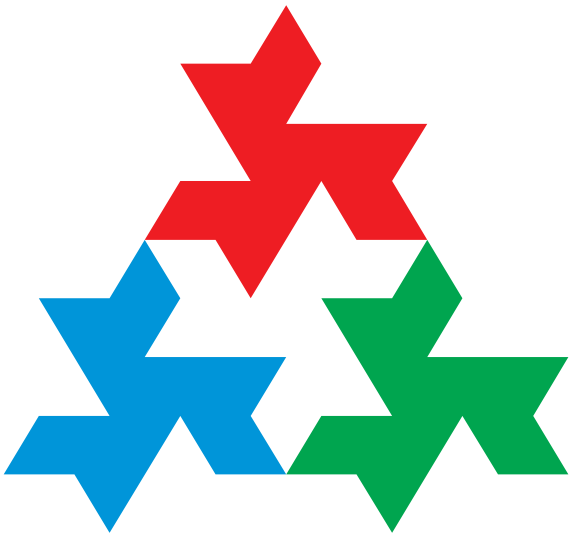
MINISTERSTWO EDUKACJI NARODOWEJ
FUNDACJA ROZWOJU INFORMATYKI
KOMITET GŁÓWNY OLIMPIADY INFORMATYCZNEJ



# XXII OLIMPIADA INFORMATYCZNA
# 2014/2015

WARSZAWA, 2016

**Autorzy tekstów:**

Krzysztof Diks
Tomasz Idziaszek
Adam Karczmarz
Jakub Łącki
Wojciech Nadara
Karol Pokorski
Jakub Radoszewski
Wojciech Rytter
Marek Sokołowski
Jacek Tomasiewicz
Michał Włodarczyk

**Autorzy programów:**

Marcin Andrychowicz
Dawid Dąbrowski
Kamil Dębowski
Bartosz Kostka
Aleksander Łukasiewicz
Bartosz Łukasiewicz
Jan Kanty Milczek
Karol Pokorski
Piotr Smulewicz
Marek Sokołowski
Marek Sommer
Tomasz Syposz
Bartosz Tarnawski

**Opracowanie i redakcja:**

Tomasz Idziaszek
Jakub Radoszewski

**Skład:**

Jakub Radoszewski

**Tłumaczenie treści zadań:**

Kamil Dębowski
Krzysztof Diks
Lech Duraj
Marek Sommer
Bartosz Szreder

Pozycja dotowana przez Ministerstwo Edukacji Narodowej.

Sponsorzy Olimpiady:



ISBN 978-83-64292-02-6

# Spis treści

**Krzysztof Diks**

# Wstęp

Drodzy Czytelnicy,

oddajemy do Waszych rąk sprawozdanie z XXII Olimpiady Informatycznej, rozgrywanej w roku szkolnym 2014/2015. Wierzymy, że wydawnictwo przygotowane pod kierunkiem dr. Jakuba Radoszewskiego zadowoli sympatyków Olimpiady Informatycznej i przyczyni się do wychowania kolejnych pokoleń uczniów-informatyków. Tak jak w każdej z dotychczasowych dwudziestu jeden niebieskich książeczek, oprócz formalnego sprawozdania z przebiegu Olimpiady znajdziemy w niej opracowania wszystkich zadań konkursowych. Zostały one przygotowane przez pomysłodawców zadań we współpracy z jurorami, którzy opracowywali programy wzorcowe oraz testy, za pomocą których sprawdzano rozwiązania uczestników zawodów olimpijskich. W książeczce znajdują się także treści zadań zawodów międzynarodowych, w których biorą udział – z licznymi sukcesami – reprezentanci Polski.

Przygotowanie Olimpiady jest pracą zespołową nauczycieli akademickich, doktorantów, studentów oraz wielu osób zaangażowanych bezpośrednio w organizację samych zawodów. Trudno wszystkich wymienić z imienia i nazwiska, ale każdemu należą się ogromne podziękowania.

W 2015 roku Olimpiada Informatyczna była organizatorem XXI Bałtyckiej Olimpiady Informatycznej. Odbyła się ona w kwietniu 2015 w Warszawie i zgromadziła 6-osobowe reprezentacje krajów regionu bałtyckiego. Olimpiada zakończyła się wielkim sukcesem, zarówno organizacyjnym, jak i sportowym. Zwycięzcą XXI Bałtyckiej Olimpiady Informatycznej został nasz reprezentant, Artur Puzio. Po raz pierwszy w historii olimpiada międzynarodowa była finansowana z pieniędzy niepublicznych, choć nie uważamy, że powinno to stać się regułą. Pieniądze zainwestowane w uzdolnionych uczniów procentują znakomicie w przyszłości. Dlatego w tym miejscu chciałbym wymienić podmioty, które wspieranie młodych polskich informatyków traktują jako misję i przyczyniły się do sukcesu organizacyjnego XXI Bałtyckiej Olimpiady Informatycznej. Są to: Bank Zachodni WBK – główny sponsor oraz Samsung – Labo, Allegro, deepsense.io, NASK, Jane Street i TEZA. Cieszy, że w wielu z tych firm znaczące role odgrywają byli olimpijczycy.

Wszystkim przyszłym olimpijczykom, którzy będą korzystać z tej publikacji, życzę licznych sukcesów w doskonaleniu umiejętności informatycznych.

*Krzysztof Diks*
*Przewodniczący Komitetu Głównego Olimpiady Informatycznej*

# Sprawozdanie z przebiegu XXII Olimpiady Informatycznej w roku szkolnym 2014/2015

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 XXII 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 XXII Olimpiady.

Zawody I stopnia rozpoczęły się 6 października 2014 roku. Ostatecznym terminem nadsyłania prac konkursowych był 3 listopada 2014 roku.

Zawody II i III stopnia były dwudniowymi sesjami stacjonarnymi, poprzedzonymi jednodniowymi sesjami próbnymi. Zawody II stopnia odbyły się w siedmiu okręgach: Białymstoku, Gdańsku, Krakowie, Poznaniu, Toruniu, Warszawie i Wrocławiu w dniach 10-12 lutego 2015 roku, natomiast zawody III stopnia odbyły się w Warszawie na Wydziale Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego, w dniach 15-18 kwietnia 2015 roku.

Uroczystość zakończenia XXII Olimpiady Informatycznej odbyła się 18 kwietnia 2015 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:
dr Przemysława Kanarek (Uniwersytet Wrocławski)
prof. dr hab. Paweł Idziak (Uniwersytet Jagielloński)

sekretarz naukowy:
dr Tomasz Idziaszek (Codility Polska)

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 Polska)
członkowie:
Joanna Bujnowska (studentka Uniwersytetu Warszawskiego)
Adam Iwaniuk (student Uniwersytetu Warszawskiego)

Adrian Jaskółka (student Uniwersytetu Warszawskiego)
mgr inż. Konrad Kozłowski (Politechnika Białostocka)

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

**Górnośląski Komitet Okręgowy**
przewodniczący:
prof. dr hab. inż. Zbigniew Czech (Politechnika Śląska w Gliwicach)
zastępca przewodniczącego:
dr inż. Krzysztof Simiński (Politechnika Śląska w Gliwicach)
członkowie:
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, Uniwersytet Jagielloński)
zastępca przewodniczącego:
dr Maciej Ślusarek (Katedra Algorytmiki, Uniwersytet Jagielloński)
sekretarz:
mgr Monika Gillert (Katedra Algorytmiki, Uniwersytet Jagielloński)
członkowie:
dr Iwona Cieślik (Katedra Algorytmiki, Uniwersytet Jagielloński)
dr Grzegorz Gutowski (Katedra Algorytmiki, Uniwersytet Jagielloński)
mgr Robert Obryk (Katedra Algorytmiki, Uniwersytet Jagielloński)
mgr Andrzej Pezarski (Katedra Algorytmiki, Uniwersytet Jagielloński)

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

**Pomorski Komitet Okręgowy**
przewodniczący:
prof. dr hab. inż. Marek Kubale (Politechnika Gdańska)
zastępca przewodniczącego:
dr hab. Andrzej Szepietowski (Uniwersytet Gdański)
sekretarz:
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)
mgr Agnieszka Polak (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 Polska)
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. Strona internetowa Komitetu Okręgowego: `http://www.ii.uni.wroc.pl/olimpiada_informatyczna_2015`

## JURY OLIMPIADY INFORMATYCZNEJ

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

Michał Adamczyk
Dawid Dąbrowski
Kamil Dębowski
Maciej Dębski
Wojciech Dyżewski
Adam Karczmarz
Bartosz Kostka
Aleksander Łukasiewicz
Bartosz Łukasiewicz
Maciej Matraszek
Jan Kanty Milczek
Marcin Parafiniuk
Karol Pokorski
Piotr Smulewicz
Marek Sokołowski
Marek Sommer
Tomasz Syposz
Bartosz Tarnawski
Olaf Tomalka
Paweł Wegner

## ZAWODY I STOPNIA

Zawody I stopnia XXII Olimpiady Informatycznej odbyły się w dniach 6 października – 3 listopada 2014 roku. Wzięło w nich udział 698 zawodników. Decyzją Komitetu Głównego zdyskwalifikowano 19 zawodników. Powodem dyskwalifikacji była niesamodzielność rozwiązań zadań konkursowych. Sklasyfikowano 679 zawodników.

Decyzją Komitetu Głównego do zawodów zostało dopuszczonych 45 uczniów gimnazjów. Pochodzili oni z następujących szkół:

| nazwa gimnazjum | miejscowość | liczba uczniów |
|---|---|---|
| Gimnazjum nr 24 w Zespole Szkół Ogólnokształcących nr 1 | Gdynia | 15 |
| XIII Gimnazjum im. Stanisława Staszica w Zespole Szkół nr 82 | Warszawa | 4 |
| Gimnazjum nr 49 z Oddziałami Dwujęzycznymi w Zespole Szkół nr 14 | Wrocław | 3 |
| Gimnazjum nr 50 w Zespole Szkół Ogólnokształcących nr 6 | Bydgoszcz | 2 |
| Publiczne Gimnazjum nr 52 Ojców Pijarów im. ks. Stanisława Konarskiego | Kraków | 2 |
| Gimnazjum nr 16 w Zespole Szkół Ogólnokształcących nr 7 | Szczecin | 2 |
| Publiczne Gimnazjum nr 1 im. Mikołaja Kopernika | Świebodzin | 2 |
| Gimnazjum nr 42 z Oddziałami Dwujęzycznymi | Warszawa | 2 |
| Publiczne Gimnazjum nr 32 Dwujęzyczne w Zespole Szkół Ogólnokształcących nr 2 | Białystok | 1 |
| Gimnazjum w Zespole Szkół | Czarna | 1 |
| Gimnazjum w Zespole Szkół i Przedszkola | Czerniewice | 1 |
| Gimnazjum w Akademii Dobrej Edukacji im. Macieja Płażyńskiego | Gdańsk | 1 |
| Gimnazjum nr 16 im. Króla Stefana Batorego | Kraków | 1 |
| Pallotyńskie Gimnazjum im. Stefana Batorego | Lublin | 1 |
| Salezjańskie Gimnazjum im. Księdza Bosko | Łódź | 1 |
| Publiczne Gimnazjum nr 23 w Zespole Szkół Ogólnokształcących nr 6 im. Jana Kochanowskiego | Radom | 1 |
| Gimnazjum nr 32 z Oddziałami Dwujęzycznymi w Zespole Szkół Ogólnokształcących nr 2 | Szczecin | 1 |
| Gimnazjum Dwujęzyczne im. Świętej Kingi w Zespole Szkół Ogólnokształcących nr 2 | Tarnów | 1 |
| Gimnazjum Akademickie w Zespole Szkół UMK | Toruń | 1 |
| Gimnazjum nr 124 im. Polskich Noblistów | Warszawa | 1 |
| Gimnazjum nr 1 im. Hugona Dionizego Steinhausa | Wrocław | 1 |

Kolejność województw pod względem liczby zawodników była następująca:

| | | | |
|---|---|---|---|
| mazowieckie | 141 zawodników | zachodniopomorskie | 23 |
| małopolskie | 91 | podkarpackie | 18 |
| dolnośląskie | 88 | wielkopolskie | 16 |
| pomorskie | 73 | łódzkie | 14 |
| kujawsko-pomorskie | 60 | opolskie | 12 |
| podlaskie | 57 | świętokrzyskie | 10 |
| śląskie | 37 | warmińsko-mazurskie | 7 |
| lubelskie | 26 | lubuskie | 6 |

W zawodach I stopnia najliczniej reprezentowane były szkoły:

| nazwa szkoły | miejscowość | liczba uczniów |
|---|---|---|
| Zespół Szkół nr 82 (XIV Liceum Ogólnokształcące im. Stanisława Staszica oraz Gimnazjum nr 13 im. Stanisława Staszica) | Warszawa | 70 |
| III Liceum Ogólnokształcące im. Marynarki Wojennej RP i Gimnazjum nr 24 | Gdynia | 50 |
| I Liceum Ogólnokształcące im. Adama Mickiewicza | Białystok | 45 |
| V Liceum Ogólnokształcące im. Augusta Witkowskiego | Kraków | 45 |
| Zespół Szkół nr 14 (Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej oraz Gimnazjum nr 49 z Oddziałami Dwujęzycznymi) | Wrocław | 41 |
| Zespół Szkół Ogólnokształcących nr 6 (VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich oraz Gimnazjum nr 50) | Bydgoszcz | 28 |
| Liceum Ogólnokształcące nr III im. Adama Mickiewicza | Wrocław | 27 |
| Zespół Szkół Uniwersytetu Mikołaja Kopernika (Gimnazjum i Liceum Akademickie) | Toruń | 22 |
| Zespół Szkół Ogólnokształcących nr 7 (XIII Liceum Ogólnokształcące oraz Gimnazjum nr 16) | Szczecin | 17 |
| Zespół Szkół nr 15 (VIII Liceum Ogólnokształcące im. Króla Władysława IV i Gimnazjum nr 58 z Oddziałami Dwujęzycznymi im. Króla Władysława IV) | Warszawa | 17 |
| I Liceum Ogólnokształcące im. Stanisława Staszica | Lublin | 15 |
| Zespół Szkół Ogólnokształcących nr 6 (VI Liceum Ogólnokształcące im. Jana Kochanowskiego oraz Publiczne Gimnazjum nr 23) | Radom | 13 |
| I Liceum Ogólnokształcące im. Tadeusza Kościuszki | Legnica | 11 |
| III Liceum Ogólnokształcące im. Adama Mickiewicza | Tarnów | 11 |
| Zespół Szkół Łączności im. Obrońców Poczty Polskiej w Gdańsku | Kraków | 7 |
| V Liceum Ogólnokształcące im. Stefana Żeromskiego | Gdańsk | 6 |
| VIII Liceum Ogólnokształcące im. Adama Mickiewicza | Poznań | 5 |
| Zespół Szkół Ogólnokształcących nr 4 (IV Liceum Ogólnokształcące im. Tadeusza Kościuszki) | Toruń | 5 |
| XVIII Liceum Ogólnokształcące im. Jana Zamoyskiego | Warszawa | 5 |
| Zespół Szkół i Placówek Oświatowych – V Liceum Ogólnokształcące | Bielsko-Biała | 4 |

| | | |
|---|---|---|
| Publiczne Liceum Ogólnokształcące nr II z Oddziałami Dwujęzycznymi im. Marii Konopnickiej | Opole | 4 |
| Zespół Szkół Ogólnokształcących nr 2 (II Liceum Ogólnokształcące im. Hetmana Jana Tarnowskiego oraz Gimnazjum Dwujęzyczne im. Świętej Kingi) | Tarnów | 4 |
| Zespół Szkół Ogólnokształcących nr 2 (II Liceum Ogólnokształcące im. Księżnej Anny z Sapiehów Jabłonowskiej i Publiczne Gimnazjum nr 32 Dwujęzyczne) | Białystok | 3 |
| Zespół Szkół Ogólnokształcących im. Stefana Żeromskiego (III Liceum Ogólnokształcące im. Stefana Żeromskiego) | Bielsko-Biała | 3 |
| IX Liceum Ogólnokształcące im. Cypriana Kamila Norwida | Częstochowa | 3 |
| VIII Liceum Ogólnokształcące im. Stanisława Wyspiańskiego | Kraków | 3 |
| Zespół Szkół Elektryczno-Mechanicznych im. gen. Józefa Kustronia | Nowy Sącz | 3 |
| Zespół Szkół Ogólnokształcących (Publiczne Liceum Ogólnokształcące nr III z Oddziałami Dwujęzycznymi im. Marii Skłodowskiej-Curie) | Opole | 3 |
| I Liceum Ogólnokształcące im. Mikołaja Kopernika | Rzeszów | 3 |
| I Liceum Ogólnokształcące im. Bolesława Prusa | Siedlce | 3 |
| Zespół Szkół Technicznych | Strzyżów | 3 |
| I Liceum Ogólnokształcące im. Marii Konopnickiej | Suwałki | 3 |

Najliczniej reprezentowane były następujące miejscowości:

| | | | |
|---|---|---|---|
| Warszawa | 109 zawodników | Łódź | 7 |
| Wrocław | 73 | Olsztyn | 6 |
| Kraków | 63 | Rzeszów | 6 |
| Białystok | 53 | Katowice | 5 |
| Gdynia | 51 | Rybnik | 5 |
| Bydgoszcz | 28 | Częstochowa | 4 |
| Toruń | 27 | Kielce | 4 |
| Lublin | 20 | Nowy Sącz | 4 |
| Szczecin | 20 | Wadowice | 4 |
| Radom | 17 | Chorzów | 3 |
| Tarnów | 17 | Gliwice | 3 |
| Gdańsk | 13 | Inowrocław | 3 |
| Poznań | 13 | Końskie | 3 |
| Legnica | 11 | Siedlce | 3 |
| Bielsko-Biała | 10 | Strzyżów | 3 |
| Opole | 9 | Suwałki | 3 |

Zawodnicy uczęszczali do następujących klas:

| | |
|---|---|
| do klasy I gimnazjum | 1 uczeń |
| do klasy II gimnazjum | 11 uczniów |
| do klasy III gimnazjum | 33 |
| do klasy I szkoły ponadgimnazjalnej | 107 |
| do klasy II szkoły ponadgimnazjalnej | 220 |
| do klasy III szkoły ponadgimnazjalnej | 282 |
| do klasy IV szkoły ponadgimnazjalnej | 25 |

W zawodach I stopnia zawodnicy mieli do rozwiązania pięć zadań:

- „Czarnoksiężnicy okrągłego stołu" autorstwa Wojciecha Ryttera,
- „Kinoman" autorstwa Tomasza Idziaszka,
- „Kwadraty" autorstwa Wojciecha Ryttera,
- „Łasuchy" autorstwa Michała Włodarczyka,
- „Pieczęć" autorstwa Jakuba Łąckiego,

każde oceniane maksymalnie po 100 punktów.

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

- **CZA** – Czarnoksiężnicy okrągłego stołu

| | **CZA** | |
|---|---|---|
| | liczba zawodników | czyli |
| 100 pkt. | 25 | 3,68% |
| 75–99 pkt. | 8 | 1,18% |
| 50–74 pkt. | 5 | 0,74% |
| 1–49 pkt. | 166 | 24,45% |
| 0 pkt. | 157 | 23,12% |
| brak rozwiązania | 318 | 46,83% |

- **KIN** – Kinoman

| | **KIN** | |
|---|---|---|
| | liczba zawodników | czyli |
| 100 pkt. | 149 | 21,95% |
| 75–99 pkt. | 21 | 3,09% |
| 50–74 pkt. | 3 | 0,44% |
| 1–49 pkt. | 291 | 42,86% |
| 0 pkt. | 69 | 10,16% |
| brak rozwiązania | 146 | 21,50% |

- **KWA** – Kwadraty

| | **KWA** | |
|---|---|---|
| | liczba zawodników | czyli |
| 100 pkt. | 193 | 28,42% |
| 75–99 pkt. | 23 | 3,39% |
| 50–74 pkt. | 8 | 1,18% |
| 1–49 pkt. | 99 | 14,58% |
| 0 pkt. | 72 | 10,60% |
| brak rozwiązania | 284 | 41,83% |

- **LAS** – Łasuchy

| | **LAS** | |
|---|---|---|
| | liczba zawodników | czyli |
| 100 pkt. | 239 | 35,20% |
| 75–99 pkt. | 15 | 2,21% |
| 50–74 pkt. | 41 | 6,04% |
| 1–49 pkt. | 21 | 3,09% |
| 0 pkt. | 226 | 33,28% |
| brak rozwiązania | 137 | 20,18% |

- **PIE** – Pieczęć

| | **PIE** | |
|---|---|---|
| | liczba zawodników | czyli |
| 100 pkt. | 335 | 49,34% |
| 75–99 pkt. | 52 | 7,66% |
| 50–74 pkt. | 21 | 3,09% |
| 1–49 pkt. | 68 | 10,02% |
| 0 pkt. | 87 | 12,81% |
| brak rozwiązania | 116 | 17,08% |

W sumie za wszystkie 5 zadań konkursowych:

| **SUMA** | liczba zawodników | czyli |
|---|---|---|
| 500 pkt. | 20 | 2,94% |
| 375–499 pkt. | 83 | 12,22% |
| 250–374 pkt. | 130 | 19,15% |
| 125–249 pkt. | 122 | 17,97% |
| 1–124 pkt. | 221 | 32,55% |
| 0 pkt. | 103 | 15,17% |

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 10–12 lutego 2015 roku w siedmiu okręgach, zakwalifikowano 354 zawodników, którzy osiągnęli w zawodach I stopnia wynik nie mniejszy niż 128 pkt.

Zawodnicy zostali przydzieleni do okręgów w następującej liczbie:

| | | | |
|---|---|---|---|
| Białystok | 45 zawodników | Toruń | 29 |
| Gdańsk | 33 | Warszawa | 89 |
| Kraków | 83 | Wrocław | 53 |
| Poznań | 22 | | |

Czterech zawodników nie stawiło się na zawody, w zawodach wzięło więc udział 350 zawodników.

Zawodnicy uczęszczali do szkół w następujących województwach:

| | | | |
|---|---|---|---|
| mazowieckie | 85 zawodników | dolnośląskie | 49 |
| małopolskie | 54 | podlaskie | 43 |

| | | | |
|---|---|---|---|
| pomorskie | 33 | wielkopolskie | 6 |
| kujawsko-pomorskie | 28 | podkarpackie | 5 |
| zachodniopomorskie | 16 | świętokrzyskie | 5 |
| lubelskie | 13 | opolskie | 3 |
| śląskie | 8 | łódzkie | 2 |

W zawodach II stopnia najliczniej reprezentowane były szkoły:

| nazwa szkoły | miejscowość | liczba uczniów |
|---|---|---|
| Zespół Szkół nr 82 (XIV Liceum Ogólnokształcące im. Stanisława Staszica oraz Gimnazjum nr 13 im. Stanisława Staszica) | Warszawa | 44 |
| I Liceum Ogólnokształcące im. Adama Mickiewicza | Białystok | 38 |
| Zespół Szkół nr 14 (Liceum Ogólnokształcące nr XIV im. Polonii Belgijskiej oraz Gimnazjum nr 49 z Oddziałami Dwujęzycznymi) | Wrocław | 32 |
| V Liceum Ogólnokształcące im. Augusta Witkowskiego | Kraków | 30 |
| III Liceum Ogólnokształcące im. Marynarki Wojennej RP i Gimnazjum nr 24 | Gdynia | 25 |
| Zespół Szkół Ogólnokształcących nr 6 (VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich oraz Gimnazjum nr 50) | Bydgoszcz | 14 |
| VI Liceum Ogólnokształcące im. Jana Kochanowskiego w Zespole Szkół Ogólnokształcących nr 6 | Radom | 13 |
| Zespół Szkół Ogólnokształcących nr 7 (XIII Liceum Ogólnokształcące oraz Gimnazjum nr 16) | Szczecin | 13 |
| Zespół Szkół Uniwersytetu Mikołaja Kopernika (Liceum Akademickie) | Toruń | 11 |
| VIII Liceum Ogólnokształcące im. Króla Władysława IV w Zespole Szkół nr 15 | Warszawa | 11 |
| Liceum Ogólnokształcące nr III im. Adama Mickiewicza | Wrocław | 11 |
| III Liceum Ogólnokształcące im. Adama Mickiewicza | Tarnów | 10 |
| I Liceum Ogólnokształcące im. Stanisława Staszica | Lublin | 7 |
| I Liceum Ogólnokształcące im. Tadeusza Kościuszki | Legnica | 5 |
| Zespół Szkół Ogólnokształcących nr 2 (II Liceum Ogólnokształcące im. Hetmana Jana Tarnowskiego) | Tarnów | 3 |

Najliczniej reprezentowane były miejscowości:

| | | | |
|---|---|---|---|
| Warszawa | 68 zawodników | Gdynia | 26 |
| Wrocław | 43 | Szczecin | 15 |
| Białystok | 41 | Bydgoszcz | 14 |
| Kraków | 38 | Radom | 13 |

| | | | |
|---|---|---|---|
| Tarnów | 13 | Gdańsk | 5 |
| Toruń | 11 | Legnica | 5 |
| Lublin | 9 | Opole | 3 |
| Poznań | 6 | Rzeszów | 3 |

10 lutego odbyła się sesja próbna, podczas której zawodnicy rozwiązywali nieliczące się do ogólnej klasyfikacji zadanie „Logistyka" autorstwa Michała Włodarczyka. W dniach konkursowych zawodnicy rozwiązywali następujące zadania:

- w pierwszym dniu zawodów (11 lutego):
  - „Podział naszyjnika" autorstwa Jacka Tomasiewicza,
  - „Pustynia" autorstwa Adama Karczmarza,
- w drugim dniu zawodów (12 lutego):
  - „Kurs szybkiego czytania" autorstwa Wojciecha Ryttera,
  - „Trzy wieże" autorstwa Jacka Tomasiewicza,

każde oceniane maksymalnie po 100 punktów.

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

- **LOG – próbne** – Logistyka

| | **LOG – próbne** | |
|---|---|---|
| | liczba zawodników | czyli |
| 100 pkt. | 34 | 9,71% |
| 75–99 pkt. | 7 | 2,00% |
| 50–74 pkt. | 31 | 8,86% |
| 1–49 pkt. | 146 | 41,71% |
| 0 pkt. | 80 | 22,86% |
| brak rozwiązania | 52 | 14,86% |

- **POD** – Podział naszyjnika

| | **POD** | |
|---|---|---|
| | liczba zawodników | czyli |
| 100 pkt. | 12 | 3,43% |
| 75–99 pkt. | 6 | 1,71% |
| 50–74 pkt. | 16 | 4,57% |
| 1–49 pkt. | 202 | 57,72% |
| 0 pkt. | 62 | 17,71% |
| brak rozwiązania | 52 | 14,86% |

- **PUS** – Pustynia

| | **PUS** | |
|---|---|---|
| | liczba zawodników | czyli |
| 100 pkt. | 3 | 0,86% |
| 75–99 pkt. | 4 | 1,14% |
| 50–74 pkt. | 11 | 3,14% |
| 1–49 pkt. | 58 | 16,57% |
| 0 pkt. | 159 | 45,43% |
| brak rozwiązania | 115 | 32,86% |

- **KUR** – Kurs szybkiego czytania

| | **KUR** | |
|---|---|---|
| | liczba zawodników | czyli |
| 100 pkt. | 10 | 2,86% |
| 75–99 pkt. | 4 | 1,14% |
| 50–74 pkt. | 7 | 2,00% |
| 1–49 pkt. | 220 | 62,86% |
| 0 pkt. | 83 | 23,71% |
| brak rozwiązania | 26 | 7,43% |

- **TRZ** – Trzy wieże

| | **TRZ** | |
|---|---|---|
| | liczba zawodników | czyli |
| 100 pkt. | 28 | 8,00% |
| 75–99 pkt. | 4 | 1,14% |
| 50–74 pkt. | 5 | 1,43% |
| 1–49 pkt. | 257 | 73,43% |
| 0 pkt. | 47 | 13,43% |
| brak rozwiązania | 9 | 2,57% |

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

| **SUMA** | liczba zawodników | czyli |
|---|---|---|
| 400 pkt. | 0 | 0,00% |
| 300–399 pkt. | 3 | 0,86% |
| 200–299 pkt. | 15 | 4,29% |
| 100–199 pkt. | 62 | 17,71% |
| 1–99 pkt. | 255 | 72,86% |
| 0 pkt. | 15 | 4,29% |

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 XXII Olimpiady Informatycznej.

## ZAWODY III STOPNIA

Zawody III stopnia odbyły się na Wydziale Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego w dniach 15–18 kwietnia 2015 roku. Do zawodów III stopnia zakwalifikowano 99 najlepszych uczestników zawodów II stopnia, którzy uzyskali wynik nie mniejszy niż 80 pkt.

Zawodnicy uczęszczali do szkół w następujących województwach:

| | | | |
|---|---|---|---|
| mazowieckie | 30 zawodników | zachodniopomorskie | 7 |
| pomorskie | 14 | lubelskie | 4 |
| dolnośląskie | 11 | śląskie | 2 |
| kujawsko-pomorskie | 10 | świętokrzyskie | 2 |
| małopolskie | 9 | łódzkie | 1 zawodnik |
| podlaskie | 8 | wielkopolskie | 1 |

Niżej wymienione szkoły miały w finale więcej niż jednego zawodnika:

| nazwa szkoły | miejscowość | liczba uczniów |
|---|---|---|
| Zespół Szkół nr 82 (XIV Liceum Ogólnokształcące im. Stanisława Staszica oraz Gimnazjum nr 13 im. Stanisława Staszica) | Warszawa | 17 |
| III Liceum Ogólnokształcące im. Marynarki Wojennej RP i Gimnazjum nr 24 | Gdynia | 12 |
| Zespół Szkół nr 14 (Liceum Ogólnokształcące nr 14 im. Polonii Belgijskiej) | Wrocław | 9 |
| I Liceum Ogólnokształcące im. Adama Mickiewicza | Białystok | 7 |
| Zespół Szkół Ogólnokształcących nr 6 (VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich) | Bydgoszcz | 7 |
| V Liceum Ogólnokształcące im. Augusta Witkowskiego | Kraków | 7 |
| Zespół Szkół Ogólnokształcących nr 7 (XIII Liceum Ogólnokształcące oraz Gimnazjum nr 16) | Szczecin | 6 |
| VI Liceum Ogólnokształcące im. Jana Kochanowskiego w Zespole Szkół Ogólnokształcących nr 6 | Radom | 5 |
| Zespół Szkół Uniwersytetu Mikołaja Kopernika (Liceum Akademickie) | Toruń | 3 |
| Zespół Szkół nr 15 (VIII Liceum Ogólnokształcące im. Króla Władysława IV) | Warszawa | 2 |
| Liceum Ogólnokształcące nr III im. Adama Mickiewicza | Wrocław | 2 |

15 kwietnia odbyła się sesja próbna, podczas której zawodnicy rozwiązywali nieliczące się do ogólnej klasyfikacji zadanie „Odwiedziny” autorstwa Wojciecha Nadary. W dniach konkursowych zawodnicy rozwiązywali następujące zadania:

- w pierwszym dniu zawodów (16 kwietnia):
  - „Myjnie” autorstwa Michała Włodarczyka,
  - „Tablice kierunkowe” autorstwa Wojciecha Nadary,
  - „Wilcze doły” autorstwa Jacka Tomasiewicza,
- w drugim dniu zawodów (17 kwietnia):
  - „Kolekcjoner Bajtemonów” autorstwa Marka Sokołowskiego,
  - „Modernizacja autostrady” autorstwa Tomasza Idziaszka,
  - „Wycieczki” autorstwa Karola Pokorskiego,

każde oceniane maksymalnie po 100 punktów.

