Na obszarze znajdującym się na zdjęciu (tym o niskiej rozdzielczości) satelita naniósł siatkę o wymiarach m na m złożoną z jednostkowych pól. Tak wiersze, jak i kolumny siatki są ponumerowane kolejno od 0 do $m-1$ (odpowiednio od góry i od lewej). Przez (s,t) oznaczamy pole znajdujące się na przecięciu wiersza s i kolumny t. Interesujący punkt numer i znajduje się na polu (r_i, c_i). Każde pole może zawierać dowolnie wiele interesujących punktów.

Nasz satelita znajduje się na orbicie przebiegającej bezpośrednio nad **główną** *przekątną siatki, przy czym główna przekątna to odcinek łączący lewy górny i prawy dolny róg siatki. Satelita może wykonać wysokiej rozdzielczości zdjęcie dowolnego obszaru, który spełnia następujące warunki:*

- *ma kształt kwadratu,*
- *dwa przeciwległe wierzchołki tego kwadratu znajdują się na głównej przekątnej siatki,*
- *każde pole siatki znajduje się albo całkowicie wewnątrz, albo całkowicie na zewnątrz fotografowanego obszaru.*

Satelita może wykonać co najwyżej k zdjęć w wysokiej rozdzielczości.

Gdy tylko satelita wykona wszystkie zdjęcia, wyśle na Ziemię wysokiej rozdzielczości obraz każdego z fotografowanych pól (niezależnie od tego, czy pole to zawiera jakiekolwiek interesujące punkty). Dane z każdego z tych pól zostaną przesłane dokładnie **raz**, *nawet jeżeli pole to zostało sfotografowane wielokrotnie.*

Tak więc musimy wybrać co najwyżej k kwadratowych obszarów, które zostaną sfotografowane, tak aby:

- *każde z pól zawierających interesujące punkty znalazło się na co najmniej jednym ze zdjęć oraz*
- *liczba pól, które znajdą się na co najmniej jednym zdjęciu, była jak najmniejsza.*

Twoim zadaniem jest wyznaczenie najmniejszej możliwej łącznej liczby pól na zdjęciach.

Szczegóły implementacji

Powinieneś zaimplementować następującą funkcję (metodę):

- `int64 take_photos(int n, int m, int k, int[] r, int[] c)`
  - `n`: *liczba interesujących punktów,*

- `m`: *liczba wierszy (a zarazem liczba kolumn) siatki,*
- `k`: *maksymalna liczba zdjęć, jakie może wykonać satelita,*
- `r`, `c`: *dwie tablice rozmiaru n opisujące współrzędne pól siatki zawierających interesujące punkty. Dla każdego $0 \leqslant i \leqslant n-1$, i-ty z interesujących punktów znajduje się na polu $(r[i], c[i])$.*
- *Funkcja powinna zwrócić najmniejszą możliwą łączną liczbę pól, które znajdą się na co najmniej jednym zdjęciu, przy założeniu, że zdjęcia pokrywają wszystkie spośród interesujących punktów.*

Szczegóły implementacji w Twoim języku programowania znajdują się w dostarczonych plikach z szablonami.

Przykłady

Przykład 1

```
take_photos(5, 7, 2, [0, 4, 4, 4, 4], [3, 4, 6, 5, 6])
```

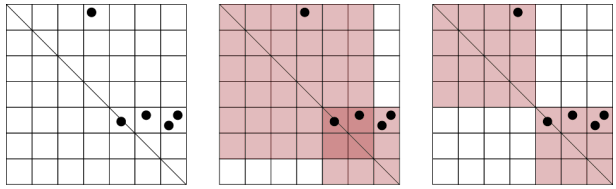
W tym przykładzie mamy siatkę o wymiarach 7×7 zawierającą 5 interesujących punktów. Interesujące punkty znajdują się łącznie na czterech różnych polach: $(0, 3)$, $(4, 4)$, $(4, 5)$ i $(4, 6)$. Satelita może wykonać co najwyżej 2 zdjęcia w wysokiej rozdzielczości.

Jednym ze sposobów uchwycenia wszystkich interesujących punktów jest wykonanie dwóch zdjęć: zdjęcia pokrywającego obszar 6×6 zawierający pola $(0, 0)$ i $(5, 5)$ i zdjęcia pokrywającego obszar 3×3 zawierający pola $(4, 4)$ i $(6, 6)$. Jeśli satelita wykona te dwa zdjęcia, wyśle na Ziemię obrazy 41 pól. Nie jest to optymalny wynik.

Optymalnym rozwiązaniem jest wykonanie jednego zdjęcia pokrywającego obszar 4×4 zawierający pola $(0, 0)$ i $(3, 3)$ i drugiego zdjęcia pokrywającego obszar 3×3 zawierający pola $(4, 4)$ i $(6, 6)$. W ten sposób na zdjęciach znajdzie się łącznie tylko 25 pól, co jest optymalnym wynikiem, więc funkcja `take_photos` *powinna zwrócić 25.*

Zauważ, że pole $(4, 6)$ wystarczy sfotografować raz, mimo iż zawiera ono dwa interesujące punkty.

Przykład ten przedstawiono na poniższych rysunkach. Rysunek po lewej pokazuje siatkę z zaznaczonymi interesującymi punktami. Na środkowym rysunku zaznaczono nieoptymalne rozwiązanie, w którym na zdjęciach znajduje się 41 pól. Rysunek po prawej przedstawia optymalne rozwiązanie.

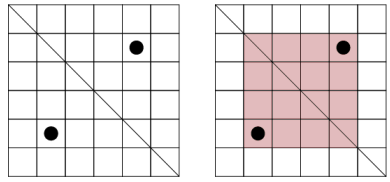


Przykład 2

```
take_photos(2, 6, 2, [1, 4], [4, 1])
```

W tym przypadku mamy 2 interesujące punkty położone symetrycznie, na polach $(1, 4)$ i $(4, 1)$. Każde poprawne zdjęcie zawierające jeden z nich zawiera także drugi z nich. Wystarczy zatem wykonać jedno zdjęcie.

Rysunki poniżej przedstawiają tenże przykład i jego optymalne rozwiązanie. W tym rozwiązaniu satelita wykonuje jedno zdjęcie pokrywające 16 pól.



Podzadania

We wszystkich podzadaniach zachodzi $1 \leqslant k \leqslant n$.

Podzadanie 1 (4 punkty): *$1 \leqslant n \leqslant 50$, $1 \leqslant m \leqslant 100$, $k = n$*
Podzadanie 2 (12 punktów): *$1 \leqslant n \leqslant 500$, $1 \leqslant m \leqslant 1000$, dla każdego i takiego że $0 \leqslant i \leqslant n - 1$, $r_i = c_i$.*
Podzadanie 3 (9 punktów): *$1 \leqslant n \leqslant 500$, $1 \leqslant m \leqslant 1000$*
Podzadanie 4 (16 punktów): *$1 \leqslant n \leqslant 4000$, $1 \leqslant m \leqslant 1\,000\,000$*
Podzadanie 5 (19 punktów): *$1 \leqslant n \leqslant 50\,000$, $1 \leqslant k \leqslant 100$, $1 \leqslant m \leqslant 1\,000\,000$*
Podzadanie 6 (40 punktów): *$1 \leqslant n \leqslant 100\,000$, $1 \leqslant m \leqslant 1\,000\,000$*

Przykładowy program sprawdzający

Przykładowy program sprawdzający wczytuje dane w następującym formacie:

- *wiersz 1: liczby całkowite n, m oraz k,*
- *wiersz $2 + i$ $(0 \leqslant i \leqslant n - 1)$: liczby całkowite r_i oraz c_i.*

Bartosz Kostka, Jakub Radoszewski
Tłumaczenie

Dostępna pamięć: 2 GB, czas: 2 s. *IOI 2016, dzień drugi, 16.08.2016*

Obrazek logiczny

Obrazki logiczne to znany typ łamigłówki. W tym zadaniu rozważamy jej prostą, jednowymiarową wersję. Rozwiązujący ma przed sobą rząd n pól. Pola te są ponumerowane od 0 do $n-1$ od lewej do prawej. Zadaniem rozwiązującego jest pokolorować pola na biało i czarno. Czarne pola oznaczamy jako `'X'`*, a białe jako* `'_'`*.*

Rozwiązujący ma zadany ciąg $c = [c_0, \ldots, c_{k-1}]$ złożony z k dodatnich liczb całkowitych, który nazywamy **wskazówką***. Jego zadaniem jest pokolorować pola w taki sposób, aby czarne pola tworzyły dokładnie k bloków kolejnych pól. Ponadto liczba czarnych pól w i-tym bloku od lewej (bloki numerujemy od 0) musi być równa c_i. Przykładowo, jeśli wskazówka to $c = [3, 4]$, to rozwiązanie łamigłówki musi zawierać dwa bloki czarnych pól: jeden o długości 3 i następnie drugi o długości 4. Tak więc jeśli $n = 10$ i $c = [3, 4]$, jednym z rozwiązań spełniających wymagania wskazówki jest* `_XXX__XXXX`*. Zauważmy, że* `XXXX_XXX__` *nie spełnia wymagań wskazówki, jako że bloki czarnych pól znajdują się w złej kolejności. Także* `__XXXXXXX_` *nie spełnia wymagań wskazówki, gdyż zawiera tylko jeden blok czarnych pól, a nie dwa osobne.*

Masz dany częściowo rozwiązany obrazek logiczny, tzn. znasz n i c i wiesz, że niektóre pola muszą być czarne, a niektóre białe. Twoim zadaniem jest wydedukować coś więcej na temat pokolorowania pól.