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

- **ODW – próbne** – Odwiedziny

| | **ODW – próbne** | |
|---|---|---|
| | liczba zawodników | czyli |
| 100 pkt. | 23 | 23,23% |
| 75–99 pkt. | 4 | 4,04% |
| 50–74 pkt. | 2 | 2,02% |
| 1–49 pkt. | 54 | 54,55% |
| 0 pkt. | 14 | 14,14% |
| brak rozwiązania | 2 | 2,02% |

- **MYJ** – Myjnie

| | **MYJ** | |
|---|---|---|
| | liczba zawodników | czyli |
| 100 pkt. | 2 | 2,02% |
| 75–99 pkt. | 0 | 0% |
| 50–74 pkt. | 0 | 0% |
| 1–49 pkt. | 24 | 24,24% |
| 0 pkt. | 23 | 23,23% |
| brak rozwiązania | 50 | 50,51% |

- **TAB** – Tablice kierunkowe

| | **TAB** | |
|---|---|---|
| | liczba zawodników | czyli |
| 100 pkt. | 5 | 5,05% |
| 75–99 pkt. | 1 | 1,01% |
| 50–74 pkt. | 3 | 3,03% |
| 1–49 pkt. | 21 | 21,21% |
| 0 pkt. | 25 | 25,25% |
| brak rozwiązania | 44 | 44,45% |

- **WIL** – Wilcze doły

| | **WIL** | |
|---|---|---|
| | liczba zawodników | czyli |
| 100 pkt. | 47 | 47,48% |
| 75–99 pkt. | 18 | 18,18% |
| 50–74 pkt. | 12 | 12,12% |
| 1–49 pkt. | 16 | 16,16% |
| 0 pkt. | 5 | 5,05% |
| brak rozwiązania | 1 | 1,01% |

- **KOL** – Kolekcjoner Bajtemonów

| | **KOL** | |
|---|---|---|
| | liczba zawodników | czyli |
| 100 pkt. | 20 | 20,2% |
| 75–99 pkt. | 16 | 16,16% |
| 50–74 pkt. | 3 | 3,03% |
| 1–49 pkt. | 46 | 46,47% |
| 0 pkt. | 10 | 10,1% |
| brak rozwiązania | 4 | 4,04% |

- **MOD** – Modernizacja autostrady

| | **MOD** | |
|---|---|---|
| | liczba zawodników | czyli |
| 100 pkt. | 9 | 9,09% |
| 75–99 pkt. | 2 | 2,02% |
| 50–74 pkt. | 6 | 6,06% |
| 1–49 pkt. | 34 | 34,35% |
| 0 pkt. | 23 | 23,23% |
| brak rozwiązania | 25 | 25,25% |

- **WYC** – Wycieczki

| | **WYC** | |
|---|---|---|
| | liczba zawodników | czyli |
| 100 pkt. | 5 | 5,05% |
| 75–99 pkt. | 6 | 6,06% |
| 50–74 pkt. | 4 | 4,04% |
| 1–49 pkt. | 40 | 40,4% |
| 0 pkt. | 8 | 8,08% |
| brak rozwiązania | 36 | 36,37% |

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

| **SUMA** | liczba zawodników | czyli |
|---|---|---|
| 450–600 pkt. | 2 | 2,02% |
| 300–449 pkt. | 12 | 12,12% |
| 150–299 pkt. | 43 | 43,44% |
| 1–149 pkt. | 42 | 42,42% |
| 0 pkt. | 0 | 0,00% |

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

Poniżej zestawiono listę wszystkich laureatów i wyróżnionych finalistów:

(1) **Przemysław Jakub Kozłowski**, 3 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza, Białystok, 482 pkt., laureat I miejsca

(2) **Jarosław Kwiecień**, 2 klasa, XIV Liceum Ogólnokształcące im. Polonii Belgijskiej, Wrocław, 480 pkt., laureat I miejsca

(3) **Konrad Paluszek**, 3 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica, Warszawa, 444 pkt., laureat I miejsca

(4) **Maciej Hołubowicz**, 3 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza, Białystok, 434 pkt., laureat I miejsca

(5) **Katarzyna Kowalska**, 3 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica, Warszawa, 395 pkt., laureatka II miejsca

(6) **Michał Zawalski**, 3 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica, Warszawa, 390 pkt., laureat II miejsca

(7) **Jan Tabaszewski**, 2 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica, Warszawa, 376 pkt., laureat II miejsca

(8) **Tomasz Garbus**, 3 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia, 363 pkt., laureat II miejsca

(9) **Mariusz Trela**, 3 klasa, Publiczne Gimnazjum nr 52 Ojców Pijarów, Kraków, 356 pkt., laureat II miejsca

(10) **Paweł Burzyński**, 1 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia, 354 pkt., laureat II miejsca

(11) **Jakub Boguta**, 1 klasa, Prywatne Liceum Ogólnokształcące im. Królowej Jadwigi, Lublin, 320 pkt., laureat II miejsca

(12) **Artur Puzio**, 1 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica, Warszawa, 318 pkt., laureat II miejsca

(13) **Konrad Majewski**, 3 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica, Warszawa, 308 pkt., laureat II miejsca

(14) **Franciszek Budrowski**, 1 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza, Białystok, 304 pkt., laureat II miejsca

(15) **Marek Zbysiński**, 3 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica, Warszawa, 295 pkt., laureat III miejsca

(16) **Tadeusz Dudkiewicz**, 3 klasa, I Liceum Ogólnokształcące im. Stanisława Staszica, Lublin, 289 pkt., laureat III miejsca

(17) **Stanisław Strzelecki**, 3 klasa, XIII Gimnazjum im. Stanisława Staszica, Warszawa, 280 pkt., laureat III miejsca

(18) **Michał Tepper**, 2 klasa, VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich, Bydgoszcz, 275 pkt., laureat III miejsca

(19) **Tomasz Grześkiewicz**, 1 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica, Warszawa, 264 pkt., laureat III miejsca

(20) **Eryk Kijewski**, 2 klasa, I Liceum Ogólnokształcące im. Marii Konopnickiej, Suwałki, 263 pkt., laureat III miejsca

(21) **Angelika Serwa**, 3 klasa, IX Liceum Ogólnokształcące im. Cypriana Kamila Norwida, Częstochowa, 251 pkt., laureatka III miejsca

(22) **Adrian Naruszko**, 3 klasa, VI Liceum Ogólnokształcące im. Jana Kochanowskiego, Radom, 248 pkt., laureat III miejsca

(23) **Agnieszka Dudek**, 2 klasa, XIV Liceum Ogólnokształcące im. Polonii Belgijskiej, Wrocław, 245 pkt., laureatka III miejsca

(24) **Bartosz Narkiewicz**, 3 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia, 233 pkt., laureat III miejsca

(24) **Jan Olkowski**, 1 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica, Warszawa, 233 pkt., laureat III miejsca

(26) **Piotr Pawlak**, 2 klasa, Ogólnokształcąca Szkoła Muzyczna I i II stopnia im. F. Nowowiejskiego, Gdańsk, 230 pkt., laureat III miejsca

(27) **Mateusz Radecki**, 2 klasa, VI Liceum Ogólnokształcące im. Jana Kochanowskiego, Radom, 228 pkt., laureat III miejsca

(28) **Adrian Siwiec**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków, 226 pkt., laureat III miejsca

(29) **Kacper Kluk**, 3 klasa, Gimnazjum nr 24 w Zespole Szkół Ogólnokształcących nr 1, Gdynia, 223 pkt., laureat III miejsca

(30) **Marek Żochowski**, 1 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia, 221 pkt., laureat III miejsca

(31) **Piotr Kowalewski**, 3 klasa, Gimnazjum nr 24 w Zespole Szkół Ogólnokształcących nr 1, Gdynia, 220 pkt., laureat III miejsca

(32) **Juliusz Pham**, 3 klasa, Gimnazjum nr 24 w Zespole Szkół Ogólnokształcących nr 1, Gdynia, 217 pkt., laureat III miejsca

(33) **Kamil Rajtar**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków, 214 pkt., laureat III miejsca

(34) **Martyna Siejba**, 3 klasa, XIV Liceum Ogólnokształcące im. Polonii Belgijskiej, Wrocław, 213 pkt., laureatka III miejsca

(35) **Filip Tokarski**, 3 klasa, Zespół Szkół UMK Gimnazjum i Liceum Akademickie, Toruń, 212 pkt., laureat III miejsca

(36) **Albert Cieślak**, 3 klasa, II Liceum Ogólnokształcące im. Marii Skłodowskiej-Curie, Końskie, 211 pkt., laureat III miejsca

(37) **Tomasz Kościuszko**, 2 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica, Warszawa, 210 pkt., laureat III miejsca

(38) **Krzysztof Piesiewicz**, 2 klasa, VIII Liceum Ogólnokształcące i 58 Gimnazjum im. Króla Władysława IV, Warszawa, 204 pkt., laureat III miejsca

(39) **Wiktoria Kośny**, 2 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica, Warszawa, 203 pkt., laureatka III miejsca

(40) **Michał Kuźba**, 3 klasa, VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich, Bydgoszcz, 202 pkt., laureat III miejsca

(41) **Mateusz Lewko**, 3 klasa, XIV Liceum Ogólnokształcące im. Polonii Belgijskiej, Wrocław, 200 pkt., laureat III miejsca

(41) **Jan Ludziejewski**, 3 klasa, VIII Liceum Ogólnokształcące i 58 Gimnazjum im. Króla Władysława IV, Warszawa, 200 pkt., laureat III miejsca

(43) **Jakub Jasiulewicz**, 2 klasa, VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich, Bydgoszcz, 183 pkt., finalista z wyróżnieniem

(44) **Bartłomiej Wróblewski**, 3 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica, Warszawa, 180 pkt., finalista z wyróżnieniem

(45) **Bartłomiej Karasek**, 3 klasa, Zespół Szkół nr 1, Grodzisk Mazowiecki, 179 pkt., finalista z wyróżnieniem

(46) **Wojciech Szwarc**, 2 klasa, XIII Liceum Ogólnokształcące, Szczecin, 178 pkt., finalista z wyróżnieniem

(46) **Dominik Traczyk**, 3 klasa, VI Liceum Ogólnokształcące im. Jana Kochanowskiego, Radom, 178 pkt., finalista z wyróżnieniem

(48) **Rafał Łyżwa**, 1 klasa, VI Liceum Ogólnokształcące im. Jana Kochanowskiego, Radom, 175 pkt., finalista z wyróżnieniem

(48) **Jakub Zadrożny**, 2 klasa, Liceum Ogólnokształcące nr III im. Adama Mickiewicza, Wrocław, 175 pkt., finalista z wyróżnieniem

(50) **Michał Kukuła**, 3 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia, 173 pkt., finalista z wyróżnieniem

(51) **Paweł Solecki**, 3 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica, Warszawa, 168 pkt., finalista z wyróżnieniem

(52) **Maciej Biernaczyk**, 3 klasa, Zespół Szkół UMK Gimnazjum i Liceum Akademickie, Toruń, 165 pkt., finalista z wyróżnieniem

(52) **Bruno Pitrus**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków, 165 pkt., finalista z wyróżnieniem

(54) **Mateusz Jurczyński**, 3 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia, 163 pkt., finalista z wyróżnieniem

(55) **Krzysztof Małysa**, 1 klasa, XIII Liceum Ogólnokształcące, Szczecin, 160 pkt., finalista z wyróżnieniem

(55) **Marcin Wawerka**, 3 klasa, VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich, Bydgoszcz, 160 pkt., finalista z wyróżnieniem

(57) **Stanisław Szcześniak**, 3 klasa, Gimnazjum nr 124 im. Polskich Noblistów, Warszawa, 150 pkt., finalista z wyróżnieniem

Lista pozostałych finalistów w kolejności alfabetycznej:

- **Adrian Akerman**, 3 klasa, Uniwersyteckie Katolickie Liceum Ogólnokształcące, Tczew
- **Anna Białokozowicz**, 2 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza, Białystok
- **Rafał Brzeziński**, 3 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia
- **Piotr Bujakowski**, 2 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków
- **Kamil Ćwintal**, 2 klasa, Zespół Szkół Ogólnokształcących im. Edwarda Szylki, Ożarów
- **Albert Dziugieł**, 3 klasa, VI Liceum Ogólnokształcące im. Tadeusza Reytana, Warszawa
- **Wojciech Fica**, 3 klasa, XIV Liceum Ogólnokształcące im. Polonii Belgijskiej, Wrocław
- **Karolina Gabara**, 1 klasa, VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich, Bydgoszcz
- **Szymon Gajda**, 2 klasa, II Liceum Ogólnokształcące im. Heleny Malczewskiej, Zawiercie
- **Gabriela Gierasimiuk**, 3 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza, Białystok
- **Daniel Grzegorzewski**, 3 klasa, I Liceum Ogólnokształcące im. Józefa Ignacego Kraszewskiego, Biała Podlaska
- **Piotr Haryza**, 3 klasa, XIII Liceum Ogólnokształcące, Szczecin
- **Artur Jamro**, 3 klasa, Zespół Szkół Ogólnokształcących nr 1, Nowy Sącz
- **Daniel Kałuża**, 3 klasa, Gimnazjum i Liceum Ogólnokształcące im. Stefana Batorego, Warszawa
- **Rafał Kaszuba**, 2 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków
- **Jan Klinkosz**, 1 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia
- **Adam Klukowski**, 2 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica, Warszawa

- **Michał Kosior**, 3 klasa, III Liceum Ogólnokształcące im. płk. Dionizego Czachowskiego, Radom
- **Adam Kuczaj**, 3 klasa, XIV Liceum Ogólnokształcące im. Polonii Belgijskiej, Wrocław
- **Robert Laskowski**, 1 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica, Warszawa
- **Jarosław Ławnicki**, 3 klasa, I Liceum Ogólnokształcące im. gen. Jana Henryka Dąbrowskiego, Kutno
- **Julia Majkowska**, 2 klasa, XIV Liceum Ogólnokształcące im. Polonii Belgijskiej, Wrocław
- **Paweł Malenda**, 2 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza, Białystok
- **Maciej Maruszczak**, 3 klasa, Gimnazjum nr 16 w Zespole Szkół Ogólnokształcących nr 7, Szczecin
- **Marcin Michorzewski**, 3 klasa, XIII Liceum Ogólnokształcące, Szczecin
- **Michał Niciejewski**, 1 klasa, XIII Liceum Ogólnokształcące, Szczecin
- **Szymon Pajzert**, 3 klasa, Zespół Szkół UMK, Liceum Akademickie, Toruń
- **Filip Plata**, 3 klasa, VI Liceum Ogólnokształcące im. Jana Kochanowskiego, Radom
- **Julian Pszczołowski**, 3 klasa, XIV Liceum Ogólnokształcące im. Polonii Belgijskiej, Wrocław
- **Szymon Rakowski**, 3 klasa, I Liceum Ogólnokształcące im. Adama Mickiewicza, Białystok
- **Łukasz Raszkiewicz**, 3 klasa, III Liceum Ogólnokształcące im. Marynarki Wojennej RP, Gdynia
- **Jakub Romanowski**, 2 klasa, XIV Liceum Ogólnokształcące im. Stanisława Staszica, Warszawa
- **Michał Siennicki**, 3 klasa, Gimnazjum z Oddziałami Dwujęzycznymi nr 42, Warszawa
- **Jan Sierpina**, 3 klasa, Liceum Ogólnokształcące nr III im. Adama Mickiewicza, Wrocław
- **Błażej Sowa**, 3 klasa, XIV Liceum Ogólnokształcące im. Polonii Belgijskiej, Wrocław
- **Maciej Sypetkowski**, 2 klasa, I Liceum Ogólnokształcące im. Władysława Jagiełły, Krasnystaw
- **Mateusz Szpyrka**, 2 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków
- **Agnieszka Świetlik**, 3 klasa, VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich, Bydgoszcz
- **Anna Urbala**, 1 klasa, Zespół Szkół Elektryczno-Elektronicznych, Szczecin
- **Dawid Wegner**, 3 klasa, VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich, Bydgoszcz

- **Bartosz Wodziński**, 3 klasa, V Liceum Ogólnokształcące im. Augusta Witkowskiego, Kraków
- **Szymon Wolarz**, 3 klasa, VIII Liceum Ogólnokształcące im. Adama Mickiewicza, Poznań

Komitet Główny Olimpiady Informatycznej przyznał następujące nagrody rzeczowe:

- puchar wręczono zwycięzcy XXII Olimpiady Przemysławowi Jakubowi Kozłowskiemu,
- złote, srebrne i brązowe medale przyznano odpowiednio laureatom I, II i III miejsca,
- stypendia przyznano wszystkim laureatom według następującego klucza:
  - laureaci I miejsca – 2500 zł,
  - laureaci II miejsca – 700 zł,
  - laureaci III miejsca – 300 zł.

  Stypendia ufundowane zostały przez Ministerstwo Edukacji Narodowej.
- roczną prenumeratę miesięcznika „Delta" przyznano wszystkim laureatom.

Komitet Główny powołał reprezentacje na:

(1) **Międzynarodową Olimpiadę Informatyczną IOI'2015**, która odbyła się w terminie 26 lipca – 2 sierpnia 2015 r. w Kazachstanie w miejscowości Ałmaty

(2) oraz **Olimpiadę Informatyczną Krajów Europy Środkowej CEOI'2015**, która odbyła się w terminie 29 czerwca – 4 lipca 2015 r. w Czechach w miejscowości Brno, w składzie:

- Przemysław Jakub Kozłowski
- Jarosław Kwiecień
- Konrad Paluszek
- Maciej Hołubowicz

rezerwowi:

- Katarzyna Kowalska
- Michał Zawalski

(3) **Bałtycką Olimpiadę Informatyczną BOI'2015**, która odbyła się w terminie 28 kwietnia – 3 maja 2015 w Warszawie–Józefowie. Wzięli w niej udział zawodnicy z klas programowo niższych niż ostatnia klasa szkoły ponadgimnazjalnej, w składzie:

- Jarosław Kwiecień
- Jan Tabaszewski
- Mariusz Trela
- Paweł Burzyński

- Jakub Boguta
- Artur Puzio

poza konkursem:

- Franciszek Budrowski
- Stanisław Strzelecki
- Michał Tepper
- Tomasz Grześkiewicz

Rezerwowymi zawodnikami byli:

- Eryk Kijewski
- Agnieszka Dudek
- Jan Olkowski

Polacy na Międzynarodowej Olimpiadzie Informatycznej spisali się znakomicie, zdobywając 1 złoty i 3 srebrne medale. Najlepszy wynik osiągnął Jarosław Kwiecień, zajmując 20. miejsce i zdobywając złoty medal. Jest to już drugi złoty medal Jarosława. Poprzedni wywalczył w zeszłym roku na Tajwanie i ma szanse zdobyć jeszcze jeden złoty medal w przyszłym roku.

Konrad Paluszek, Maciej Hołubowicz i Przemysław Jakub Kozłowski zdobyli srebrne medale.

Młodzi Polscy informatycy od lat należą do czołówki światowej. W nieoficjalnej klasyfikacji medalowej zajmują czwarte miejsce (zobacz `http://stats.ioinformatics.org/`). Pierwsza piątka klasyfikacji przedstawia się następująco:

| Kraj | Złota | Srebra | Brązy | RAZEM |
|---|---|---|---|---|
| Chiny | 72 | 23 | 12 | 107 |
| Rosja | 52 | 32 | 12 | 96 |
| USA | 42 | 33 | 15 | 90 |
| Polska | 34 | 36 | 27 | 97 |
| Korea Południowa | 33 | 33 | 26 | 92 |

Pod względem liczby wszystkich zdobytych medali Polska zajmuje drugie miejsce, wyprzedzając takie kraje jak USA i Rosja.

Zwycięzcą XXII Środkowoeuropejskiej Olimpiady Informatycznej (ex-aequo z zawodnikiem z Chorwacji) został Przemysław Jakub Kozłowski. Złoty medal zdobył także Jarosław Kwiecień, srebrny Maciej Hołubowicz, a brązowy Konrad Paluszek.

W Bałtyckiej Olimpiadzie Informatycznej Polacy także odnieśli spektakularne sukcesy – Artur Puzio został zwycięzcą tejże Olimpiady, a wszyscy pozostali zawodnicy zdobyli medale:

- Jan Tabaszewski – złoty medal, piąte miejsce w rankingu,
- Paweł Burzyński – srebrny medal, siódme miejsce w rankingu,
- Mariusz Trela – srebrny medal,
- Jarosław Kwiecień – srebrny medal,

- Jakub Boguta – srebrny medal.

Zawodnicy startujący poza konkursem uplasowali się na następujących pozycjach:

- Tomasz Grześkiewicz i Franciszek Budrowski – srebrne medale,
- Stanisław Strzelecki i Michał Tepper – brązowe medale.

Komitet Główny podjął następujące uchwały o udziale młodzieży w obozach:

- w Obozie Naukowo-Treningowym im. A. Kreczmara, który odbył się w Warszawie w terminie 10-18 lipca, wzięli udział reprezentanci na IOI, wraz z zawodnikami rezerwowymi, oraz finaliści Olimpiady, którzy nie uczęszczali w tym roku szkolnym do programowo najwyższej klasy szkoły ponadgimnazjalnej,
- w Obozie Krajów Wyszehradzkich, który odbył się w miejscowości Danisovce na Słowacji w terminie 13-19 czerwca, wzięło udział trzech reprezentantów na IOI oraz jeden zawodnik rezerwowy.

Sekretariat wystawił łącznie 42 zaświadczenia o uzyskaniu tytułu laureata, 15 zaświadczeń o uzyskaniu tytułu wyróżnionego finalisty oraz 42 zaświadczenia o uzyskaniu tytułu finalisty XXII Olimpiady Informatycznej.

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

- Michał Adamczyk (student Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego)
  - Stanisław Szcześniak – finalista z wyróżnieniem
- Jacek Banasik (XIV Liceum Ogólnokształcące im. Stanisława Staszica w Warszawie)
  - Konrad Paluszek – laureat I miejsca
- Iwona Bujnowska (I Liceum Ogólnokształcące im. Adama Mickiewicza w Białymstoku)
  - Przemysław Jakub Kozłowski – laureat I miejsca
  - Maciej Hołubowicz – laureat I miejsca
  - Anna Białokozowicz – finalistka
  - Gabriela Gierasimiuk – finalistka
  - Szymon Rakowski – finalista
- Ireneusz Bujnowski (I Liceum Ogólnokształcące im. Adama Mickiewicza w Białymstoku)
  - Przemysław Jakub Kozłowski – laureat I miejsca
  - Maciej Hołubowicz – laureat I miejsca
  - Franciszek Budrowski – laureat II miejsca
  - Anna Białokozowicz – finalistka
  - Gabriela Gierasimiuk – finalistka
  - Paweł Malenda – finalista
  - Szymon Rakowski – finalista

- Filip Chmielewski (Szczecin)
  - Anna Urbala – finalistka
- Patryk Czajka (student Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego)
  - Konrad Majewski – laureat II miejsca
  - Artur Puzio – laureat II miejsca
  - Robert Laskowski – finalista
  - Jakub Romanowski – finalista
- Jolanta Ćwintal (Zespół Szkół Ogólnokształcących im. Edwarda Szylki w Ożarowie)
  - Kamil Ćwintal – finalista
- Marian Domozych (Zespół Szkół Elektryczno-Elektronicznych w Szczecinie)
  - Anna Urbala – finalistka
- Czesław Drozdowski (XIII Liceum Ogólnokształcące w Szczecinie)
  - Krzysztof Małysa – finalista z wyróżnieniem
  - Wojciech Szwarc – finalista z wyróżnieniem
  - Piotr Haryza – finalista
  - Maciej Maruszczak – finalista
  - Marcin Michorzewski – finalista
  - Michał Niciejewski – finalista
  - Anna Urbala – finalistka
- Bartłomiej Dudek (student Wydziału Matematyki i Informatyki Uniwersytetu Wrocławskiego)
  - Jarosław Kwiecień – laureat I miejsca
  - Agnieszka Dudek – laureatka III miejsca
  - Mateusz Lewko – laureat III miejsca
  - Martyna Siejba – laureatka III miejsca
  - Jakub Zadrożny – finalista z wyróżnieniem
  - Wojciech Fica – finalista
  - Julian Pszczołowski – finalista
- Lech Duraj (Katedra Algorytmiki Uniwersytetu Jagiellońskiego)
  - Kamil Rajtar – laureat III miejsca
  - Adrian Siwiec – laureat III miejsca
  - Bruno Pitrus – finalista z wyróżnieniem
  - Bartosz Wodziński – finalista
- Andrzej Dyrek (V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie)
  - Kamil Rajtar – laureat III miejsca
  - Adrian Siwiec – laureat III miejsca
  - Bruno Pitrus – finalista z wyróżnieniem
  - Piotr Bujakowski – finalista
  - Rafał Kaszuba – finalista
  - Mateusz Szpyrka – finalista

  - Bartosz Wodziński – finalista
- Jarosław Dzikowski (student Wydziału Matematyki i Informatyki Uniwersytetu Wrocławskiego)
  - Mateusz Lewko – laureat III miejsca
  - Martyna Siejba – laureatka III miejsca
  - Wojciech Fica – finalista
- Karol Farbiś (student Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego)
  - Jakub Romanowski – finalista
- Elżbieta Gajda (Gimnazjum w Ogrodzieńcu)
  - Szymon Gajda – finalista
- Marek Gajda (Zespół Szkół im. Stanisława Staszica w Zawierciu)
  - Szymon Gajda – finalista
- Grzegorz Herman (Katedra Algorytmiki Uniwersytetu Jagiellońskiego)
  - Piotr Bujakowski – finalista
  - Rafał Kaszuba – finalista
  - Mateusz Szpyrka – finalista
- Andrzej Jackowski (Społeczne Gimnazjum nr 2 STO w Białymstoku)
  - Franciszek Budrowski – laureat II miejsca
- Wiktor Janas (Google Zurich)
  - Mateusz Lewko – laureat III miejsca
  - Martyna Siejba – laureatka III miejsca
  - Wojciech Fica – finalista
  - Adam Kuczaj – finalista
  - Julian Pszczołowski – finalista
- Sebastian Jaszczur (student Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego)
  - Michał Tepper – laureat III miejsca
  - Dawid Wegner – finalista
- Adam Kaczyński (Liceum Ogólnokształcące nr III im. Adama Mickiewicza we Wrocławiu)
  - Jan Sierpina – finalista
- Szymon Kanonowicz (VIII Liceum Ogólnokształcące im. Króla Władysława IV w Warszawie)
  - Krzysztof Piesiewicz – laureat III miejsca
- Henryk Kawka (Gimnazjum nr 24 w Lublinie)
  - Jakub Boguta – laureat II miejsca
- Krzysztof Kiewicz (student Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego)
  - Krzysztof Piesiewicz – laureat III miejsca

- Konrad Kijewski (SOFTAX S.J. w Warszawie)
  - Eryk Kijewski – laureat III miejsca
- Bartosz Kostka (student Wydziału Matematyki i Informatyki Uniwersytetu Wrocławskiego)
  - Jarosław Kwiecień – laureat I miejsca
  - Agnieszka Dudek – laureatka III miejsca
  - Jakub Zadrożny – finalista z wyróżnieniem
  - Julia Majkowska – finalistka
- Paweł Kura (student Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego)
  - Daniel Grzegorzewski – finalista
- Jan Kwaśniak (student Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego)
  - Albert Cieślak – laureat III miejsca
- Anna Beata Kwiatkowska (Zespół Szkół Uniwersytetu Mikołaja Kopernika, Gimnazjum i Liceum Akademickie w Toruniu)
  - Filip Tokarski – laureat III miejsca
  - Maciej Biernaczyk – finalista z wyróżnieniem
  - Szymon Pajzert – finalista
- Krzysztof Loryś (Wydział Matematyki i Informatyki Uniwersytetu Wrocławskiego)
  - Julia Majkowska – finalistka
- Sławomir Majkowski (Zespół Szkół Elektryczno-Elektronicznych w Szczecinie)
  - Anna Urbala – finalistka
- Michał Malarski (Gimnazjum i Liceum Ogólnokształcące im. Stefana Batorego w Warszawie)
  - Daniel Kałuża – finalista
- Dawid Matla (XIV Liceum Ogólnokształcące im. Polonii Belgijskiej we Wrocławiu)
  - Agnieszka Dudek – laureatka III miejsca
  - Mateusz Lewko – laureat III miejsca
  - Wojciech Fica – finalista
  - Julia Majkowska – finalistka
- Mirosław Mortka (VI Liceum Ogólnokształcące im. Jana Kochanowskiego w Radomiu)
  - Adrian Naruszko – laureat III miejsca
  - Mateusz Radecki – laureat III miejsca
  - Dominik Traczyk – finalista z wyróżnieniem
  - Rafał Łyżwa – finalista z wyróżnieniem
  - Filip Plata – finalista
- Rafał Nowak (Wydział Matematyki i Informatyki Uniwersytetu Wrocławskiego)
  - Agnieszka Dudek – laureatka III miejsca

  - Jakub Zadrożny – finalista z wyróżnieniem
  - Julian Pszczołowski – finalista
- Małgorzata Piekarska (VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich w Bydgoszczy)
  - Michał Kuźba – laureat III miejsca
  - Michał Tepper – laureat III miejsca
  - Jakub Jasiulewicz – finalista z wyróżnieniem
  - Marcin Wawerka – finalista z wyróżnieniem
  - Agnieszka Świetlik – finalistka
  - Dawid Wegner – finalista
- Mirosław Pietrzycki (I Liceum Ogólnokształcące im. Stanisława Staszica w Lublinie)
  - Tadeusz Dudkiewicz – laureat III miejsca
- Karol Pokorski (Google Zurich)
  - Jarosław Kwiecień – laureat I miejsca
  - Mateusz Lewko – laureat III miejsca
  - Martyna Siejba – laureatka III miejsca
  - Wojciech Fica – finalista
  - Adam Kuczaj – finalista
  - Julia Majkowska – finalistka
  - Julian Pszczołowski – finalista
- Adam Polak (Katedra Algorytmiki Uniwersytetu Jagiellońskiego)
  - Kamil Rajtar – laureat III miejsca
  - Adrian Siwiec – laureat III miejsca
  - Bruno Pitrus – finalista z wyróżnieniem
  - Bartosz Wodziński – finalista
- Wojciech Roszczyński (VIII Liceum Ogólnokształcące im. Adama Mickiewicza w Poznaniu)
  - Szymon Wolarz – finalista
- Agnieszka Samulska (VIII Liceum Ogólnokształcące im. Króla Władysława IV w Warszawie)
  - Krzysztof Piesiewicz – laureat III miejsca
- Piotr Smok (IX Liceum Ogólnokształcące im. Cypriana Kamila Norwida w Częstochowie)
  - Angelika Serwa – laureatka III miejsca
- Marek Sommer (student Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego)
  - Konrad Paluszek – laureat I miejsca
  - Katarzyna Kowalska – laureatka II miejsca
  - Konrad Majewski – laureat II miejsca
  - Jan Tabaszewski – laureat II miejsca
  - Michał Zawalski – laureat II miejsca
  - Tomasz Grześkiewicz – laureat III miejsca

  - Tomasz Kościuszko – laureat III miejsca
  - Marek Zbysiński – laureat III miejsca
  - Stanisław Szcześniak – finalista z wyróżnieniem
  - Jakub Romanowski – finalista
- Krzysztof Stanisławek (student Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego)
  - Daniel Grzegorzewski – finalista
- Szymon Stankiewicz (student Katedry Algorytmiki Uniwersytetu Jagiellońskiego)
  - Adrian Naruszko – laureat III miejsca
  - Mateusz Radecki – laureat III miejsca
  - Dominik Traczyk – finalista z wyróżnieniem
  - Rafał Łyżwa – finalista z wyróżnieniem
- Tomasz Syposz (student Wydziału Matematyki i Informatyki Uniwersytetu Wrocławskiego)
  - Jarosław Kwiecień – laureat I miejsca
- Ryszard Szubartowski (III Liceum Ogólnokształcące im. Marynarki Wojennej RP w Gdyni, Stowarzyszenie Talent)
  - Paweł Burzyński – laureat II miejsca
  - Tomasz Garbus – laureat II miejsca
  - Bartosz Narkiewicz – laureat III miejsca
  - Marek Żochowski – laureat III miejsca
  - Mateusz Jurczyński – finalista z wyróżnieniem
  - Michał Kukuła – finalista z wyróżnieniem
  - Rafał Brzeziński – finalista
  - Jan Klinkosz – finalista
  - Łukasz Raszkiewicz – finalista
- Michał Śliwiński (Liceum Ogólnokształcące nr III im. Adama Mickiewicza we Wrocławiu)
  - Jakub Zadrożny – finalista z wyróżnieniem
- Joanna Śmigielska (XIV Liceum Ogólnokształcące im. Stanisława Staszica w Warszawie)
  - Katarzyna Kowalska – laureatka II miejsca
  - Konrad Majewski – laureat II miejsca
  - Jan Tabaszewski – laureat II miejsca
  - Michał Zawalski – laureat II miejsca
  - Tomasz Grześkiewicz – laureat III miejsca
  - Tomasz Kościuszko – laureat III miejsca
  - Wiktoria Kośny – laureatka III miejsca
  - Jan Olkowski – laureat III miejsca
  - Stanisław Strzelecki – laureat III miejsca
  - Marek Zbysiński – laureat III miejsca
  - Paweł Solecki – finalista z wyróżnieniem

  - Stanisław Szcześniak – finalista z wyróżnieniem
  - Bartłomiej Wróblewski – finalista z wyróżnieniem
  - Albert Dziugieł – finalista
  - Adam Klukowski – finalista
  - Robert Laskowski – finalista
  - Jakub Romanowski – finalista
- Agnieszka Tarnówka-Stec (V Liceum Ogólnokształcące im. Augusta Witkowskiego w Krakowie)
  - Bartosz Wodziński – finalista
- Jacek Tomasiewicz (Codility Polska)
  - Przemysław Jakub Kozłowski – laureat I miejsca
- Michał Wenderlich (VI Liceum Ogólnokształcące im. Jana i Jędrzeja Śniadeckich w Bydgoszczy)
  - Michał Kuźba – laureat III miejsca
- Paweł Żyliński (Instytut Informatyki Uniwersytetu Gdańskiego)
  - Adrian Akerman – finalista

Podobnie jak w ubiegłych latach w przygotowaniu jest publikacja zawierająca pełną informację o XXII 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 2015 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 PRZEPISY SZCZEGÓŁOWE

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

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

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

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

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

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

### §6 TRYB ODWOŁAWCZY

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

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

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

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

### §7 REJESTRACJA PRZEBIEGU ZAWODÓW

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

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

## Rozdział III – Uprawnienia i nagrody

### §8 UPRAWNIENIA I NAGRODY

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

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

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

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

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

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

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

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

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

# Rozdział IV – Olimpiada międzynarodowa

## §9 UDZIAŁ W OLIMPIADZIE MIĘDZYNARODOWEJ

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

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

# Rozdział V – Postanowienia końcowe

## §10 POSTANOWIENIA KOŃCOWE

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

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

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

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

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

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

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

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

## §11 FINANSOWANIE OLIMPIADY

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

# Zasady organizacji zawodów XXII Olimpiady Informatycznej w roku szkolnym 2014/2015

## §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** – 6 października – 3 listopada 2014 roku
  ogłoszenie wyników w witrynie Olimpiady – 21 listopada 2014 roku godz. 20.00
- **zawody II stopnia** – 10–12 lutego 2015 roku
  ogłoszenie wyników w witrynie Olimpiady – 20 lutego 2015 roku godz. 20.00
- **zawody III stopnia** – 14–17 kwietnia 2015 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 3 listopada 2014 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 3 listopada 2014 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 14 listopada 2014 roku poprzez SIO każdy zawodnik będzie mógł zapoznać się ze wstępną oceną swojej pracy.

(12) Do środy 19 listopada 2014 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 19 listopada 2014 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 XXVII Międzynarodową Olimpiadę Informatyczną w 2015 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 XVI Obozie Naukowo-Treningowym im. Antoniego Kreczmara, który odbędzie się w czasie wakacji 2015 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 XXII Olimpiady Informatycznej

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

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

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

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

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

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

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

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

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

(10) Podczas każdej sesji:

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

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

3. W SIO umieszczane będą publiczne odpowiedzi na pytania zawodników. Odpowiedzi te mogą zawierać ważne informacje dotyczące toczących się zawodów, więc wszyscy uczestnicy zawodów proszeni są o regularne zapoznawanie się z ukazującymi się odpowiedziami.
4. Jakikolwiek inny sposób komunikowania się z członkami Jury co do treści i sposobów rozwiązywania zadań jest niedopuszczalny.
5. Komunikowanie się z innymi uczestnikami Olimpiady (np. ustnie, telefonicznie lub poprzez sieć) w czasie przeznaczonym na rozwiązywanie zadań jest zabronione pod rygorem dyskwalifikacji.
6. Każdy zawodnik ma prawo drukować wyniki swojej pracy w sposób opisany w Ustaleniach technicznych.
7. Każdy zawodnik powinien umieścić ostateczne rozwiązania zadań w SIO, za pomocą przeglądarki lub za pomocą skryptu do wysyłania rozwiązań `submit`. Skrypt `submit` działa także w przypadku awarii sieci, wówczas rozwiązanie zostaje automatycznie dostarczone do SIO, gdy komputer odzyska łączność z siecią. Tylko zgłoszone w podany sposób rozwiązania zostaną ocenione.
8. Po zgłoszeniu rozwiązania każdego z zadań SIO dokona wstępnego sprawdzenia i udostępni jego wyniki zawodnikowi. Wstępne sprawdzenie polega na uruchomieniu programu zawodnika na testach przykładowych (wyniki sprawdzenia tych testów nie liczą się do końcowej klasyfikacji). Te same testy przykładowe są używane do wstępnego sprawdzenia za pomocą skryptu do weryfikacji rozwiązań na komputerze zawodnika (skryptu „ocen").
9. Podczas zawodów III stopnia, w przypadku zadań wskazanych przez Komitet Główny, zawodnicy będą mogli poznać wynik punktowy swoich pięciu wybranych zgłoszeń. Przez ostatnie 30 minut zawodów ta opcja nie będzie dostępna.
10. Rozwiązanie każdego zadania można zgłosić co najwyżej 10 razy. Spośród tych zgłoszeń oceniane jest jedynie najpóźniejsze poprawnie kompilujące się rozwiązanie.