Mianowicie, przez **poprawne rozwiązanie** *rozumiemy rozwiązanie spełniające wymagania wskazówki, które ponadto jest zgodne z kolorami wskazanych pól. Twój program powinien stwierdzić, które pola w dowolnym poprawnym rozwiązaniu będą pokolorowane na czarno i które pola w dowolnym poprawnym rozwiązaniu będą białe.*

Możesz założyć, że wejście jest dobrane w taki sposób, że istnieje co najmniej jedno poprawne rozwiązanie.

Szczegóły implementacji

Powinieneś napisać jedną funkcję (metodę):

- `string solve_puzzle(string s, int[] c)`
  - `s`: *napis o długości n. Dla każdego i ($0 \leqslant i \leqslant n-1$), znak i to:*
    - `'X'`*, jeśli pole i musi być czarne,*
    - `'_'`*, jeśli pole i musi być białe,*
    - `'.'`*, jeśli nic nie wiadomo o polu i.*
  - `c`: *tablica rozmiaru k zawierająca wskazówkę, zdefiniowana powyżej.*
  - *Funkcja powinna zwrócić napis długości n. Dla każdego i ($0 \leqslant i \leqslant n-1$), znak i wynikowego napisu powinien być równy:*
    - `'X'`*, jeśli pole i jest czarne w każdym poprawnym rozwiązaniu,*
    - `'_'`*, jeśli pole i jest białe w każdym poprawnym rozwiązaniu,*
    - `'?'`*, w przeciwnym przypadku (tzn. jeśli istnieją dwa poprawne rozwiązania, takie że w pierwszym z nich pole i jest czarne, a w drugim białe).*

W języku C sygnatura funkcji jest minimalnie inna:

- `void solve_puzzle(int n, char* s, int k, int* c, char* result)`
  - `n`: *długość napisu* `s` *(liczba pól),*
  - `k`: *rozmiar tablicy* `c` *(długość wskazówki),*
  - *pozostałe parametry są takie same jak powyżej,*
  - *zamiast zwracać napis złożony z n znaków, funkcja powinna go zapisać do napisu* `result`*.*

Kody ASCII znaków występujących w tym zadaniu to:

- *'*`X`*': 88,*
- *'*`_`*': 95,*
- *'*`.`*': 46,*
- *'*`?`*': 63.*

Szczegóły implementacji w Twoim języku programowania znajdują się w dostarczonych plikach z szablonami.

Przykłady

Przykład 1

```
solve_puzzle(".........", [3, 4])
```

Oto wszystkie poprawne rozwiązania łamigłówki:

- `XXX_XXXX__`,
- `XXX__XXXX_`,
- `XXX___XXXX`,
- `_XXX_XXXX_`,
- `_XXX__XXXX`,
- `__XXX_XXXX`.

Można zauważyć, że pola o indeksach (numerowanych od 0) 2, 6 i 7 w każdym poprawnym rozwiązaniu są czarne. Każde inne pole może, ale nie musi być czarne. Poprawną odpowiedzią jest zatem `??X???XX??`*.*

Przykład 2

```
solve_puzzle(".......", [3, 4])
```

W tym przykładzie całe rozwiązanie jest wyznaczone jednoznacznie i poprawną odpowiedzią jest `XXX_XXXX`*.*

Przykład 3

```
solve_puzzle("..._._....", [3])
```

W tym przykładzie możemy wywnioskować, że pole o indeksie 4 musi być białe – nie ma możliwości pokolorowania trzech kolejnych pól na czarno pomiędzy białymi polami o indeksach 3 i 5. Zatem poprawną odpowiedzią jest `???___????`.

Przykład 4

```
solve_puzzle(".X........", [3])
```

Są jedynie dwa poprawne rozwiązania spełniające powyższy opis:

- `XXX_______`,
- `_XXX______`.

Tak więc poprawną odpowiedzią jest `?XX?______`.

Podzadania

We wszystkich podzadaniach zachodzi $1 \leqslant k \leqslant n$ *oraz* $1 \leqslant c_i \leqslant n$ *dla każdego* $0 \leqslant i \leqslant k-1$.

Podzadanie 1 (7 punktów): $n \leqslant 20$, $k = 1$, *s zawiera jedynie* `'.'` *(pusta łamigłówka)*
Podzadanie 2 (3 punkty): $n \leqslant 20$, *s zawiera jedynie* `'.'`
Podzadanie 3 (22 punkty): $n \leqslant 100$, *s zawiera jedynie* `'.'`
Podzadanie 4 (27 punktów): $n \leqslant 100$, *s zawiera jedynie* `'.'` *oraz* `'_'` *(są tylko informacje o białych polach)*
Podzadanie 5 (21 punktów): $n \leqslant 100$
Podzadanie 6 (10 punktów): $n \leqslant 5\,000$, $k \leqslant 100$
Podzadanie 7 (10 punktów): $n \leqslant 200\,000$, $k \leqslant 100$

Przykładowy program sprawdzający

Przykładowy program sprawdzający wczytuje dane w następującym formacie:

- *wiersz 1: napis s,*
- *wiersz 2: liczba k, po której następuje k liczb całkowitych* $c_0, \ldots, c_{k-1}$.

Bartosz Kostka, Jakub Radoszewski
Tłumaczenie

Dostępna pamięć: 2 GB, czas: 2 s. *IOI 2016, dzień drugi, 16.08.2016*

Wykrywanie wrednej usterki

Ilshat jest inżynierem oprogramowania pracującym nad wydajnymi strukturami danych. Pewnego dnia wymyślił on nową strukturę danych, która przechowuje zbiór **nieujemnych** *n-bitowych liczb całkowitych, gdzie n jest potęgą dwójki (czyli $n = 2^b$ dla pewnego nieujemnego, całkowitego b).*

Struktura danych początkowo jest pusta. Program używający tej struktury danych musi przestrzegać następujących zasad:

- *Program może dodawać do struktury danych elementy, które są n-bitowymi liczbami całkowitymi, każdą pojedynczo, używając funkcji* `add_element(x)`*. Jeżeli program spróbuje dodać do struktury danych element już się w niej znajdujący, nic się nie dzieje.*
- *Po dodaniu ostatniego elementu program powinien wywołać funkcję* `compile_set()` *(dokładnie raz).*
- *Następnie program może wywoływać funkcję* `check_element(x)`*, aby sprawdzać, czy element x znajduje się w strukturze danych. Ta funkcja może być używana wielokrotnie.*

Kiedy Ilshat zaimplementował prototyp swojej struktury danych, miał on buga w funkcji `compile_set()`*. Bug ten zmienia kolejność cyfr binarnych każdego elementu zbioru w pewien ustalony sposób. Ilshat chciałby, abyś wykrył, jakie przestawienie cyfr powoduje bug.*

Formalnie, rozważmy ciąg $p = [p_0, \ldots, p_{n-1}]$, w którym każda liczba od 0 do $n - 1$ występuje dokładnie raz. Taki ciąg nazywamy **permutacją***. Rozważmy element zbioru, którego cyfry w zapisie binarnym to kolejno $a_0, \ldots, a_{n-1}$ (gdzie a_0 jest najbardziej znaczącym bitem). W momencie wywołania funkcji* `compile_set()`*, element ten jest podmieniany na element $a_{p_0}, a_{p_1}, \ldots, a_{p_{n-1}}$.*

Ta sama permutacja p jest używana do zamiany kolejności cyfr każdego elementu zbioru. Permutacja ta może być dowolna. W szczególności, może się zdarzyć, że $p_i = i$ dla każdego $0 \leqslant i \leqslant n - 1$.

Dla przykładu, załóżmy, że $n = 4$, $p = [2, 1, 3, 0]$, a Ty dodałeś do struktury liczby, których binarna reprezentacja to `0000`*,* `1100` *oraz* `0111`*. Wywołanie funkcji* `compile_set` *zmienia te elementy odpowiednio na* `0000`*,* `0101` *i* `1110`*.*

Twoim zadaniem jest napisanie programu, który odkryje permutację p poprzez zadawanie zapytań do struktury danych. Program powinien (w następującej kolejności):

1. *wybrać zbiór liczb n-bitowych,*
2. *dodać te liczby do struktury danych,*
3. *wywołać funkcję* `compile_set`*, aby spowodować buga,*
4. *zbadać istnienie pewnych elementów w strukturze danych,*
5. *użyć tych informacji aby odkryć i zwrócić permutację p.*

Zauważ, że Twój program może wywołać funkcję `compile_set` *jedynie raz.*

Dodatkowo istnieją pewne ograniczenia na liczbę wywołań funkcji bibliotecznych. Twój program może

- *wywołać funkcję* `add_element` *co najwyżej w razy (w jest od angielskiego słowa zapisy – „writes”),*
- *wywołać funkcję* `check_element` *co najwyżej r razy (r jest od angielskiego słowa odczyty – „reads”).*

Szczegóły implementacji

Twoim zadaniem jest zaimplementowanie jednej funkcji (metody):

- `int[] restore_permutation(int n, int w, int r)`
  - `n`: *liczba bitów w binarnej reprezentacji każdego elementu zbioru (a także długość permutacji p),*
  - `w`: *maksymalna liczba operacji* `add_element`*, którą może wykonać Twój program,*
  - `r`: *maksymalna liczba operacji* `check_element`*, którą może wykonać Twój program.*
  - *Funkcja powinna zwracać znalezioną permutację p.*

W języku C sygnatura funkcji jest minimalnie inna:

- `void restore_permutation(int n, int w, int r, int* result)`
  - `n`, `w` *i* `r` *mają takie same znaczenie jak powyżej.*
  - *Funkcja powinna zwracać znalezioną permutację p poprzez zapisanie jej do dostarczonej tablicy* `result`*: dla każdego i, powinna ona zapisać wartość* p_i *do* `result[i]`.

Funkcje biblioteczne

Do komunikowania się ze strukturą danych Twój program powinien używać następujących trzech funkcji (metod):

- `void add_element(string x)`
  - *Funkcja ta dodaje element opisany przez* `x` *do struktury danych.*
  - `x`: *napis złożony ze znaków* `0` *i* `1` *będący reprezentacją binarną liczby, która powinna być dodana do struktury danych. Długość* `x` *musi wynosić n.*
- `void compile_set()`
  - *Funkcja ta powinna być wywołana dokładnie raz. Twój program nie może wywołać funkcji* `add_element()` *po wywołaniu opisywanej funkcji. Twój program nie może także wywołać funkcji* `check_element()` *przed wywołaniem opisywanej funkcji.*

- `boolean check_element(string x)`
  - *Ta funkcja sprawdza, czy element* `x` *znajduje się w strukturze danych (po zastosowaniu permutacji* p*).*
  - `x`: *napis złożony ze znaków* `0` *oraz* `1`*, będący reprezentacją elementu, którego istnienie chcemy sprawdzić. Długość* `x` *musi wynosić* n*.*
  - *zwraca* `true`*, jeżeli element* `x` *znajduje się w strukturze danych, natomiast* `false` *w przeciwnym razie.*

Pamiętaj, że jeżeli Twój program złamie którąś z wymienionych reguł, wynikiem jego sprawdzania będzie „Zła odpowiedź”.

Dla każdego napisu pierwszy znak odpowiada za najbardziej znaczący bit odpowiadającej liczby.

Program sprawdzający ustala permutację p *przed wywołaniem funkcji* `restore_permutation`*.*

Szczegóły implementacji w Twoim języku programowania znajdują się w dostarczonych plikach z szablonami.

Przykład

Program sprawdzający wykonuje następujące wywołanie:

```
restore_permutation(4, 16, 16).
```

Mamy $n = 4$*, a program może wykonać co najwyżej 16 zapisów („writes”) i 16 odczytów („reads”).*

Program zawodnika wykonuje następujące wywołania:

- `add_element("0001")`
- `add_element("0011")`
- `add_element("0100")`
- `compile_set()`
- `check_element("0001")` *zwraca* `false`
- `check_element("0010")` *zwraca* `true`
- `check_element("0100")` *zwraca* `true`
- `check_element("1000")` *zwraca* `false`
- `check_element("0011")` *zwraca* `false`
- `check_element("0101")` *zwraca* `false`
- `check_element("1001")` *zwraca* `false`
- `check_element("0110")` *zwraca* `false`

- `check_element("1010")` *zwraca* `true`
- `check_element("1100")` *zwraca* `false`

Jest tylko jedna permutacja zgodna z wynikami funkcji `check_element()`*, a mianowicie permutacja* $p = [2, 1, 3, 0]$*. Tak więc* `restore_permutation` *powinna zwrócić* `[2, 1, 3, 0]`*.*

Podzadania

Podzadanie 1 (20 punktów): $n = 8$, $w = 256$, $r = 256$, $p_i \neq i$ *zachodzi dla co najwyżej dwóch indeksów* i $(0 \leqslant i \leqslant n - 1)$.
Podzadanie 2 (18 punktów): $n = 32$, $w = 320$, $r = 1024$
Podzadanie 3 (11 punktów): $n = 32$, $w = 1024$, $r = 320$
Podzadanie 4 (21 punktów): $n = 128$, $w = 1792$, $r = 1792$
Podzadanie 5 (30 punktów): $n = 128$, $w = 896$, $r = 896$

Przykładowy program sprawdzający

Przykładowy program sprawdzający wczytuje dane w następującym formacie:

- *wiersz 1: liczby całkowite* n, w, r,
- *wiersz 2:* n *liczb całkowitych będących elementami* p.

XXII Bałtycka Olimpiada Informatyczna,

Helsinki, Finlandia, 2016

Michał Glapa, Karol Kaszuba, Bartosz Kostka
Tłumaczenie

Dostępna pamięć: 256 MB, czas: 2,5 s. *BOI 2016, dzień pierwszy, 12.05.2016*

Park

W stolicy Bajtlandii znajduje się prostokątny park ogrodzony płotem. Drzewa w parku oraz odwiedzający ten park będą w tym zadaniu reprezentowani za pomocą kół.

W parku są cztery bramy, po jednej w każdym rogu (brama 1: na dole po lewej, brama 2: na dole po prawej, brama 3: na górze po prawej, brama 4: na górze po lewej). Odwiedzający mogą wchodzić do parku i wychodzić z niego jedynie poprzez te bramy.

Odwiedzający park może wejść do parku lub wyjść z niego w chwili, gdy reprezentujące go koło dotyka obu boków rogu odpowiedniej bramy. Odwiedzający może dowolnie poruszać się po parku, ale reprezentujące go koło nie może nachodzić na żadne z drzew ani na płot.

Twoim zadaniem jest, dla każdego odwiedzającego park oraz dla danego numeru bramy, przez którą wszedł do parku, określić numery bram, przez które może on wyjść z parku.

Wejście

W pierwszym wierszu wejścia znajdują się dwie liczby całkowite n i m: odpowiednio liczba drzew w parku oraz liczba odwiedzających park.

W drugim wierszu wejścia znajdują się dwie liczby całkowite w oraz h: szerokość oraz wysokość prostokąta opisującego park. Brama w lewym dolnym rogu ma współrzędne $(0, 0)$, natomiast ta w prawym górnym rogu znajduje się w punkcie (w, h).

Następnie dane jest n wierszy opisujące drzewa. Każdy wiersz składa się z trzech liczb całkowitych x, y oraz r: środek koła reprezentującego drzewo to (x, y), a jego promień wynosi r. Drzewa nie nachodzą na siebie nawzajem ani na płot.

W ostatnich m wierszach opisani są odwiedzający park. Każdy wiersz zawiera dwie liczby całkowite r oraz e: promień koła reprezentującego odwiedzającego oraz numer bramy, przez którą wejdzie do parku.

Dodatkowo, żadne drzewo nie nachodzi na kwadratowy obszar $2k \times 2k$ w żadnym rogu, gdzie k to maksymalny promień koła odwiedzającego park.

Wyjście

Dla każdego odwiedzającego park, Twój program powinien wypisać jeden wiersz zawierający numery bram, przez które odwiedzający może opuścić park, w kolejności rosnącej i bez spacji pomiędzy kolejnymi liczbami.

Uwagi

Dwa obiekty dotykają się, kiedy mają punkt wspólny. Dwa obiekty nachodzą na siebie, kiedy mają więcej niż jeden punkt wspólny.

Przykład

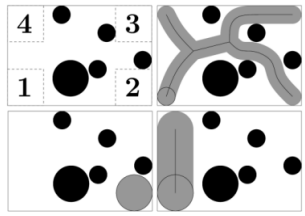
Dla danych wejściowych:

```
5 3
16 11
11 8 1
6 10 1
7 3 2
10 4 1
15 5 1
1 1
2 2
2 1
```

poprawnym wynikiem jest:

```
1234
2
14
```

Obrazek poniżej pokazuje lokalizacje bram oraz możliwe drogi każdego z odwiedzających park:



Podzadania

We wszystkich podzadaniach zachodzi $4k < w, h \leqslant 10^9$*, gdzie* k *jest maksymalnym promieniem koła odwiedzającego park.*

Podzadanie 1 (27 punktów)

- $1 \leqslant n \leqslant 2000$
- $m = 1$

Podzadanie 2 (31 punktów)

- $1 \leqslant n \leqslant 200$
- $1 \leqslant m \leqslant 10^5$

Podzadanie 3 (42 punkty)

- $1 \leqslant n \leqslant 2000$
- $1 \leqslant m \leqslant 10^5$

Michał Glapa, Karol Kaszuba, Bartosz Kostka
Tłumaczenie

Dostępna pamięć: 256 MB, czas: 1,5 s. *BOI 2016, dzień pierwszy, 12.05.2016*

Restrukturyzacja firmy

W firmie, w której zatrudnionych jest n pracowników, będzie przeprowadzana restrukturyzacja. Po jej zakończeniu struktura firmy ma mieć postać drzewa ukorzenionego, którego wierzchołki reprezentują pracowników firmy, a pracownik odpowiadający danemu wierzchołkowi jest bezpośrednim przełożonym pracowników odpowiadających synom tego wierzchołka.

Każdy pracownik przedstawił listę potencjalnych przełożonych, których może zaakceptować. Dodatkowo, należy określić pensje pracowników. Pensja musi być liczbą całkowitą dodatnią oraz pensja każdego pracownika musi być większa od sumy pensji jego bezpośrednich podwładnych.

Twoim zadaniem jest zrestrukturyzować firmę tak, aby wszystkie powyższe warunki były spełnione, a suma wszystkich pensji była jak najmniejsza.

Wejście

W pierwszym wierszu wejścia znajduje się liczba całkowita n: liczba pracowników. Pracownicy są ponumerowani kolejno liczbami $1, 2, \ldots, n$. Następnie wejście zawiera n wierszy, które opisują preferencje pracowników. Wśród tych wierszy, wiersz numer i zawiera liczbę całkowitą k_i, a następnie listę k_i liczb całkowitych. Lista ta zawiera wszystkich pracowników, których i-ty pracownik zaakceptuje jako swojego przełożonego.

Wyjście

Należy wypisać najmniejszą sumaryczną pensję spośród wszystkich sposobów przeprowadzenia restrukturyzacji firmy. Możesz założyć, że istnieje co najmniej jedno poprawne rozwiązanie.

Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
8
```

Podzadanie 1 (22 punkty)

- $2 \leqslant n \leqslant 10$
- $\sum_{i=1}^{n} k_i \leqslant 20$

Podzadanie 2 (45 punktów)

- $2 \leqslant n \leqslant 100$
- $\sum_{i=1}^{n} k_i \leqslant 200$

Podzadanie 3 (33 punkty)

- $2 \leqslant n \leqslant 5000$
- $\sum_{i=1}^{n} k_i \leqslant 10\,000$

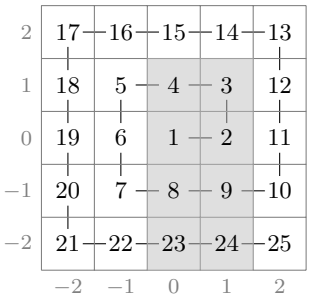
Michał Glapa, Karol Kaszuba, Bartosz Kostka
Tłumaczenie

Dostępna pamięć: 256 MB, czas: 1,5 s. *BOI 2016, dzień pierwszy, 12.05.2016*

Spirala

W pola tablicy o wymiarach $(2n+1)\times(2n+1)$ składającej się z jednostkowych, kwadratowych pól wpisano kolejne liczby naturalne. Liczbę 1 wpisano w środkowe pole, liczbę 2 wpisano w pole znajdujące się po prawej stronie pola z liczbą 1 i sąsiadujące z tym polem, a kolejne liczby wpisano w pola tablicy spiralnie, w kierunku odwrotnym do kierunku poruszania się wskazówek zegara (patrz rysunek).

Twoim zadaniem będzie udzielenie odpowiedzi na q pytań o sumę liczb w prostokątnej podtablicy tej tablicy (modulo 10^9+7). Dla przykładu, w tablicy poniżej mamy $n = 2$, a suma liczb w zacienionym prostokącie to 74:



Wejście

Pierwszy wiersz wejścia zawiera dwie liczby n oraz q: rozmiar tablicy i liczbę pytań.

W każdym z kolejnych q wierszy znajdują się cztery liczby całkowite x_1, y_1, x_2 oraz y_2 $(-n \leqslant x_1 \leqslant x_2 \leqslant n,\ -n \leqslant y_1 \leqslant y_2 \leqslant n)$. Taka czwórka oznacza, że i-te zapytanie dotyczy prostokąta, w którego rogach znajdują się pola o współrzędnych (x_1, y_1) i (x_2, y_2).

Wyjście

Dla każdego pytania wypisz w osobnym wierszu odpowiedź na to pytanie (modulo 10^9+7).

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`2 3`	`74`
`0 -2 1 1`	`9`
`-1 0 1 0`	`14`
`1 2 1 2`	

Podzadania

We wszystkich podzadaniach zachodzi $1 \leqslant q \leqslant 100$.

Podzadanie 1 (12 punktów)

- $1 \leqslant n \leqslant 1000$

Podzadanie 2 (15 punktów)

- $1 \leqslant n \leqslant 10^9$
- $x_1 = x_2$ *i* $y_1 = y_2$

Podzadanie 3 (17 punktów)

- $1 \leqslant n \leqslant 10^5$

Podzadanie 4 (31 punktów)

- $1 \leqslant n \leqslant 10^9$
- $x_1 = y_1 = 1$

Podzadanie 5 (25 punktów)

- $1 \leqslant n \leqslant 10^9$

Michał Glapa, Karol Kaszuba, Bartosz Kostka
Tłumaczenie

BOI 2016, dzień drugi, 13.05.2016

Labirynt

Uolevi napisał grę, w której gracz zbiera monety w labiryncie. Aktualnie problemem jest, że gra jest zbyt prosta. Czy możesz zaprojektować trochę labiryntów stanowiących wyzwanie dla grających w grę Uoleviego?

Każdy labirynt reprezentowany jest poprzez prostokątną tablicę zawierającą wolne pola (`.`) i ściany (`#`). Jedno pole jest bazą (`x`), a niektóre pola zawierają monety (`o`). Gracz rozpoczyna grę w bazie i może poruszać się w lewo, w prawo, w górę i w dół. Zadaniem gracza jest zebranie wszystkich monet i powrót do bazy.

Trudność labiryntu określana jest jako długość najkrótszej ścieżki, która rozpoczyna się w bazie, przechodzi przez pola z wszystkimi monetami, a na koniec wraca do bazy.

Wejście

Wejście rozpoczyna się pojedynczą liczbą całkowitą t: liczbą wymaganych labiryntów. Dalej następuje t wierszy. Każdy taki wiersz zawiera trzy liczby całkowite n, m oraz k. Oznaczają one, że rozmiar szukanego labiryntu wynosi $n \times m$ oraz że musi się w nim znaleźć dokładnie k monet.

Wyjście

Wyjście powinno zawierać t opisów labiryntów oddzielonych pustymi wierszami, w tej samej kolejności, w jakiej były podawane na wejściu. Każdy labirynt musi być rozwiązywalny.

Przykład

Dla danych wejściowych:

```
2
3 3 1
4 7 2
```

jednym z możliwych poprawnych wyników jest:

```
###
#.x
#o#

.o.####
.#..x.#
...##.#
###o...
```

Trudność pierwszego labiryntu wynosi 4, a drugiego 18.

Zgłoszenie

W tym zadaniu powinieneś zgłosić odpowiedni plik wyjściowy. Jest tylko jeden plik wejściowy, który podajemy poniżej (w dwóch kolumnach). Twój plik wyjściowy musi zawierać wszystkie labirynty wyszczególnione w pliku wejściowym.

```
50
10 18 2
11 14 3
9 17 4
8 9 6
5 11 4
10 20 5
9 10 8
7 6 9
12 20 6
12 20 12
8 14 8
11 18 4
14 5 7
12 11 4
18 6 6
9 17 6
10 13 4
8 13 6
12 12 5
11 5 5
13 12 11
13 13 6
7 15 8
5 5 4
12 9 12
7 17 10
12 16 2
10 10 4
10 20 6
19 11 10
14 13 6
17 13 8
7 19 1
8 16 8
14 9 12
13 9 2
16 5 2
7 10 1
13 17 6
18 7 11
16 13 1
16 13 12
11 12 3
15 9 5
13 19 12
5 17 1
8 16 8
6 6 10
20 13 2
11 7 3
```

Ocenianie

Dla każdego labiryntu Twoim wynikiem będzie $max(0, 100 - 3(d - x))$*, przy czym* x *jest trudnością Twojego labiryntu, a* d *jest trudnością najbardziej skomplikowanego labiryntu znalezionego przez jury. Twój całkowity wynik za zadanie będzie średnią wyników zaokrągloną w dół do najbliższej liczby całkowitej.*

Michał Glapa, Karol Kaszuba, Bartosz Kostka
Tłumaczenie

Dostępna pamięć: 256 MB, czas: 2 s. *BOI 2016, dzień drugi, 13.05.2016*

Miasta

W Bajtlandii jest n miast, spośród których pewne k to ważne miasta, często odwiedzane przez króla tej krainy.

W krainie jest m dróg łączących pewne pary miast. Są one w bardzo złym stanie, przez co król Bajtlandii nie może na nich rozwinąć prędkości maksymalnej swojego sportowego BMW z obawy przed pęknięciem opon.

Dla każdej drogi znany jest koszt jej naprawy. Twoim zadaniem jest wybrać, które z nich naprawić, tak aby między każdą parą ważnych miast było połączenie używające jedynie naprawionych dróg, a sumaryczny koszt naprawy był najmniejszy możliwy.

Wejście

W pierwszym wierszu wejścia znajdują się trzy liczby całkowite n, k i m: liczba miast, liczba ważnych miast oraz liczba dróg. Miasta są ponumerowane kolejnymi liczbami całkowitymi $1, 2, \ldots, n$.

Drugi wiersz wejścia zawiera k liczb całkowitych oznaczających numery ważnych miast.

Każdy z kolejnych m wierszy zawiera opis jednej drogi. Taki opis składa się z trzech liczb całkowitych a, b i c, oznaczających, że dana droga łączy miasta o numerach a i b, a koszt jej naprawy wynosi c.

Możesz założyć, że z każdego miasta da się dojechać do każdego innego.

Wyjście

W jedynym wierszu wyjścia Twój program powinien wypisać jedną liczbę całkowitą: minimalny koszt naprawy, która zapewnia, że wszystkie ważne miasta są ze sobą nawzajem połączone naprawionymi drogami.

Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
11
```

Podzadania

We wszystkich podzadaniach zachodzi $1 \leqslant c \leqslant 10^9$ *oraz* $n \geqslant k$.

Podzadanie 1 (22 punkty)

- $2 \leqslant k \leqslant 5$
- $n \leqslant 20$
- $1 \leqslant m \leqslant 40$

Podzadanie 2 (14 punktów)

- $2 \leqslant k \leqslant 3$
- $n \leqslant 10^5$
- $1 \leqslant m \leqslant 2 \cdot 10^5$

Podzadanie 3 (15 punktów)

- $2 \leqslant k \leqslant 4$
- $n \leqslant 1000$
- $1 \leqslant m \leqslant 2000$

Podzadanie 4 (23 punkty)

- $k = 4$
- $n \leqslant 10^5$
- $1 \leqslant m \leqslant 2 \cdot 10^5$

Podzadanie 5 (26 punktów)

- $k = 5$
- $n \leqslant 10^5$
- $1 \leqslant m \leqslant 2 \cdot 10^5$

Michał Glapa, Karol Kaszuba, Bartosz Kostka
Tłumaczenie

Dostępna pamięć: 256 MB, czas: 1 s. *BOI 2016, dzień drugi, 13.05.2016*

Zamiany

Dany jest ciąg n liczb $x_1, x_2, \ldots, x_n$. Każda liczba $1, 2, \ldots, n$ występuje w tym ciągu dokładnie raz.

Możesz modyfikować ten ciąg, zamieniając pewne pary elementów podczas niektórych spośród $n-1$ tur ponumerowanych $k = 2, 3, \ldots, n$. W turze numer k możesz (ale nie musisz) zamienić miejscami liczby x_k i $x_{\lfloor k/2 \rfloor}$.