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

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

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

(14) Od czwartku 12 lutego 2015 roku od godz. 20.00 do poniedziałku 16 lutego 2015 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

**Wojciech Rytter**
Treść zadania, Opracowanie

**Bartosz Kostka**
Program

**Dostępna pamięć: 128 MB.** *OI, etap I, 6.10–3.11.2014*

# Czarnoksiężnicy okrągłego stołu

*Czarnoksiężnicy okrągłego stołu kolejny raz spotykają się na tajnej naradzie i kolejny raz nie mogą się zgodzić ze sobą, w jakiej kolejności powinni usiąść przy stole. W naradzie uczestniczy $n$ czarnoksiężników. Każdy z nich jest jednoznacznie identyfikowany przez wysokość swojego spiczastego kapelusza. Wysokości kapeluszy są różnymi liczbami całkowitymi z zakresu od 1 do $n$ (im wyższy kapelusz, tym większy staż czarnoksiężnika). Żeby nie zakłócić estetyki przy stole, wysokości kapeluszy czarnoksiężników siedzących obok siebie nie mogą się różnić o więcej niż $p$.*

*Ponadto nie wszyscy czarnoksiężnicy przepadają za sobą – jeśli czarnoksiężnik $a$ nie lubi czarnoksiężnika $b$, to czarnoksiężnik $b$ nie może siedzieć bezpośrednio po prawej stronie czarnoksiężnika $a$. Zakładamy, że przewodniczący narady (mający kapelusz o wysokości $n$) wybrał już swoje miejsce przy okrągłym stole. Na ile sposobów pozostali czarnoksiężnicy mogą do niego dołączyć?*

## Wejście

*Pierwszy wiersz standardowego wejścia zawiera trzy liczby całkowite $n$, $k$ i $p$ ($1 \leqslant n \leqslant 1\,000\,000$, $0 \leqslant k \leqslant 100\,000$, $0 \leqslant p \leqslant 3$) pooddzielane pojedynczymi odstępami, oznaczające liczbę czarnoksiężników, liczbę informacji o ich niechęciach względem innych oraz maksymalną różnicę wysokości kapeluszy.*

*Kolejne $k$ wierszy zawiera uporządkowane pary: $i$-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 czarnoksiężnik w kapeluszu wysokości $a_i$ nie lubi czarnoksiężnika w kapeluszu wysokości $b_i$. Każda uporządkowana para czarnoksiężników może się pojawić na wejściu co najwyżej raz.*

*W testach wartych 16% punktów zachodzi $n \leqslant 5$. W innych testach wartych 16% punktów zachodzi $p \leqslant 2$.*

## Wyjście

*Jedyny wiersz standardowego wyjścia powinien zawierać liczbę całkowitą, będącą resztą z dzielenia przez $10^9 + 7$ liczby możliwości usadzenia czarnoksiężników.*

## Przykład

| *Dla danych wejściowych:* | *poprawnym wynikiem jest:* |
|---|---|
| 5 2 3<br>1 3<br>5 4 | 6 |

**Wyjaśnienie do przykładu:** *Czarnoksiężnicy mogą usiąść przy okrągłym stole na jeden z sześciu sposobów: 53124, 53142, 52143, 53412, 52314, 53214.*

**Testy „ocen”:**

`1ocen`: *mały test, w którym jeden z czarnoksiężników nie lubi nikogo;*

`2ocen`: $n = 5$, $k = 0$, $p = 3$;

`3ocen`: $n = 1\ 000\ 000$, $k = 0$, $p = 2$.

# Rozwiązanie

Zadanie sprowadza się do policzenia cykli Hamiltona w skierowanym grafie $n$-wierzchołkowym, w którym krawędź między każdą parą wierzchołków $u, v$ spełnia $|u - v| \leqslant p$ oraz niektóre krawędzie są zabronione (opowiadają konfliktom czarnoksiężników). Zajmiemy się jedynie przypadkiem $p = 3$; przypadki $0 \leqslant p \leqslant 2$ są dosyć proste.

Dla wygody jako nazw wierzchołków będziemy używać liczb będących odległością od $n$ (czyli $0 := n$, $1 := n - 1$, ...). Tak więc liczbie $n$ (przewodniczącemu narady) odpowiada teraz wierzchołek 0. Przyjmijmy, że przewodniczący narady siedzi na miejscu 0.

Oznaczmy przez $X$ zbiór uporządkowanych par $(i, j)$, gdzie $i, j \in [0, n-1]$, reprezentujących konflikty (inaczej mówiąc, w naszych permutacjach nie może wystąpić para ze zbioru $X$). Trzeba jasno powiedzieć, że są to pary uporządkowane; jeśli np. $(9, 5) \in X$ oraz $(5, 9) \notin X$, to 9 nie może siedzieć obok 5 bezpośrednio z lewej strony, ale może siedzieć z prawej.

Tak więc rozważamy permutacje $\pi = (\pi_0, \pi_1, \ldots, \pi_{n-1})$ zbioru $\{0, \ldots, n-1\}$, w których zawsze $\pi_0 = 0$, $\pi_{n-1} \in \{1, 2, 3\}$, każde dwa kolejne elementy różnią się co najwyżej o 3 oraz żadna para ze zbioru $X$ nie występuje jako dwa kolejne elementy w $\pi$ (wliczając $\pi_{n-1}, \pi_0$). Oznaczmy liczbę tych permutacji przez $ILE(n, X)$. Naszym zadaniem jest wyznaczenie tej liczby.

Niech $ILE_k(n, X)$ będzie liczbą tych permutacji spełniających powyższe warunki, od których dodatkowo żądamy, żeby $\pi_{n-1} = k$, gdzie $k \in \{1, 2, 3\}$. Oczywiście końcowy wynik to

$$ILE(n, X) \;=\; ILE_1(n, X) + ILE_2(n, X) + ILE_3(n, X).$$

## Permutacje typu $(0 \diamond \diamond \ldots \diamond 1)$ i obliczanie pomocniczych zmiennych $x_i$, $y_i$

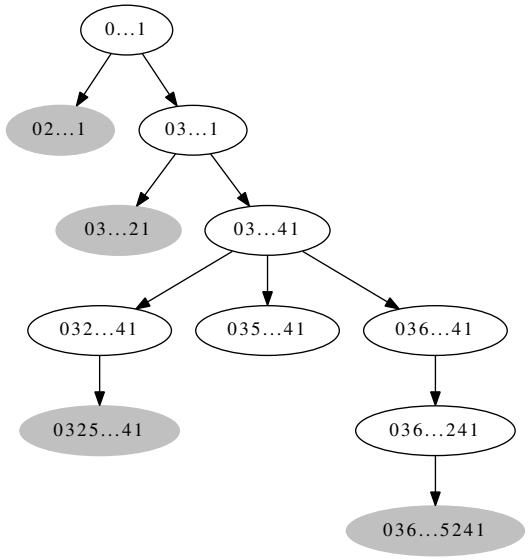
Definiujemy $x_i$, $y_i$ jako liczby bezkonfliktowych permutacji liczb z przedziału $[i, n-1]$, w których pierwszy i ostatni wyraz należą do zbioru $\{i, i+1\}$. W przypadku $x_i$ żądamy, aby pierwszym wyrazem było $i$, a w przypadku $y_i$, aby pierwszym wyrazem było $i+1$. Zauważmy, że $x_0 = ILE_1(n, X)$.

**Obserwacja 1.** Pozornie wydaje się, że zawsze $x_i = y_i$ (symetryczna definicja), ale nie zawsze tak jest, bo konflikty (zbiór $X$) nie muszą być symetryczne.

Naszym celem jest wyznaczyć wzór rekurencyjny na $x_i$ oraz $y_i$. Zrobimy to na początek dla $i = 0$, przy założeniu, że $n \geqslant 8$. W tym celu przeanalizujemy możliwe prefiksy/sufiksy permutacji dające w sumie pewną liczbę kolejnych elementów (poczynając od 0). Symbol $\diamond$ oznacza *cokolwiek* (liczbę jeszcze nie wybraną). Jeśli pierwszym elementem jest 0 i ostatnim jest 1, to wszystkie możliwe sytuacje to:

$$0\,2\,\diamond\ldots\diamond\,1 \qquad 0\,3\,\diamond\ldots\diamond\,2\,1 \qquad 0\,3\,2\,5\,\diamond\ldots\diamond\,4\,1 \qquad 0\,3\,6\,\diamond\ldots\diamond\,5\,2\,4\,1.$$

Aby się o tym przekonać, możemy narysować drzewo, w którego każdym węźle na wszystkie możliwe sposoby próbujemy wydłużyć prefiks albo sufiks permutacji tak, by nie zaburzyć warunku, że każde dwa kolejne elementy różnią się co najwyżej o 3. Rozwijanie przerywamy, jeśli prefiks i sufiks zawierają w sumie $k$ pierwszych elementów, jeden z nich kończy się liczbą $k-1$, a drugi $k-2$.




Jeśli pierwszym elementem jest 1 i ostatnim jest 0, to mamy sytuacje symetryczne:

$$1\,\diamond\ldots\diamond\,2\,0 \qquad 1\,2\,\diamond\ldots\diamond\,3\,0 \qquad 1\,4\,\diamond\ldots\diamond\,5\,2\,3\,0 \qquad 1\,4\,2\,5\,\diamond\ldots\diamond\,6\,3\,0.$$

Przyjmijmy na razie $X = \emptyset$. Powyższe sytuacje odnoszą się do $[0, n-1]$. Przeskalujmy je teraz do $[i, n-1]$ (zastępujemy zero przez $i$). Ogólnie dla $n - i \geqslant 8$ otrzymujemy:

$$x_i \;=\; y_{i+1} + y_{i+2} + y_{i+4} + y_{i+5}, \quad y_i \;=\; x_{i+1} + x_{i+2} + x_{i+4} + x_{i+5}. \tag{1}$$

Jeśli $X \neq \emptyset$, to niektóre składniki znikną. Na przykład jeśli $i = 0$ oraz $X = \{\,(0,2),\,(2,5)\,\}$, to

$$x_i \;=\; y_{i+2} + y_{i+4}, \qquad y_i \;=\; x_{i+1} + x_{i+2} + x_{i+4}.$$

Podsumowując, mamy:

$ILE_1(n, X) \;=\; x_0 \;=\; y_1 + y_2 + y_4 + y_5$ (bez składników konfliktowych ze zbioru $X$).

## Permutacje typu $(0 \diamond \diamond \ldots \diamond 2)$ i $(0 \diamond \diamond \ldots \diamond 3)$

Przeanalizujmy teraz permutacje kończące się na 2. Podobnie jak poprzednio, możemy rozrysować drzewo możliwych konfiguracji typu prefiks/sufiks, które szybko się kończy. Ostatecznie dowiadujemy się, że permutacje takie mają dla $n \geqslant 8$ jedną z postaci:

$$0\,1 \diamond \ldots \diamond 2 \qquad 0\,3 \diamond \ldots \diamond 4\,1\,2 \qquad 0\,3\,1\,4 \diamond \ldots \diamond 5\,2.$$

Tak więc

$$ILE_2(n, X) \;=\; x_1 + x_3 + x_4 \;\text{ (ewentualnie bez składników konfliktowych).}$$

Podobnie rozumujemy dla permutacji kończących się na 3. W tym przypadku drzewo możliwości jest nieco większe. Ostatecznie okazuje się, że możliwe są tylko następujące konfiguracje:

$$0\,1\,2 \diamond \ldots \diamond 3 \quad 0\,1\,4 \diamond \ldots \diamond 5\,2\,3 \quad 0\,1\,4\,2\,5 \diamond \ldots \diamond 6\,3 \quad 0\,2\,1\,4 \diamond \ldots \diamond 3 \quad 0\,2\,5 \diamond \ldots \diamond 4\,1\,3.$$

Tak więc

$$ILE_3(n, X) \;=\; x_2 + x_4 + x_5 + y_3 + y_4 \;\text{ (ewentualnie bez składników konfliktowych).}$$

## Rozwiązanie wzorcowe

Niech $\langle\alpha\rangle$ będzie funkcją zero-jedynkową, która sprawdza, czy permutacja $\alpha$ zawiera zabronioną parę ze zbioru $X$: jeśli zawiera, to wynikiem jest zero.

Poniższy pseudokod pokazuje, jak ostatecznie liczymy wynik końcowy:

```
1:  Algorytm ILE(n, X)
2:    for i := n downto n − 7 do oblicz x_i, y_i brutalnie;
3:    for i := n − 8 downto 0 do begin
4:      { oblicz x_i, korzystając z równania (1), dbając o to, by pominąć }
5:      { składniki wynikające z zabronionych par (ze zbioru X) }
6:      x_i := y_{i+1} · ⟨i+1, i, i+2⟩ + y_{i+2} · ⟨i+2, i+1, i, i+3⟩+
7:           y_{i+4} · ⟨i+4, i+1, i, i+3, i+2, i+5⟩ + y_{i+5} · ⟨i+5, i+2, i+4, i+1, i, i+3, i+6⟩;
8:      analogicznie oblicz y_i;
9:    end
10:
11:   { wynik = ILE_1(n, X) + ILE_2(n, X) + ILE_3(n, X), gdzie ILE_1(n, X) = x_0 }
12:   wynik := x_0 + x_1 · ⟨201⟩ + x_3 · ⟨41203⟩ + x_4 · ⟨520314⟩ + x_2 · ⟨3012⟩ +
13:        x_4 · ⟨523014⟩ + x_5 · ⟨6301425⟩ + y_3 · ⟨30214⟩ + y_4 · ⟨413025⟩;
14:   return wynik;
15: end
```

Algorytm wykonuje liniową liczbę operacji arytmetycznych. Sprawdzanie konfliktów (implementacja funkcji $\langle\alpha\rangle$) działa także w czasie liniowym, gdyż każdy wierzchołek występuje w stałej liczbie (sensownych) par ze zbioru $X$.

**Tomasz Idziaszek**
Treść zadania, Opracowanie

**Karol Pokorski**
Program

**Dostępna pamięć: 128 MB.**

*OI, etap I, 6.10–3.11.2014*

# Kinoman

*Bajtazar jest zapalonym kinomanem, dlatego ucieszył się, gdy jego ulubione kino studyjne przygotowało bardzo ciekawą promocję na lato. Każdego z n dni lata w kinie będzie wyświetlany jeden z m filmów. Promocyjny karnet uprawnia do bezpłatnego wejścia na dowolną liczbę seansów, pod warunkiem że jego właściciel nie będzie robił przerw (tzn. ominięcie seansu unieważnia karnet; pierwszy seans można wybrać dowolnie).*

*Na podstawie internetowych recenzji Bajtazar przyporządkował każdemu z m filmów jego współczynnik fajności. Bajtazar chciałby wykorzystać promocyjny karnet w taki sposób, aby zmaksymalizować sumę współczynników fajności obejrzanych filmów. Niestety nie jest to takie proste, gdyż Bajtazar okropnie nie lubi oglądać dwa razy tego samego filmu. Powtórny seans nuży go i odbiera przyjemne wspomnienia, które wiązał z filmem. Zatem chce on tak naprawdę zmaksymalizować sumę współczynników fajności tych filmów, które obejrzy dokładnie raz.*

## Wejście

*W pierwszym wierszu standardowego wejścia znajdują się dwie liczby całkowite $n$ i $m$ ($1 \leqslant m \leqslant n \leqslant 1\,000\,000$) oddzielone pojedynczym odstępem, oznaczające liczbę dni promocji oraz liczbę filmów. Dla ułatwienia filmy numerujemy liczbami od 1 do $m$.*

*W drugim wierszu znajduje się ciąg $n$ liczb całkowitych $f_1, f_2, \ldots, f_n$ ($1 \leqslant f_i \leqslant m$) pooddzielanych pojedynczymi odstępami: liczba $f_i$ oznacza numer filmu wyświetlanego $i$-tego dnia promocji. W trzecim wierszu znajduje się ciąg $m$ liczb całkowitych $w_1, w_2, \ldots, w_m$ ($1 \leqslant w_j \leqslant 1\,000\,000$) pooddzielanych pojedynczymi odstępami: liczba $w_j$ oznacza współczynnik fajności filmu o numerze $j$. Może się tak zdarzyć, że pewne spośród podanych $m$ filmów nie będą w ogóle wyświetlane w trakcie trwania letniej promocji.*

*W testach wartych 70% punktów zachodzi dodatkowy warunek $n \leqslant 100\,000$, a w podzbiorze tych testów wartym 20% punktów zachodzi warunek $n \leqslant 8000$.*

## Wyjście

*W jedynym wierszu standardowego wyjścia należy wypisać jedną liczbę całkowitą oznaczającą sumaryczną fajność filmów, które Bajtazar obejrzy dokładnie raz, jeśli optymalnie wykorzysta promocyjny karnet.*

## Przykład

| *Dla danych wejściowych:* | *poprawnym wynikiem jest:* |
|---|---|
| `9 4`<br>`2 3 1 1 4 1 2 4 1`<br>`5 3 6 6` | `15` |

**Wyjaśnienie do przykładu:** *Bajtazar może wykorzystać karnet, aby obejrzeć 6 seansów, poczynając od drugiego dnia. W ten sposób obejrzy dokładnie raz filmy o numerach 2, 3 i 4.*

**Testy „ocen”:**

`1ocen`: *$n = 10$, $m = 5$, losowy;*

`2ocen`: *$n = 100$, $m = 50$, losowy;*

`3ocen`: *$n = 1\,000\,000$, $m = 1\,000\,000$, wszystkie filmy poza jednym mają fajność 200 000 i nie powtarzają się; jeden ma fajność 1 000 000 i powtarza się raz na 10 dni.*

# Rozwiązanie

W opracowaniu założymy dla uproszczenia, że współczynnik fajności każdego filmu jest równy jego numerowi. W ten sposób, odzierając zadanie z warstwy fabularnej, możemy je wysłowić następująco: dany jest ciąg $n$ liczb $a_1, a_2, \ldots, a_n$; należy znaleźć taki spójny fragment w tym ciągu, że suma liczb występujących dokładnie raz w tym fragmencie jest możliwie jak największa.

Dla przykładu rozważmy ciąg $a = 5, 6, 2, 5, 2, 8, 5, 5, 4, 4, 2$ i zastanówmy się, jak wyglądają sumy fragmentów, które zaczynają się pierwszym wyrazem ciągu (czyli sumy prefiksów tego ciągu). Niech $s_i$ oznacza sumę prefiksu kończącego się wyrazem na pozycji $i$. Mamy wtedy:

| $a_i$ | 5 | 6 | 2 | 5 | 2 | 8 | 5 | 5 | 4 | 4 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $s_i$ | 5 | 11 | 13 | 8 | 6 | 14 | 14 | 14 | 18 | 14 | 14 |

Przykładowo prefiks $5, 6, 2, 5, 2, 8, 5$ zawiera dokładnie jedną szóstkę i ósemkę (a pozostałe liczby występują w nim wielokrotnie), więc jego suma wynosi $s_7 = 6 + 8 = 14$. Jak wyznaczyć ciąg $s$? Rozważmy dowolną wartość $x$ występującą w ciągu $a$ i niech $\{j_1, j_2, \ldots, j_k\}$ będzie zbiorem pozycji, na których ta wartość się znajduje (czyli $x = a_{j_1} = a_{j_2} = \ldots = a_{j_k}$). Wartość $x$ będzie liczyła się do sumy tych prefiksów, w których występuje dokładnie raz; będą to więc prefiksy kończące się na pozycjach od $j_1$ do $j_2 - 1$. Jeśli zatem zdefiniujemy ciąg $p$ w taki sposób, że dla wartości $x$ ustalimy $p_{j_1} = +x$, $p_{j_2} = -x$ oraz $p_{j_3} = \ldots = p_{j_k} = 0$, to nie będzie zaskoczeniem, że ciąg $s$ uzyskamy, licząc sumy prefiksowe ciągu $p$ (czyli ze wzoru $s_i = s_{i-1} + p_i$):

| $a_i$ | 5 | 6 | 2 | 5 | 2 | 8 | 5 | 5 | 4 | 4 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $p_i$ | +5 | +6 | +2 | −5 | −2 | +8 | 0 | 0 | +4 | −4 | 0 |
| $s_i$ | 5 | 11 | 13 | 8 | 6 | 14 | 14 | 14 | 18 | 14 | 14 |

Przyjrzyjmy się teraz, jak zmienią się ciągi $p$ i $s$, jeśli usuniemy z ciągu $a$ pierwszy wyraz (będzie to odpowiadało rozważaniu tych fragmentów, które zaczynają się drugim wyrazem ciągu $a$). Zauważmy, że w ciągu $p$ będziemy musieli uaktualnić jedynie trzy wyrazy odpowiadające wartości pierwszego wyrazu ciągu $a$:

| $a_i$ | – | 6 | 2 | 5 | 2 | 8 | 5 | 5 | 4 | 4 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $p_i$ | **0** | +6 | +2 | **+5** | −2 | +8 | **−5** | 0 | +4 | −4 | 0 |
| $s_i$ | 0 | 6 | 8 | 13 | 11 | 19 | 14 | 14 | 18 | 14 | 14 |

W analogiczny sposób możemy usuwać kolejne wyrazy ciągu $a$. W ogólności, jeśli usuwamy wyraz o wartości $x$ z pozycji $j_i$, to musimy uaktualnić wyrazy ciągu $p$ na pozycjach $j_i$, $j_{i+1}$ oraz $j_{i+2}$ (o ile istnieją). W ten sposób rozważymy sumy wszystkich spójnych fragmentów ciągu $a$. Zauważmy, że odpowiedzią do zadania jest największa wartość, która kiedykolwiek pojawiła się w ciągu $s$.

W ten sposób można otrzymać rozwiązanie o złożoności $O(n^2)$, zaimplementowane w pliku `kins2.cpp`.

## Rozwiązanie wzorcowe

W celu efektywnej implementacji powyższego rozwiązania, możemy wykorzystać strukturę danych, która umożliwi nam szybkie wykonywanie następujących operacji na ciągu liczb $p_1, p_2, \ldots, p_n$:

- zmiana wartości jednego wyrazu ciągu $p$,
- wyznaczenie największej wartości w ciągu sum prefiksowych ciągu $p$.

Zauważmy, że nie potrzebujemy w całości konstruować ciągu $s$ – wystarczy nam jedynie znajomość największego wyrazu tego ciągu. Powyższą strukturę danych można zaimplementować jako statyczne drzewo przedziałowe, w którym obie operacje będą realizowane w czasie $O(\log n)$. Taka struktura danych pojawiła się już na XVI Olimpiadzie Informatycznej w zadaniu *Łyżwy* [16] (patrz także opis w książce [42]).

Na strukturze danych co najwyżej $3n$ razy wykonamy operację zmiany wartości wyrazu i $n$ razy wykonamy operację wyznaczenia maksimum sum prefiksowych. Zatem cały algorytm będzie miał złożoność czasową $O(n \log n)$. Potrzebujemy jeszcze wyznaczyć indeksy $j_1, j_2, \ldots, j_k$ dla wszystkich wartości $x$ z ciągu $a$, ale można to zrobić na początku programu w sumarycznym czasie $O(n)$. Zainteresowanych Czytelników odsyłamy do rozwiązania wzorcowego zawartego w pliku `kin.cpp`.

**Wojciech Rytter**
**Treść zadania, Opracowanie**

**Jan Kanty Milczek**
**Program**

**Dostępna pamięć: 64 MB.**

*OI, etap I, 6.10–3.11.2014*

# Kwadraty

*W tym zadaniu rozważamy rozkłady dodatnich liczb całkowitych na sumy* **różnych** *kwadratów dodatnich liczb całkowitych (dalej będziemy je nazywać w skrócie rozkładami). Na przykład, liczba 30 ma dwa rozkłady:* $1^2 + 2^2 + 5^2 = 1^2 + 2^2 + 3^2 + 4^2 = 30$*, natomiast dla liczby 8 nie istnieje żaden rozkład.*

*Interesuje nas odpowiedź na pytanie, jak duża musi być największa liczba w rozkładzie danej liczby* $n$*. Innymi słowy, chcemy wyznaczyć wartość* $k(n)$ *będącą minimum z największych liczb występujących we wszystkich rozkładach liczby* $n$*. Dla uproszczenia przyjmijmy, że jeśli liczby* $n$ *nie da się rozłożyć, to* $k(n) = \infty$*. Dla przykładu,* $k(1) = 1$*,* $k(8) = \infty$*,* $k(30) = 4$*,* $k(378) = 12$*,* $k(380) = 10$*.*

*Liczbą* **przerośniętą** *nazwiemy taką liczbę* $x$*, dla której istnieje liczba* $y > x$ *taka, że* $k(y) < k(x)$*. Z poprzedniego przykładu widzimy więc, że 378 jest liczbą przerośniętą.*

*Dla zadanej liczby* $n$ *oblicz* $k(n)$ *oraz liczbę liczb przerośniętych mniejszych lub równych* $n$*.*

## Wejście

*W pierwszym i jedynym wierszu standardowego wejścia znajduje się jedna liczba całkowita* $n$ *(*$1 \leqslant n \leqslant 10^{18}$*). W testach wartych 45% punktów zachodzi dodatkowy warunek* $n \leqslant 50\,000\,000$*, w podzbiorze tych testów wartym 30% punktów warunek* $n \leqslant 1\,000\,000$*, a w podzbiorze tych testów wartym 20% punktów warunek* $n \leqslant 1000$*.*

## Wyjście

*Twój program powinien wypisać na standardowe wyjście dwie liczby całkowite oddzielone pojedynczym odstępem: pierwsza z nich to* $k(n)$*, a druga to liczba liczb przerośniętych z przedziału od 1 do* $n$*. Jeśli* $k(n) = \infty$*, to zamiast pierwszej liczby należy wypisać znak* `-` *(minus).*

## Przykład

| *Dla danych wejściowych:* | *poprawnym wynikiem jest:* |
|---|---|
| `30` | `4 15` |
| *natomiast dla danych wejściowych:* | *poprawnym wynikiem jest:* |
| `8` | `- 5` |

# Rozwiązanie

W zadaniu tym trzeba było trochę poeksperymentować komputerowo na małych liczbach, żeby zauważyć pewne własności. Dlatego też zadanie pojawiło się na I etapie,

by zawodnicy mieli czas na takie eksperymenty. Na początek trzeba zauważyć, że wartości $k(n)$ jesteśmy w stanie wyznaczyć algorytmem plecakowym (programowanie dynamiczne). Do zbioru przedmiotów dodajemy po kolei przedmioty o rozmiarach $1^2, 2^2, \ldots$ i dla każdej liczby $m$ zapisujemy najmniejszy przedmiot, po dodaniu którego jest możliwe uzyskanie liczby $m$ jako sumy rozmiarów różnych przedmiotów dodanych do tej pory. Jeżeli chcemy wyznaczyć wartości $k(1), \ldots, k(n)$, to dodawanie przedmiotów powinniśmy przerwać w momencie, w którym kolejny dodawany przedmiot miałby rozmiar większy od $n$. W tym momencie uzyskaliśmy rozwiązanie siłowe, które co prawda na zawodach dostawało mało punktów (zgodnie z treścią zadania – 20 punktów), jednak przyda nam się w rozwiązaniu wzorcowym.

Aby zbadać, jak zachowuje się funkcja $k$, możemy uruchomić nasze rozwiązanie siłowe wyznaczające jej początkowe wartości np. dla $n \leqslant 1000$. Na początku zastanówmy się nad bardziej podstawowym pytaniem: dla jakich liczb $n$ zachodzi $k(n) = \infty$? Przyglądając się wynikom naszego rozwiązania siłowego, możemy zauważyć, że największa znaleziona liczba bez żadnego przedstawienia w postaci sumy różnych kwadratów liczb całkowitych dodatnich to 128, a wszystkie liczby z przedziału $[129, 1000]$ posiadają jakieś przedstawienie. To może nam nasunąć podejrzenie, że nie istnieją liczby większe od 128, dla których $k(n) = \infty$. Spróbujemy zatem udowodnić następujący fakt:

**Fakt 1.** *Jeżeli* $n \geqslant 129$*, to* $k(n) \neq \infty$*.*

Zanim jednak zabierzemy się do jego dowodu, zastanówmy się jeszcze chwilę i popatrzmy na wyniki wyprodukowane przez rozwiązanie siłowe. Zdefiniujmy następujące funkcje:

$$\sigma(m) \;=\; 1^2 + 2^2 + 3^2 + \ldots + m^2, \quad \mathit{ind}(n) \;=\; \min\{\, m \;:\; \sigma(m) \;\geqslant\; n \,\}.$$

Jest jasne, że $k(n) \geqslant \mathit{ind}(n)$, innymi słowy $\sigma(k(n)) \geqslant n$ (ponieważ suma pewnych spośród liczb $1^2, 2^2, \ldots, k(n)^2$ nie może przekraczać sumy wszystkich tych liczb). Jeżeli popatrzymy na wyniki naszego rozwiązania siłowego, dojdziemy do wniosku, że dla odpowiednio dużych liczb (dajmy na to, większych od 400) często zachodzi $k(n) = \mathit{ind}(n)$, a dla pozostałych liczb zachodzi $k(n) = \mathit{ind}(n) + 1$. Udowodnimy następujący fakt:

**Fakt 2.** *Jeżeli* $m \geqslant 11$ *oraz* $n \in [129, \sigma(m)]$*, to* $k(n) \leqslant m + 1$*.*

Zauważmy, że z tak sformułowanego stwierdzenia wynika w sposób natychmiastowy prawdziwość poprzedniego faktu.

**Dowód:** Dowód przeprowadzimy przez indukcję. Dla $m = 11$ teza jest prawdziwa na mocy wyników uzyskanych przez nasze rozwiązanie siłowe.

Niech teraz $m > 11$ oraz $n \in [129, \sigma(m)]$. Będziemy chcieli udowodnić tezę indukcyjną dla $m$, wiedząc, że jest prawdziwa dla $m - 1$. Rozpatrzmy dwa przypadki:

**Przypadek 1:** $n \leqslant 128 + (m+1)^2$**.** W tym przypadku mamy $128 \leqslant \sigma(m-3)$ (jako że już $\sigma(8) = 204$) oraz $(m+1)^2 \leqslant (m-2)^2 + (m-1)^2$ (co zachodzi dla $m \geqslant 8$). Po zsumowaniu stronami tych dwóch nierówności otrzymujemy $128 + (m+1)^2 \leqslant \sigma(m-1)$, zatem $n \in [129, \sigma(m-1)]$. Na mocy założenia indukcyjnego $k(n) \leqslant m \leqslant m + 1$, więc w tym przypadku teza istotnie zachodzi.

**Przypadek 2:** $n \geqslant 129+(m+1)^2$**.** Wiemy, że $n-(m+1)^2 \in [129, \sigma(m)-(m+1)^2]$, zatem tym bardziej $n-(m+1)^2 \in [129, \sigma(m-1)]$. Na mocy założenia indukcyjnego $k(n-(m+1)^2) \leqslant m$, zatem $n-(m+1)^2 = a_1^2+a_2^2+\ldots+a_l^2$, dla pewnych $1 \leqslant a_1 < a_2 < \ldots < a_l \leqslant m$. Stąd wynika, że $n = a_1^2+a_2^2+\ldots+a_l^2+(m+1)^2$, co jest przedstawieniem spełniającym warunki zadania. Ostatecznie uzyskujemy $k(n) \leqslant m+1$, co dowodzi tezy indukcyjnej. ■

Wiemy już całkiem dużo na temat wartości funkcji $k$; dla *małych* liczb (nazwijmy tak liczby całkowite równe co najwyżej $\sigma(11) = 506$) mamy je *explicite* wyznaczone przez nasze rozwiązanie siłowe, a dla *dużych* $n$ (nazwijmy tak liczby większe od $\sigma(11)$) zachodzi $k(n) \in \{ind(n), ind(n)+1\}$.

Do poznania wszystkich wartości funkcji $k$ pozostaje nam jeszcze jedynie stwierdzić, dla których *dużych* liczb zachodzi $k(n) = ind(n)$, a dla których $k(n) = ind(n)+1$. Ustalmy zatem pewną liczbę $n$, dla której $k(n) = ind(n)$. Wtedy $n = a_1^2+a_2^2+\ldots+a_l^2$, gdzie $1 \leqslant a_1 < a_2 < \ldots < a_l = ind(n)$. Niech teraz liczby $b_1, b_2, \ldots, b_p$ będą to wszystkie liczby całkowite z przedziału $[1, ind(n)]$ różne od liczb $a_1, a_2, \ldots, a_l$. Skoro liczby $a_1, a_2, \ldots, a_l$ oraz $b_1, b_2, \ldots, b_p$ to sumarycznie wszystkie liczby od 1 do $ind(n)$, to

$$\sigma(ind(n)) = a_1^2+\ldots+a_l^2+b_1^2+\ldots+b_p^2 = n+b_1^2+\ldots+b_p^2,$$

zatem $\sigma(ind(n))-n = b_1^2+\ldots+b_p^2$, czyli $k(\sigma(ind(n))-n) < \infty$. Stąd wniosek, że jeżeli dla jakiejś *dużej* liczby $n$ zachodzi $k(\sigma(ind(n))-n) = \infty$, to wtedy $k(n) = ind(n)+1$.

Sprawdźmy natomiast, co się dzieje, jeżeli $k(\sigma(ind(n)) - n) < \infty$. Jako że $n > \sigma(ind(n)-1)$, to

$$\sigma(ind(n)) - n < ind(n)^2 < \sigma(ind(n)-1).$$

Stąd na mocy faktu 2 otrzymujemy, że $k(\sigma(ind(n)) - n) \leqslant ind(n)$, co oznacza, że $\sigma(ind(n)) - n = b_1^2+\ldots+b_p^2$ dla pewnych $1 \leqslant b_1 < \ldots < b_p \leqslant ind(n)$. Jeżeli zatem za liczby $a_1, \ldots, a_l$ weźmiemy wszystkie liczby od 1 do $ind(n)$ różne od $b_1, \ldots, b_p$, to wtedy

$$a_1^2+\ldots+a_l^2 = \sigma(ind(n)) - (b_1^2+\ldots+b_p^2) = \sigma(ind(n)) - (\sigma(ind(n)) - n) = n.$$

Zatem w istocie $n$ ma przedstawienie w postaci sumy kwadratów różnych liczb całkowitych nie większych niż $ind(n)$, czyli $k(n) = ind(n)$, co dopełnia charakteryzacji wartości funkcji $k$. Podsumowując:

**Fakt 3 (Metoda obliczania $k(n)$).**
*Jeżeli $n$ jest małą liczbą, to $k(n)$ jest wyznaczone przez algorytm siłowy. Jeżeli $n$ jest dużą liczbą i $\sigma(ind(n)) - n > 128$, to $k(n) = ind(n)$. Jeżeli $n$ jest dużą liczbą i $\sigma(ind(n)) - n \leqslant 128$, to $k(n) = ind(n)$, jeżeli $k(\sigma(ind(n)) - n) \neq \infty$, a $k(n) = ind(n)+1$ w przeciwnym przypadku.*

Przejdźmy teraz do drugiej części zadania, czyli do zliczania liczb *przerośniętych*. Udowodnimy następujące stwierdzenie:

**Fakt 4.** *Liczba $n$* **nie** *jest przerośnięta wtedy i tylko wtedy, gdy $k(n) = ind(n)$.*

**Dowód:** Jeżeli $k(n) = ind(n)$ oraz $p > n$, to $k(p) \geqslant ind(p) \geqslant ind(n) = k(n)$. Tak więc $n$ w istocie nie jest liczbą przerośniętą.

Mamy także $\sigma(ind(n)) \geqslant n$, zatem jeżeli $k(n) > ind(n)$, to $k(\sigma(ind(n))) = ind(n) < k(n)$. W tym przypadku $n$ rzeczywiście jest liczbą przerośniętą, co dowodzi postawionego faktu. ■

Na przykład liczba 522 jest przerośnięta, ponieważ w tym przypadku mamy: $ind(522) = 12$, $\sigma(11) = 506 < 522 \leqslant \sigma(12) = 650$, ale $k(522) = 13$.

Dla $m \geqslant 12$ zdefiniujmy teraz

$$\Delta_m = (\sigma(m-1),\, \sigma(m)],\ L_m = (\sigma(m-1),\, \sigma(m)-129],\ R_m = (\sigma(m)-129,\, \sigma(m)].$$

Już wiemy, że jeżeli $n \in L_m$ oraz $n$ jest *duże*, to $n$ nie jest przerośnięte. Ponadto dla każdego $m \geqslant 12$ w przedziale $R_m$ jest tyle samo liczb przerośniętych, gdyż jeżeli $n \in R_m$, to to, czy $n$ jest przerośnięte, zależy jedynie od tego, czy $k(\sigma(m) - n) = \infty$. Konkretnie, w każdym takim przedziale znajduje się tyle liczb przerośniętych, ile istnieje takich $m$, że $k(m) = \infty$, czyli dokładnie 31 (wszystkie takie liczby znamy, gdyż wiemy, że są one równe co najwyżej 128).