Ciąg $a_1, a_2, \ldots, a_n$ jest leksykograficznie mniejszy niż ciąg $b_1, b_2, \ldots, b_n$, jeśli istnieje indeks j ($1 \leqslant j \leqslant n$) taki że $a_k = b_k$ dla każdego $k < j$ i $a_j < b_j$.

Jaki jest najmniejszy leksykograficznie ciąg, który możesz otrzymać w wyniku tych zamian?

Wejście

W pierwszym wierszu znajduje się liczba całkowita n. W drugim wierszu wejścia znajduje się n liczb całkowitych oznaczających kolejne wyrazy ciągu x.

Wyjście

W jedynym wierszu wyjścia Twój program powinien wypisać n liczb całkowitych: najmniejszy leksykograficznie ciąg, który możesz otrzymać.

Przykład

Dla danych wejściowych:

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

poprawnym wynikiem jest:

```
2 1 3 4 5
```

Podzadanie 1 (10 punktów)

- $1 \leqslant n \leqslant 20$

Podzadanie 2 (11 punktów)

- $1 \leqslant n \leqslant 40$

Podzadanie 3 (27 punktów)

- $1 \leqslant n \leqslant 1000$

Podzadanie 4 (20 punktów)

- $1 \leqslant n \leqslant 5 \cdot 10^4$

Podzadanie 5 (32 punkty)

- $1 \leqslant n \leqslant 2 \cdot 10^5$

XXIII Olimpiada Informatyczna Europy Środkowej,

Piatra-Neamţ, Rumunia 2016

Kamil Dębowski, Dominik Klemba
Tłumaczenie

Dostępna pamięć: 256 MB, czas: 1 s. *CEOI 2016, dzień pierwszy, 20.07.2016*

Astronauta

Juliusz – Pan Astronauta – rozpoczął właśnie staż na Międzynarodowej Stacji Kosmicznej (MSK). Jego zadaniem w trakcie stażu jest zbadanie cywilizacji Terran z planety Mar Sara. Terranie dotychczas wybudowali N miast, numerowanych od 1 do N, ale nie wybudowali jeszcze żadnych dróg pomiędzy nimi.

Gdy tylko Juliusz zaczął swój staż, cywilizacja Terran rozpoczęła budowę dróg na Mar Sara. Wiadomo, że Terranie muszą połączyć miasta w możliwie krótkim czasie, więc nie będą nigdy budować drogi łączącej miasta, między którymi istnieje już ścieżka (złożona z jednej lub wielu dróg). Za każdym razem, gdy Terranie budują nową drogę, Juliusz chce szybko dowiedzieć się, pomiędzy którymi dwoma miastami została ona wybudowana. Zależy mu, aby poznać odpowiedź, zanim wybudowana zostanie kolejna droga.

Szczęśliwie, Juliusz posiada dostęp do najnowszego satelity SETI, pozwalającego mu na analizę infrastruktury planety. Dla danych dwóch rozłącznych zbiorów miast SETI stwierdza, czy istnieje **bezpośrednia** *droga pomiędzy jakimś miastem z pierwszego zbioru oraz jakimś miastem z drugiego zbioru.*

Przebieg zdarzeń jest następujący:

1. *Terranie budują nową drogę.*
2. *Astronauta używa dobrodziejstw SETI, by oznaczyć dodaną drogę.*
3. *Udaje się on do MSK i podaje indeksy dwóch miast połączonych nową drogą.*
4. *Jeśli nie zostało jeszcze wybudowanych $N - 1$ dróg, wróć do punktu pierwszego.*

Napisz program rozwiązujący problem Astronauty.

Interakcja

Powinieneś zaimplementować funkcję `run()`*, która będzie wywołana raz, na początku działania programu. Z tej funkcji możesz wywołać funkcję* `query()` *za każdym razem, gdy chciałbyś użyć SETI. Za każdym razem, gdy określisz, która droga została wybudowana, powinieneś wywołać* `setRoad()`*. Możesz wołać* `query()` *wielokrotnie pomiędzy wywołaniami funkcji* `setRoad()`*.*

Funkcja biblioteki: `query()`

- *C/C++:* `int query(int size_a, int size_b, int a[], int b[]);`
- *Pascal:* `function query(size_a, size_b : longint, a, b : array of longint) : longint;`

Wywołuj tę funkcję, aby skorzystać z SETI. Argumenty `size_a`*,* `size_b` *powinny określać rozmiar zbiorów, o które pytamy. Tablice* `a[]`*,* `b[]` *powinny reprezentować zbiory (zawierać indeksy miast). Pamiętaj, że zbiory powinny być rozłączne! Funkcja zwraca 1, jeśli istnieje para miast z dwóch zbiorów połączona bezpośrednio drogą. W przeciwnym razie zwraca 0.*

Procedura biblioteki: `setRoad()`

- *C/C++:* `void setRoad(int a, int b);`
- *Pascal:* `procedure setRoad(a, b : longint);`

Użyj tej procedury, aby zakomunikować, iż odkryłeś nowo powstałą drogę pomiędzy miastami `a` *i* `b`*. Jeśli się pomylisz, otrzymasz zero punktów za ten test. W przeciwnym razie, wszystkie przyszłe wywołania* `query()` *będą uwzględniały tę drogę.*

Twoja procedura: `run()`

- *C/C++:* `void run(int N);`
- *Pascal:* `procedure run(N : longint);`

Twoje rozwiązanie musi implementować tę procedurę. W tej procedurze powinieneś wywoływać `query()` *oraz* `setRoad()`*. Przekazywany argument* `N` *reprezentuje liczbę miast wybudowanych przez Terran. Jeśli procedura zakończy się przed odkryciem* $\texttt{N} - 1$ *ścieżek, nie otrzymasz punktów za dany test.*

Uwagi dotyczące języka programowania

- *C/C++: dodaj* `#include "icc.h"`
- *Pascal: musisz zdefiniować* `unit icc` *oraz użyć* `uses graderhelplib`*.*

Punktacja

Zadanie jest podzielone na sześć podzadań. Testy należące do jednego podzadania będą posiadały tę samą wartość N*. Punkty za podzadanie otrzymasz, jeśli poprawnie odgadniesz wszystkie drogi w każdym teście i nie przekroczysz limitu* M *wywołań funkcji* `query()`*. Wartości* N *i* M *podane są wraz z punktami za podzadania w tabeli poniżej.*

Numer podzadania	N *= liczba miast*	M *= limit wywołań* `query()`	*Punkty*
1	*15*	*1500*	*7*
2	*50*	*2500*	*11*
3	*100*	*2250*	*22*
4	*100*	*2000*	*21*
5	*100*	*1775*	*29*
6	*100*	*1625*	*10*

Przykładowe wykonanie

Program zawodnika	*Sprawdzaczka*	*Wyjaśnienie*
`-`	`run(4)`	*Jest $N = 4$ miast. Pierwsza droga powstaje pomiędzy miastami 2 i 4 (uczestnik nie dysponuje tą informacją).*
`query(1, 3, {1}, {2, 3, 4})`	`return 0`	*Zapytanie o zbiory $\{1\}$ i $\{2, 3, 4\}$. Odpowiedź to 0: nie istnieje bezpośrednia droga do miasta 1 z żadnego miasta z drugiego zbioru.*
`query(1, 2, {2}, {3, 4})`	`return 1`	*Dla zbiorów $\{2\}$ i $\{3, 4\}$ odpowiedzią jest 1, bo istnieje droga z miasta 2 do miasta 4.*
`query(1, 1, {2}, {3})`	`return 0`	
`setRoad(2, 4)`	`-`	*Droga została poprawnie określona jako $(2, 4)$. Nowa droga powstaje pomiędzy miastami 1 i 3.*
`query(2, 2, {2, 4}, {1, 3})`	`return 0`	*Zapytanie o zbiory $\{2, 4\}$ i $\{1, 3\}$. Odpowiedź to 0.*
`setRoad(1, 3)`	`-`	*Poprawne określenie drogi $(1, 3)$. Nowa droga jest budowana pomiędzy miastami 1 i 4.*
`query(2, 2, {2, 4}, {1, 3})`	`return 1`	*Pytanie jak powyżej. Tym razem odpowiedź to 1 ze względu na nowo powstałą drogę.*
`query(1, 2, {2}, {1, 3})`	`return 0`	
`query(1, 1, {4}, {3})`	`return 0`	
`setRoad(4, 1)`	`exit`	*Ostatnia droga $(4, 1)$ została poprawnie określona. Program otrzymuje pełną punktację za ten test.*

Kamil Dębowski, Dominik Klemba
Tłumaczenie

Dostępna pamięć: 256 MB, czas: 0,2 s. *CEOI 2016, dzień pierwszy, 20.07.2016*

Kangur

Ogrodnicy nie cierpią kangurów, gdyż te wyżerają im marchew. Rozważmy ogród w kształcie długiego paska złożonego z N pól, ponumerowanych od 1 do N. Początkowo każde pole zawiera jedną marchewkę.

W polu o numerze x pojawił się kangur. W celu zjedzenia wszystkich marchewek ma zamiar odwiedzić wszystkie pola, każde dokładnie raz. Jest on w stanie skoczyć z dowolnego pola na dowolne inne pole. Chce on skończyć na polu o numerze y. Oczywiście, musi on wykonać dokładnie $N - 1$ skoków.

Kangur nie chce zostać złapany, więc z każdym skokiem musi on zmieniać kierunek. Innymi słowy, nie może on skoczyć dwa razy z rzędu w prawo (z pola a do pola b, po czym z b do c, gdzie $a < b < c$) oraz nie może on skoczyć dwa razy z rzędu w lewo (z a do b, po czym z b do c, gdzie $c < b < a$).

Znając liczbę pól N, początkowe pole x oraz końcowe pole y, znajdź liczbę różnych możliwych tras kangura.