Stąd wynika już prosty algorytm na zliczanie liczb przerośniętych równych co najwyżej $n$. Jeżeli $n$ jest *małe*, to wynik odczytujemy z rozwiązania siłowego. W przeciwnym przypadku przedział $[1, n]$ dzielimy na liczby *małe*, czyli $[1, \sigma(11)]$, w którym znamy liczbę liczb przerośniętych (wynosi ona 175), oraz na przedział $[\sigma(11) + 1, n]$, który to z kolei dzielimy na przedziały $\Delta_{12} \cup \ldots \cup \Delta_{ind(n)-1} \cup S$, gdzie $S = (\sigma(ind(n)-1), n]$. W każdym z przedziałów od $\Delta_{12}$ do $\Delta_{ind(n)-1}$ znajduje się dokładnie 31 liczb przerośniętych. Jeżeli $n \leqslant \sigma(m) - 129$, to $S \subseteq L_{ind(n)}$, zatem w takim przypadku w $S$ nie ma żadnych liczb przerośniętych. Jeżeli jednak $n \geqslant \sigma(m) - 128$, to $S = L_{ind(n)} \cup (S \cap R_{ind(n)})$ i w $S$ znajduje się tyle liczb przerośniętych, co w $S \cap R_{ind(n)}$. Przedział ten zawiera co najwyżej 129 liczb, zatem możemy przejrzeć wszystkie jego elementy i dla każdego z nich stwierdzić, korzystając z naszej charakteryzacji (fakty 3 i 4), czy jest on liczbą przerośniętą.

Takim oto sposobem otrzymaliśmy algorytm, który na początku musi wykonać pewien *preprocessing*, aby wyznaczyć wartości funkcji $k$ dla *małych* argumentów, potem wyznaczyć $ind(n)$ i na koniec ewentualnie przeiterować po przedziale o długości co najwyżej 129, stwierdzając, które liczby z niego są przerośnięte. Czas działania naszego algorytmu jest zatem stały, nie licząc czasu wyznaczenia $ind(n)$. Jako że $\sigma(m) = \Theta(m^3)$, to $ind(n) = \Theta(\sqrt[3]{n})$, zatem iterowanie po kolejnych wartościach $\sigma(1), \sigma(2), \ldots$ da nam algorytm działający w czasie $O(\sqrt[3]{n})$. To w zupełności wystarcza, aby zmieścić się w limicie czasowym. Możemy też skorzystać ze znanego wzoru $\sigma(m) = \frac{m \cdot (m+1) \cdot (2m+1)}{6}$ i wyszukać wartość $ind(n)$ binarnie, co pozwoli nam otrzymać algorytm działający w czasie $O(\log n)$.

**Michał Włodarczyk**
Treść zadania, Opracowanie

**Piotr Smulewicz, Marek Sommer**
Program

**Dostępna pamięć: 64 MB.**

*OI, etap I, 6.10–3.11.2014*

# Łasuchy

*Na uroczystej kolacji wieńczącej tegoroczny Zlot Bajtockich Miłośników Słodyczy przy okrągłym stole usiadło n łasuchów. Na stole postawiono n tortów. Torty różnią się między sobą wielkością, wyglądem i smakiem, ale dla łasuchów najważniejszą cechą charakteryzującą tort jest jego kaloryczność (i-ty tort ma kaloryczność $c_i$). Torty postawiono tak, że pomiędzy każdą parą sąsiadujących łasuchów znajduje się jeden tort. Każdy z łasuchów może wybrać, czy chce jeść tort znajdujący się po jego lewej stronie, czy ten znajdujący się po jego prawej stronie. Jeśli dwóch łasuchów wybierze ten sam tort, to dzielą się nim po połowie.*

*Każdy łasuch chce zmaksymalizować kaloryczność tortu (lub połówki tortu), który zje. Łasuch będzie niezadowolony, jeśli okaże się, że wybrał źle – czyli zjadłby więcej, gdyby wybrał drugi z dostępnych mu tortów (przy założeniu, że reszta łasuchów nie zmienia swojego wyboru). Pomóż łasuchom dokonać wyboru tak, aby żaden z nich nie był niezadowolony.*

## Wejście

*Pierwszy wiersz standardowego wejścia zawiera liczbę całkowitą $n$ ($2 \leqslant n \leqslant 1\,000\,000$), oznaczającą liczbę łasuchów (i zarazem liczbę tortów). Drugi wiersz zawiera ciąg $n$ liczb całkowitych $c_1, c_2, \ldots, c_n$ ($1 \leqslant c_i \leqslant 1\,000\,000\,000$) pooddzielanych pojedynczymi odstępami; liczba $c_i$ oznacza kaloryczność i-tego tortu. Zakładamy, że i-ty łasuch (dla $1 \leqslant i < n$) może wybrać tort i-ty lub $(i+1)$-szy, natomiast n-ty łasuch może wybrać tort n-ty lub pierwszy.*

*W testach wartych 50% punktów zachodzi $n \leqslant 1000$, a w podzbiorze tych testów wartym 20% punktów zachodzi $n \leqslant 20$.*

## Wyjście

*Jeśli łasuchy nie mogą wybrać tortów tak, aby każdy z nich był zadowolony, w pierwszym i jedynym wierszu standardowego wyjścia powinno znajdować się jedno słowo* `NIE`*. W przeciwnym wypadku, pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać ciąg n liczb całkowitych pooddzielanych pojedynczymi odstępami; i-ta liczba ma oznaczać numer tortu wybranego przez i-tego łasucha. Jeśli jest więcej niż jedna poprawna odpowiedź, Twój program powinien wypisać dowolną z nich.*

## Przykład

*Dla danych wejściowych:*

```
5
5 3 7 2 9
```

*poprawnym wynikiem jest:*

```
2 3 3 5 1
```

**Testy „ocen”:**

`1ocen`: *$n = 20$, torty o nieparzystych indeksach mają kaloryczność 7, a torty o parzystych indeksach mają kaloryczność 3; każdy łasuch będzie zadowolony, jeśli wybierze tort o kaloryczności 7;*

`2ocen`: *$n = 1\,000\,000$, kaloryczność każdego tortu jest liczbą wylosowaną z przedziału $[500\,000\,001, 1\,000\,000\,000]$; jeśli wszystkie łasuchy wybiorą tort z prawej strony lub jeśli wszystkie łasuchy wybiorą tort z lewej strony, to każdy z nich będzie zadowolony.*

# Rozwiązanie

Zadania na Olimpiadzie Informatycznej można zazwyczaj zaklasyfikować do jednego z czterech rodzajów: zliczanie obiektów, optymalizacja, symulacja wydarzeń lub szukanie strategii wygrywającej w grze dwuosobowej. Zadanie *Łasuchy* odbiega od powyższego kanonu i dlatego na początku nie za bardzo wiadomo, jak się za nie zabrać. Jesteśmy proszeni o zaproponowanie dowolnego przyporządkowania tortów, w którym żaden łasuch nie będzie chciał zmieniać swojego przydziału. Choć na pierwszy rzut oka zagadnienie to wydaje się być trochę oderwane od rzeczywistości, okazuje się, że istnieje cała teoria matematyczna, blisko związana z ekonomią, badająca podobne problemy. Znajomość tej teorii nie jest niezbędna do rozwiązania zadania, niemniej jednak wygodnie będzie nam zaczerpnąć z niej kilka pojęć.

## Krótki wstęp do niekooperacyjnej teorii gier

**Definicja 1.** *Grą* nazywamy obiekt matematyczny, składający się ze:

1. zbioru graczy $N = \{1, 2, \ldots, n\}$,
2. $n$ zbiorów strategii – dla $i$-tego gracza jest to $A_i$,
3. rodziny funkcji $u_i : A_1 \times A_2 \times \ldots \times A_n \longrightarrow \mathbb{R}$ dla $i = 1, \ldots, n$, mówiących jakie *wypłaty* dostają gracze w zależności od decyzji podjętych przez wszystkich.

Każdy gracz może wybrać jedną strategię ze swojego zbioru $A_i$. Celem każdego z nich jest maksymalizacja własnego zysku, jednak gracze mogą sobie wzajemnie w tym przeszkadzać.

Definicja ta została wprowadzona przez słynnego matematyka Johna Nasha[1] i okazała się świetnym narzędziem do modelowania sytuacji konfliktu w ekonomii, naukach społecznych, ruchu drogowym czy sztucznej inteligencji. Najważniejszym narzędziem pomocnym w zrozumieniu mechaniki gry jest pojęcie *równowagi*.

**Definicja 2.** *Równowagą Nasha (czystą)* nazwiemy takie przyporządkowanie graczom strategii, w którym żadnemu z graczy nie opłaca się zmieniać strategii, przy założeniu, że pozostali gracze również pozostaną przy swoich wyborach.

[1] John Forbes Nash (1928 – 2015) – amerykański matematyk i ekonomista, współlaureat Nagrody Nobla w dziedzinie ekonomii w 1994 roku. Nash cierpiał na schizofrenię paranoidalną, o czym opowiada film *Piękny umysł*.

Oprócz równowagi *czystej* ważne jest też ogólniejsze pojęcie równowagi *mieszanej*, w której pozwalamy graczom na losowanie strategii.

**Twierdzenie 1 (Nash, 1951).** *Każda gra ze skończoną liczbą graczy i strategii posiada przynajmniej jedną mieszaną równowagę Nasha.*

Sytuacja się komplikuje, kiedy pytamy o istnienie czystej równowagi. Nie dość, że nie każda gra takową posiada (zachęcamy do wymyślenia przykładu – można ograniczyć się do dwóch graczy, z których każdy ma dwie strategie), to rozstrzyganie o jej istnieniu na podstawie funkcji wypłaty jest problemem NP-trudnym, co oznacza, że nie jest znany żaden algorytm odpowiadający na to pytanie w czasie wielomianowym. Oczywiście istnieją gry, w których poszukiwanie czystej równowagi jest prostsze, i na szczęście właśnie z taką grą będziemy mieć do czynienia.

W dalszej części opracowania ograniczamy się tylko do pojęcia równowagi czystej. Więcej o teorii gier można dowiedzieć się z internetowego skryptu `http://mst.mimuw.edu.pl/lecture.php?lecture=wtg`.

## Równowaga przy stole

Nasze zadanie sprowadza się do znalezienia czystej równowagi Nasha w grze, w której graczami są łasuchy, każdy z nich ma dwie strategie (wzięcie lewego bądź prawego tortu), a wypłaty zależą od wybranego tortu oraz od decyzji podjętych przez sąsiadów.

Najprostsze rozwiązanie polega na sprawdzeniu wszystkich scenariuszy, jakie mogą wydarzyć się przy stole. Skoro każdy z graczy ma do wyboru dwie strategie, to złożoność obliczeniowa takiego algorytmu wynosi $O(2^n)$, co pozwala na zdobycie maksymalnie 20 punktów.

Spróbujemy innego podejścia. Wybierzmy pewne początkowe przyporządkowanie tortów i sprawdźmy, kto jest niezadowolony. Dopóki przynajmniej jeden gracz będzie niezadowolony, zmieniamy jego strategię, licząc na to, że w końcu dojdziemy do momentu, w którym wszyscy będą zadowoleni.

Taki pomysł może budzić wątpliwości: co w sytuacji, kiedy równowaga nie istnieje? Okazuje się, że autor zadania był trochę złośliwy, bo dla każdych danych wejściowych istnieje przydział, w którym wszyscy są zadowoleni[2]. Fakt ten stanie się oczywisty podczas analizy przedstawionych algorytmów.

No dobrze, ale jak dobrać stan początkowy oraz kolejność poprawiania łasuchów, aby zagwarantować, że po niewielkiej liczbie poprawek dojdziemy do stanu równowagi? Przecież hipotetycznie moglibyśmy wpaść w pętlę podczas zmieniania strategii graczy i napisać program, który nigdy się nie zakończy! Spokojnie, pod koniec opracowania udowodnimy, że nawet w najgorszej implementacji powyższego algorytmu taka sytuacja nie może nigdy mieć miejsca. Zaczniemy jednak od zaprezentowania kilku metod poprawiania stanu gry, które prowadzą do efektywnych rozwiązań.

[2]Na swoje usprawiedliwienie autor twierdzi, że niemożność wymyślenia przykładu z odpowiedzią *NIE* miała skłonić zawodników do zastanowienia się, dlaczego zawsze istnieje poprawny przydział, co mogło naprowadzić ich na trop algorytmu. Tym razem możemy mu wybaczyć.

**Rozwiązanie wzorcowe**

Pierwszy pomysł na początkowe ustawienie to przypisanie każdemu łasuchowi tortu po jego prawej stronie, czyli $i$-ty łasuch otrzyma tort o kaloryczności $c_{i+1}$ (ustalamy, że indeksy rosną w prawo; ponadto dla prostoty opisu utożsamiamy tort o numerze $n+1$ z tortem o numerze 1). Zastanówmy się, kto może być niezadowolony przy takim ustawieniu. Jeśli $i$-ty gracz woli wybrać lewy tort, który musiałby dzielić z lewym sąsiadem, zamiast prawego tortu, którego nie musi współdzielić, to $c_i > 2c_{i+1}$. Zauważmy, że niezależnie od sytuacji w sąsiedztwie, $i$-ty gracz zawsze będzie preferował lewy tort, więc po pojedynczej zmianie już nigdy nie będziemy musieli poprawiać jego strategii.

Przypuśćmy, że zmodyfikowaliśmy wybór $i$-tego gracza. Mogło to nie spodobać się jego sąsiadom, którzy teraz również mogą chcieć zmienić swoje preferencje. Jednak jeśli gracz o numerze $i+1$ rezygnuje z całego tortu lub gracz o numerze $i-1$ wybiera obecnie tort, który będzie musiał współdzielić, to znowu żadne przyszłe wydarzenia nie zmienią już ich zdania. Spróbujmy sformalizować to rozumowanie.

**Lemat 1.** Rozpoczynając od przypisania każdemu łasuchowi tortu po jego prawej stronie i modyfikując wybory niezadowolonych łasuchów, nigdy nie znajdziemy się w sytuacji, w której pewien łasuch chciałby zmienić tort po raz drugi.

**Dowód:** Indukcja po liczbie graczy, którym zmieniliśmy przydział. Ustalmy niezadowolonego łasucha o numerze $i$, któremu zmieniamy przydział z prawego tortu na lewy. Rozpatrzymy cztery przypadki opisujące jego sąsiedztwo.

A *Żaden z sąsiadów nie był jeszcze modyfikowany.* Jak wcześniej zauważyliśmy, implikuje to $c_i > 2c_{i+1}$, zatem łasuch zawsze będzie preferował lewy tort.

B *Zmodyfikowaliśmy wcześniej tylko lewego sąsiada.* Zgodnie z założeniem indukcyjnym lewy sąsiad wybiera teraz lewy tort i już nigdy tego nie zmieni, natomiast prawy sąsiad wybiera prawy tort, ale może w przyszłości zmienić decyzję. Wynika z tego, że $c_i > c_{i+1}$ i niezależnie od preferencji prawego sąsiada $i$-ty gracz zawsze pozostanie przy lewym torcie.

C *Zmodyfikowaliśmy wcześniej tylko prawego sąsiada.* Rozumowanie analogiczne do przypadku B.

D *Obaj sąsiedzi zostali już zmodyfikowani.* Założenie indukcyjne mówi, że obaj sąsiedzi pozostaną przy swoich wyborach, więc nic nie przeszkodzi już $i$-temu łasuchowi w uczcie.

Pokazaliśmy, że obecnie poprawiany gracz zawsze pozostanie przy swoim nowym torcie, zatem udowodniliśmy tezę indukcyjną, co kończy dowód. ■

Możemy trzymać niezadowolonych łasuchów w kolejce, do której dodajemy łasuchy w momencie, w którym przestaje im odpowiadać ich decyzja. Jako że każdy może trafić do kolejki tylko raz, otrzymujemy rozwiązanie liniowe.

Jeszcze prostszy algorytm dostaniemy, wykorzystując fakt, że każdy początkowo niezadowolony łasuch może być źródłem dwóch ciągów poprawek: w lewo oraz

w prawo. Aby uwzględnić je wszystkie, można najpierw „obejść stół w prawo" sprawdzając zadowolenie kolejnych łasuchów, a następnie w lewo. Uwaga: nie wystarczy przejść od łasucha 1 do łasucha $n$ i z powrotem, ponieważ gra toczy się na okręgu i pominiemy w ten sposób np. ciąg poprawek $(n-3, n-2, n-1, n, 1, 2, 3)$. Na szczęście dwukrotne obejście stołu w każdym z kierunków jest już wystarczające. Takie rozwiązanie zaimplementowano w pliku `las.cpp`.

Obie powyższe implementacje mogły liczyć na 100 punktów, natomiast algorytm sprawdzający łasuchów w kolejności od 1 do $n$ do momentu, aż wszyscy będą zadowoleni, może działać w czasie $\Theta(n^2)$, przez co przechodzi jedynie testy warte 50 punktów.

### Rozwiązanie alternatywne

Inny pomysł to rozpoczęcie od przypisania każdemu łasuchowi tortu o wyższej kaloryczności (w przypadku remisu wybieramy dowolny). Co ciekawe, tym razem wystarczy sprawdzić każdego łasucha tylko raz (w dowolnej kolejności) i zmienić strategię niezadowolonych. Poniższy lemat gwarantuje, że taka procedura doprowadzi do stanu równowagi.

**Lemat 2.** Rozpoczynając od przypisania każdemu łasuchowi tortu o wyższej kaloryczności, wystarczy poprawić strategię każdego łasucha co najwyżej raz, w dowolnej kolejności.

**Dowód:** Chcemy pokazać, że zmiana przydziału dla gracza o numerze $i$ nie popsuje zadowolenia już sprawdzonych łasuchów. Przypuśćmy bez straty ogólności, że $c_i \geqslant c_{i+1}$, czyli omawiany łasuch początkowo wybierał lewy tort. Skoro obecnie jest on z niego niezadowolony, to oznacza, że $i$-ty tort jest współdzielony, zaś tort o numerze $i+1$ nie ma obecnie żadnego amatora. Zatem obaj jego sąsiedzi muszą wybierać torty po swojej prawej.

Kiedy zmienimy decyzję $i$-tego gracza, oba najbliższe torty będą przypisane pojedynczym graczom. Wypłata gracza $i-1$ zwiększy się dwukrotnie, natomiast gracz $i+1$, którzy nie był zainteresowany kalorycznością $c_{i+1}$, tym bardziej nie skusi się na kaloryczność $\frac{c_{i+1}}{2}$. Wobec tego potencjalna zmiana strategii $i$-tego gracza nie wpłynie na wcześniejsze ustalenia i po poprawieniu wszystkich niezadowolonych łasuchów znajdziemy się w stanie równowagi. ■

Implementację tego rozwiązania można znaleźć w pliku `las2.cpp`.

## Dalsza analiza

Co jeżeli nie byliśmy wystarczająco sprytni i zaczęliśmy od innego przyporządkowania początkowego? W dalszym ciągu mamy szansę na przejście kilku testów, ponieważ nasz niekoniecznie efektywny algorytm zawsze znajdzie pewną czystą równowagę Nasha. Aby to pokazać, wprowadzimy przydatne pojęcie *potencjału*.

**Definicja 3.** Dla ustalonego przydziału tortów $p$, jego *potencjałem* nazwiemy wielkość

$$\Phi(p) = \sum_{i=1}^{n} \alpha_i^p c_i, \tag{1}$$

gdzie $\alpha_i^p$ zależy od $p$ i równa się:

0: jeśli $i$-ty tort nie został nikomu przydzielony,

1: jeśli dokładnie jeden gracz chce zjeść $i$-ty tort,

$\frac{3}{2}$: jeśli dwóch graczy zdecydowało się na $i$-ty tort.

Przedstawiona definicja może wydawać się „wyciągnięta z kapelusza", chociażby ze względu na pojawiającą się znikąd stałą $\frac{3}{2}$. Pochodzenie wzoru (1) staje się jednak jasne w dowodzie kolejnego lematu.

**Lemat 3.** Jeśli przydział $q$ otrzymujemy z $p$ poprzez zmianę decyzji pewnego gracza, to wypłata tego gracza wzrasta o $\Phi(q) - \Phi(p)$.

**Dowód:** Przypuśćmy, że interesujący nas gracz zmienia swój tort z $i$-tego na $j$-ty. Z definicji ciągu $(\alpha_i^p)$ wynika, że jego nowa wypłata równa się $(\alpha_j^q - \alpha_j^p)c_j$, zaś poprzednia wynosiła $(\alpha_i^p - \alpha_i^q)c_i$. Jako że dla pozostałych tortów wartości $\alpha$ nie zmieniają się, to różnica pomiędzy tymi wypłatami równa się $\Phi(q) - \Phi(p)$. ■

Powyższa obserwacja pozwala na prosty dowód zapowiedzianego wcześniej twierdzenia.

**Twierdzenie 2.** *Algorytm modyfikujący decyzje niezadowolonych łasuchów zawsze znajduje pewną czystą równowagę Nasha po skończonej liczbie kroków.*

**Dowód:** Oznaczmy ciąg przydziałów generowany w trakcie działania ustalonego algorytmu przez $p_1, p_2, \ldots$ Zmiana decyzji niezadowolonego gracza prowadzi do wzrostu jego wypłaty, toteż z lematu 3 wnioskujemy, że dla każdego $i$ zachodzi $\Phi(p_{i+1}) > \Phi(p_i)$.

Dodatnia różnica potencjałów nie może być mniejsza niż $\frac{1}{2}$, więc gdyby wygenerowany ciąg przydziałów był nieskończony, to ciąg $(\Phi(p_i))_{i=1}^{\infty}$ byłby nieograniczony. Nie jest to możliwe, ponieważ potencjał nie może przekroczyć $\frac{3}{2}\sum_{i=1}^{n} c_i$. Zatem algorytm w pewnym momencie znajduje przydział, w którym wszyscy są zadowoleni, i kończy działanie. ■

W praktyce taki algorytm mógł zdobyć sporo punktów, jeśli zaczynał od losowego przydziału i efektywnie znajdował niezadowolonych łasuchów.

**Teoria gier a algorytmika**

Funkcja potencjału wygląda na zbyt eleganckie narzędzie, by używać go tylko do badania przydziałów tortów. W rzeczywistości zostało ono wprowadzone do badania tzw. *routing games*, modelujących m.in. ruch drogowy albo przesyłanie pakietów danych po sieci. Każdy z uczestników takiej gry ma za zadanie przesłać pewien ładunek pomiędzy dwoma wierzchołkami grafu, a koszt przesyłania ładunku wzdłuż krawędzi zależy od tego, jak mocno jest ona eksploatowana przez innych graczy.

Oczywiście każdy gracz stara się minimalizować tylko własny koszt (albo własny czas przesyłu). Badanie równowagi Nasha w takich grach doprowadziło do odkrycia m.in. paradoksu Braessa, czyli przykładu sieci, w której dodanie nowej krawędzi w grafie prowadzi do pogorszenia sytuacji każdego z graczy. Co ciekawe, nie jest to jedynie teoretyczna koncepcja. Naukowcy zajmujący się zagadnieniami transportu drogowego wielokrotnie zaobserwowali sytuację, w której zamknięcie drogi w dużym mieście paradoksalnie doprowadziło do poprawienia przepustowości w całej sieci drogowej.

Analiza *routing games* jest częścią stosunkowo nowej teorii naukowej zwanej *algorytmiczną teorią gier*.

## Jeszcze inne rozwiązania

Inny wniosek z lematu 3 jest taki, że każdy przydział maksymalizujący funkcję potencjału jest równowagą Nasha. Taki przydział można znaleźć przy pomocy programowania dynamicznego w czasie $O(n)$, co daje nam kolejne efektywne rozwiązanie.

Możliwe jest też bardziej bezpośrednie wykorzystanie programowania dynamicznego. Niech $i \geqslant 3$ oraz $t_i[A, B, C] = 1$, jeśli możliwe jest zadowolenie wszystkich łasuchów z przedziału $[2, i-1]$, gdy $A$ koduje decyzję pierwszego łasucha, natomiast $B, C$ – decyzje łasuchów o numerach $i-1, i$. W przeciwnym przypadku $t_i[A, B, C] = 0$. Dla każdego $i$ musimy wyznaczyć tylko 8 wartości. Łatwo zauważyć, że możemy to zrobić, znając wartości $t_{i-1}[\cdot, \cdot, \cdot]$. Tak obliczona tablica $t$ wystarcza do odtworzenia całego rozwiązania.

## Testy

Testy zostały podzielone na trzy rodzaje:

1. Rosnący (lub malejący) ciąg tortów, w którym nie ma pary sąsiednich tortów, takich że jeden jest co najmniej dwukrotnie bardziej kaloryczny od drugiego. Wyjątek stanowi sytuacja obok pierwszego oraz ostatniego tortu. Całość jest przesunięta cyklicznie o losową wartość.

2. Wiele segmentów o kalorycznościach postaci $[x, x+1, x+2, \ldots]$ przedzielonych bardzo kalorycznymi tortami.

3. Testy całkowicie losowe.

**Jakub Łącki**
Treść zadania, Opracowanie

**Dawid Dąbrowski, Karol Pokorski**
Program

**Dostępna pamięć: 64 MB.**

*OI, etap I, 6.10–3.11.2014*

# Pieczęć

*Bajtek znalazł dziś w swojej poczcie dziwny dokument. Było to zawiadomienie o tym, że w spadku po swoim wuju Bajtazarze otrzymał gigantyczną sumę pieniędzy. Pismo jest opieczętowane wielokrotnie pieczęcią Królestwa Bajtocji. Bajtek woli się upewnić, że nie pada ofiarą naciągaczy. Chce więc sprawdzić, czy pieczęć jest prawdziwa.*

*Bajtek wie, jak wygląda pieczęć Królestwa Bajtocji. Na otrzymanym piśmie jest jednak tyle tuszu, że trudno powiedzieć, czy skrupulatny urzędnik przystawił pieczęć bardzo wiele razy, czy też jest to nieudolna próba zmylenia Bajtka przez oszustów. Pomóż Bajtkowi i napisz program, który mając dany wzór z otrzymanego pisma oraz opis pieczęci, stwierdzi, czy Bajtek jest w posiadaniu autentycznego zawiadomienia o spadku.*

*Pieczęć ma specjalne zabezpieczenia, w związku z czym: (1) nie można jej podczas przystawiania obracać, (2) nie można jej przyłożyć tak, żeby pozostawiła tusz poza stemplowanym dokumentem, oraz (3) każde miejsce na dokumencie może być pokryte tuszem z pieczęci co najwyżej raz.*

## Wejście

*W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita $q$ ($1 \leqslant q \leqslant 10$), określająca liczbę zestawów danych. W kolejnych wierszach znajdują się opisy kolejnych zestawów danych.*

*W pierwszym wierszu opisu pojedynczego zestawu danych znajdują się cztery liczby całkowite $n$, $m$, $a$ oraz $b$ ($1 \leqslant n, m, a, b \leqslant 1000$) pooddzielane pojedynczymi odstępami.*

*W kolejnych $n$ wierszach znajduje się opis wzoru z pisma. Każdy z tych wierszy składa się z $m$ znaków, z których każdy to* `.` *(kropka) lub* `x`*. Kropka oznacza, że w danym miejscu kartki nie ma śladów tuszu, a* `x`*, że tusz pozostawił ślad.*

*Następnie opisany jest wygląd przykładowego dokumentu po jednokrotnym opieczętowaniu go pieczęcią Królestwa Bajtocji. Jest on podany w takim samym formacie jak opis dokumentu otrzymanego przez Bajtka, w $a$ wierszach zawierających po $b$ znaków* `.` *oraz* `x`*. Możesz założyć, że zarówno opis wzoru z pisma, jak i opis przykładowego dokumentu zawiera jakiś ślad tuszu.*

*W testach wartych łącznie 44% punktów dla każdego zestawu danych zachodzi: $n, m, a, b \leqslant 150$.*

## Wyjście

*Twój program powinien wypisać na standardowe wyjście dokładnie $q$ wierszy. W $i$-tym wierszu powinna się znaleźć odpowiedź dla $i$-tego zestawu danych.*

*Jeśli dokument otrzymany przez Bajtka mógł zostać opieczętowany pieczęcią, należy wypisać jedno słowo* `TAK`*. Jeśli zaś pismo otrzymane przez Bajtka na pewno jest fałszerstwem, należy wypisać jedno słowo* `NIE`*.*

### Przykład

| *Dla danych wejściowych:* | *poprawnym wynikiem jest:* |
|---|---|
| `2` | `TAK` |
| `3 4 4 2` | `NIE` |
| `xx..` | |
| `.xx.` | |
| `xx..` | |
| `x.` | |
| `.x` | |
| `x.` | |
| `..` | |
| `2 2 2 2` | |
| `xx` | |
| `xx` | |
| `.x` | |
| `x.` | |

**Testy „ocen”:**

`1ocen`: *Jeden przypadek: dokument o wymiarach $2 \times 3$ cały pokryty tuszem, pieczątka o wymiarach $1 \times 2$. Odpowiedź* `NIE`.

`2ocen`: *Jeden przypadek: dokument–szachownica o wymiarach $8 \times 8$, pieczątka–szachownica o wymiarach $3 \times 3$. Odpowiedź* `NIE`.

`3ocen`: *$q = 10$, wszystkie przypadki takie same: dokumenty $1000 \times 1000$, pieczątki losowe, wymiaru $20 \times 20$, zawsze na dokumencie pieczątka odciśnięta jest dokładnie raz. Wszystkie odpowiedzi* `TAK`.

## Rozwiązanie

Naszym celem jest sprawdzenie, czy dokument otrzymany przez Bajtka mógł zostać wykonany przez wielokrotne odciskanie na nim pieczęci Królestwa Bajtocji. Dokument podany jest w postaci tablicy rozmiaru $n \times m$. Będziemy mówić, że składa się on z *$n$ wierszy* oraz *$m$ kolumn*. W każdym wierszu mamy *$m$ pól*, a każde pole jest albo zamalowane, albo białe. Pieczęć jest podana w taki sam sposób jako tablica złożona z $a$ wierszy oraz $b$ kolumn.

W trakcie odciskania pieczęci Królestwa Bajtocji na dokumencie należy przestrzegać trzech ograniczeń. Po pierwsze, pieczęci nie wolno obracać. Po drugie, za każdym razem należy ją przykładać tak, by zamalowywała ona jedynie pola leżące w granicach dokumentu. Wreszcie po trzecie, każde pole dokumentu może być pokryte tuszem co najwyżej jednokrotnie. Jeśli końcowy wygląd pewnego dokumentu da się uzyskać za pomocą pewnej liczby przystawień pieczęci, przy zachowaniu podanych reguł, dokument taki nazwiemy *poprawnym*. W szczególności pusty dokument, tj. taki, na którym nie ma ani jednego zamalowanego pola, uznajemy za poprawny (choć opis takiego dokumentu nie może się pojawić na wejściu).

Aby sprawdzić, czy dokument jest poprawny, spróbujemy odtworzyć sekwencję odciśnięć pieczęci. Spójrzmy na proces pieczętowania dokumentu od końca. Wyobraźmy sobie, że nagraliśmy film, który przedstawia cały proces przykładania pieczęci królestwa Bajtocji do dokumentu, i teraz oglądamy go wstecz, tj. od końca do początku. Na początku filmu dokument jest zamalowany tuszem. Każde przyłożenie pieczęci wymazuje ślady tuszu z niektórych pól, a w końcu cały dokument jest pusty. Chcielibyśmy teraz wymazać wszystkie ślady tuszu z dokumentu, postępując podobnie jak na puszczonym w tył filmie.

Wyobraźmy sobie *pieczęć wymazującą*, która może posłużyć do cofania wcześniejszych odciśnięć pieczęci. Ma ona taki sam kształt, jak opisana na wejściu pieczęć Królestwa Bajtocji, tj. działa na dokładnie tak samo rozmieszczone pola dokumentu i wymaga przestrzegania analogicznych reguł: nie wolno jej obracać ani przykładać tak, by działała na pola poza dokumentem. Ponadto, każde wymazywane pole musi być zaciemnione w momencie wymazywania. Aby rozwiązać zadanie, wystarczy sprawdzić, czy za pomocą pieczęci wymazującej możemy w podanym dokumencie zamienić wszystkie zamalowane pola na pola białe.

Na początku przyjrzyjmy się najprostszemu możliwemu algorytmowi: przykładamy pieczęć wymazującą w dowolnym miejscu (w którym wolno nam ją przyłożyć) i powtarzamy tę operację tak długo, jak się da. Jeśli na końcu otrzymamy pusty dokument, mamy dowód, że podany na wejściu dokument był poprawny. Mógł on przecież powstać przez odciskanie pieczęci Królestwa Bajtocji w dokładnie tych samych miejscach, w których użyliśmy pieczęci wymazującej. Jednak co w przypadku, gdy na kartce pozostało jeszcze trochę śladów tuszu i nie możemy już więcej użyć pieczęci wymazującej?

Spójrzmy na następujący dokument rozmiaru $1 \times 4$:

```
xxxx
```

oraz pieczęć zamalowującą dwa sąsiednie pola:

```
xx
```

Skorzystajmy z naszego algorytmu i w początkowym dokumencie wymażmy dwa pola – drugie i trzecie od lewej:

```
x..x
```

Dostaliśmy dokument, na którym nie da się już użyć naszej pieczęci wymazującej. Widać jednocześnie, że gdybyśmy przykładali pieczęć wymazującą w inny sposób, np. najpierw do pierwszego i drugiego pola dokumentu, a potem do trzeciego i czwartego, udałoby się nam uzyskać pusty dokument.

Nie możemy zatem przykładać pieczęci wymazującej w dowolnym miejscu. Potrzebujemy nieco sprytniejszego sposobu. Łatwo przekonać się, że jeśli nasza pieczęć ma taki kształt, jak w powyższym przykładzie, wystarczy przykładać lewe pole pieczęci do pierwszego od lewej zamalowanego pola na dokumencie. Jeśli w ten sposób uda nam się wymazać cały tusz, dokument jest poprawny, a w przeciwnym razie – nie. Jak zaraz pokażemy, to podejście daje się rozszerzyć na dowolne dokumenty i pieczęcie.

Spójrzmy na dokument i znajdźmy w nim pierwszy od góry wiersz, w którym co najmniej jedno pole jest zamalowane. Następnie w tym wierszu znajdźmy pierwsze od

lewej zamalowane pole. To pole nazwiemy *pierwszym* polem dokumentu. Analogicznie zdefiniujmy pierwsze pole pieczęci. Czas na kluczową obserwację.

**Obserwacja 1.** Rozważmy dokument, który został poprawnie opieczętowany pewną liczbą odciśnięć pieczęci Królestwa Bajtocji. Wówczas pierwsze pole dokumentu zostało zamalowane pierwszym polem pieczęci.

Uzasadnienie obserwacji jest bardzo proste. Niech $p$ będzie pierwszym polem otrzymanego dokumentu. Rozważmy moment, w którym zamalowane zostało pole $p$. Zastanówmy się, gdzie mogła wówczas zostać przyłożona pieczęć. Ponieważ $p$ to pierwsze pole ostatecznego dokumentu, nie mogliśmy zamalować żadnego pola w wierszach powyżej $p$ oraz żadnego pola na lewo od $p$ w tym samym wierszu. Stąd już wprost wnioskujemy, że dopuszczalne jest tylko takie przyłożenie pieczęci, w którym pole $p$ zostaje zamalowane przez pierwsze pole pieczęci.