Wejście

Plik wejściowy `kangaroo.in` *będzie zawierał trzy liczby całkowite N, x, y, oddzielone odstępami.*

Wyjście

Do pliku wyjściowego `kangaroo.out` *powinieneś wypisać jedną liczbę całkowitą – liczbę różnych możliwych tras kangura modulo 1 000 000 007.*

Limity i uwagi

- $2 \leqslant N \leqslant 2000$
- $1 \leqslant x \leqslant N$
- $1 \leqslant y \leqslant N$
- $x \neq y$
- *W testach wartych 6 punktów zachodzi $N \leqslant 8$.*
- *W testach wartych 36 punktów zachodzi $N \leqslant 40$.*
- *W testach wartych 51 punktów zachodzi $N \leqslant 200$.*
- *Każda trasa jest jednoznacznie wyznaczona przez kolejność odwiedzania pól.*

- *Jest zagwarantowane, że w każdym teście istnieje co najmniej jedna możliwa trasa kangura.*
- *Kangur może wykonać pierwszy skok (z pola x) w dowolnym kierunku.*

Przykład

Dla pliku wejściowego `kangaroo.in`*:*

```
4 2 3
```

poprawnym wynikiem jest plik wyjściowy `kangaroo.out`*:*

```
2
```

Wyjaśnienie do przykładu: *Kangur zaczyna z pola 2 i musi skończyć na polu 3. Dwie możliwe trasy to* $2 \to 1 \to 4 \to 3$ *oraz* $2 \to 4 \to 1 \to 3$.

Kamil Dębowski, Dominik Klemba
Tłumaczenie

Dostępna pamięć: 256 MB, czas: 0,6 s. *CEOI 2016, dzień pierwszy, 20.07.2016*

Sztuczka

Grupa turystów, która spokojnie zwiedzała zamek w Branie, została schwytana przez Hrabiego Drakulę. Szczęśliwie, do grupy należy magik i jego dwaj asystenci. Są oni teraz ostatnią nadzieją schwytanych. Magik postanowił wykonać przed Drakulą niesamowitą sztuczkę. Jeśli uda się zachwycić Hrabiego Drakulę, ten wypuści schwytanych.

Po rozpoczęciu sztuczki asystenci nie mogą komunikować się z magikiem ani ze sobą nawzajem. Magik wręczy Drakuli talię $2N + 1$ kart, ponumerowanych od 0 do $2N$. Hrabia wybierze i schowa jedną kartę. Następnie Hrabia wybierze N z pozostałych $2N$ kart i wręczy wybrane N kart pierwszemu asystentowi, a pozostałe N kart drugiemu asystentowi. Każdy z asystentów wybierze dwie ze swoich N kart i wręczy je magikowi, jedna po drugiej. Wtedy, znając tylko dwie kolejne karty wręczone przez pierwszego asystenta i dwie kolejne karty wręczone przez drugiego asystenta, magik zgadnie kartę schowaną przez Hrabiego Drakulę.

Pomóż magikowi i nie pozwól na to, by turyści stali się kolacją wampira!

Twój program zostanie uruchomiony trzy razy dla każdego testu. Podczas pierwszego uruchomienia Twój program przejmie rolę pierwszego asystenta. Podczas drugiego uruchomienia Twój program przejmie rolę drugiego asystenta. Podczas trzeciego uruchomienia Twój program przejmie rolę magika.

Wejście

Pierwszy wiersz pliku `trick.in` *będzie zawierał jedną liczbę całkowitą T, oznaczającą liczbę przypadków testowych (sztuczka ma być wykonana tyle razy w tym teście). Drugi wiersz będzie zawierał jedną liczbę R należącą do zbioru $\{1, 2, 3\}$, określającą rolę, którą przejmie Twój program we wszystkich przypadkach testowych w tym teście.*

Jeśli $R = 1$, Twój program przejmie rolę pierwszego asystenta. Jeśli $R = 2$, Twój program przejmie rolę drugiego asystenta. W obu przypadkach, wiersz o numerze $2i + 1$ ($1 \leqslant i \leqslant T$) będzie zawierał liczbę całkowitą N_i, mówiącą, że w i-tym przypadku testowym talia będzie składała się z $2N_i + 1$ kart. Wiersz o numerze $2i + 2$ ($1 \leqslant i \leqslant T$) będzie zawierał N_i liczb całkowitych, opisujących zbiór kart otrzymanych przez kontrolowanego asystenta w i-tym przypadku testowym.

Jeśli $R = 3$, Twój program przejmie rolę magika. Wiersz o numerze $2i + 1$ ($1 \leqslant i \leqslant T$) będzie zawierał liczbę N_i oznaczającą to samo co w poprzednim akapicie. Wiersz o numerze $2i + 2$ ($1 \leqslant i \leqslant T$) będzie zawierał cztery liczby całkowite – dwie liczby wypisane przez Twój program dla $R = 1$ oraz dwie liczby wypisane przez Twój program dla $R = 2$ (są to oczywiście liczby wypisane w i-tym przypadku testowym). Te dwie pary liczb podane zostaną w tej samej kolejności, w jakiej wypisał je Twój program dla $R = 1$ i $R = 2$.

Wyjście

Jeśli $R = 1$ lub $R = 2$, to Twój program powinien wypisać w i-tym wierszu ($1 \leqslant i \leqslant T$) pliku wyjściowego `trick.out` *dwie (oddzielone odstępem) liczby całkowite, reprezentujące dwie*

karty wybrane przez danego asystenta i wręczone magikowi w i-tym przypadku testowym. Te dwie liczby muszą być różne i muszą należeć do zbioru N_i liczb danych w pliku wejściowym.

Jeśli $R = 3$, Twój program powinien w i-tym wierszu ($1 \leqslant i \leqslant T$) wypisać jedną liczbę całkowitą, reprezentującą kartę schowaną przez Hrabiego Drakulę w i-tym przypadku testowym.

Limity

- $6 \leqslant N_i \leqslant 1\,234\,567$ $(1 \leqslant i \leqslant T)$
- $S_N \leqslant 1\,234\,567$, gdzie $S_N = N_1 + N_2 + \ldots + N_T$
- *W testach wartych 29 punktów zachodzi* $N_i = 6$ $(1 \leqslant i \leqslant T)$.
- *W innych testach wartych 19 punktów zachodzi* $6 \leqslant N_i \leqslant 30$ $(1 \leqslant i \leqslant T)$ *oraz* $S_N \leqslant 123\,456$.
- *W jeszcze innych testach wartych 30 punktów zachodzi* $6 \leqslant N_i \leqslant 500$ $(1 \leqslant i \leqslant T)$, $S_N \leqslant 123\,456$ *oraz będzie co najwyżej 10 przypadków testowych z* $N_i > 50$.

Przykład

Dla pliku wejściowego `trick.in`*:*

```
2
1
6
6 1 2 5 7 10
6
9 8 2 0 4 6
```

możliwym wynikiem jest plik wyjściowy `trick.out`*:*

```
1 2
8 4
```

dla pliku wejściowego `trick.in`*:*

```
2
2
6
3 0 4 9 12 8
6
7 1 11 10 3 5
```

możliwym wynikiem jest plik wyjściowy `trick.out`*:*

```
4 3
1 3
```

dla pliku wejściowego `trick.in`*:*

```
2
3
6
1 2 4 3
6
8 4 1 3
```

poprawnym wynikiem jest plik wyjściowy `trick.out`*:*

```
11
12
```

Wyjaśnienie do przykładu: *Przedstawione zostały trzy wywołania programu dla tego samego testu, zawierającego dwa przypadki testowe.*

W pierwszym przypadku testowym, pierwszy asystent otrzymuje karty 1, 2, 5, 6, 7 i 10 i wręcza magikowi karty 1 i 2, w tej kolejności. Drugi asystent otrzymuje karty 0, 3, 4, 8, 9 i 12 i wręcza magikowi karty 4 i 3, w tej kolejności. Wtedy magik stwierdza, że kartą schowaną przez Hrabiego Drakulę jest 11.

Kamil Dębowski, Dominik Klemba
Tłumaczenie

Dostępna pamięć: 256 MB, czas: 0,1 s. *CEOI 2016, dzień drugi, 22.07.2016*

Wyrażenie nawiasowe

Poprawnym wyrażeniem nawiasowym jest:

- *ciąg pusty;*
- *ciąg* `(`B`)`*, gdzie B jest poprawnym wyrażeniem nawiasowym;*
- *LR – sklejenie dwóch poprawnych wyrażeń nawiasowych L, R.*

Niech B będzie poprawnym wyrażeniem nawiasowym długości N. i-ty element ciągu B oznaczmy jako B_i. Dla dwóch indeksów i, j (gdzie $1 \leqslant i < j \leqslant N$) powiemy, że B_i, B_j są dopasowanymi nawiasami wtedy i tylko wtedy, gdy spełnione są następujące warunki:

- *$B_i =$* `(`*, $B_j =$* `)` *oraz*
- *$i = j - 1$ lub $C = B_{i+1}B_{i+2}\ldots B_{j-1}$ jest poprawnym wyrażeniem nawiasowym.*

Przyjmijmy, że S jest ciągiem małych liter alfabetu angielskiego. i-ty element ciągu S oznaczmy jako S_i. Powiemy, że poprawne wyrażenie nawiasowe B pasuje do S, jeśli:

- *B i S są tej samej długości oraz*
- *dla dowolnej pary indeksów i oraz j, gdzie $i < j$, jeżeli B_i i B_j są dopasowanymi nawiasami, to $S_i = S_j$.*

Dla danego ciągu S, składającego się z N małych liter alfabetu angielskiego, znajdź leksykograficznie najmniejsze poprawne wyrażenie nawiasowe, które pasuje do S, albo wypisz -1, jeżeli takie wyrażenie nie istnieje.