## Prosty algorytm

Poczyniona obserwacja wskazuje, w których miejscach pieczęć została przyłożona do dokumentu. W naszym rozwiązaniu odtwarzamy sekwencję przyłożeń pieczęci i zgodnie z tą sekwencją używamy pieczęci wymazującej na podanym na wejściu dokumencie. W tym celu znajdujemy pierwsze pole dokumentu i przykładamy w nim pierwsze pole pieczęci wymazującej. Krok ten powtarzamy tak długo, jak to możliwe. Jeśli w pewnym momencie otrzymamy pusty dokument, wiemy, że podany na wejściu dokument był poprawny. Druga opcja jest taka, że dokument zawiera jeszcze zamalowane pola, jednak nie możemy na nim w żaden sposób odcisnąć pieczęci wymazującej. Taki dokument uznamy za niepoprawny.

Zapiszmy nasze dotychczasowe ustalenia w postaci algorytmu. Dopóki co najmniej jedno pole dokumentu jest zamalowane, powtarzamy kolejno następujące kroki:

1. Znajdź pierwsze pole dokumentu.
2. Sprawdź, czy można odcisnąć pieczęć wymazującą na dokumencie, przykładając pierwsze pole pieczęci do pierwszego pola dokumentu (należy sprawdzić, czy przyłożona pieczęć nie wystaje poza dokument oraz czy pola, które mają zostać wymazane, są aktualnie zamalowane).
    - Jeśli nie jest to możliwe, zgłoś wynik „dokument niepoprawny”.
    - W przeciwnym razie odciśnij pieczęć wymazującą na dokumencie, tj. wymaż tusz z odpowiednich pól dokumentu.

Jeśli po zakończeniu wykonania algorytmu otrzymamy pusty dokument, dokument uznajemy za poprawny.

Zastanówmy się teraz, jaki jest czas działania naszego rozwiązania. Znalezienie pierwszego pola dokumentu realizujemy w najprostszy możliwy sposób: po prostu przeglądamy dokument od samej góry wiersz po wierszu. Wymaga to czasu $O(nm)$. Podobnie w czasie $O(ab)$ znajdujemy pierwsze pole pieczęci. Wreszcie sprawdzenie, czy pieczęć można przyłożyć, łatwo zrealizować w czasie $O(ab)$, porównując wszystkie pola

opisu pieczęci z odpowiadającymi polami dokumentu. W takim samym czasie działa też wymazywanie z dokumentu odpowiednich pól. Zatem jednokrotne wykonanie opisanych powyżej kroków wymaga czasu $O(nm+ab)$. Dla uproszczenia dalszej analizy tego algorytmu załóżmy, że $ab \leqslant nm$, co pozwala nam uprościć czas działania do $O(nm)$.

Ile razy będziemy musieli powtórzyć kroki 1 i 2? Oznaczmy przez $d$ liczbę pól, które zamalowuje pieczęć. Za każdym razem, gdy wykonamy obydwa kroki, wymażemy z dokumentu dokładnie $d$ zamalowanych pól. Zatem łącznie co najwyżej $O(nm/d)$ razy wykonamy obydwa kroki algorytmu.

W pesymistycznym przypadku mamy jednak $d = 1$ i wtedy najlepsze ograniczenie na czas działania całego algorytmu to $O(n^2m^2)$. Jeśli poprawnie zaimplementujemy takie rozwiązanie, otrzymamy 44% punktów.

## Rozwiązanie wzorcowe

Opiszemy teraz, jak przyspieszyć nasz algorytm. Zastanówmy się, jak działa aktualna wersja algorytmu w pesymistycznym przypadku. Jak wcześniej zauważyliśmy, zachodzi on wtedy, gdy podana na wejściu pieczęć zamalowuje tylko jedno pole dokumentu. W tej sytuacji opis pieczęci podany na wejściu może być bardzo nieefektywny. Przyjmijmy, że $n = m = a = b = 1000$. Wtedy opis pieczęci to tablica $1000 \times 1000$, w której tylko jedno pole jest zamalowane. Co więcej, przy każdym przyłożeniu pieczęci przeglądamy tę tablicę w całości!

Skoro interesują nas jedynie zamalowane pola pieczęci, będziemy ją reprezentować w inny sposób. Przyjmijmy ponownie, że pieczęć zamalowuje dokładnie $d$ pól. Wówczas do opisu pieczęci wystarczy nam lista składająca się z $d$ elementów: każdy element opisywać będzie współrzędne jednego pola, które zamalowuje pieczęć. Ponumerujmy wiersze i kolumny pieczęci kolejnymi liczbami naturalnymi, poczynając od 1. Wiersze numerujemy od góry do dołu, a kolumny od lewej do prawej. Przykładowo, pieczęć

```
..
.x
xx
```

możemy zapisać w postaci listy $(2,2),(3,1),(3,2)$. Każdy element na tej liście określa numer wiersza oraz numer kolumny jednego zamalowanego pola. Na potrzeby implementacji prościej nam jednak będzie zapamiętać położenia zamalowanych pól *względem* położenia pierwszego pola. Dlaczego? Gdy znajdziemy pierwsze pole dokumentu, chcemy do niego przyłożyć pierwsze pole pieczęci. A zatem potrzebujemy wiedzieć, gdzie względem niego leżą inne zamalowane pola pieczęci. Pieczęć z powyższego przykładu będziemy więc reprezentować w postaci listy $(0,0),(1,-1),(1,0)$. Jeśli położenie pierwszego pola to $(x,y)$, wystarczy zmodyfikować pierwotną reprezentację, odejmując od każdej pierwszej współrzędnej $x$, a od każdej drugiej $y$. Dzięki temu drugi krok algorytmu możemy wykonać w czasie proporcjonalnym do liczby zamalowanych pól na pieczęci, czyli $O(d)$. Oczywiście na samym początku algorytmu musimy jeszcze przetworzyć reprezentację pieczęci do nowej postaci, co można jednak prosto wykonać w czasie $O(ab)$. Rozwiązanie oparte na tym pomyśle znaleźć można w pliku `pies1.cpp`.

Nadal jednak sporo czasu zużywamy w pierwszym kroku algorytmu na znalezienie pierwszego pola dokumentu. Używamy tu oczywistego algorytmu, który przegląda dokument wiersz za wierszem i zatrzymuje się, gdy tylko znajdzie zamalowane pole. Zauważmy jednak, że w trakcie wykonania algorytmu możemy jedynie zamieniać zamalowane pola na pola białe. A zatem pierwsze pole dokumentu może się jedynie przesuwać coraz „dalej". Jeśli więc w pewnym momencie stwierdzimy, że pierwszym polem dokumentu jest $p$, to gdy następnym razem szukamy pierwszego pola dokumentu, możemy pominąć wszystkie pola leżące powyżej pola $p$ oraz w tym samym wierszu na lewo od niego. Dzięki temu w trakcie *wszystkich* wykonań pierwszego kroku algorytmu każde pole przejrzymy co najwyżej jednokrotnie. W efekcie sumaryczny czas spędzony w pierwszym kroku to $O(nm)$.

Implementację tego algorytmu znaleźć można w pliku `pie.cpp`. Zajmijmy się teraz czasem działania. Na początku budujemy listową reprezentację pieczęci w czasie $O(ab)$, otrzymując listę długości $d$. Następnie, algorytm wykonuje $O(\frac{nm}{d})$ faz. Krok drugi zajmuje za każdym razem czas $O(d)$, co łącznie daje $O(nm)$, czyli tyle samo ile wynosi łączny czas wykonywania pierwszego kroku. Musimy jeszcze umieć szybko stwierdzać, czy dokument posiada zamalowane pole. W tym celu wystarczy stale utrzymywać w pomocniczej zmiennej liczbę aktualnie zamalowanych pól. Dopóki zmienna ta ma dodatnią wartość, dokument posiada zamalowane pola. Tym samym poprawność dokumentu potrafimy sprawdzić w czasie $O(nm + ab)$, czyli w czasie liniowym od rozmiaru danych wejściowych.

# Zawody II stopnia

opracowania zadań

**Michał Włodarczyk**
Treść zadania, Opracowanie

**Bartosz Łukasiewicz**
Program

**Dostępna pamięć: 256 MB.** *OI, etap II, dzień próbny, 10.02.2015*

# Logistyka

*Bajtazar ma firmę logistyczną. Klienci firmy często zlecają przewiezienie dużych ilości towarów, które nie mieszczą się w pojedynczej ciężarówce. Wtedy Bajtazar wysyła konwój. Czasami do konwoju jest przypisanych więcej kierowców niż ciężarówek. Zapasowi kierowcy jadą wtedy jako pasażerowie. Przyjmujemy, że każda ciężarówka może zabrać dowolnie wielu pasażerów. W każdej chwili kierowcy mogą zdecydować się na postój. Wtedy cały konwój zatrzymuje się, a przed wznowieniem jazdy kierowcy mogą wsiąść do dowolnych ciężarówek i zamieniać się za kierownicą. Nie ma żadnych dolnych ani górnych ograniczeń na liczbę postojów na trasie.*

*Aby zwiększyć bezpieczeństwo na drogach, bajtockie ministerstwo transportu wprowadziło ograniczenia czasu pracy kierowców ciężarówek. Każdy z kierowców, po przejściu okresowych testów psychofizycznych, dostaje wpis do prawa jazdy, ile kilometrów może spędzić za kierownicą pojazdu w czasie jednej podróży.*

*Bajtazar poprosił Cię o napisanie programu, który pomoże mu zarządzać jego zespołem n kierowców. Program musi obsługiwać dwa typy zdarzeń:*

- *Uaktualnienie wpisu w prawie jazdy i-tego kierowcy. Zakładamy, że na początku żaden kierowca nie ma wpisu w prawie jazdy. Dopóki go nie otrzyma, nie może prowadzić ciężarówki.*
- *Zapytanie, czy jest możliwe wysłanie konwoju złożonego z c ciężarówek na trasę o długości s kilometrów. Podczas trasy kierowcy mogą jeździć jako pasażerowie i przesiadać się pomiędzy ciężarówkami. Zlecenia są obsługiwane pojedynczo, tzn. kolejny konwój rusza w trasę dopiero po powrocie poprzedniego.*

## Wejście

*Pierwszy wiersz standardowego wejścia zawiera dwie liczby całkowite $n$ i $m$ ($1 \leqslant n, m \leqslant 1\,000\,000$) oddzielone pojedynczym odstępem, oznaczające liczbę kierowców i liczbę zdarzeń. W kolejnych $m$ wierszach znajdują się opisy kolejnych zdarzeń.*

*Jeśli jest to zdarzenie uaktualnienia wpisu, to wiersz składa się z litery* `U` *oraz dwóch liczb całkowitych $k$ i $a$ ($1 \leqslant k \leqslant n$, $0 \leqslant a \leqslant 1\,000\,000\,000$) oznaczających, że $k$-ty kierowca może od tej pory przejechać za kierownicą $a$ kilometrów podczas jednej podróży. Jeśli jest to zapytanie, to wiersz składa się z litery* `Z` *oraz dwóch liczb całkowitych $c$ i $s$ ($1 \leqslant c \leqslant n$, $1 \leqslant s \leqslant 1\,000\,000\,000$) oznaczających pytanie, czy jest możliwe przejechanie $c$ ciężarówkami na trasie o długości $s$ kilometrów.*

*W testach wartych 33% punktów zachodzi dodatkowy warunek $n, m \leqslant 1000$. W testach wartych 66% punktów zachodzi dodatkowy warunek $a, s \leqslant 1\,000\,000$.*

## Wyjście

*Jeśli na wejściu znajduje się $q$ zapytań, to na standardowe wyjście należy wypisać $q$ wierszy: w $i$-tym z nich powinno znajdować się słowo* `TAK` *lub* `NIE` *oznaczające odpowiedź na $i$-te zapytanie z wejścia.*

### Przykład

| *Dla danych wejściowych:* | *poprawnym wynikiem jest:* |
|---|---|
| `3 8` | `NIE` |
| `U 1 5` | `TAK` |
| `U 2 7` | `NIE` |
| `Z 2 6` | `TAK` |
| `U 3 1` | |
| `Z 2 6` | |
| `U 2 2` | |
| `Z 2 6` | |
| `Z 2 1` | |

**Testy „ocen”:**

`1ocen`: *jeden kierowca, kilka wpisów i zapytań z odpowiedziami pozytywnymi i negatywnymi;*

`2ocen`: *1000 kierowców, każdy może przejechać po 1000 km; chcemy wysłać 1000 ciężarówek w trasę 1 km;*

`3ocen`: *500 000 kierowców mogących przejechać po 1 km; wysyłamy jedną ciężarówkę w trasę 500 000 km.*

## Rozwiązanie

Jesteśmy proszeni o wielokrotne modyfikowanie bazy uprawnień kierowców i odpowiadanie na zapytania o zdolność wykonania zleceń firmy Bajtazara. W zadaniach tego typu zazwyczaj opłaca się poświęcić trochę czasu na opracowanie struktury danych, która nie będzie wymagała obliczania wszystkiego od nowa po każdej drobnej zmianie (doświadczeni Czytelnicy już w tym momencie pomyślą o strukturach drzewiastych). Zanim jednak zaczniemy projektowanie struktury danych, sprowadźmy zadawane nam pytanie do możliwie prostego warunku.

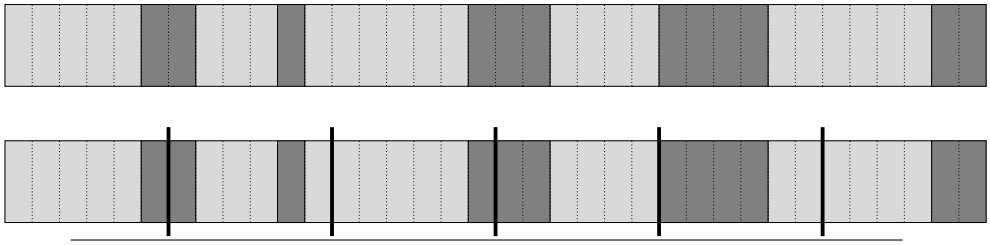
### Warunek dla pojedynczego zlecenia

Oznaczmy przez $(a_i)_{i=1}^n$ ciąg ograniczeń na długość przejechanej trasy kierowców. Dopóki $i$-ty kierowca nie posiada żadnego wpisu w swoim prawie jazdy, przyjmujemy $a_i = 0$. Musimy odpowiedzieć na pytanie, czy taki zespół jest w stanie przejechać $s$ kilometrów konwojem złożonym z $c$ ciężarówek.

**Lemat 1.** Zespół kierowców opisany ciągiem $(a_i)_{i=1}^n$ może wykonać zlecenie na $s$ kilometrów i $c$ ciężarówek wtedy i tylko wtedy, gdy

$$\sum_{i=1}^{n} \min(a_i, s) \geqslant sc. \tag{1}$$

**Dowód:** Łatwo udowodnić, że powyższy warunek jest konieczny do powodzenia podróży. Sumaryczna liczba kilometrów do przejechania wynosi $sc$, zaś $i$-ty kierowca może przejechać w trasie maksymalnie $\min(a_i, s)$ kilometrów.



Rys. 1: Ilustracja do dowodu lematu 1. W przykładzie $s = 6$, $c = 6$, zaś elementy ciągu $a$ to $(5, 2, 3, 1, 10, 3, 4, 4, 20, 5)$.

Wykorzystamy geometryczną interpretację wzoru (1), aby pokazać, że przestawiony warunek jest także wystarczający. Naszkicujmy pasek (patrz rysunek 1) o długości $sc$ jednostek i przypiszmy początkowe $\min(a_1, s)$ jednostek z lewej strony pierwszemu kierowcy. Kolejne $\min(a_2, s)$ jednostek przyporządkujmy drugiemu kierowcy, po czym kontynuujmy ten proces, aż cały pasek będzie zajęty – nierówność (1) gwarantuje nam, że uda się go zapełnić do końca.

Następnie podzielmy pasek na $c$ równych części, odpowiadających poszczególnym ciężarówkom. Zauważmy, że czytając $i$-tą część paska od lewej strony do prawej, dostajemy pewien plan prowadzenia $i$-tej ciężarówki, czyli informację, który kierowca będzie ją prowadzić na poszczególnych kilometrach trasy. Musimy jedynie się upewnić, że jeden kierowca nie będzie przypisany w tym samym momencie do dwóch różnych ciężarówek. Na szczęście taka sytuacja jest niemożliwa, ponieważ gdyby odcinek zawierał segmenty paska odpowiadające temu samemu kilometrowi trasy dla różnych ciężarówek, to musiałby mieć długość co najmniej $s + 1$ jednostek, podczas gdy każdemu kierowcy przyporządkowaliśmy spójny odcinek o długości nie większej niż $s$ jednostek. ■

Bezpośrednie sprawdzanie warunku (1) dla każdego zlecenia prowadzi do algorytmu o złożoności czasowej $O(nm)$, który działa wystarczająco szybko na testach spełniających warunek $n, m \leqslant 1000$, wartych w sumie 33 punkty. Implementacja takiego rozwiązania znajduje się w pliku `logs0.cpp`.

## Rozwiązanie wzorcowe

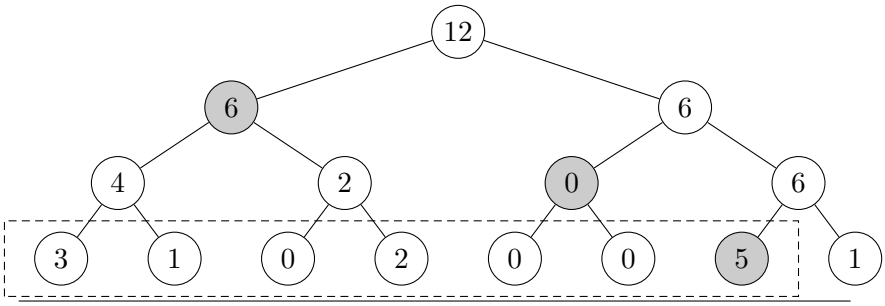
Oznaczmy przez $w_i$ liczbę kierowców, którzy mogą przejechać za kierownicą dokładnie $i$ kilometrów, oraz niech $x_i = iw_i$. Przekształćmy sumę z warunku (1) do nieco wygodniejszej postaci:

$$\sum_{i=1}^{n} \min(a_i, s) = \sum_{a_i \leqslant s} a_i + \sum_{a_i > s} s = \sum_{i \leqslant s} x_i + s \sum_{i > s} w_i. \tag{2}$$

Kiedy dowiadujemy się o zmianie wpisu w uprawnieniach kierowcy z $a$ na $b$ kilometrów, wystarczy wprowadzić następujące modyfikacje:

1. $w_a := w_a - 1$,
2. $x_a := x_a - a$,
3. $w_b := w_b + 1$,
4. $x_b := x_b + b$.

Musimy teraz tylko umieć dynamicznie wyznaczać sumy częściowe ciągów $w_i$ oraz $x_i$. Standardową strukturą danych pomocną w takiej sytuacji jest *drzewo przedziałowe*[1]. Aby obliczyć sumę dla dowolnego przedziału, rozbijamy go na $O(\log n)$ przedziałów bazowych (patrz rysunek 2), dla których będziemy pamiętać obliczone sumy. Podobnie zmiana pojedynczej wartości ciągu będzie wymagać $O(\log n)$ operacji.




Rys. 2: Przykład działania drzewa przedziałowego dla ciągu długości 8. Na szaro zaznaczono rozbicie przedziału $[1 : 7]$ na przedziały bazowe $[1 : 4]$, $[5 : 6]$, $[7]$.

Według treści zadania, w testach wartych 66% punktów zachodzi dodatkowy warunek $a, s \leqslant 10^6$. Oczywiście $w_i, x_i$ będą równe zero dla $i$ powyżej maksymalnej wartości $a$, zatem przy takim ograniczeniu moglibyśmy przechowywać oba ciągi dla $i = 1, \ldots, 10^6$. Przykład takiego rozwiązania znajdziemy w pliku `logb0.cpp`. Jeśli jednak chcemy zdobyć pełne 100 punktów za zadanie, musi nam wystarczyć słabsze założenie $a, s \leqslant 10^9$.

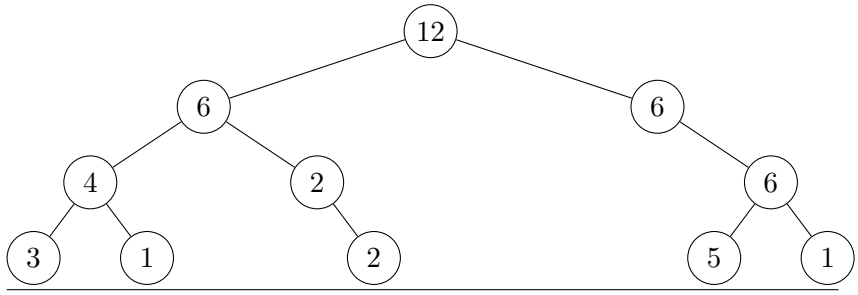
### Dynamiczne drzewo przedziałowe

Limit pamięci uniemożliwia nam alokację nawet pojedynczej tablicy długości $10^9$. Zauważmy jednak, że liczba wszystkich wpisów w dokumentach kierowców jest ograniczona przez $m$, która to wielkość jest nie większa niż $10^6$. W takim razie liczba wszystkich $i$, dla których $w_i, x_i$ będzie niezerowe, jest stosunkowo niewielka i możemy ograniczyć się do pamiętania tylko takich wyrazów ciągów.

W najprostszej implementacji drzewa przedziałowego, najpierw budujemy drzewo dla ciągu o ustalonej długości, a następnie modyfikujemy tylko wartości pamiętane w węzłach drzewa. Nazywamy to **statycznym** drzewem przedziałowym. Nieco większe możliwości daje **dynamiczna** wersja tej struktury danych. Jeśli zrezygnujemy

[1]Dokładny opis tej struktury pojawił się w opracowaniach zadań *Tetris 3D* z XIII Olimpiady Informatycznej [13] oraz *Koleje* z IX Olimpiady Informatycznej [9], a także w *Wykładach z Algorytmiki Stosowanej*, `http://was.zaa.mimuw.edu.pl`.

z wygodnej reprezentacji tablicowej drzewa, możemy dynamicznie dodawać węzły, dopiero kiedy będziemy ich potrzebować. Wysokość takiego drzewa nigdy nie przekroczy $\lceil \log_2(10^9) \rceil = 30$, zatem wszystkie operacje dalej będziemy wykonywać wystarczająco szybko.




Rys. 3: Przykład działania dynamicznego drzewa przedziałowego dla tych samych danych co na rysunku 2.

Pozostaje pytanie, czy zmieścimy się w dostępnym limicie pamięci. Ponieważ maksymalna liczba liści drzewa wynosi $m = 10^6$, a maksymalna wysokość drzewa wynosi 30, to zgrubne szacowanie liczby węzłów daje $30m$ (na każdym poziomie drzewa bierzemy maksymalną liczbę liści). Spróbujmy jednak znaleźć lepsze oszacowanie.

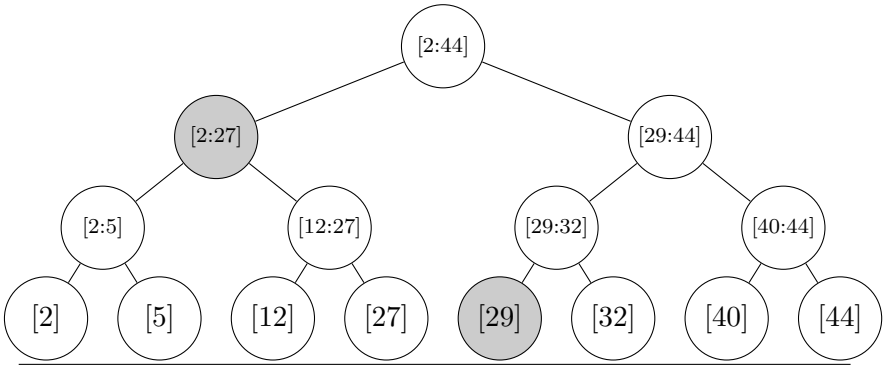
Maksymalna liczba węzłów znajdujących się w drzewie do głębokości 20 to $2^{21} - 1$. Na wszystkich niższych poziomach drzewa (których jest dokładnie 10) każdy liść daje maksymalnie 10 węzłów. Takie oszacowanie daje w sumie nie więcej niż $2^{21} + 10m \leqslant 1{,}21 \cdot 10^7$ węzłów. W pojedynczym węźle przechowujemy sumę podciągu ($w_i$) (zmienna typu `int`, czyli 4 bajty), sumę podciągu ($x_i$) (zmienna typu `long long`, czyli 8 bajtów) oraz wskaźniki do lewego i prawego syna (w sumie 8 bajtów). Cała struktura zajmie nie więcej niż $1{,}21 \cdot 10^7 \cdot 20B \leqslant 242MB$,[2] więc zostało nam jeszcze $14MB$ na pozostałe zmienne.

Powyższe rozwiązanie można znaleźć w pliku `log1.cpp`. Jego złożoność czasowa wynosi $O(n + m \log a_{max})$. Algorytm jest wystarczająco szybki, by przejść wszystkie przygotowane testy. Jednakże dynamiczne drzewo przedziałowe okazuje się nieprzyjemne w implementacji, zwłaszcza jeśli dysponujemy ograniczonym czasem na zawodach. Dlatego zanim zasiądziemy do klawiatury, warto pomyśleć, czy nie istnieje prostsze rozwiązanie.

### Rozwiązanie offline

Okazuje się, że wystarczy nam statyczne drzewo przedziałowe, o ile dopuścimy się drobnego „oszustwa". Pisząc prawdziwe oprogramowanie, musielibyśmy być przygotowani na obsłużenie dowolnych danych, jednak w zadaniu możemy na początku podejrzeć przyszłość i sprawdzić, jakie wartości $a_i$ pojawią się w całej historii zapytań. Następnie zbiór takich wartości możemy posortować i zbudować na nim od razu skompresowane drzewo przedziałowe (patrz rysunek 4).

[2] Lub $231MB$, jeśli zakładamy, że $1MB = 1024 \cdot 1024B$.




Rys. 4: W przykładzie pojawia się 8 wartości $a_i$. W każdym węźle zapisano kontrolowany przez niego przedział. Na szaro zaznaczono przedziały bazowe dla zapytania o przedział $[1 : 30]$.

Tym razem otrzymujemy złożoność czasową $O(n + m \log m)$ oraz pamięciową $O(n+m)$. Implementację takiego rozwiązania można znaleźć w plikach `log.cpp` oraz `log3.pas`.

## Testy

Podstawowy rodzaj testów został zaprojektowany z myślą o następujących założeniach:

- podobny rozkład zapytań typu `U` i `Z`,
- podobny rozkład odpowiedzi `TAK` i `NIE`,
- wykrywanie rozwiązań, które w zapytaniach typu `Z` jedną z sum ze wzoru (2) obliczają siłowo,
- wielokrotne odwoływanie się do tego samego kierowcy, aby wymusić zmianę wpisu,
- wielokrotne przypisywanie tego samego wpisu różnym kierowcom, aby w ciągu $(w_i)$ pojawiały się liczby większe od 1.

Pozostałe testy należały do jednej z poniższych grup:

1. test zupełnie losowy,
2. jeśli odpowiedź to `TAK`, to w każdej ciężarówce pojedzie dwóch kierowców,
3. najpierw dodawane są dodatnie wpisy, potem wpisy są znowu zerowane,
4. każdy kierowca może przejechać co najwyżej 1 kilometr.

**Jacek Tomasiewicz** Treść zadania | **Jakub Radoszewski** Opracowanie | **Dawid Dąbrowski, Tomasz Syposz** Program

**Dostępna pamięć: 128 MB.** *OI, etap II, dzień pierwszy, 11.02.2015*

# Podział naszyjnika

*Mamy naszyjnik złożony z $n$ koralików, z których każdy jest jednego z $k$ rodzajów. Koraliki numerujemy liczbami całkowitymi od 1 do $n$. Koralik o numerze $i$ sąsiaduje w naszyjniku z koralikami o numerach $i+1$ oraz $i-1$ (o ile koraliki o takich numerach istnieją), a ponadto koraliki o numerach 1 oraz $n$ również sąsiadują ze sobą. Chcemy podzielić naszyjnik dwoma cięciami na dwie niepuste części tak, aby każdy rodzaj koralika występował dokładnie w jednej z części (tzn. jeśli jedna z części zawiera koralik rodzaju $j$, to druga część nie może zawierać żadnego koralika rodzaju $j$). Na ile sposobów możemy to zrobić oraz jaka jest minimalna różnica długości otrzymanych części?*

## Wejście

*Pierwszy wiersz standardowego wejścia zawiera dwie liczby całkowite $n$ i $k$ ($2 \leqslant k \leqslant n \leqslant 1\,000\,000$) oddzielone pojedynczym odstępem, oznaczające długość naszyjnika i liczbę rodzajów koralików. Rodzaje koralików numerujemy liczbami od 1 do $k$. Drugi wiersz wejścia zawiera ciąg $n$ liczb całkowitych $r_1, r_2, \ldots, r_n$ ($1 \leqslant r_i \leqslant k$) pooddzielanych pojedynczymi odstępami; liczba $r_i$ oznacza rodzaj koralika o numerze $i$. Możesz założyć, że każdy rodzaj koralika wystąpi w naszyjniku co najmniej raz.*

*W testach wartych łącznie 30% punktów zachodzi dodatkowy warunek $n \leqslant 1000$.*

## Wyjście

*Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać dwie liczby całkowite oddzielone pojedynczym odstępem. Pierwsza z nich ma oznaczać liczbę sposobów, na jakie można podzielić naszyjnik (możesz założyć, że dla danych wejściowych co najmniej jeden podział jest możliwy). Druga liczba ma oznaczać minimalną różnicę długości otrzymanych części.*

## Przykład

*Dla danych wejściowych:*

```
9 5
2 5 3 2 2 4 1 1 3
```

*poprawnym wynikiem jest:*

```
4 3
```

**Wyjaśnienie do przykładu:** *Są cztery możliwe podziały; krótsza część może zawierać koraliki $(5)$, $(4)$, $(1,1)$ lub $(4,1,1)$. W ostatnim przypadku uzyskujemy optymalną różnicę długości $6-3=3$.*

**Testy „ocen”:**

`1ocen`: *krótki naszyjnik z dwoma możliwymi podziałami ($n = 10$);*

`2ocen`: *naszyjnik długości 1000, wszystkie koraliki mają różne rodzaje, wszystkie podziały są poprawne;*

`3ocen`: *$n = 1\,000\,000$, $k = n/2$, naszyjnik postaci $1, \ldots, k, k, \ldots, 1$.*

# Rozwiązanie

Dany jest cykliczny naszyjnik złożony z $n$ koralików, które możemy ponumerować od 1 do $n$. Dla każdego koralika znamy jego rodzaj $r_i$ określony liczbą z zakresu od 1 do $k$. W zadaniu interesują nas podziały naszyjnika za pomocą dwóch cięć na dwie części, takie że każdy rodzaj koralika występuje w dokładniej jednej z części. Każdy taki podział nazwiemy *prawidłowym*. Naszym zadaniem jest wyznaczenie liczby prawidłowych podziałów naszyjnika oraz prawidłowego podziału na dwie części o jak najbardziej zbliżonych długościach. Wiemy przy tym, że w naszyjniku występuje każdy rodzaj koralika od 1 do $k$ oraz że istnieje przynajmniej jeden prawidłowy podział naszyjnika.

## Od cyklu do ciągu

Jeśli nasze dwa cięcia wykonujemy przed koralikami o numerach $a$ oraz $b$, dla $1 \leqslant a < b \leqslant n$, to naszyjnik rozpada się na części $a, \ldots, b-1$ oraz $b, \ldots, n, 1, \ldots, a-1$. To oznacza, że zawsze jedna z części będzie spójnym fragmentem oryginalnego ciągu $(r_i)$. Tak więc każdy podział naszyjnika odpowiada jakiemuś fragmentowi ciągu $(r_i)$, i odwrotnie: każdy fragment ciągu $(r_i)$ odpowiada jakiemuś podziałowi naszyjnika. Może się jednak zdarzyć, że dwa fragmenty ciągu $(r_i)$ będą odpowiadać temu samemu podziałowi naszyjnika. Przy powyższych oznaczeniach zachodzi to tylko w przypadku, gdy $a = 1$: wówczas otrzymane części naszyjnika to $1, \ldots, b-1$ oraz $b, \ldots, n$. Aby uniknąć dwukrotnego rozpatrywania tego samego podziału, możemy na przykład rozważać tylko fragmenty ciągu kończące się przed pozycją $n$ (odpowiadają one wtedy zawsze fragmentom $a, \ldots, b-1$ z powyższego opisu). Równoważnie moglibyśmy też rozważać tylko fragmenty niezaczynające się na pierwszej pozycji.

Fragment ciągu nazwiemy *prawidłowym*, jeśli odpowiadający mu podział naszyjnika jest prawidłowy. Fragment jest więc prawidłowy, jeśli każdy rodzaj koralika występuje tylko w nim albo nie występuje w nim wcale. Używając tej definicji, możemy teraz sformułować zadanie w wygodniejszy sposób, unikając cykliczności naszyjnika: zamiast zliczać prawidłowe podziały naszyjnika, możemy równoważnie zliczać prawidłowe fragmenty ciągu kończące się przed pozycją $n$, a zamiast szukać prawidłowego podziału naszyjnika na części o możliwie zbliżonych długościach, wystarczy szukać prawidłowego fragmentu ciągu o długości możliwie zbliżonej do $\frac{n}{2}$ (fragment ten może być dłuższy lub krótszy od $\frac{n}{2}$, jako że nie wiemy, czy odpowiada on dłuższej, czy też krótszej części naszyjnika po podziale).

## Jakie fragmenty ciągu są prawidłowe?

Zastanówmy się, na jakiej pozycji $j$ może kończyć się prawidłowy fragment ciągu zaczynający się na danej pozycji $i$. Wiemy już, że dany fragment ciągu jest prawidłowy, jeśli dla każdego rodzaju koralika albo (1) wszystkie koraliki tego rodzaju występują w tym fragmencie, albo (2) żaden koralik tego rodzaju nie występuje w tym fragmencie. W ten sposób każdy rodzaj koralika nakłada pewne warunki na pozycję $j$. Spróbujmy określić dokładniej jakie to warunki.

Załóżmy, że koraliki ustalonego rodzaju $c \in \{1, \ldots, k\}$ występują w ciągu $(r_i)$ na pozycjach $p_{c,1} < p_{c,2} < \ldots < p_{c,n_c}$. Zauważmy, że jeśli $n_c = 1$, czyli koralik tego rodzaju występuje w ciągu dokładnie raz, to albo ten koralik wystąpi w rozważanym fragmencie i wtedy dla fragmentu będzie spełniony warunek (1), albo też koralik ten nie wystąpi w tym fragmencie i wtedy dla fragmentu będzie spełniony warunek (2). To oznacza, że taki rodzaj koralika nie nakłada żadnych ograniczeń na koniec fragmentu.

Przyjmijmy teraz, że $n_c \geqslant 2$. Wówczas wszystko zależy od położenia $i$ w ciągu pozycji $p_{c,1}, \ldots, p_{c,n_c}$. Jeśli $i \leqslant p_{c,1}$, to warunek (1) dla tego rodzaju koralików będzie spełniony dla pozycji $j \geqslant p_{c,n_c}$, a warunek (2) dla pozycji $j < p_{c,1}$. Tak więc przedział pozycji *zabronionych*, na których nie może kończyć się rozważany fragment, to $[p_{c,1}, p_{c,n_c} - 1]$. Jeśli $p_{c,l} < i \leqslant p_{c,l+1}$ dla pewnego $l$, to warunek (2) będzie spełniony dla pozycji $j < p_{c,l+1}$, zaś warunek (1) nigdy nie będzie spełniony, gdyż koralik znajdujący się na pozycji $p_{c,1}$ na pewno znajdzie się poza fragmentem. W ten sposób zabronione są pozycje $[p_{c,l+1}, n]$. Wreszcie jeśli $i > p_{c,n_c}$, to dowolne $j \leqslant n$ spełnia warunek (2), więc żadna pozycja nie jest zabroniona.

Ostatecznie dla każdego rodzaju koralika otrzymaliśmy co najwyżej jeden przedział pozycji zabronionych, czyli takich, na których nie może kończyć się prawidłowy fragment zaczynający się na pozycji $i$.

## Wyznaczanie przedziałów zabronionych

Aby pójść naprzód, musimy jednak umieć efektywnie wyznaczać przedziały pozycji zabronionych (w skrócie: przedziały zabronione) dla każdej kolejnej pozycji początkowej fragmentu $i$. Dla pozycji $i = 1$ możemy to zrobić siłowo. Natomiast gdy przemieszczamy się o jedną pozycję dalej (z $i$ do $i + 1$), zmianie może ulec jedynie przedział zabroniony odpowiadający koralikowi z opuszczanej właśnie pozycji.

Okazuje się, że do wyznaczenia zmieniających się przedziałów wystarczą dwie tablice:

- *next*$[i]$ – przechowująca dla każdej pozycji $i \in \{1, \ldots, n\}$, pozycję następnego koralika rodzaju $r_i$ lub $\infty$, jeśli takiego koralika nie ma,
- *interval*$[c]$ – przechowująca dla każdego rodzaju koralika $c \in \{1, \ldots, k\}$, przedział zabroniony wynikający z tego rodzaju koralika dla bieżącej pozycji $i$.

Faktycznie, załóżmy, że przesuwamy się z pozycji $i$ na pozycję $i + 1$. Mamy wtedy kilka przypadków. Niech $c = r_i$.

- Jeśli *interval*$[c]$ był pusty, to pozostaje pusty; odpowiada to przypadkowi $n_c = 1$.

- W przeciwnym razie, jeśli $next[i] < \infty$, to $interval[c]$ staje się $[next[i], n]$; odpowiada to przypadkowi $i = p_{c,l}$ dla $l < n_c$, czyli $p_{c,l} < i + 1 \leqslant p_{c,l+1}$.
- Jeśli zaś $next[i] = \infty$, to $interval[c]$ staje się pusty; odpowiada to przypadkowi $i = p_{c,n_c}$, czyli $i + 1 > p_{c,n_c}$.

Zauważmy, że na powyższej liście (oczywiście) nie występuje przypadek $i + 1 \leqslant p_{c,1}$.

Elementy tablicy *next* można wyznaczyć w czasie $O(n + k) = O(n)$. W tym celu wystarczy przejść po wszystkich elementach ciągu od końca do początku, przechowując w tablicy rozmiaru $k$ pozycje ostatnio napotkanych koralików każdego rodzaju. Podobnie w czasie $O(n)$ można wyznaczyć początkowe przedziały zabronione dla pozycji 1. Wreszcie wyznaczenie zmieniających się przedziałów zabronionych dla poszczególnych pozycji $i$ zgodnie z opisaną procedurą wykonujemy w czasie $O(n)$. Aby nigdy nie rozważać fragmentów kończących się na pozycji $n$, na samym początku dodajemy dodatkowy przedział zabroniony $[n, n]$.

## Drzewo przedziałowe

Umiemy już efektywnie wyznaczać przedziały zabronione dla poszczególnych pozycji początkowych. Musimy teraz wymyślić, jak na tej podstawie uzyskać odpowiedzi na interesujące nas pytania. Nie będzie niespodzianką, że możemy w tym celu zastosować (statyczne) *drzewo przedziałowe*. Więcej o tej strukturze danych można przeczytać np. w opracowaniu zadania *Logistyka* w tej książeczce i w zawartych tam odnośnikach.

Skoncentrujmy się na razie na pierwszym pytaniu z zadania. Aby na nie odpowiedzieć, wystarczy wyznaczyć liczbę prawidłowych fragmentów zaczynających się na poszczególnych pozycjach ciągu. Liczba ta zależy wprost od liczby pozycji końcowych zawartych w przedziałach zabronionych, czyli od łącznej długości sumy tych przedziałów.

W drzewie przedziałowym będziemy przechowywać aktualne przedziały zabronione. Standardowo, gdy dodajemy nowy przedział zabroniony, wstawiamy go do drzewa z wagą $+1$, a gdy zmieniamy przedział zabroniony, to stary przedział wstawiamy z wagą $-1$, a następnie nowy z wagą $+1$. Wstawienie przedziału do drzewa polega na rozbiciu go na *przedziały bazowe*, odpowiadające węzłom drzewa. W każdym węźle drzewa $v$ przechowujemy, jako $w[v]$, sumę wag ze wstawionych przedziałów, dla których $v$ znalazł się w rozkładzie na przedziały bazowe. Dzięki temu, że z wagą $-1$ wstawiamy przedziały, które wcześniej wstawiliśmy z wagą $+1$, wartości $w$ będą zawsze nieujemne.

W każdym węźle drzewa $v$ będziemy także przechowywać liczbę $f[v]$ zabronionych pozycji w jego poddrzewie. Dokładniej, będzie to liczba pozycji zabronionych, biorąc pod uwagę jedynie wartości $w$ z poddrzewa $v$ – jeśli jakiś przodek węzła $v$ ma dodatnią wartość $w$, to niezależnie od faktycznej wartości $f[v]$ każda pozycja w poddrzewie jest zabroniona, jednak węzeł $v$ nie będzie o tym wiedział. Ostatecznie szukaną liczbę prawidłowych fragmentów wyznaczymy na podstawie wartości $f$ w korzeniu.

Trzeba jeszcze pokazać, że umiemy aktualizować wartości $f$ przy wstawianiu przedziałów. Standardowo, po wykonaniu każdego wstawienia, aktualizujemy te wartości we wszystkich węzłach znajdujących się w przodkach węzłów z rozkładu na przedziały

bazowe. Wystarczy teraz zauważyć, że do wyznaczenia wartości $f$ dla węzła $v$ wystarczą nam dane przechowywane w nim i w jego dzieciach: jeśli $w[v] = 0$, to $f[v]$ jest sumą wartości $f$ dla jego dzieci $left[v]$ i $right[v]$, a w przeciwnym razie jest ona równa długości odpowiadającego mu przedziału bazowego $[a[v], b[v]]$.

Zbudowanie drzewa przedziałowego zajmuje czas $O(n)$. Łącznie wykonujemy co najwyżej $2n$ wstawień przedziałów. Każde takie wstawienie na drzewie przedziałowym działa w czasie $O(\log n)$. Ostatecznie ta część rozwiązania działa w czasie $O(n \log n)$.

## Najbardziej zrównoważony podział

Nieco więcej uwagi wymaga zastosowanie drzewa przedziałowego do rozwiązania drugiej części zadania, polegającej na znalezieniu prawidłowego fragmentu o długości najbliższej $\frac{n}{2}$. W chwili rozpatrywania danej pozycji początkowej fragmentu $i$ weźmiemy pod uwagę pozycję $j = i + \lfloor \frac{n}{2} \rfloor - 1$, a jeśli wypada ona za pozycją $n$, to pozycję $j = n$. Naszym celem będzie teraz znalezienie końcowej pozycji prawidłowego fragmentu jak najbliższej pozycji $j$. Innymi słowy, będziemy chcieli znaleźć najbliższą pozycji $j$ pozycję *dozwoloną*, czyli niezawierającą się w żadnym z przedziałów zabronionych. Będziemy jej szukać zarówno wśród wcześniejszych pozycji (będzie to tzw. *dozwolony poprzednik* pozycji $j$), jak i późniejszych pozycji (tzw. *dozwolony następnik* pozycji $j$). Wyszukiwanie tak zdefiniowanego poprzednika i następnika pozycji będzie przypominać nieco analogiczne procedury używane w drzewie wyszukiwań binarnych (np. jako podprocedury w implementacji usuwania węzła z drzewa BST).

Przede wszystkim na początku warto sprawdzić, czy sama pozycja $j$ nie jest dozwolona. Jest tak dokładnie wtedy, gdy suma wartości $w$ na ścieżce od liścia odpowiadającego pozycji $j$ ($leaf[j]$) do korzenia drzewa ($root$) jest równa zeru. Do przechodzenia w górę drzewa użyjemy tablicy $parent$, wskazującej na ojca węzła.

```
1:  v := leaf[j];
2:  sumw := w[v];
3:  while v ≠ root do begin
4:    v := parent[v];
5:    sumw := sumw + w[v];
6:  end
7:  if sumw = 0 then pozycja j jest dozwolona;
```

Jeśli pozycja $j$ okazała się dozwolona, kandydatem na wynik pochodzącym od pozycji $i$ jest właśnie pozycja $j$. W przeciwnym razie musimy wyznaczyć dozwolonego poprzednika i następnika pozycji $j$. Skoncentrujemy się tylko na pierwszym z nich; drugiego szuka się symetrycznie (jeśli w ogóle $j < n$).

Aby znaleźć dozwolonego poprzednika pozycji $j$, przechodzimy ścieżką od liścia $leaf[j]$ do korzenia drzewa w poszukiwaniu pierwszego węzła, do którego ścieżka ta wchodzi od strony prawego syna, natomiast poddrzewo jego lewego syna zawiera jakąś dozwoloną pozycję. Wiemy wtedy, że szukany poprzednik znajduje się w owym lewym poddrzewie. To, czy poddrzewo węzła $v$ zawiera jakąś dozwoloną pozycję, sprawdzamy, porównując wartość $f[v]$ z długością przedziału bazowego $[a[v], b[v]]$ *oraz* biorąc pod uwagę sumę wartości $w$ na ścieżce od węzła $v$ do korzenia drzewa. Sumę tę

uzyskujemy z obliczonej przed chwilą zmiennej *sumw*. Znaleziony węzeł zapisujemy w zmiennej *node*.

```
1:  v := leaf[j];
2:  { Używamy zmiennej sumw z poprzedniego pseudokodu! }
3:  node := nil;
4:  while v ≠ root do begin
5:    sumw := sumw − w[v];
6:    if v = right[parent[v]] then begin
7:      v' := left[parent[v]];
8:      if (sumw = 0) and (f[v'] < b[v'] − a[v'] + 1) then begin
9:        node := v';
10:       break;
11:     end
12:   end
13:   v := parent[v];
14: end
15: return node;
```

Jeśli na koniec *node* = **nil**, to pozycja $j$ nie ma dozwolonego poprzednika. W przeciwnym razie będzie nim skrajnie prawy dozwolony węzeł w poddrzewie węzła *node*. W poniższym pseudokodzie pokazujemy, jak znaleźć taki węzeł. Zauważmy, że nie musimy już używać zmiennej *sumw*, gdyż wiemy, że dla węzła *node* jest ona równa 0. Wynikiem funkcji jest dozwolony poprzednik pozycji $j$.

```
1:  v := node;
2:  while a[v] ≠ b[v] do begin { dopóki v nie jest liściem }
3:    v' := right[v];
4:    if f[v'] < b[v'] − a[v'] + 1 then v := v';
5:    else v := left[v];
6:  end
7:  return a[v];
```

Chociaż w powyższym opisie do oznaczania parametrów węzłów używaliśmy notacji tablicowej, doświadczeni w używaniu drzewa przedziałowego zawodnicy wiedzą, że wartości typu *parent*$[v]$, *left*$[v]$ oraz *right*$[v]$ można łatwo wyznaczyć na podstawie numeru węzła, a przedziały $[a[v], b[v]]$ na podstawie numeru węzła i jego głębokości – przy założeniu, że drzewo przedziałowe pamiętane jest w statycznej tablicy. Podobnie rzecz ma się z wyznaczaniem węzła *leaf*$[j]$ na podstawie pozycji $j$.

Wyznaczenie dozwolonego poprzednika i następnika pozycji $j$ wymaga tylko stałej liczby przejść w górę i w dół drzewa przedziałowego, co wykonujemy w czasie $O(\log n)$. Druga część rozwiązania działa zatem także w czasie $O(n \log n)$. Taka jest więc ostateczna złożoność czasowa całego rozwiązania.

Implementacja rozwiązania wzorcowego zgodna z powyższym opisem znajduje się w pliku `pod4.cpp`, natomiast inne implementacje można znaleźć w plikach `pod.cpp` oraz `pod[1-3].cpp`. Przy pisaniu programu trzeba było pamiętać o tym, że do przechowywania pierwszego z obliczanych wyników potrzebna była zmienna całkowita 64-bitowa.

**Adam Karczmarz**
Treść zadania, Opracowanie

**Piotr Smulewicz, Bartosz Tarnawski**
Program

**Dostępna pamięć: 128 MB.** *OI, etap II, dzień pierwszy, 11.02.2015*

# Pustynia

*Droga z Bajtadu do Bajtary wiedzie przez piaski Wielkiej Pustyni Bajtockiej. Jest to męcząca wędrówka, zwłaszcza że na całej trasie znajduje się tylko $s$ studni. Widząc, że gospodarka Bajtocji zależy w dużej mierze od dostępności szlaków komunikacyjnych, władca Bajtocji postanowił wykopać nowe studnie na tej trasie. Odległość z Bajtadu do Bajtary wynosi $n + 1$ bajtomil i w każdym punkcie w odległości całkowitej liczby bajtomil od Bajtadu znajduje się lub może znajdować się studnia. Im głębiej jest położona woda w danym miejscu, tym trudniejsze i bardziej kosztowne jest wykopanie w tym miejscu nowej studni.*

*Władca zlecił zatem zbadanie sytuacji nadwornemu geologowi Bajtazarowi. Bajtazar dysponuje $m$ pomiarami wykonanymi za pomocą sieci satelitarnej. Niestety, informacje dostarczone przez satelity nie dają wprost informacji na temat głębokości wody. Każdy pomiar wykonany jest na spójnym fragmencie trasy i wskazuje jedynie, że w pewnych punktach na tym fragmencie woda znajduje się głębiej niż w pozostałych. Dodatkowo wiadomo, że woda w każdym punkcie leży na całkowitej głębokości od 1 do $10^9$ bajtometrów.*

*Pomóż Bajtazarowi i wyznacz, jak może wyglądać rzeczywista głębokość wody w każdym punkcie trasy. Może się okazać, że dane satelitarne są sprzeczne.*

## Wejście

*Pierwszy wiersz standardowego wejścia zawiera trzy liczby całkowite $n$, $s$ i $m$ ($1 \leqslant s \leqslant n \leqslant 100\,000$, $1 \leqslant m \leqslant 200\,000$) pooddzielane pojedynczymi odstępami, opisujące odległość między miastami, liczbę studni na trasie oraz liczbę pomiarów satelitarnych.*

*Kolejne $s$ wierszy opisuje studnie: $i$-ty z nich zawiera dwie liczby całkowite $p_i$ i $d_i$ ($1 \leqslant p_i \leqslant n$, $1 \leqslant d_i \leqslant 1\,000\,000\,000$), oznaczające, że $i$-ta studnia znajduje się w odległości $p_i$ bajtomil od Bajtadu i ma głębokość $d_i$ bajtometrów (tzn. w punkcie, w którym znajduje się studnia, woda jest na głębokości $d_i$ bajtometrów). Studnie podane są w kolejności rosnących wartości $p_i$.*

*Kolejne $m$ wierszy opisuje wykonane pomiary satelitarne: $i$-ty z nich zawiera trzy liczby całkowite $l_i$, $r_i$ i $k_i$ ($1 \leqslant l_i < r_i \leqslant n$, $1 \leqslant k_i \leqslant r_i - l_i$), po których następuje ciąg $k_i$ liczb całkowitych $x_1, x_2, \ldots, x_{k_i}$ ($l_i \leqslant x_1 < x_2 < \ldots < x_{k_i} \leqslant r_i$). Oznacza to pomiar na odcinku od $l_i$ do $r_i$ (włącznie), w wyniku którego ustalono, że woda w punktach $x_1, \ldots, x_{k_i}$ znajduje się* **ściśle głębiej** *niż woda w pozostałych punktach z tego odcinka. Suma wszystkich wartości $k_i$ nie przekracza 200 000.*

*W testach wartych łącznie 60% punktów zachodzą dodatkowe warunki $n, m \leqslant 1000$. W testach wartych łącznie 30% punktów zachodzi dodatkowy warunek, że suma wszystkich wartości $k_i$ nie przekracza 1000.*

## Wyjście

*Jeśli nie istnieje układ głębokości zgodny z wykonanymi pomiarami, pierwszy wiersz standardowego wyjścia powinien zawierać jedno słowo* `NIE`*. W przeciwnym wypadku w pierwszym wierszu*

*wyjścia powinno znaleźć się słowo* `TAK`*, natomiast drugi wiersz powinien zawierać ciąg n liczb całkowitych z przedziału od 1 do 1 000 000 000 oznaczający głębokości wody w kolejnych punktach na trasie (idąc od Bajtadu). Jeśli istnieje więcej niż jedno poprawne rozwiązanie, Twój program powinien wypisać dowolne z nich.*

## Przykład

| *Dla danych wejściowych:* | *jednym z poprawnych wyników jest:* |
|---|---|
| `5 2 2`<br>`2 7`<br>`5 3`<br>`1 4 2 2 3`<br>`4 5 1 4` | `TAK`<br>`6 7 1000000000 6 3` |
| *Dla danych wejściowych:* | *poprawnym wynikiem jest:* |
| `3 2 1`<br>`2 3`<br>`3 5`<br>`1 3 1 2` | `NIE` |
| *Również dla danych wejściowych:* | *poprawnym wynikiem jest:* |
| `2 1 1`<br>`1 1000000000`<br>`1 2 1 2` | `NIE` |

**Testy „ocen":**

`1ocen`: *n = 100 000, pomiary wskazują, że woda w punkcie i jest głębiej niż we wszystkich wcześniejszych punktach trasy (dla* $i = 2, \ldots, n$*);*

`2ocen`: *n = 100 000, z jednego pomiaru wynika, że woda w punktach o numerach parzystych jest głębiej niż w punktach o numerach nieparzystych.*

# Rozwiązanie

Naszym zadaniem jest konstrukcja pewnego ciągu liczb całkowitych $a_1, \ldots, a_n$ (czasami w skrócie będziemy go nazywać $a$). Liczbę $a_i$ definiujemy jako głębokość studni położonej w odległości $i$ bajtomil od Bajtadu. Szukany ciąg musi spełniać następujące warunki:

(1) $1 \leqslant a_i \leqslant 10^9$ dla $i = 1, \ldots, n$,

(2) dla $i \in \mathcal{F}_0$, gdzie $\mathcal{F}_0 \subseteq \{1, \ldots, n\}$, $a_i$ jest ustalone i dane na wejściu,

(3) dla każdego spośród $m$ danych na wejściu ograniczeń postaci $(l, r, x_1, \ldots, x_k)$, gdzie $l \leqslant x_1 < \ldots < x_k \leqslant r$, każda z liczb $a_{x_1}, \ldots, a_{x_k}$ jest ostro większa niż dowolne $a_j$, takie że $j \in \{l, \ldots, r\} \setminus \{x_1, \ldots, x_k\}$.

Może się także okazać, że podane warunki są sprzeczne – nasz program ma wtedy wypisać na wyjście słowo `NIE`.

## Reprezentacja ciągu i ograniczeń

Naszą strategią będzie iteracyjne upraszczanie ograniczeń nałożonych na nasz ciąg, aż dotrzemy do ograniczeń, które jednocześnie są trywialne do sprawdzenia i pozwalają łatwo znaleźć przykładowe rozwiązanie. Mówiąc ściślej, będziemy utrzymywać:

- ograniczenia górne $c_1, \ldots, c_n$, oznaczające, że szukany ciąg powinien spełniać $1 \leqslant a_i \leqslant c_i$ dla $i = 1, \ldots, n$. Początkowo ustawiamy $c_i := 10^9$.
- Zbiór $\mathcal{F}$ zawierający indeksy $i \in \{1, \ldots, n\}$, dla których $a_i$ jest już określone. Początkowo mamy $\mathcal{F} = \mathcal{F}_0$.
- Ważony graf skierowany $G = (V, E)$ o wierzchołkach $V = \{v_1, \ldots, v_n\}$, reprezentujący nierówności pomiędzy elementami ciągu $a$. Krawędź od $v_i$ do $v_j$ o wadze $d$ w takim grafie oznacza, że $a_i \geqslant a_j + d$.

Początkowy graf $G$ konstruujemy na podstawie danych ograniczeń typu (3). Dla każdego ograniczenia postaci $(l, r, x_1, \ldots, x_k)$ i każdej pary $(a_i, a_j)$ takiej, że $i \in \{x_1, \ldots, x_k\}$ i $j \in [l, r] \setminus \{x_1, \ldots, x_k\}$, tworzymy krawędź od $v_i$ do $v_j$ o wadze 1.

## Upraszczanie ograniczeń

Dopóki będzie to możliwe, będziemy stosować jedną z dwóch reguł upraszczających ograniczenia. Intuicyjnie, chcemy zastępować ograniczenia wynikające z grafu $G$ takimi, które wyrażone są za pomocą ograniczeń górnych $c_i$ i zbioru $\mathcal{F}$. Każda z tych reguł ma następujące własności.

(1) Jeżeli istnieje ciąg spełniający ograniczenia przed zastosowaniem reguły, to istnieje też ciąg spełniający ograniczenia uproszczone.

(2) Każdy ciąg spełniający ograniczenia po zastosowaniu reguły spełnia również ograniczenia przed zastosowaniem reguły.

**Reguła 1.** Jeśli istnieje takie $i$, że $i \notin \mathcal{F}$ i w $G$ nie ma krawędzi wchodzącej do $v_i$, to dodajemy $i$ do $\mathcal{F}$ i ustawiamy $a_i := c_i$.

Musimy pokazać, że jeśli istnieje ciąg $a$ spełniający ograniczenia przed zastosowaniem reguły, to istnieje też ciąg spełniający uproszczone ograniczenia. Istotnie, jako że $a$ spełnia ograniczenia przed zastosowaniem reguły, to $1 \leqslant a_i \leqslant c_i$. Ponieważ do $v_i$ nie wchodzi w $G$ żadna krawędź, nie mamy ograniczeń typu $a_j \geqslant a_i + d$. Być może istnieją ograniczenia postaci $a_i \geqslant a_j + d$, ale każde z nich jest tym bardziej spełnione, jeśli wartość $a_i$ zwiększymy do wartości $c_i$. To pokazuje własność (1) dla tej reguły.

Zauważmy, że ograniczenia przed zastosowaniem reguły są podzbiorem ograniczeń po jej zastosowaniu. Stąd łatwo wynika własność (2).

**Reguła 2.** Jeśli istnieje takie $i$, że $i \in \mathcal{F}$ i istnieje krawędź $(v_i, v_j) \in E$ o wadze $d$, to przypisujemy $c_j := \min(c_j, a_i - d)$. Ponadto, usuwamy z $E$ krawędź $(v_i, v_j)$.

Dla dowolnego ciągu $a$, ograniczenia $a_j \leqslant c_j$ i $a_i \geqslant a_j + d$ są równoważne ograniczeniu $a_j \leqslant \min(c_j, a_i - d)$. To dowodzi jednocześnie obu własności (1) i (2).

Zastanówmy się teraz, w jakiej sytuacji nie jesteśmy w stanie zastosować żadnej z reguł upraszczających. Dzieje się tak dokładnie wtedy, gdy do każdego $v_i$, $i \notin \mathcal{F}$, wchodzi co najmniej jedna krawędź (w przeciwnym razie można by zastosować regułę 1), a dla każdego $j \in \mathcal{F}$, z $v_j$ nie wychodzi w $G$ żadna krawędź (w przeciwnym razie można by zastosować regułę 2). Każda krawędź w grafie $G$ musi mieć zatem początek w $v_j$ takim, że $j \notin \mathcal{F}$.

Może się zdarzyć, że $\mathcal{F} = \{1, \ldots, n\}$. Wtedy w grafie $G$ nie ma ani jednej krawędzi. Własności (1) i (2) stosowanych reguł gwarantują, że jeżeli $1 \leqslant a_i \leqslant c_i$ dla każdego $i$, to $a_1, \ldots, a_n$ spełnia oryginalne ograniczenia. W przeciwnym wypadku, na mocy własności (1), mamy pewność, że rozwiązanie nie istnieje.

Jeśli natomiast $\mathcal{F} \neq \{1, \ldots, n\}$, to po usunięciu z grafu wierzchołków $\{v_i : i \in \mathcal{F}\}$ dostajemy niepusty graf skierowany, w którym do każdego wierzchołka wchodzi przynajmniej jedna krawędź. Łatwo pokazać, że w takim grafie musi istnieć cykl, na przykład $w_1 \to \ldots \to w_p \to w_1$, gdzie $p > 1$. Ponieważ każda krawędź w $G$ ma wagę 1, cykl taki odpowiada $p$ nierównościom: $w_i \geqslant w_{i+1} + 1$ dla każdego $i = 1, 2, \ldots, p-1$ i $w_p \geqslant w_1 + 1$. Dodając stronami wszystkie te nierówności, otrzymujemy $w_1 + \ldots + w_p \geqslant w_1 + \ldots + w_p + p$, nierówność w oczywisty sposób nieprawdziwą. Zatem na mocy własności (1), w tym przypadku ograniczenia są sprzeczne.

## Implementacja

Aby efektywnie zaimplementować stosowanie reguł, oprócz ograniczeń $c_i$, zbioru $\mathcal{F}$ i grafu $G$, utrzymujemy kolejki $Q_j$, dla $j = 1, 2$, zawierające indeksy $i$, dla których można zastosować regułę $j$. Po zastosowaniu reguły 1, usuwamy indeks $i$ z kolejki $Q_1$ i wstawiamy go do $Q_2$, o ile z $v_i$ wychodzi aktualnie co najmniej jedna krawędź. Po zastosowaniu reguły 2, indeks $i$ usuwamy z kolejki $Q_2$ tylko wtedy, gdy krawędź $(v_i, v_j)$ była jedyną krawędzią wychodzącą z $v_i$. Dodatkowo jeśli w wyniku usunięcia tej krawędzi $v_j$ staje się wierzchołkiem, do którego nie wchodzi żadna krawędź, wstawiamy $j$ do $Q_1$, jeśli $j \notin \mathcal{F}$.

Każdy indeks jest wstawiany co najwyżej raz do $Q_1$ i co najwyżej raz do $Q_2$. Dodatkowo, reguła 2 jest stosowana dla każdego indeksu $i$ w kolejce $Q_2$ co najwyżej tyle razy, ile początkowo krawędzi wychodzi z $v_i$. Stąd, na kolejce $Q_1$ wykonujemy $O(n) = O(|V|)$ operacji, a na kolejce $Q_2$ wykonujemy $O(|E|)$ operacji. Nasz algorytm działa zatem w czasie $O(|V| + |E|)$.

Aby ostatecznie obliczyć złożoność tego rozwiązania, musimy oszacować liczbę krawędzi w grafie $G$ przed pierwszym uproszczeniem ograniczeń. Każde wejściowe ograniczenie postaci $(l, r, x_1, \ldots, x_k)$ tworzy $(r-l+1-k) \cdot k \leqslant nk$ krawędzi. Oznaczmy przez $S$ sumę liczb $k$ we wszystkich ograniczeniach. Wówczas złożoność czasowa tego algorytmu szacuje się przez $O(n \cdot S)$. Zauważmy, że w grafie mogą znajdować się krawędzie wielokrotne. Takie rozwiązanie było oceniane na zawodach na ok. 30% punktów. Przykładowa implementacja znajduje się w pliku `puss5.cpp`.

## Zmniejszanie rozmiaru grafu

Rozważmy $g$-te ograniczenie postaci $(l, r, x_1, \ldots, x_k)$. Zauważmy, że dla każdego ciągu $a$ spełniającego to ograniczenie, istnieje liczba $a^*_g$ taka, że $a_{x_i} \geqslant a^*_g + 1$ dla $i = 1, \ldots, k$ i jednocześnie $a^*_g \geqslant a_j$ dla każdego $j \in [l, r] \setminus \{x_1, \ldots, x_k\}$. Nic nie stoi zatem na przeszkodzie, żeby rozszerzyć poszukiwany ciąg $a_1, \ldots, a_n$ o dodatkowe $m$ pomocniczych liczb $a^*_1, \ldots, a^*_m$, które nasz algorytm będzie próbował wyznaczyć, a następnie zignoruje je przy wypisywaniu wyniku.

Przypomnijmy, że algorytm upraszczający ograniczenia działał w czasie proporcjonalnym do rozmiaru grafu $G$. Dodatkowe elementy ciągu pozwolą nam na zmniejszenie rozmiaru początkowego grafu $G$, ale nie wpłyną na sam algorytm upraszczający.

Nasz nowy graf $G$ ma większy zbiór wierzchołków $V = \{v_1, \ldots, v_n, v^*_1, \ldots, v^*_m\}$. Podobnie jak poprzednio, ważone i skierowane krawędzie pomiędzy wierzchołkami odpowiadają nierównościom pomiędzy elementami $a_1, \ldots, a_n, a^*_1, \ldots, a^*_m$. Aby zakodować $g$-te ograniczenie postaci $(l, r, x_1, \ldots, x_k)$, do grafu dodajemy krawędzie $(v_{x_i}, v^*_g)$ dla $i = 1, \ldots, k$, każdą o wadze 1. Dodatkowo, dla każdego $j \in [l, r] \setminus \{x_1, \ldots, x_k\}$, dodajemy do grafu krawędź $(v^*_g, v_j)$ o wadze 0.

Inaczej niż poprzednio, tym razem w grafie $G$ mamy dwie możliwe wagi na krawędziach – 0 i 1. Aby zastosować algorytm upraszczający ograniczenia, należy się upewnić, że dowolny cykl w takim grafie wciąż prowadzi do sprzeczności. Tak istotnie jest: w dowolnym cyklu w $G$ na zmianę występują wierzchołki odpowiadające liczbom $a_i$ i te odpowiadające liczbom $a^*_j$. Stąd, co druga krawędź na cyklu ma wagę 1, a więc każdy cykl ma dodatnią wagę. Na mocy analogicznego jak uprzednio argumentu otrzymujemy sprzeczność.

Aby oszacować złożoność tego rozwiązania, musimy obliczyć rozmiar grafu $G$. Mamy $n + m$ wierzchołków. Dla każdego z $m$ ograniczeń typu $(l, r, x_1, \ldots, x_k)$ tworzymy $k + (r - l + 1 - k) = r - l + 1 \leqslant n$ krawędzi. Stąd, sumaryczna liczba krawędzi szacuje się przez $nm$ i złożonością czasową całego rozwiązania jest $O(nm)$. Takie rozwiązanie otrzymywało na zawodach około 60% punktów. Program `puss4.cpp` realizuje ten pomysł.

## Rozwiązanie wzorcowe

W rozwiązaniu wzorcowym postępujemy podobnie jak poprzednio: zmniejszamy rozmiar grafu $G$, wprowadzając pomocnicze wyrazy ciągu, które, odpowiednio zdefiniowane, pozwolą nam na zmniejszenie liczby krawędzi potrzebnych do zakodowania wszystkich ograniczeń.

W poprzednim rozwiązaniu, rozważając $g$-te ograniczenie $(l, r, x_1, \ldots, x_k)$, do $G$ dodawaliśmy $r - l + 1 - k$ ograniczeń postaci $(v^*_g, v_j)$. Liczba $r - l + 1$ mogła być jednak duża, rzędu $\Theta(n)$. Spróbujmy uporać się z tym problemem. Zauważmy, że zbiór $[l, r] \setminus \{x_1, \ldots, x_k\}$ jest tak naprawdę sumą $k + 1$ (być może pustych) przedziałów: $[l, x_1 - 1], [x_1 + 1, x_2 - 1], \ldots, [x_{k-1} + 1, x_k - 1], [x_k + 1, r]$. Gdybyśmy dla każdego przedziału $[x, y]$ (gdzie $1 \leqslant x \leqslant y \leqslant n$) mieli dodatkowy element ciągu $a_{[x,y]}$ taki, że $a_{[x,y]} \geqslant a_z$ dla każdego $z = x, \ldots, y$, to dla $g$-tego ograniczenia wystarczyłoby nam tylko $k$ dodatkowych krawędzi w $G$. Niestety, wprowadzenie elementów $a_{[x,y]}$ dla każdego możliwego przedziału $[x, y]$ wiązałoby się z dodatkowymi krawędziami w grafie:

moglibyśmy na przykład zakodować za pomocą krawędzi ograniczenia $a_{[x,y]} \geqslant a_z$ dla każdych $1 \leqslant x \leqslant z \leqslant y \leqslant n$, otrzymując w ten sposób $O(n^3)$ dodatkowych krawędzi. Moglibyśmy również postąpić odrobinę sprytniej i brać pod uwagę tylko ograniczenia postaci $a_{[x,y]} \geqslant a_{[x+1,y]}$ i $a_{[x,y]} \geqslant a_{[x,y-1]}$ dla każdych $x < y$, generując w ten sposób tylko $O(n^2)$ nowych krawędzi. Ponieważ $n$ może być dość duże, takie rozwiązanie nie jest jednak wystarczające do uzyskania kompletu punktów.

Nie rezygnujemy jednak z pomysłu użycia elementów ciągu ograniczających z góry pewne spójne grupy elementów oryginalnego ciągu. Skorzystamy z pomysłu powtarzającego się wielokrotnie w zadaniach olimpijskich: użyjemy tak zwanych *przedziałów bazowych*. Przypomnijmy pokrótce, czym są przedziały bazowe. Dla liczby naturalnej $B$, zbiór $P_B$ przedziałów bazowych definiujemy jako najmniejszy zbiór taki, że:

- przedział $[1, 2^B]$ należy do $P_B$,
- jeśli przedział $[a, b]$ należy do $P_B$ i $a < b$, to przedziały $[a, s]$ i $[s + 1, b]$, gdzie $s = \lfloor \frac{a+b}{2} \rfloor$, także należą do $P_B$.

Przykładowo, jeśli $B = 2$, to do zbioru przedziałów bazowych $P_B$ należą przedziały $[1, 4], [1, 2], [3, 4], [1, 1], [2, 2], [3, 3]$ i $[4, 4]$.

Do zbioru $P_B$ należy dokładnie $2^{B+1} - 1$ przedziałów bazowych. Każdy przedział $[x, y]$, gdzie $x \leqslant y$ i $x, y \in \{1, \ldots, 2^B\}$, można rozbić na $O(B)$ parami rozłącznych przedziałów bazowych, które pokrywają wszystkie elementy całkowite zawarte w $[x, y]$. Co więcej, rozbicie to można obliczyć w czasie $O(B)$. Przedziały bazowe łączy się z konstrukcją drzewa przedziałowego, o którym można przeczytać np. w opracowaniu zadania *Logistyka* w tej książeczce i w zawartych tam odnośnikach.