Wejście

Plik wejściowy `match.in` *zawiera ciąg N małych liter alfabetu angielskiego S.*

Wyjście

Do pliku wyjściowego `match.out` *powinieneś wypisać albo ciąg składający się z N znaków, który przedstawia najmniejsze leksykograficznie poprawne wyrażenie nawiasowe pasujące do danego ciągu S, albo -1, jeżeli takie wyrażenie nie istnieje.*

Limity oraz inne informacje

- *$2 \leqslant N \leqslant 100\,000$*
- *W testach wartych łącznie 10 punktów zachodzi $N \leqslant 18$.*

- *W innych testach wartych łącznie 27 punktów zachodzi* $N \leqslant 2000$.
- *Jeśli* A *i* B *są poprawnymi wyrażeniami nawiasowymi długości* N*, to powiemy, że wyrażenie* A *jest leksykograficznie mniejsze od wyrażenia* B*, jeżeli istnieje indeks* i $(1 \leqslant i \leqslant N)$*, taki że* $A_j = B_j$ *dla każdego* $j < i$*, oraz* $A_i < B_i$.
- *Znak* `(` *jest uznawany za mniejszy leksykograficznie niż znak* `)`.

Przykład

Dla pliku wejściowego `match.in`*:*

```
abbaaa
```

poprawnym wynikiem jest plik wyjściowy `match.out`*:*

```
(()())
```

Wyjaśnienie do przykładu: *Innym pasującym wyrażeniem nawiasowym mogłoby być* `(())()`*, ale nie jest ono najmniejsze leksykograficznie.*

Dla pliku wejściowego `match.in`*:*

```
abab
```

poprawnym wynikiem jest plik wyjściowy `match.out`*:*

```
-1
```

Wyjaśnienie do przykładu: *Nie istnieje poprawne wyrażenie nawiasowe pasujące do tego ciągu.*

Kamil Dębowski, Dominik Klemba
Tłumaczenie

Dostępna pamięć: 256 MB, czas: 2.5 s. *CEOI 2016, dzień drugi, 22.07.2016*

Popeala

Rumuńskie słowo 'popeală' pochodzi z rumuńskiej historycznej noweli „Alexandru Lăpuşneanul", w której tytułowy Książę Mołdawii używa wariacji tego słowa do opisania swojej nadchodzącej zemsty na uzurpatorach. Wyrażenie ostatnio wróciło niespodziewanie do łask w kontekście konkursów programistycznych. Jest używane do opisania każdej sytuacji, w której organizatorzy utrudniają życie zawodnikom w nietypowy i (zazwyczaj) niezamierzony sposób: bardzo restrykcyjne limity, niepoprawne testy, błędne stwierdzenia, literówki i inne takie wszelakie. To zadanie dotyczy właśnie takiego... popeală.

Rozważmy konkurs programistyczny, w którym bierze udział N uczestników. Uczestnicy mają do rozwiązania jedno zadanie sprawdzane za pomocą T testów. Organizatorzy chcą pogrupować testy w co najwyżej S grup.

Jak działają grupy: *każdy test należy do* **dokładnie jednej** *grupy. Grupa może zawierać dowolną dodatnią liczbę testów. Jeśli program zawodnika nie zaliczy* **jakiegokolwiek** *testu z danej grupy, to otrzyma za tę grupę 0 punktów. W przeciwnym przypadku zdobędzie liczbę punktów równą sumie punktów za testy w tej grupie.*

Organizatorzy są złośliwi, więc chcą pogrupować testy po zakończeniu konkursu. Dla każdego zawodnika wiedzą, które testy jego program rozwiązał poprawnie, i chcą **zminimalizować łączną liczbę punktów** *zdobytych przez wszystkich zawodników.*

Formalnie: Dana jest tablica liczb całkowitych Points[] rozmiaru T. i-ty test warty jest Points[i] punktów. Dana jest również dwuwymiarowa tablica Results[][] rozmiaru $N \cdot T$. Results[i][j] jest równe 1 wtedy i tylko wtedy, gdy i-ty zawodnik poprawnie rozwiązał j-ty test. W przeciwnym przypadku to pole będzie równe 0. Organizatorzy zadecydowali, że grupy będą zawierały **spójne przedziały testów**. *Innymi słowy, jeżeli testy X i Y będą w tej samej grupie, to każdy test Z ($X \leqslant Z \leqslant Y$) będzie znajdował się w grupie z nimi.*

Masz złe serce, więc chcesz pomóc organizatorom. Chcą oni wiedzieć, dla każdej wartości $1 \leqslant K \leqslant S$, jaka jest minimalna łączna liczba punktów zdobytych przez zawodników w konkursie, jeżeli organizatorzy zdecydują się podzielić testy na K grup.

Wejście

Plik wejściowy `popeala.in` *w pierwszym wierszu zawiera trzy oddzielone odstępami dodatnie liczby całkowite N, T, S. Drugi wiersz zawiera T oddzielonych odstępami dodatnich liczb całkowitych, reprezentujących kolejne elementy tablicy Points[]. Kolejne N wierszy zawiera binarne ciągi długości T, reprezentujące wiersze tablicy Results[][].*

Wyjście

Plik wyjściowy `popeala.out` *powinien składać się z S wierszy. i-ty z nich powinien zawierać jedną liczbę całkowitą: minimalną* **łączną** *liczbę punktów możliwą do uzyskania przez zawodników, jeżeli organizatorzy zdecydują się podzielić testy na i grup.*

Limity i inne informacje

- $1 \leqslant T \leqslant 20\,000$
- $1 \leqslant N \leqslant 50$
- $1 \leqslant S \leqslant min(50, T)$
- $1 \leqslant Points[i] \leqslant 10\,000$, dla każdego $1 \leqslant i \leqslant T$.
- $(Points[1] + Points[2] + \ldots + Points[T]) \cdot N \leqslant 2\,000\,000\,000$
- W testach wartych 8 punktów zachodzi $T \leqslant 40$.
- W innych testach wartych 9 punktów zachodzi $40 < T \leqslant 500$.
- W jeszcze innych testach wartych 9 punktów zachodzi $500 < T \leqslant 4000$.

Przykład

Dla pliku wejściowego `popeala.in`*:*	*poprawnym wynikiem jest plik wyjściowy* `popeala.out`*:*

`2 3 3`
`4 3 5`
`101`
`110`
	`0`
	`8`
	`16`

Wyjaśnienie do przykładu: *Jest $N = 2$ zawodników, $T = 3$ testów oraz najwyżej $S = 3$ grup. Tablica $Points[]$ to $[4, 3, 5]$.*

W przypadku pojedynczej grupy łączna liczba punktów będzie równa 0, ponieważ żaden z zawodników nie rozwiązał wszystkich testów poprawnie (wszystkie testy muszą być w jednej grupie).

W przypadku dwóch grup są dwa sposoby podzielenia testów. Jeden z nich daje łączną liczbę punktów równą 12, drugi skutkuje łączną liczbą punktów 8. Ponieważ chcemy zminimalizować wynik, wybieramy tę drugą wartość.

Kamil Dębowski, Dominik Klemba
Tłumaczenie

CEOI 2016, dzień drugi, 22.07.2016

Ruter

Adam i Bartek zostali zatrudnieni przez firmę informatyczną z miasta Piatra Neamt. Ich pierwszym projektem jest stworzenie nowego rodzaju rutera, niesamowitego **Connect Ethernet Operating Interface 2016**. *Ruter powinien składać się z:*

- N *wierzchołków wejściowych, ponumerowanych od 1 do* N*;*
- N *wierzchołków wyjściowych, ponumerowanych od* $N + 1$ *do* $2 \cdot N$*;*
- K *wierzchołków wewnętrznych, ponumerowanych od* $2 \cdot N + 1$ *do* $2 \cdot N + K$*;*
- M *skierowanych krawędzi między parami różnych wierzchołków.*

Wierzchołek X *wysyła dane do wierzchołka* Y *(czyli* Y *otrzymuje dane z* X*), jeśli:*

- $X = Y$ *lub*
- *istnieje wierzchołek* Z*, taki że* X *wysyła dane do* Z *oraz istnieje bezpośrednia krawędź z wierzchołka* Z *do wierzchołka* Y*.*

Jeśli wierzchołek X *wysyła dane do innego wierzchołka* Y*, to definiujemy* **ścieżkę danych** *z* X *do* Y *jako zbiór bezpośrednich krawędzi* $\{(A_1, A_2), (A_2, A_3), \ldots, (A_{L-1}, A_L)\}$ *dla pewnego* $L \geqslant 2$*, taki że* $A_1 = X$ *i* $A_L = Y$*.*

Ruter działa poprawnie, jeśli:

- *każdy wierzchołek wejściowy wysyła dane do każdego wierzchołka wyjściowego;*
- *każdy wierzchołek wejściowy otrzymuje dane tylko od siebie samego;*
- *każdy wierzchołek wyjściowy wysyła dane tylko do siebie samego;*
- *dla każdych dwóch różnych wierzchołków* X *i* Y*, jeśli* X *wysyła dane do* Y*, to* Y *nie wysyła danych do* X*;*
- *dla każdych dwóch różnych wierzchołków* X *i* Y*, jeśli* X *wysyła dane do* Y*, to istnieje tylko jedna ścieżka danych z* X *do* Y*. W szczególności, każde dwa wierzchołki* X *i* Y *są połączone co najwyżej jedną bezpośrednią krawędzią.*

Jak każde urządzenie elektroniczne, ruter potrzebuje prądu. Moc potrzebna do funkcjonowania wierzchołka X *jest określona jako* $P_X = IN_X \cdot OUT_X$*, gdzie* IN_X *jest liczbą wierzchołków wejściowych wysyłających dane do* X*, natomiast* OUT_X *jest liczbą wierzchołków wyjściowych otrzymujących dane z* X*. Maksymalną mocą rutera nazywamy*

$$P_{max} = max(P_1, P_2, \ldots, P_{2 \cdot N+K}).$$

Przełożony dostarczył Adamowi i Bartkowi specyfikacje techniczne kilku ruterów testowych, które umieściliśmy w tabeli na następnej stronie. Dla każdej specyfikacji przełożony żąda projektu rutera, który:

- *ma dokładnie N wierzchołków wejściowych i N wierzchołków wyjściowych;*
- *ma co najwyżej M_{lim} bezpośrednich krawędzi;*
- *używa maksymalnej mocy co najwyżej P_{lim};*
- *ma łącznie co najwyżej 500 000 wierzchołków (czyli $N_{total} = 2 \cdot N + K \leqslant 500\,000$).*

Numer testu	N	M_{lim}	P_{lim}	*Punkty*
1	*118*	*1 000 000*	*1 000 000*	*4*
2	*223*	*1 000 000*	*1 000 000*	*5*
3	*1250*	*500 000*	*500 000*	*6*
4	*5101*	*500 000*	*500 000*	*6*
5	*9934*	*500 000*	*500 000*	*26*
6	*9955*	*500 000*	*100 000*	*30*
7	*9978*	*100 000*	*100 000*	*23*

Adam i Bartek dostaną pewną liczbę punktów za każdy poprawnie zbudowany ruter, co wyspecyfikowano w ostatniej kolumnie powyższej tabeli.