Powróćmy teraz do naszego rozwiązania. Niech $B$ będzie najmniejszą liczbą naturalną taką, że $n \leqslant 2^B$. Mamy $n > 2^{B-1}$, a więc zbiór przedziałów bazowych $P_B$ zawiera $O(n)$ przedziałów. Wiemy też, że każdy przedział $[x, y]$, gdzie $1 \leqslant x \leqslant y \leqslant n$, można rozbić na $O(\log n)$ przedziałów z $P_B$. Dodajemy do konstruowanego ciągu $a_1, \ldots, a_n, a_1^*, \ldots, a_m^*$ jeszcze co najwyżej $2^B - 1 = O(n)$ elementów: dla każdego przedziału bazowego $[p, q] \subseteq [1, n]$, gdzie $p < q$, mamy element $a_{[p,q]}$ taki, że $a_{[p,q]} \geqslant a_z$ dla każdego $z \in [p, q]$. Będziemy też używali zapisu $a_{[p,p]}$ do oznaczenia elementu $a_p$. Dla każdego dodanego elementu mamy także w grafie $G$ odpowiadający mu wierzchołek $v_{[p,q]}$. Aby zakodować za pomocą grafu ograniczenia postaci $a_{[p,q]} \geqslant a_z$ dla każdego $z \in [p, q]$, dodajemy do $G$ krawędzie $(v_{[p,q]}, v_{[p,s]})$ i $(v_{[p,q]}, v_{[s+1,q]})$, gdzie $s = \lfloor \frac{p+q}{2} \rfloor$ – obie o wadze 0. Definicja przedziałów bazowych gwarantuje, że $[p, s]$ i $[s + 1, q]$ także należą do $P_B$ i zawierają się w $[1, n]$. Przez indukcję po długości przedziału bazowego można łatwo udowodnić, że dla $z \in [p, q]$ istnieje w grafie $G$ skierowana ścieżka od $v_{[p,q]}$ do $v_z$ o wadze 0. Stąd, ograniczenia zakodowane w grafie implikują, że $a_{[p,q]} \geqslant a_z$. W ten sposób dodajemy do grafu co najwyżej $2|P_B| = O(n)$ wierzchołków i $O(n)$ krawędzi.

Pozostaje ustalić, jak zakodować $g$-te ograniczenie $(l, r, x_1, \ldots, x_k)$ za pomocą krawędzi grafu. Podobnie jak poprzednio, mamy w $G$ krawędzie $(v_{x_i}, v_g^*)$ o wadze 1. Każdy niepusty przedział $I$ spośród $[l, x_1 - 1], [x_1 + 1, x_2 - 1], \ldots, [x_{k-1} + 1, x_k - 1], [x_k + 1, r]$ rozbijamy na $O(\log n)$ przedziałów bazowych $I'_1, I'_2, \ldots, I'_h$, a następnie dla każdego $j = 1, \ldots, h$ dodajemy do grafu krawędź $(v_g^*, v_{I'_j})$ o wadze 0. Ponieważ $I = I'_1 \cup \ldots \cup I'_h$, krawędzie te będą implikować nierówność $a_g^* \geqslant a_j$ dla każdego $j \in [l, r] \setminus \{x_1, \ldots, x_k\}$. Ostatecznie, w związku z tym ograniczeniem do $G$ dodajemy $O(k \log n)$ krawędzi.

Graf $G$ ma zatem $O(n+m)$ wierzchołków i $O(S \log n)$ krawędzi. Stosując ponownie algorytm upraszczający ograniczenia, otrzymujemy rozwiązanie wzorcowe o złożoności czasowej $O(n + S \log n)$. Implementacja tego rozwiązania znajduje się w pliku `pus3.cpp`.

## Testy

Przygotowano 10 grup testów. Większość testów z odpowiedzią `TAK` była generowana w dwóch niezależnych fazach:

1. Wybór przykładowego ciągu spełniającego warunki i wybór zbioru $\mathcal{F}_0$. Ciągi były generowane za pomocą kilku procedur o różnych stopniach losowości.
2. Wybór ograniczeń spełnionych dla ustalonego ciągu. Dwie przykładowe metody wyboru ograniczeń to metoda losowa i wybór dla każdego $i$ maksymalnego przedziału, dla którego $a_i$ jest maksymalnym elementem tego przedziału.

W testach z odpowiedzią `NIE`, generowany był graf $G$ z cyklem, który następnie był zamieniany na wejściowe ograniczenia, bądź wymuszane było, aby niektóre elementy $a_i$ dla $i \notin \mathcal{F}_0$ musiały pochodzić spoza przedziału $[1, 10^9]$.

**Wojciech Rytter**
Treść zadania, Opracowanie

**Jan Kanty Milczek**
Program

**Dostępna pamięć: 128 MB.** *OI, etap II, dzień drugi, 12.02.2015*

# Kurs szybkiego czytania

*Bajtazar zapisał się na kurs szybkiego czytania, na którym nauczył się wielu ćwiczeń poprawiających spostrzegawczość. Jego ulubionym ćwiczeniem jest znajdowanie wzorca w ciągu symboli. Aby przygotować ćwiczenie, Bajtazar wykorzystuje komputer do wygenerowania bardzo długiego ciągu zer i jedynek. Wybiera liczby $n$, $a$, $b$, $p$ takie, że $n$ i $a$ są względnie pierwsze, a komputer generuje ciąg $c_0, c_1, \ldots, c_{n-1}$, gdzie $c_i = 0$ wtedy i tylko wtedy, gdy $(ai + b) \bmod n < p$. Następnie Bajtazar wymyśla drugi, krótszy ciąg $m$ symboli $w_0, w_1, \ldots, w_{m-1}$. Jego zadaniem jest jak najszybsze znalezienie wszystkich wystąpień krótszego ciągu w ciągu wygenerowanym przez komputer. Ciebie poprosił o pomoc w napisaniu programu, który sprawdzi, czy znalazł wszystkie.*

## Wejście

*Pierwszy wiersz standardowego wejścia zawiera pięć liczb całkowitych $n$, $a$, $b$, $p$ i $m$ ($2 \leqslant n \leqslant 1\,000\,000\,000$, $1 \leqslant p, a, b, m < n$, $1 \leqslant m \leqslant 1\,000\,000$) pooddzielanych pojedynczymi odstępami. Liczby $a$ i $n$ są względnie pierwsze. W drugim wierszu zapisane jest jedno $m$-literowe słowo $w_0, w_1, \ldots, w_{m-1}$ złożone z symboli* `0` *i* `1`.

*Istnieją następujące, rozłączne grupy testów:*

- *w testach wartych 8% punktów zachodzi warunek $n \leqslant 1000$;*
- *w innych testach wartych 8% punktów zachodzi warunek $n \leqslant 1\,000\,000$;*
- *w jeszcze innych testach wartych 66% punktów zachodzi warunek $m \leqslant 1000$.*

## Wyjście

*Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać liczbę całkowitą, będącą liczbą wystąpień ciągu $w_0, w_1, \ldots, w_{m-1}$ w ciągu $c_0, c_1, \ldots, c_{n-1}$.*

## Przykład

*Dla danych wejściowych:*

```
9 5 6 4 3
101
```

*poprawnym wynikiem jest:*

```
3
```

**Wyjaśnienie do przykładu:** *Dla $n = 9$, $a = 5$, $b = 6$ i $p = 4$ komputer wygeneruje ciąg zgodnie z poniższą tabelką. Ciąg* `101` *występuje w wygenerowanym ciągu trzy razy.*

| $i$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| $ai + b$ | 6 | 11 | 16 | 21 | 26 | 31 | 36 | 41 | 46 |
| $(ai + b) \bmod n$ | 6 | 2 | 7 | 3 | 8 | 4 | 0 | 5 | 1 |
| $c_i$ | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |

**Testy „ocen”:**

`1ocen:` *szukamy wystąpień ciągu* `0010` *w ciągu* `10010000100100100100`*;*

`2ocen:` *szukamy wystąpień ciągu* `00000` *w ciągu* `00000001000000010000000000000001`*;*

`3ocen:` $n = 1\,000\,000\,000$*,* $m = 1\,000\,000$*. Szukamy wystąpień ciągu* `011...11` *(cyfra 0 oraz same jedynki) w ciągu* `00...0011...110` *(499 999 999 zer, 500 000 000 jedynek, 1 zero).*

# Rozwiązanie

Zadanie to, pozornie natury tekstowej, należy zakwalifikować raczej jako teoriolicz-bowe. W zadaniu mamy ciągi $\mathbf{c}$ oraz $w$, a naszym celem jest zliczyć wystąpienia ciągu $w$ (wzorca) w ciągu $\mathbf{c}$ (tekście). Problemem jest duży rozmiar $\mathbf{c}$. Rozwiązanie otrzymamy, wykorzystując arytmetykę postępów arytmetycznych i przedziałów na cyklu.

Niech $\oplus$ oznacza operację dodawania modulo $n$. Powiemy, że wzorzec $w$ występuje cyklicznie w tekście $\mathbf{c}$ na pozycji $i$, gdy

$$w[k] = \mathbf{c}[i \oplus k] \quad \text{dla wszystkich } k = 0, \ldots, m-1.$$

Wystąpienie cykliczne, które nie jest standardowym wystąpieniem, nazywamy *nadmiarowym*.

W przykładzie z treści zadania $w$ występuje w $\mathbf{c}$ trzy razy, natomiast cyklicznie występuje cztery razy. Wystąpienie cykliczne zaczynające się na przedostatniej pozycji jest nadmiarowe.

Definiujemy

$$\mathbf{P}(k) \;=\; \{\, i \;:\; w[k] = \mathbf{c}[i \oplus k] \,\}.$$

**Obserwacja 1.** Zbiór pozycji wystąpień cyklicznych $w$ jest równy $\bigcap_{k=0}^{m-1} \mathbf{P}(k)$.

Zgrubnie nasz algorytm wygląda następująco.

1: **Algorytm**
2: Oblicz $\bigcap_{k=0}^{m-1} \mathbf{P}(k)$;
3: *nadmiar* := liczba nadmiarowych wystąpień cyklicznych;
4: **return** $|\bigcap_{k=0}^{m-1} \mathbf{P}(k)| - \mathit{nadmiar}$;

## Reprezentacja zbiorów wystąpień

Teraz zajmiemy się taką reprezentacją zbiorów $\mathbf{P}(k)$, by pojedyncze przecięcia tych zbiorów (w odpowiedniej kolejności) i liczenie nadmiaru były operacjami wykonywalnymi możliwie szybko.

Zdefiniujmy $v_i = (a \cdot i + b) \mod n$; wówczas $\mathbf{c}[i] = 0$ wtedy i tylko wtedy, gdy $v_i < p$. Przypomnijmy, że $a$ oraz $n$ są względnie pierwsze.

**Fakt 1.** *Liczby* $v_0, \ldots, v_{n-1}$ *są parami różne.*

**Dowód:** Załóżmy przez sprzeczność, że tak nie jest. To znaczy, że istnieją indeksy $0 \leqslant i < j < n$, takie że $v_i = v_j$. Wtedy $v_i = v_j \Leftrightarrow n|(a \cdot j + b) - (a \cdot i + b) \Leftrightarrow n|a(j-i)$. Skoro $a$ i $n$ są względnie pierwsze, to $n|j-i$, ale $0 < j - i < n$. Otrzymaliśmy sprzeczność dowodzącą prawdziwości faktu. ■

Skoro liczb $v_0, \ldots, v_{n-1}$ jest $n$, przyjmują wartości od 0 do $n-1$ i są wszystkie różne, to znaczy, że są permutacją liczb od 0 do $n-1$. Nie jest to jednak zupełnie dowolna permutacja. Aby to zauważyć, zastanówmy się, jak wygląda permutacja odwrotna do $v$, tzn. ciąg $\mathbf{C}_i$ oznaczający pozycję, na której w ciągu $v$ stoi wartość $i$ ($v_{\mathbf{C}_i} = i$).

*Odwrotnością a modulo n* nazywamy taką liczbę $r$, że $a \cdot r \equiv 1 \pmod{n}$. Jeśli $a$ jest względnie pierwsze z $n$, to jest ona wyznaczona jednoznacznie. Odwrotność $a$ modulo $n$ oznaczamy przez $a^{-1} \bmod n$. Można ją wyznaczyć w czasie $O(\log n)$ za pomocą rozszerzonego algorytmu Euklidesa.

**Fakt 2.** $\mathbf{C}_k = ((k-b) \cdot a^{-1}) \bmod n$*, a zatem ciąg* $\mathbf{C}$ *jest cyklicznym ciągiem arytmetycznym modulo* $n$ *o różnicy* $a^{-1} \bmod n$.

**Dowód:** $\mathbf{C}_k$ jest równe takiej wartości $i$, dla której $ai + b \equiv k \pmod{n}$. Szukana wartość $i$ to rzeczywiście $((k-b) \cdot a^{-1}) \bmod n$. To oznacza, że $\mathbf{C}_{k \oplus 1} - \mathbf{C}_k = a^{-1} \bmod n$, czyli $\mathbf{C}$ jest cyklicznym ciągiem arytmetycznym modulo $n$. ■

**Przykład 1.** Przyjrzyjmy się przykładowi z treści zadania. W tym przypadku ciąg $v$ to ciąg

$$6,\ 2,\ 7,\ 3,\ 8,\ 4,\ 0,\ 5,\ 1.$$

Ciąg $\mathbf{C}$ jest cyklicznym ciągiem arytmetycznym modulo 9 o różnicy $r = 5^{-1} \bmod 9 = 2$:

$$\underline{6 \to 8 \to 1 \to 3} \to \overline{5 \to 7 \to 0 \to 2 \to 4}.$$

Zauważmy, że $4 + 2 = 6$ (pierwszy element cyklu jest następnikiem ostatniego).

Na pozycjach w $\mathbf{c}$ odpowiadających pierwszym czterem elementom ciągu $\mathbf{C}$ są symbole 0, a na kolejnych pięciu symbole 1. Generowany tekst $\mathbf{c}$ to

$$1\,0\,1\,0\,1\,1\,0\,1\,0.$$

Jeśli $A$ jest podzbiorem $\{0, \ldots, n-1\}$ i $k \in \{0, \ldots, n-1\}$, to oznaczymy

$$A \ominus k = \{\, a \oplus (-k) \,:\, a \in A \,\}.$$

Analiza ciągu $\mathbf{C}$ prowadzi nas do eleganckiej charakteryzacji zbiorów $\mathbf{P}(k)$.

**Fakt 3.** $\mathbf{P}(k)$ *jest ciągiem arytmetycznym modulo* $n$ *o różnicy* $a^{-1} \bmod n$.

**Dowód:** Z definicji zbioru $\mathbf{P}(k)$ mamy:

$$\mathbf{P}(k) = \{\, i \,:\, w[k] = \mathbf{c}[i] \,\} \ominus k.$$

Dla $j \in \{0, 1\}$ oznaczmy przez $S^{(j)} = \{\, i \,:\, \mathbf{c}[i] = j \,\}$ zbiór pozycji tekstu $\mathbf{c}$, na których znajduje się cyfra $j$. Wówczas wzór na $\mathbf{P}(k)$ przyjmuje postać $\mathbf{P}(k) = S^{(w[k])} \ominus k$.

Wiemy, że $S^{(0)}$ jest prefiksem ciągu $\mathbf{C}$, a $S^{(1)}$ jest sufiksem ciągu $\mathbf{C}$. Każdy z nich jest więc ciągiem arytmetycznym modulo $n$ o różnicy $a^{-1} \bmod n$, więc po odjęciu od wszystkich jego wyrazów tej samej liczby $k$ wciąż jest takim właśnie ciągiem. ■

Na podstawie dowodu faktu łatwo podać konkretne wzory na długość oraz pierwszy wyraz ciągu arytmetycznego reprezentowanego przez $\mathbf{P}(k)$. Pozostawiamy to Czytelnikom jako ćwiczenie.

**Przykład 2.** Wróćmy jeszcze raz do przykładu z treści zadania. Mamy:

$$\mathbf{P}(0) = \{\, i \;:\; \mathbf{c}[i] = w[0] = 1 \,\} = \{\, 5, 7, 0, 2, 4 \,\}$$
$$\mathbf{P}(1) = \{\, i \;:\; \mathbf{c}[i \oplus 1] = w[1] = 0 \,\} = \{\, 6, 8, 1, 3 \,\} \ominus 1 = \{5, 7, 0, 2\}$$
$$\mathbf{P}(2) = \{\, i \;:\; \mathbf{c}[i \oplus 2] = w[2] = 1 \,\} = \{\, 5, 7, 0, 2, 4 \,\} \ominus 2 = \{3, 5, 7, 0, 2\}.$$

Mamy więc $\mathbf{P}(0) \cap \mathbf{P}(1) \cap \mathbf{P}(2) = \{0, 2, 5, 7\}$.

## Zliczanie wystąpień cyklicznych

Przedział cykliczny to taki, w którym po pozycji $n-1$ następują pozycje $0, 1, 2, \ldots$ Na przykład dla $n = 9$ przedział cykliczny $[7..2]$ odpowiada zbiorowi $\{7, 8, 0, 1, 2\}$. Odtąd zamiast zbiorów

$$\mathbf{P}(k) \;=\; \{\, i \;:\; w[k] = \mathbf{c}[i \oplus k] \,\}$$

będziemy posługiwać się zbiorami

$$\mathbf{V}(k) \;=\; \{\, v_i \;:\; w[k] = \mathbf{c}[i \oplus k] \,\}.$$

Innymi słowy, zbiór $\mathbf{V}(k)$ oznacza zakres indeksów w ciągu $\mathbf{C}$ odpowiadających elementom zbioru $\mathbf{P}(k)$. Na mocy faktu 3 mamy więc:

**Fakt 4.** *Zbiory* $\mathbf{V}(k)$ *są przedziałami cyklicznymi.*

**Przykład 3.** Dla poprzedniego przykładu mamy:

$$\mathbf{V}(0) = [4..8], \quad \mathbf{V}(1) = [4..7], \quad \mathbf{V}(2) = [3..7].$$

Ostatecznie $\mathbf{V}(0) \cap \mathbf{V}(1) \cap \mathbf{V}(2) = [4..7]$.

Zamiast przecinać zbiory $\mathbf{P}(k)$, będziemy przecinać zbiory $\mathbf{V}(k)$. Wyznaczenie przecięcia zbiorów $\mathbf{V}(k)$ sprowadza się do kolejnego wykonywania przecięcia pojedynczych przedziałów cyklicznych, co jest łatwe, o ile wynik też jest przedziałem cyklicznym. W naszym przypadku kolejność obliczania przecięć jest istotna. Skorzystamy z następującej oczywistej obserwacji.

**Obserwacja 2.** Przecięcie dwóch cyklicznych przedziałów zbioru $[0..n-1]$, których suma długości nie przekracza $n$, jest pojedynczym przedziałem cyklicznym.

W algorytmie startujemy z przedziałem $[0..n-1]$, a potem przecinamy go kolejno z przedziałami $\mathbf{V}(k)$, gdzie $w[k] = a$ oraz symbol $a$ jest rzadziej występującym w tekście $\mathbf{c}$. Skoro $a$ jest rzadziej występującym symbolem, to długość $\mathbf{V}(k)$ jest równa co najwyżej $\frac{n}{2}$, czyli suma długości takich przedziałów nie przekracza $n$. Następnie przecinamy rezultat kolejno ze zbiorami $\mathbf{V}(k)$ dla pozycji $k$ zawierających częstszy symbol. Z powyższej obserwacji wynika, że zawsze wynikiem będzie przedział cykliczny.

Rozmiar ostatniego przedziału daje nam liczbę wystąpień cyklicznych (pamiętamy, że każdej wartości $v$ odpowiada dokładnie jedna pozycja, na której w tekście występuje taka wartość).

## Zliczanie wystąpień nadmiarowych

Pozostaje jeszcze problem obliczenia liczby *nadmiar*. Możemy go rozwiązać, sprawdzając dla każdego $i > n - m$ (czyli dla pozycji blisko końca tekstu $\mathbf{c}$), czy $v_i$ należy do otrzymanego poprzednio przedziału wynikowego. Liczba pozycji, dla których to jest spełnione, jest równa liczbie *nadmiar*. Takich sprawdzeń jest $m$ i są one łatwe, ponieważ sprawdzamy, czy jakaś liczba należy do danego przedziału cyklicznego.

Otrzymaliśmy zatem rozwiązanie całego zadania działające w złożoności czasowej $O(m + \log n)$ (składnik $O(\log n)$ pochodzi z obliczania odwrotności $a$ modulo $n$) i pamięciowej $O(m)$.

## Alternatywne spojrzenie na przecinanie przedziałów cyklicznych

W poprzednim rozwiązaniu zawsze przecinaliśmy przedziały cykliczne o sumie długości nieprzekraczającej $n$, dzięki czemu mogliśmy stosować obserwację 2. Warto jednak wspomnieć, że istnieje ogólny algorytm przecinania przedziałów cyklicznych. Jeżeli mamy do czynienia z przedziałem cyklicznym $[a..b]$, takim że $a \leqslant b$, to reprezentujemy go po prostu jako przedział $[a..b]$, a jeżeli $a > b$, to reprezentujemy go jako sumę przedziałów $[a..n-1]$ oraz $[0..b]$. Następnie wszystkie standardowe przedziały użyte w reprezentacjach przedziałów cyklicznych (jest ich co najwyżej $2m$) umieszczamy na osi liczbowej, sortujemy niemalejąco ich początki i końce i przetwarzamy je wszystkie w takiej kolejności, utrzymując w każdym momencie liczbę otwartych przedziałów. Jeżeli w pewnym momencie przetwarzania jest ona równa $m$, to znaczy, że aktualnie rozpatrywany punkt należy do wszystkich przedziałów cyklicznych. Konkretniej, ostateczna reprezentacja takiego przecięcia będzie sumą (być może wielu) standardowych przedziałów, a ich łączna długość jest rozmiarem przecięcia zbiorów $\mathbf{V}(k)$.

Nie będąc świadomym, że otrzymane przedziały w naszym zadaniu tworzą jeden przedział cykliczny, trudniej jest wykluczyć nadmiarowe wystąpienia w rozsądnym czasie. Aby to zrobić, powinniśmy przetwarzać je razem ze zdarzeniami typu „początek przedziału" oraz „koniec przedziału", które obsługiwaliśmy przy wyznaczaniu przecięcia przedziałów cyklicznych.

To alternatywne rozwiązanie ma tę niewielką wadę, że ze względu na sortowanie działa w złożoności czasowej $O(m \log m + \log n)$.

**Jacek Tomasiewicz**
**Treść zadania, Opracowanie**

**Marek Sommer, Paweł Wegner**
**Program**

*XXII OI, etap II, dzień drugi, 12.02.2015*

# Trzy wieże

*Bitoni uwielbia się bawić. W swoim pokoju ułożył w jednym rzędzie n klocków. Każdy z klocków ma jeden z trzech kolorów: biały, szary lub czarny. Bitoni chciałby wybrać pewien spójny fragment rzędu klocków, a następnie z klocków z tego fragmentu zbudować wieże.*

*Każda wieża może się składać z klocków tylko jednego koloru i nie może być dwóch wież o tym samym kolorze (zatem Bitoni zbuduje co najwyżej trzy wieże). Ponadto nie może być dwóch wież o tej samej wysokości (tzn. każda wieża musi być zbudowana z innej liczby klocków niż pozostałe). Zakładamy, że Bitoni musi wykorzystać wszystkie wybrane przez siebie klocki. Pomóż Bitoniemu i napisz program, który znajdzie najdłuższy fragment rzędu klocków spełniający jego wymagania.*

## Wejście

*Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą n ($1 \leqslant n \leqslant 1\,000\,000$), oznaczającą liczbę klocków. Kolejny wiersz zawiera napis złożony z n liter $a_1a_2\ldots a_n$, w którym $a_i$ jest jedną z liter* `B`, `S` *lub* `C` *i oznacza kolor i-tego klocka w rzędzie (litera* `B` *oznacza klocek koloru białego, litera* `S` *klocek szary, a litera* `C` *klocek czarny).*

*W testach wartych 30% punktów zachodzi dodatkowy warunek $n \leqslant 2500$.*

## Wyjście

*Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą, równą liczbie klocków w najdłuższym spójnym fragmencie rzędu, który spełni wymagania Bitoniego.*

## Przykład

| *Dla danych wejściowych:* | *poprawnym wynikiem jest:* |
|---|---|
| `9`<br>`CBBSSBCSC` | `6` |

**Wyjaśnienie do przykładu:** *Bitoni może wybrać fragment złożony z 6 klocków:* `BSSBCS`*, z których zbuduje szarą wieżę złożoną z trzech klocków, białą z dwóch klocków oraz czarną z jednego klocka.*

**Testy „ocen":**

- `1ocen`*: $n = 2500$, rząd klocków jest następujący:* $\texttt{B}^{1248}\underline{\texttt{CSB}^{1250}}$ *(napis* $\texttt{B}^k$ *oznacza k-krotne powtórzenie litery* `B`*); najdłuższy fragment rzędu klocków, który Bitoni może wybrać, został podkreślony;*
- `2ocen`*: $n = 1\,000\,000$, rząd klocków jest okresowy:* `BSCBSCBSC`$\ldots$`BSCBSCB`*; Bitoni może zbudować tylko jedną wieżę z jednego klocka.*

# Rozwiązanie

W zadaniu mamy dane słowo $Z$ o długości $n$ składające się z literek `A`, `B`, `C` (dla czytelności opisu zamiast literki `S` używamy literki `A`) reprezentujących kolory kolejnych klocków ułożonych w rzędzie. Musimy znaleźć najdłuższy fragment ciągu klocków, taki że liczba wystąpień klocków każdego koloru w tym fragmencie będzie inna (tak aby wysokości wież zbudowanych z klocków tego samego koloru były różne). Fragment (niekoniecznie najdłuższy) spełniający ten warunek nazwiemy *poprawnym*.

## Rozwiązanie siłowe $O(n^3)$

Dla każdego spójnego fragmentu ciągu klocków możemy sprawdzić, czy jest on poprawny. Łatwo to wykonać w czasie liniowym: wystarczy policzyć i porównać liczbę wystąpień klocków każdego koloru. Jako że wszystkich spójnych fragmentów jest $O(n^2)$, a pojedyncze sprawdzenie zajmuje czas liniowy względem długości fragmentu, rozwiązanie to ma złożoność czasową $O(n^3)$.

Rozwiązanie to zaimplementowane jest w pliku `trzs2.cpp`. Za poprawne zaprogramowanie takiego rozwiązania na zawodach można było uzyskać około 15% punktów.

## Rozwiązanie wolne $O(n^2)$

Sprawdzenie poprawności fragmentu możemy wykonać szybciej, w czasie $O(1)$, po wcześniejszym przetworzeniu ciągu klocków w czasie $O(n)$. Aby móc efektywnie obliczać liczbę wystąpień klocków każdego koloru w dowolnym fragmencie, w pierwszym kroku wyznaczymy ciąg sum częściowych słowa $Z$ dla każdego koloru oddzielnie. Załóżmy, że rozpatrujemy kolor `A`.

Niech $w_i = 1$, jeśli $Z_i =$ `A`, zaś w przeciwnym przypadku $w_i = 0$. $i$-tą sumę częściową (dla $1 \leqslant i \leqslant n$) definiujemy jako $b_i = w_1 + w_2 + \ldots + w_i$, jednocześnie przyjmując $b_0 = 0$. Zauważmy, że $b_i = b_{i-1} + w_i$, więc ciąg $b_1, \ldots, b_n$ można obliczyć w czasie $O(n)$. Wartości $b_i$ pozwalają obliczyć liczbę wystąpień klocków koloru `A` w dowolnym fragmencie klocków $Z_i, \ldots, Z_j$ w czasie stałym ze wzoru $b_j - b_{i-1}$. W analogiczny sposób możemy wyznaczyć ciąg sum częściowych dla koloru `B` i `C`.

Implementacja takiego rozwiązania znajduje się w pliku `trzs3.cpp`. Rozwiązanie tego typu otrzymywało na zawodach około 30% punktów.

## Rozwiązanie wzorcowe $O(n)$

Udowodnimy, że optymalne wyniki znajdują się blisko jednego z końców ciągu klocków. Dokładniej, jeśli $[i, j]$ jest przedziałem reprezentującym optymalny fragment ciągu klocków (spośród wszystkich optymalnych wybierzmy ten o najmniejszym $i$), to $i \leqslant 3$ lub $j \geqslant n-2$. Jeśli tak rzeczywiście jest, to można wyznaczyć $O(n)$ kandydatów (przedziałów), wśród których znajduje się optymalny wynik. Te przedziały to $[1, j]$, $[2, j]$, $[3, j]$ oraz $[i, n]$, $[i, n-1]$, $[i, n-2]$ dla wszystkich $i, j$, takich że przedziały te będą poprawnie zdefiniowane – początek przedziału nie będzie większy od końca przedziału.

Po przetworzeniu ciągu w czasie $O(n)$, będziemy mogli w czasie stałym stwierdzić, czy przedział odpowiada poprawnemu fragmentowi (powiemy wtedy, że przedział jest poprawny). Ostatecznie, rozwiązanie to ma złożoność czasową $O(n)$ i zostało zaimplementowane w pliku `trz2.cpp`. Zaimplementowanie takiego rozwiązania w trakcie zawodów było nagradzane maksymalną liczbą punktów. Autorem tego rozwiązania jest Marek Sommer.

## Dowód

Niech przedział $[i, j]$ będzie najdłuższym poprawnym przedziałem, a spośród wszystkich najdłuższych niech będzie tym, który ma najmniejsze $i$. Spróbujemy założyć, że przedział nie znajduje się blisko żadnego z końców ciągu, czyli $i > 3$ i $j < n - 2$. To oznacza, że z każdej strony przedziału znajdują się przynajmniej po trzy klocki, których nie można dołożyć do tego przedziału.

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\cdots$ | ? | ? | ? | $Z_i$ | $Z_{i+1}$ | $\cdots$ | $Z_{j-1}$ | $Z_j$ | ? | ? | ? | $\cdots$ |

Dowód będzie opierał się na rozważeniu wielu przypadków (ale dzięki temu nie ma ich już w algorytmie) i pokazaniu, że w każdym z nich dochodzimy do sprzeczności – pokażemy, że będzie istniał lepszy lub równoważny przedział, który znajduje się blisko jednego z końców ciągu. Najpierw jednak zachęcamy Czytelnika do próby samodzielnego przeprowadzenia dowodu, który można wykonać w łatwy sposób, rozpisując różne przypadki na kartce. Poniżej przedstawiamy sposoby rozpatrzenia poszczególnych przypadków.

### W przedziale $[i, j]$ znajdują się klocki tylko jednego koloru

Mamy więc $Z_i = Z_{i+1} = \ldots = Z_{j-1} = Z_j$. Jeśli $i \neq j$, to dokładając element $Z_{i-1}$, dostaniemy również poprawny przedział (klocków koloru $Z_i$ jest co najmniej 2, więc nowy klocek tego samego bądź innego koloru niczego nie popsuje). Z tego wynika, że przedział $[i, j]$ nie jest najdłuższy poprawny – dostajemy sprzeczność. Jeśli natomiast $i = j$, to przedział jest długości 1. Dowolny przedział długości 1 jest poprawny, więc poprawny jest również przedział $[1, 1]$. Otrzymujemy sprzeczność, bo przedział $[i, j]$ miał być optymalnym rozwiązaniem o najmniejszym $i$.

### W przedziale $[i, j]$ znajdują się klocki dokładnie dwóch kolorów

W tym przypadku możemy przyjąć, że w przedziale $[i, j]$ znajdują się klocki trzech kolorów, przy czym jedna z utworzonych wież ma wysokość 0, co sprowadza się do następnego przypadku, opisanego poniżej.

### W przedziale $[i, j]$ znajdują się klocki dokładnie trzech kolorów

Niech $|X|$ oznacza liczbę wystąpień klocków koloru $X$ w przedziale $[i, j]$. Bez straty ogólności możemy założyć, że $|\texttt{A}| < |\texttt{B}| < |\texttt{C}|$. Co teraz może się kryć pod znakami zapytania?

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\cdots$ | $?_1$ | $?_2$ | $?_3$ | $Z_i$ | $Z_{i+1}$ | $\cdots$ | $Z_{j-1}$ | $Z_j$ | $?_4$ | $?_5$ | $?_6$ | $\cdots$ |

Znaki $?_3$ i $?_4$ nie mogą być równe `C` (w przeciwnym przypadku można by było powiększyć nimi rozwiązanie).

**Przypadek, w którym** $?_3 =$ `B`

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\cdots$ | $?_1$ | $?_2$ | `B` | $Z_i$ | $Z_{i+1}$ | $\cdots$ | $Z_{j-1}$ | $Z_j$ | $?_4$ | $?_5$ | $?_6$ | $\cdots$ |

Czy $?_4$ może być równe `B`? Nie, ponieważ wtedy przedział $[i, j]$ sąsiadowałby z dwiema literkami `B` i w zależności od tego, czy $|\texttt{B}| + 1 = |\texttt{C}|$, czy nie, moglibyśmy powiększyć rozwiązanie albo przy pomocy jednej, albo dwóch literek `B`. Znak $?_4$ nie może być też równy `C` (co było powiedziane wcześniej). Zatem otrzymujemy $?_4 =$ `A`.

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\cdots$ | $?_1$ | $?_2$ | `B` | $Z_i$ | $Z_{i+1}$ | $\cdots$ | $Z_{j-1}$ | $Z_j$ | `A` | $?_5$ | $?_6$ | $\cdots$ |

Wiadomo, że $|\texttt{A}| + 1 = |\texttt{B}|$ i $|\texttt{B}| + 1 = |\texttt{C}|$ (gdyby tak nie było, to można by było powiększyć rozwiązanie o jedną literkę `A` lub jedną literkę `B`). Możemy więc zapisać, że $|\texttt{A}| = x$, $|\texttt{B}| = x + 1$ i $|\texttt{C}| = x + 2$ dla pewnego $x$.

Czy $?_5$ może być równe `C`? Nie, ponieważ wtedy moglibyśmy dodać znaki $?_3$, $?_4$ i $?_5$, otrzymując lepsze rozwiązanie, gdzie $|\texttt{A}| = x + 1$, $|\texttt{B}| = x + 2$ i $|\texttt{C}| = x + 3$. Czy $?_5$ może być równe `B`? Nie, ponieważ wtedy moglibyśmy dodać znaki $?_3$, $?_4$ i $?_5$, otrzymując lepsze rozwiązanie, gdzie $|\texttt{A}| = x + 1$, $|\texttt{C}| = x + 2$ i $|\texttt{B}| = x + 3$. Zatem otrzymujemy $?_5 =$ `A`.

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\cdots$ | $?_1$ | $?_2$ | `B` | $Z_i$ | $Z_{i+1}$ | $\cdots$ | $Z_{j-1}$ | $Z_j$ | `A` | `A` | $?_6$ | $\cdots$ |

$?_6$ nie może być równe `A` ani `C`, ponieważ dodając znaki $?_4$, $?_5$ i $?_6$, otrzymalibyśmy lepsze rozwiązanie. Zatem otrzymujemy $?_6 =$ `B`.

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\cdots$ | $?_1$ | $?_2$ | `B` | $Z_i$ | $Z_{i+1}$ | $\cdots$ | $Z_{j-1}$ | $Z_j$ | `A` | `A` | `B` | $\cdots$ |

$?_2$ nie może być równe `B` ani `C`, ponieważ dodając znaki $?_2$, $?_3$ i $?_4$, otrzymalibyśmy lepsze rozwiązanie. Zatem otrzymujemy $?_2 =$ `A`.

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\cdots$ | $?_1$ | `A` | `B` | $Z_i$ | $Z_{i+1}$ | $\cdots$ | $Z_{j-1}$ | $Z_j$ | `A` | `A` | `B` | $\cdots$ |

$?_1$ nie może być równe `B` ani `C`, ponieważ dodając znaki $?_1$, $?_2$ i $?_3$, otrzymalibyśmy lepsze rozwiązanie. Zatem otrzymujemy $?_1 =$ `A`.

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\cdots$ | `A` | `A` | `B` | $Z_i$ | $Z_{i+1}$ | $\cdots$ | $Z_{j-1}$ | $Z_j$ | `A` | `A` | `B` | $\cdots$ |

Okazuje się, że dodając wszystkie znaki od $?_1$ do $?_6$, otrzymalibyśmy lepsze rozwiązanie, w którym $|\texttt{C}| = x + 2$, $|\texttt{B}| = x + 3$ i $|\texttt{A}| = x + 4$, więc mamy sprzeczność.