Wejście

W tym zadaniu nie powinieneś wysyłać swojego programu. W archiwum pobranym ze strony konkursu znajdziesz pliki `1-router.in`, `2-router.in`, *...*, `7-router.in`. *Pliki te zawierają dane wejściowe dla każdego z testów.*

Każdy z plików wejściowych `1-router.in`, `2-router.in`, *...*, `7-router.in` *opisuje pojedynczy test. Plik zawiera w jednym wierszu trzy liczby całkowite oddzielone odstępami: N (liczbę wierzchołków wejściowych i jednocześnie liczbę wierzchołków wyjściowych), M_{lim} (maksymalną liczbę bezpośrednich krawędzi) oraz P_{lim} (ograniczenie na maksymalną moc rutera).*

Wyjście

Dla każdego pliku wejściowego powinieneś stworzyć odpowiedni plik wyjściowy `1-router.out`, `2-router.out`, *...*, `7-router.out`. *Umieść wszystkie te pliki w folderze nazwanym* `router-out` *i stwórz archiwum* `zip` *zawierające ten katalog. Jako rozwiązanie powinieneś wysłać właśnie to archiwum.*

W każdym z plików wyjściowych `1-router.out`, `2-router.out`, *...*, `7-router.out` *wypisz dwie liczby całkowite (oddzielone odstępem): $N_{total} = 2 \cdot N + K$ (łączna liczba wierzchołków rutera) oraz M (liczba bezpośrednich krawędzi). W każdym z następnych M wierszy wypisz dwie liczby całkowite X i Y, oznaczające bezpośrednią krawędź z wierzchołka X do wierzchołka Y.*

Pomocnicze skrypty

W pobranym archiwum znajdziesz też dwa skrypty `gen-out.sh` *i* `check.sh`, *a także plik wykonywalny* `verif_contestant`. *Jeśli umieścisz te trzy pliki razem z plikami wejściowymi i Twoim plikiem wykonywalnym* `router` *w jednym katalogu, będziesz mógł użyć komendy* `bash gen-out.sh`, *by wygenerować pliki wyjściowe stworzone przez Twój plik wykonywalny dla każdego*

z plików wejściowych. Możesz wtedy użyć komendy `bash check.sh`*, by sprawdzić poprawność wygenerowanych plików wyjściowych dla każdego testu. Plik wykonywalny* `router` *powinien powstać przez skompilowanie Twojego programu, który czyta wejście z pliku* `router.in` *i wypisuje wyjście do pliku* `router.out`*.*

Przykład

Dla pliku wejściowego `router.in`*:*

```
3 100 200
```

poprawnym wynikiem jest plik wyjściowy `router.out`*:*

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

Wyjaśnienie do przykładu: *Adam i Bartek muszą skonstruować ruter z trzema wierzchołkami wejściowymi i trzema wierzchołkami wyjściowymi. Ruter powinien zawierać co najwyżej 100 bezpośrednich krawędzi, a jego maksymalna moc nie powinna być większa niż 200.*

Adam i Bartek użyli łącznie 9 wierzchołków:

- *wierzchołki wejściowe 1, 2 i 3;*
- *wierzchołki wyjściowe 4, 5 i 6;*
- *wierzchołki wewnętrzne 7, 8 i 9.*

Użyli też 8 bezpośrednich krawędzi.

Maksymalna moc rutera to 9. Taką moc ma wierzchołek 8, który:

- *otrzymuje dane z $IN_8 = 3$ wierzchołków wejściowych;*
- *wysyła dane do $OUT_8 = 3$ wierzchołków wyjściowych.*

Dla pliku wejściowego `router.in`*:*

```
3 100 200
```

poprawnym wynikiem jest także plik wyjściowy `router.out`*:*

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

Wyjaśnienie do przykładu: *Dla tej samej zadanej specyfikacji technicznej inny poprawny ruter zawiera jedynie 6 wierzchołków (3 wejściowe i 3 wyjściowe).*

Maksymalna moc tego rutera to 3: każdy wierzchołek wejściowy otrzymuje dane tylko od siebie samego oraz wysyła dane do wszystkich do wszystkich trzech wierzchołków wyjściowych. Podobnie, każdy wierzchołek wyjściowy otrzymuje dane z wszystkich trzech wierzchołków wejściowych i wysyła dane tylko do siebie samego.

Bibliografia

[1] *Książki Olimpiady Informatycznej 1993/1994 – 2014/2015*. Warszawa–Wrocław/ Warszawa, 1994-2016.

[2] A. V. Aho, J. E. Hopcroft, J. D. Ullman. *Projektowanie i analiza algorytmów*. Helion, Warszawa, 2003.

[3] L. Banachowski, A. Kreczmar. *Elementy analizy algorytmów*. WNT, Warszawa, 1982.

[4] L. Banachowski, K. Diks, W. Rytter. *Algorytmy i struktury danych*. WNT, Warszawa, 1996.

[5] J. Bentley. *Perełki oprogramowania*. Helion, Warszawa, 2011.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Wprowadzenie do algorytmów*. PWN, Warszawa, 2013.

[7] M. de Berg, M. van Kreveld, M. Overmars. *Geometria obliczeniowa. Algorytmy i zastosowania*. WNT, Warszawa, 2007.

[8] R. L. Graham, D. E. Knuth, O. Patashnik. *Matematyka konkretna*. PWN, Warszawa, 2017.

[9] D. Harel. *Algorytmika. Rzecz o istocie informatyki*. WNT, Warszawa, 2008.

[10] D. E. Knuth. *Sztuka programowania*. WNT, Warszawa, 2002.

[11] W. Lipski. *Kombinatoryka dla programistów*. WNT, Warszawa, 2004.

[12] E. M. Reingold, J. Nievergelt, N. Deo. *Algorytmy kombinatoryczne*. WNT, Warszawa, 1985.

[13] K. A. Ross, C. R. B. Wright. *Matematyka dyskretna*. PWN, Warszawa, 2012.

[14] R. Sedgewick. *Algorytmy w C++. Grafy*. RM, 2003.

[15] S. S. Skiena, M. A. Revilla. *Wyzwania programistyczne*. WSiP, Warszawa, 2004.

[16] P. Stańczyk. *Algorytmika praktyczna. Nie tylko dla mistrzów*. PWN, Warszawa, 2009.

[17] M. M. Sysło. *Algorytmy*. Helion, Warszawa, 2016.

[18] M. M. Sysło. *Piramidy, szyszki i inne konstrukcje algorytmiczne.* Helion, Warszawa, 2015.

[19] J. Tomasiewicz. *Zaprzyjaźnij się z algorytmami. Przewodnik dla początkujących i średniozaawansowanych.* PWN, Warszawa, 2016.

[20] R. J. Wilson. *Wprowadzenie do teorii grafów.* PWN, Warszawa, 2007.

[21] N. Wirth. *Algorytmy + struktury danych = programy.* WNT, Warszawa, 2004.

[22] *W poszukiwaniu wyzwań. Wybór zadań z konkursów programistycznych Uniwersytetu Warszawskiego.* Warszawa, 2012.

[23] *W poszukiwaniu wyzwań 2. Zadania z Akademickich Mistrzostw Polski w Programowaniu Zespołowym 2011-2014.* Warszawa, 2015.

[24] P. Gawrychowski, T. Kociumaka, J. Radoszewski, W. Rytter, T. Waleń. Universal reconstruction of a string. In *Algorithms, Data Structures - 14th International Symposium, WADS 2015*, pages 386–397, 2015.

Niniejsza publikacja stanowi wyczerpujące źródło informacji o zawodach XXIII Olimpiady Informatycznej, przeprowadzonych w roku szkolnym 2015/2016. 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 XXVIII Międzynarodowej Olimpiady Informatycznej, XXII Bałtyckiej Olimpiady Informatycznej i XXIII Olimpiady Informatycznej Krajów Europy Środkowej.

XXIII Olimpiada Informatyczna to pozycja polecana szczególnie uczniom przygotowującym się do udziału w zawodach następnych Olimpiad oraz nauczycielom informatyki, którzy znajdą w niej interesujące i przystępnie sformułowane problemy algorytmiczne wraz z rozwiązaniami. Książka może być wykorzystywana także jako pomoc do zajęć z algorytmiki na studiach informatycznych.

Sponsorzy Olimpiady:

Asseco Poland · Jane Street · IIIT · PTI Polskie Towarzystwo Informatyczne Oddział Mazowiecki · codility · AdPilot · OEiiZK

ISBN 978-83-64292-03-3