**Przypadek, w którym** $?_3 = \texttt{A}$

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\cdots$ | $?_1$ | $?_2$ | A | $Z_i$ | $Z_{i+1}$ | $\cdots$ | $Z_{j-1}$ | $Z_j$ | $?_4$ | $?_5$ | $?_6$ | $\cdots$ |

Gdyby $?_4 = \texttt{B}$, to otrzymalibyśmy przypadek symetryczny do poprzedniego przypadku, w którym $?_3 = \texttt{B}$ (który był sprzeczny), więc $?_4$ nie może być równe B ani też C (co było powiedziane wcześniej). Zatem otrzymujemy $?_4 = \texttt{A}$.

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\cdots$ | $?_1$ | $?_2$ | A | $Z_i$ | $Z_{i+1}$ | $\cdots$ | $Z_{j-1}$ | $Z_j$ | A | $?_5$ | $?_6$ | $\cdots$ |

Wiadomo, że $|\texttt{A}| + 1 = |\texttt{B}|$ i $|\texttt{A}| + 2 = |\texttt{C}|$ (gdyby tak nie było, to można by było dodać jedną lub dwie literki A, otrzymując lepsze rozwiązanie). Możemy więc zapisać, że $|\texttt{A}| = x$, $|\texttt{B}| = x + 1$ i $|\texttt{C}| = x + 2$, dla pewnego $x$.

$?_5$ nie może być równe A ani C, ponieważ dodając znaki $?_3$, $?_4$ i $?_5$, otrzymalibyśmy lepsze rozwiązanie. Zatem otrzymujemy $?_5 = \texttt{B}$.

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\cdots$ | $?_1$ | $?_2$ | A | $Z_i$ | $Z_{i+1}$ | $\cdots$ | $Z_{j-1}$ | $Z_j$ | A | B | $?_6$ | $\cdots$ |

Znak $?_2$ jest w tym przypadku symetryczny do $?_5$, więc tak samo możemy wywnioskować, że $?_2 = \texttt{B}$.

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\cdots$ | $?_1$ | B | A | $Z_i$ | $Z_{i+1}$ | $\cdots$ | $Z_{j-1}$ | $Z_j$ | A | B | $?_6$ | $\cdots$ |

$?_6$ nie może być równe B ani C, ponieważ dodając znaki $?_4$, $?_5$ i $?_6$, otrzymalibyśmy lepsze rozwiązanie. Zatem otrzymujemy $?_6 = \texttt{A}$.

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\cdots$ | $?_1$ | B | A | $Z_i$ | $Z_{i+1}$ | $\cdots$ | $Z_{j-1}$ | $Z_j$ | A | B | A | $\cdots$ |

Znak $?_1$ jest w tym przypadku symetryczny do $?_6$, więc tak samo możemy wywnioskować, że $?_1 = \texttt{A}$.

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\cdots$ | A | B | A | $Z_i$ | $Z_{i+1}$ | $\cdots$ | $Z_{j-1}$ | $Z_j$ | A | B | A | $\cdots$ |

Okazuje się, że dodając wszystkie znaki od $?_1$ do $?_6$, otrzymalibyśmy lepsze rozwiązanie, w którym $|\texttt{C}| = x + 2$, $|\texttt{B}| = x + 3$ i $|\texttt{A}| = x + 4$, więc mamy sprzeczność.

## Mniejsza liczba kandydatów

Udowodniliśmy, że wystarczy sprawdzić przedziały, które są w odległości nie większej niż 2 od któregoś z końców ciągu klocków. Jest to najmniejsze możliwe ograniczenie. Gdyby algorytm sprawdzał tylko przedziały w odległości nie większej niż 1, to źle odpowiedziałby w następującym przypadku:

ABCACBACBA

Nie wystarczy również sprawdzać przedziałów znajdujących się blisko początku:

BBBBBBCAACBBBBBBBBBBBBBBBBBBB

# Zawody III stopnia

opracowania zadań

**Wojciech Nadara** — Treść zadania, Opracowanie
**Jan Kanty Milczek** — Program

**Dostępna pamięć: 256 MB.** *OI, etap III, dzień próbny, 15.04.2015*

# Odwiedziny

*Bajtazar to wyjątkowy człowiek – przez ostatnie 21 lat był listonoszem, bankierem, łyżwiarzem, a nawet królem! Nic dziwnego, że ma mnóstwo znajomych. Niestety, ciągłe zmiany miejsc pracy sprawiły, że z wieloma z nich zaczął tracić kontakt... Czas to zmienić! Bajtazar wykona wielkie tournée po Bajtocji, aby odnowić stare znajomości.*

*W Bajtocji znajduje się $n$ miast, połączonych siecią $n-1$ dwukierunkowych dróg. Nasz bohater chce odwiedzić każde z miast kraju i ustalił już konkretną kolejność wizyt. Trasę między każdymi dwoma kolejnymi miastami pokona, korzystając z samochodu wypożyczonego w BMW (Bajtockiej Motoryzacji Wycieczkowej). Wypożyczenie każdego samochodu nie kosztuje nic, ale auta trzeba tankować – samochód o pojemności baku $k$ trzeba zatankować w mieście początkowym trasy i każdorazowo po przejechaniu dokładnie $k$ dróg. BMW, znając plan tournée Bajtazara oraz wiedząc, że każdą trasę będzie on chciał pokonać jak najszybciej, tak dobrało pojemność baków wypożyczanych samochodów, aby musiał on każdy z nich zatankować również w mieście docelowym.*

*Znając kolejność, w jakiej Bajtazar odwiedzi miasta, ceny tankowania w każdym z nich oraz pojemności baków wypożyczanych samochodów, wyznacz, ile będzie go kosztowało przejechanie każdej trasy.*

## Wejście

*W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita $n$ ($2 \leqslant n \leqslant 50\,000$), oznaczająca liczbę miast w Bajtocji. Miasta są numerowane od 1 do $n$. W kolejnym wierszu znajduje się ciąg $n$ liczb całkowitych $c_1, \ldots, c_n$ ($1 \leqslant c_i \leqslant 10\,000$) pooddzielanych pojedynczymi odstępami, oznaczających ceny paliwa w miastach Bajtocji: liczba $c_i$ oznacza koszt napełnienia baku dowolnego samochodu w mieście o numerze $i$.*

*Dalej następuje $n-1$ wierszy zawierających opisy dróg w Bajtocji. W każdym z nich podane są dwie liczby całkowite $a$, $b$ ($1 \leqslant a, b \leqslant n$) oddzielone pojedynczym odstępem, oznaczające, że w Bajtocji istnieje dwukierunkowa droga łącząca miasta o numerach $a$ i $b$.*

*W następnym wierszu znajduje się ciąg $n$ liczb całkowitych $t_1, \ldots, t_n$ pooddzielanych pojedynczymi odstępami, opisujący kolejność, w jakiej Bajtazar ma zamiar odwiedzić miasta (każda z liczb od 1 do $n$ pojawi się w tym ciągu dokładnie raz). Ostatni wiersz wejścia zawiera ciąg $n-1$ liczb całkowitych $k_1, \ldots, k_{n-1}$ pooddzielanych pojedynczymi odstępami, opisujący pojemności baków wypożyczanych samochodów: liczba $k_i$ oznacza, że podczas przejazdu z miasta o numerze $t_i$ do miasta o numerze $t_{i+1}$, Bajtazar będzie musiał tankować samochód co $k_i$ dróg. Możesz założyć, że $k_i$ zawsze dzieli odległość między tymi miastami.*

*W testach wartych 28% punktów zachodzi dodatkowy warunek $n \leqslant 1000$, a w testach wartych 36% punktów zachodzi dodatkowy warunek $n \leqslant 10\,000$.*

### Wyjście

*Na standardowe wyjście Twój program powinien wypisać $n-1$ wierszy, w każdym po jednej liczbie całkowitej. Liczba w $i$-tym wierszu ma oznaczać łączny koszt tankowania podczas trasy z miasta o numerze $t_i$ do miasta o numerze $t_{i+1}$.*

### Przykład

| *Dla danych wejściowych:* | *poprawnym wynikiem jest:* |
|---|---|
| 5 | 10 |
| 1 2 3 4 5 | 6 |
| 1 2 | 10 |
| 2 3 | 5 |
| 3 4 | |
| 3 5 | |
| 4 1 5 2 3 | |
| 1 3 1 1 | |

## Rozwiązanie

Przedstawimy rozwiązanie nieco ogólniejszego problemu. Dane jest drzewo o $n$ wierzchołkach (czyli spójny graf o $n$ wierzchołkach i $n-1$ krawędziach). Każdy wierzchołek ma przypisaną wagę. Dodatkowo dana jest również lista $q$ *zapytań*. Każde zapytanie opisane jest za pomocą pary wierzchołków $u$ i $v$ oraz liczby $k$. Takie zapytanie polega na obliczeniu sumy wag wierzchołków leżących na ścieżce łączącej $u$ i $v$, przy czym do sumy wliczamy jedynie wagi co $k$-tego wierzchołka. Możemy przy tym założyć, że odległość między $u$ i $v$ jest podzielna przez $k$. W dalszym opisie załóżmy, że $i$-te zapytanie dane jest za pomocą trzech parametrów: $u_i$, $v_i$ oraz $k_i$.

### Rozwiązanie brutalne

Zacznijmy od dosyć bezpośredniego rozwiązania. Zauważmy, że dla ustalonego $k$, w algorytmie przeszukiwania w głąb (DFS) możemy bez trudu utrzymywać sumę wag co $k$-tego wierzchołka na ścieżce od korzenia do aktualnie przetwarzanego wierzchołka. Dla każdego z $q$ zapytań uruchamiamy więc algorytm DFS startujący w $u_i$ i gdy tylko dotrzemy do $v_i$ (który leży na głębokości podzielnej przez $k$), możemy odczytać w nim wynik zapytania. To bardzo proste rozwiązanie działa w złożoności czasowej $O(nq)$ i pamięciowej $O(n)$, jednak niestety nie przydaje się zbytnio na drodze do szybszego rozwiązania.

### Początkowe przemyślenia

Zadanie jest daleko idącym uogólnieniem klasycznego problemu, w którym dany jest ciąg liczb, a naszym celem jest odpowiadanie na zapytania dotyczące sumy liczb

z podanego fragmentu tego ciągu. Problem ten ma dwa standardowe rozwiązania. Pierwsze z nich opiera się na obliczeniu sum prefiksowych ciągu, natomiast drugie korzysta z drzewa przedziałowego. Drugie z tych rozwiązań jest istotnie bardziej skomplikowane oraz wolniejsze. Warto je stosować dopiero w przypadku, gdy w oryginalnym problemie zostały wprowadzone pewne utrudnienia, na przykład liczby w danym ciągu mogą się zmieniać w czasie. W naszym zadaniu musimy poradzić sobie z dwiema przeszkodami. Po pierwsze, rozważamy dowolne drzewo, a nie zwykły ciąg (który odpowiadałby bardzo specyficznemu drzewu). Po drugie, sumujemy wagi nie każdego, lecz co $k$-tego wierzchołka.

Aby dojść do rozwiązania naszego zadania, zastanówmy się, jak wcześniej wspomniane metody radzą sobie z tymi utrudnieniami. Po chwili namysłu można dojść do wniosku, że drzewo przedziałowe w przypadku naszego zadania nie ma przewagi nad sumami prefiksowymi – jego główna zaleta (możliwość modyfikacji elementów w czasie) nie przydaje się. Możemy zatem zastanowić się, jak zmodyfikować rozwiązanie używające sum prefiksowych, aby stosowało się do naszego zadania.

Zajmijmy się na chwilę jedynie zapytaniami z $k = 1$. Ukorzeńmy rozważane drzewo w dowolnym wierzchołku. Wówczas ścieżkę $u - v$ między dowolnymi dwoma wierzchołkami $u$ i $v$ można rozbić na dwie ścieżki $u - s$ i $s - v$, gdzie $s$ jest *najniższym wspólnym przodkiem* wierzchołków $u$ i $v$ (ang. *lowest common ancestor*, *LCA*). Co więcej, ścieżki te biegną od $u$ i $v$ cały czas w stronę korzenia. Dla wygody oznaczmy przez $s = LCA(u, v)$ najniższego wspólnego przodka wierzchołków $u$ i $v$. Teraz korzystamy z techniki sum prefiksowych. Dla każdego wierzchołka $v$ utrzymujemy sumę wag $suma[v]$ wierzchołków na ścieżce od $v$ do korzenia drzewa. Aby wyznaczyć sumę wag wierzchołków na ścieżce od $u$ do $v$, wystarczy teraz obliczyć $suma[u] + suma[v] - 2 \cdot suma[LCA(u, v)] + waga[LCA(u, v)]$.

Ten pomysł, połączony ze strukturą danych pozwalającą efektywnie obliczać $LCA(u, v)$, pozwala rozwiązać zadanie w przypadku $k = 1$. Całkiem dobrze poradziliśmy sobie z faktem, że musimy pracować z drzewem a nie ciągiem liczb. Jednak aby rozwiązać zadanie dla dowolnych wartości $k$, będziemy potrzebowali całkiem nowego pomysłu.

## Rozwiązanie wzorcowe

W naszym rozwiązaniu wielokrotnie będziemy potrzebować odpowiedzi na zapytania postaci „Czy wierzchołek $a$ jest przodkiem wierzchołka $b$?”. Problem ten daje się rozwiązać dosyć standardową techniką. Uruchamiamy algorytm DFS i dla każdego wierzchołka zapamiętujemy czas wejścia i wyjścia. Znając te wartości, na wspomniane zapytania możemy odpowiadać w czasie stałym.

Jak już wspomnieliśmy, będziemy operowali na ścieżkach w drzewie, dlatego przyda nam się także struktura, która pozwala odpowiadać na zapytania o najniższego wspólnego przodka dowolnych dwóch wierzchołków. Można ją zaimplementować na kilka różnych sposobów, jednak, jak się za chwilę okaże, w naszym przypadku jeden z nich będzie szczególnie wygodny.

Podzielmy zapytania na dwie grupy: te z dużymi parametrami $k_i$ (dalej zwane *dużymi*) i te z małymi parametrami $k_i$ (dalej zwane *małymi*). Parametr $k_i$ uznajemy

za duży, gdy $k_i \geqslant t$, przy czym wartość $t$ dobierzemy później. Odpowiedzi na duże i małe zapytania będziemy uzyskiwać na różne sposoby.

### Duże zapytania

Zajmijmy się najpierw dużymi zapytaniami. Zauważmy, że jeżeli $k_i \geqslant t$, to poszukiwana suma składa się z co najwyżej $\frac{n}{t} + 1$ wag wierzchołków. Zastosujemy podejście brutalne: składniki będziemy dodawać po jednym. Jak już wspomnieliśmy, każdą ścieżkę w ukorzenionym drzewie między dwoma wierzchołkami $u_i$ oraz $v_i$ można rozbić na dwie ścieżki: od $u_i$ do $LCA(u_i, v_i)$ oraz od $LCA(u_i, v_i)$ do $v_i$ (potencjalnie któraś z tych części składa się z tylko jednego wierzchołka). Każdą taką ścieżką zajmiemy się osobno. Wierzchołki, które musimy wliczyć do sumy na takiej ścieżce, można uzyskać w taki sposób, że zaczynamy od (dla ustalenia uwagi) $u_i$ i skaczemy co $k_i$ wierzchołków w górę drzewa, do momentu, w którym znajdziemy się powyżej $LCA(u_i, v_i)$. To, czy przeskoczyliśmy $LCA(u_i, v_i)$, czy nie, sprawdzamy, pytając, czy nasz aktualny wierzchołek jest przodkiem $v_i$. Pozostaje nam rozwiązać następujący problem: jak odpowiadać na zapytania postaci „Jaki jest wierzchołek o $k$ w górę od wierzchołka $v$?". Standardowe rozwiązanie korzysta z tzw. *jump pointers*, których używa się w jednym ze sposobów obliczania LCA, jednak ich wykorzystanie spowoduje, że na takie pytania będziemy odpowiadać w czasie $O(\log k)$. Zastosujemy zatem inne, szybsze podejście.

Dla każdego z dużych zapytań $u_i$, $v_i$, $k_i$ zapiszmy w wierzchołkach drzewa $u_i$ oraz $v_i$, że należą one do $i$-tego zapytania. Ukorzeńmy drzewo w dowolnym wierzchołku. Następnie uruchommy algorytm DFS, który w pomocniczej tablicy utrzymuje numery wierzchołków na ścieżce od korzenia do aktualnie rozpatrywanego wierzchołka. Aktualizacja tej tablicy przy zejściu w głąb wymaga jedynie dopisania jednego numeru wierzchołka na końcu tablicy. Z kolei po powrocie z wywołania rekurencyjnego wystarczy usunąć z niej ostatni element. Mając taką tablicę pomocniczą, w dowolnym momencie, dla dowolnego parametru $k$ możemy odczytać z niej numer wierzchołka, który leży dokładnie $k$ poziomów powyżej aktualnie rozpatrywanego wierzchołka. Taką pomocniczą tablicę można zaimplementować za pomocą zwykłej tablicy pamiętającej indeks ostatniego elementu lub struktury `vector` z C++.

Pokażemy teraz, jak to wykorzystać w naszym algorytmie. W momencie, gdy przeszukiwanie *powraca* z wierzchołka $u_i$ (lub odpowiednio $v_i$), dodajmy wagę wierzchołka $u_i$ (lub $v_i$) do licznika odpowiadającego za odpowiedź na $i$-te zapytanie. Następnie w wierzchołku, który jest o $k$ w górę od $u_i$ (dostęp do takiego mamy teraz w czasie stałym!), zapiszmy, że mamy go później przetworzyć w taki sam sposób, w jaki przetworzyliśmy $u_i$ (o ile jeszcze nie leży on powyżej $LCA(u_i, v_i)$). W taki sposób na każdy z wierzchołków, które musimy wliczyć do odpowiedzi, poświęcamy stałą liczbę operacji, zatem odpowiedzi na wszystkie duże zapytania jesteśmy w stanie wyznaczyć w czasie $O(\frac{nq}{t})$. Należy przy tym uważać na mały szczegół techniczny: łatwo pomylić się i wagę LCA policzyć w tej sumie dwukrotnie.

Warto tu jeszcze zwrócić uwagę na złożoność pamięciową. Każde duże zapytanie w dowolnym momencie wykonania algorytmu odpowiada za istnienie co najwyżej dwóch elementów na listach zapytań, które utrzymujemy w wierzchołkach. Łącznie daje to $O(q)$ elementów. Jeśli jednak implementujemy algorytm w języku C++ i do przechowywania tych list używamy struktur `vector`, możemy wpaść w pułapkę. Struktura `vector` nie gwarantuje, że zajmuje pamięć proporcjonalną do długości *aktualnej* listy

elementów. Pesymistycznie może zajmować pamięć proporcjonalną do maksymalnego *dotychczas* osiągniętego rozmiaru. To powoduje, że złożoność pamięciowa naszego programu niepotrzebnie wzrasta do $O(\frac{nq}{t})$. Aby pozbyć się tego problemu, po przetworzeniu listy z danego wierzchołka powinniśmy wyczyścić wektor i wywołać metodę `shrink_to_fit`, która powoduje, że wektor zwalnia nieużywaną pamięć. To poprawia złożoność pamięciową do $O(q)$.

### Małe zapytania

Zajmijmy się teraz małymi zapytaniami. Tu przyda nam się idea sum prefiksowych. Załóżmy, że dla każdego $k < t$ i każdego wierzchołka $v$ znamy sumę wag co $k$-tego wierzchołka na ścieżce do korzenia (sumę prefiksową). Oznaczmy tę wartość przez $suma[v][k]$. Wówczas odpowiedź na zapytanie możemy wyznaczyć następująco. Ponownie rozdzielamy ścieżkę z $u_i$ do $v_i$ na dwie ścieżki od $u_i$ i $v_i$ do $LCA(u_i, v_i)$ i następnie obliczamy sumę odpowiednich wag na każdej z nich za pomocą uprzednio spamiętanych sum prefiksowych. Trzeba przy tym uważać, aby wagi $LCA(u_i, v_i)$ nie policzyć dwa razy.

Aby obliczyć sumę wag na ścieżce od $u_i$ do $LCA(u_i, v_i)$, znajdujemy najniższego przodka $u_i$, który leży powyżej $LCA(u_i, v_i)$ i którego odległość od $u_i$ jest podzielna przez $k_i$. Oznaczmy znaleziony wierzchołek przez $w$. Wówczas suma wag na ścieżce od $u_i$ do $LCA(u_i, v_i)$ to $suma[u_i][k_i] - suma[w][k_i]$ (jeśli $w$ nie istnieje, szukana wartość to $suma[u_i][k_i]$). Zauważmy, że ponownie pojawia się tutaj pytanie o przodka ustalonego wierzchołka na danej głębokości, zatem i tu posłużymy się algorytmem DFS z pomocniczą tablicą. Jeśli znamy $LCA(u_i, v_i)$ i głębokości wierzchołków, wyznaczenie takiego $w$ nie stanowi problemu. Nasze zadanie sprowadziliśmy zatem do problemu obliczenia tablicy $suma$, co już jest dosyć proste. Uruchamiamy algorytm DFS z pomocniczą tablicą i przy rozpatrywaniu wierzchołka $v$ dla wszystkich liczb $k$ z przedziału $[1, t)$ uaktualniamy $suma[v][k] := waga[v] + suma[s][k]$, gdzie $s$ jest przodkiem $v$ odległym o $k$ (jeśli taki wierzchołek $s$ nie istnieje, wówczas po prostu przyjmujemy $suma[v][k] = waga[v]$).

Takim oto sposobem uzyskaliśmy rozwiązanie tej części zadania w złożoności czasowej $O(nt+q \log n)$ i pamięciowej $O(nt)$. Składnik $O(q \log n)$ pochodzi z wyznaczania LCA – o czym piszemy w następnej sekcji.

### Wyznaczanie LCA

Jak już się przekonaliśmy, w całym naszym rozwiązaniu wykorzystujemy algorytm DFS pamiętający ścieżkę od korzenia. Jeśli mamy tę ścieżkę oraz funkcję odpowiadającą w czasie stałym na pytanie „Czy $a$ jest przodkiem $b$?", wyznaczenie $LCA(u, v)$ jest bardzo proste – w momencie, gdy algorytm DFS odwiedza wierzchołek $u$, wystarczy na ścieżce od korzenia do $u$ znaleźć najniższy wierzchołek, który jest przodkiem $v$. Ścieżkę od korzenia do $u$ mamy wówczas zapisaną w tablicy, dlatego możemy w tym celu użyć wyszukiwania binarnego. Co ciekawe, jeżeli przyjrzymy się bliżej naszemu algorytmowi, okaże się, że nigdy nie potrzebujemy obliczać samego LCA. W przypadku dużych zapytań skakaliśmy co $k$, aż znaleźliśmy się w przodku drugiego wierzchołka z zapytania, a w przypadku małych możemy bezpośrednio od razu znaleźć $w$, szukając

najniższego wierzchołka będącego przodkiem $v_i$ (którego poprzednik na ścieżce nie jest przodkiem $v_i$) i rozpatrując jedynie przodków $u_i$ w odległości podzielnej przez $k_i$.

### Dobór parametru $t$

Umiemy już w miarę szybko odpowiadać zarówno na duże jak i na małe zapytania. Pozostało jedynie dobrać optymalną wartość parametru $t$. Na duże zapytania odpowiadamy w sumarycznym czasie $O(\frac{nq}{t})$, a na małe w czasie $O(nt + q \log n)$, przez co optymalna wartość $t$ wynosi $\sqrt{q}$. Wówczas otrzymujemy rozwiązanie o złożoności czasowej $O(n\sqrt{q} + q \log n)$ oraz złożoności pamięciowej $O(n\sqrt{q})$. Doświadczeni zawodnicy pewnie od początku przeczuwali, że w złożoności pojawi się pierwiastek z $q$ lub z $n$. W takich sytuacjach warto jednak nie polegać ślepo na intuicji i przeprowadzić dokładną analizę w zależności od wprowadzonego parametru. Może się na przykład zdarzyć, że czasy działania dwóch faz to, powiedzmy, $O(nt)$ i $O(\frac{n}{t} \log n)$. Wtedy dla $t = \sqrt{n}$ czas działania całego algorytmu to $O(n\sqrt{n} \log n)$, jednak dla $t = \sqrt{n \log n}$ cały algorytm działa w czasie $O(n\sqrt{n \log n})$, czyli lepszym o czynnik $\sqrt{\log n}$. Taka różnica może być kluczowa do otrzymania kompletu punktów za rozwiązanie.

## Redukcja złożoności pamięciowej

Warto wspomnieć, że złożoność pamięciową można zmniejszyć do liniowej, choć to usprawnienie nie było wymagane od zawodników. Opisaliśmy już, jak ustrzec się przed zwiększonym zużyciem pamięci w przypadku rozpatrywania dużych zapytań, zatem czemu nie spróbować zredukować jej w przypadku małych zapytań? Aby odpowiadać na małe zapytania, najpierw wykonaliśmy kosztowny preprocessing w czasie $O(n\sqrt{q})$, a potem raz przetworzyliśmy zapytania w czasie $O(q \log n + n)$, aby uzyskać wszystkie odpowiedzi. Nie trzeba jednak deklarować od razu wielkiej tablicy *suma* zajmującej $O(n\sqrt{q})$ pamięci. Cały algorytm można podzielić na $t$ faz, gdzie w $p$-tej fazie odpowiadamy na zapytania, w których $k_i = p$. Przetwarzając wszystkie zapytania z ustalonym $k$, potrzebujemy jedynie sum prefiksowych skaczących co $k$, co redukuje pamięć do $O(n)$. Ostatecznie otrzymujemy algorytm rozwiązujący zadanie w złożoności czasowej $O(n\sqrt{q} + q \log n)$ i pamięciowej $O(n + q)$.

## Możliwe udogodnienia implementacyjne

W rozwiązaniu wzorcowym wielokrotnie używaliśmy zapytań postaci „Jaki jest przodek wierzchołka $v$ w odległości $k$?", na które odpowiadaliśmy za pomocą algorytmu DFS z pomocniczą tablicą. Wymagało to uprzedniego wczytania wszystkich zapytań i dzielenia algorytmu na poszczególne fazy. Co zrobić, aby na wspomniane wyżej zapytania odpowiadać *online*? Standardowym rozwiązaniem tego problemu jest metoda *jump pointers*, która jest chyba najpopularniejszym sposobem obliczania $LCA$. Na początku dla każdej pary $(v, d)$ wyznaczamy przodka $v$ w odległości $2^d$ (ograniczamy się do takich $d$, że $v$ leży na głębokości co najmniej $2^d$). Zauważmy, że wyniki dla par $(\cdot, d+1)$ łatwo obliczyć na podstawie wyników dla par $(\cdot, d)$. Mając takie informacje, jesteśmy w stanie odpowiadać na pytania postaci „Jaki jest przodek wierzchołka $v$

w odległości $k$?" w złożoności czasowej $O(\log n)$, zapisując $k$ w systemie binarnym. Takie rozwiązanie skutkuje zwiększeniem złożoności pamięciowej do $O(n \log n)$ (co jednak nie jest wielką przeszkodą, bo nawet $O(n\sqrt{n})$ było dopuszczane), ale także złożoności czasowej do $O(n\sqrt{q} \log n)$. Jednakże limity czasu i pamięci były na tyle duże, że i takie rozwiązania dostawały na zawodach komplet punktów.

## Rozwiązanie alternatywne

Opiszemy jeszcze stosunkowo proste i bardzo pomysłowe rozwiązanie alternatywne. Wcześniej rozpatrywaliśmy tablicę *suma* o wymiarach $n \times \sqrt{q}$, gdyż ograniczyliśmy się do wartości $k < \sqrt{q}$. Teraz pozbądźmy się tego ograniczenia i wyobraźmy sobie, że używamy analogicznej tablicy o wymiarach $n \times n$. Wartości tej tablicy będziemy wyznaczali jedynie tam, gdzie jest to potrzebne. Nie będziemy także oczywiście tak wielkiej tablicy deklarowali jawnie. Zamiast tego posłużymy się słownikową strukturą danych. W C++ najlepiej do tego celu nadaje się `unordered_map`, będąca implementacją tablicy z haszowaniem.

Zdefiniujmy funkcję $F(a, k)$, która oblicza to samo, co w poprzednim rozwiązaniu znaczyło $suma[a][k]$, czyli sumę wag co $k$-tych wierzchołków na drodze do korzenia, zaczynając od wierzchołka $a$. Obliczmy $F(a, k)$ za pomocą następującego algorytmu: jeżeli już kiedyś obliczyliśmy $F(a, k)$, to podajemy zapamiętany wynik, a jeżeli nie, to wynikiem jest $waga[a] + F(b, k)$, gdzie $waga[a]$ to oczywiście waga wierzchołka $a$, a $b$ jest $k$-tym przodkiem $a$ (o ile istnieje). Zauważmy, że w sumie w całym naszym programie funkcja $F$ nie zostanie wywołana więcej niż $O(q + n\sqrt{q})$ razy! Dzieje się tak z tego samego powodu, z którego poprzednie rozwiązanie działało szybko, tzn. oddzielnie szacujemy liczbę wywołań $F$ dla małych i dużych zapytań. Zaletą takiego podejścia jest jednak to, że podział na małe i duże zapytania odbywa się jedynie w analizie złożoności czasowej (i pamięciowej) i nie musimy się nim przejmować w implementacji.

Wyznaczanie przodka w odległości $k$ możemy zrealizować albo w czasie logarytmicznym za pomocą wcześniej opisanego sposobu, albo w czasie stałym przy użyciu algorytmu DFS z pomocniczą tablicą. Struktura `unordered_map` działa z kolei w oczekiwanym czasie stałym. Zatem w ten sposób możemy otrzymać rozwiązanie w złożoności czasowej $O(n\sqrt{q})$. Jego jedyną wadą w porównaniu do poprzedniego jest gorsza złożoność pamięciowa – $O(n\sqrt{q})$. Oczywiście trzeba tu jeszcze zadbać o kilka kwestii podobnie jak w poprzednim rozwiązaniu, np. wyznaczanie pierwszego wierzchołka ponad LCA w odległości podzielnej przez $k$.

**Michał Włodarczyk**
Treść zadania, Opracowanie

**Karol Pokorski**
Program

**Dostępna pamięć: 256 MB.** *OI, etap III, dzień pierwszy, 16.04.2015*

# Myjnie

*Bajtazar ma zamiar wystartować w przetargu na zbudowanie n myjni samochodowych wzdłuż głównej drogi ekspresowej w Bajtocji. Zanim jednak to zrobi, chciałby przeliczyć, czy to przedsięwzięcie mu się opłaci.*

*W tym celu zlecił profesjonalnej firmie badanie rynku. Z wyników badań wynika, że drogą będzie jeździć m potencjalnych klientów oraz że i-ty będzie jeździł odcinkiem pomiędzy myjniami $a_i$ i $b_i$ (włącznie) oraz jest zainteresowany myciem samochodu, o ile cena w myjni nie będzie wyższa niż $c_i$ bajtalarów. Bajtazar w każdej myjni ustala cenę niezależnie. Zakładając, że każdy klient wybierze najtańszą myjnię na swojej trasie (lub nie wybierze żadnej, jeżeli we wszystkich ceny przekroczą jego budżet), Bajtazar chce dobrać takie ceny w myjniach, by zmaksymalizować swój sumaryczny zysk.*

## Wejście

*Pierwszy wiersz standardowego wejścia zawiera dwie liczby całkowite n i m ($1 \leqslant n \leqslant 50$, $1 \leqslant m \leqslant 4000$) oddzielone pojedynczym odstępem, oznaczające liczbę myjni i liczbę klientów. Myjnie ponumerowane są liczbami od 1 do n. W kolejnych m wierszach znajdują się opisy klientów: i-ty z tych wierszy zawiera trzy liczby całkowite $a_i$, $b_i$ i $c_i$ ($1 \leqslant a_i \leqslant b_i \leqslant n$, $1 \leqslant c_i \leqslant 500\,000$) pooddzielane pojedynczymi odstępami, oznaczające, że i-ty klient jeździ pomiędzy myjniami $a_i$ a $b_i$ i posiada budżet $c_i$ bajtalarów.*

*W testach wartych łącznie 75% punktów zachodzi dodatkowy warunek $m \leqslant 250$.*

## Wyjście

*Pierwszy wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą s oznaczającą maksymalny sumaryczny zysk Bajtazara w bajtalarach. Drugi wiersz powinien zawierać przykładowy cennik, pozwalający uzyskać zysk s, a konkretnie ciąg n liczb całkowitych $p_1, p_2, \ldots, p_n$ ($1 \leqslant p_i \leqslant 500\,000$) pooddzielanych pojedynczymi odstępami: liczba $p_i$ ma oznaczać cenę w i-tej myjni. Jeżeli jest więcej niż jedna poprawna odpowiedź, Twój program powinien wypisać dowolną z nich.*

## Przykład

*Dla danych wejściowych:*

```
7 5
1 4 7
3 7 13
5 6 20
6 7 1
1 2 5
```

*poprawnym wynikiem jest:*

```
43
5 5 13 13 20 20 13
```

### Ocenianie

*Jeśli pierwszy wiersz wyjścia będzie nieprawidłowy, otrzymasz 0 punktów za test. Jeśli pierwszy wiersz wyjścia będzie prawidłowy, jednak reszta wyjścia nie będzie prawidłowa, otrzymasz 60% punktów za test. Stanie się tak, nawet jeśli odpowiedź nie będzie zgodna z formatem wyjścia (np. zostanie wypisany tylko jeden wiersz wyjścia lub zostaną wypisane więcej niż dwa wiersze, lub cennik myjni będzie nieprawidłowy lub w złym formacie).*

**Testy „ocen":**

`1ocen`: *$n = 5$, $m = 2$; pierwszy klient przejeżdża obok wszystkich myjni i ma budżet 10, drugi zaś przejeżdża tylko obok trzeciej myjni i ma budżet 9; w optymalnym rozwiązaniu cena paliwa w trzeciej myjni powinna być równa 9, a w pozostałych myjniach powinna być większa lub równa 9 – wówczas obaj klienci kupią paliwo w cenie 9 bajtalarów i zysk Bajtazara będzie równy $2 \cdot 9 = 18$;*

`2ocen`: *$n = 2$, $m = 8$; trzech klientów przejeżdża obok wszystkich myjni (mają budżet 3), trzech klientów przejeżdża tylko obok pierwszej myjni (mają budżet 1) oraz dwóch klientów przejeżdża tylko obok drugiej myjni (mają budżet 1); w optymalnym rozwiązaniu cena paliwa w każdej myjni powinna być równa 3 bajtalary – wówczas tylko pierwsi trzej klienci kupią paliwo i zysk Bajtazara będzie równy $3 \cdot 3 = 9$;*

`3ocen`: *$n = 50$, $m = 1000$; $i$-ty klient jedzie od pierwszej do ostatniej myjni i ma budżet $500 \cdot i$; w optymalnym rozwiązaniu cena paliwa w każdej myjni powinna być równa 250 000 bajtalarów – wówczas tylko klienci o numerach od 500 do 1000 kupią paliwo i zysk Bajtazara będzie równy $501 \cdot 250\,000 = 125\,250\,000$.*

## Rozwiązanie

Często pierwszym krokiem w pracy nad zadaniem algorytmicznym jest postawienie sobie pytania, „od której strony" będziemy konstruować rozwiązanie. Przykładowe pytania, które powinniśmy sobie zadać, to:

- „Kto będzie korzystać z myjni znajdującej się na początku drogi?"
- „Gdzie będzie mył samochód najbogatszy klient?"
- „Która myjnia powinna być najtańsza?"

I to ostatnie pytanie okazuje się kluczem do rozwiązania zadania.

Przypuśćmy, że myjnia o numerze $x$ (w skrócie: myjnia $x$) jest najtańsza i można z niej skorzystać za jedyne $s$ bajtalarów. Daje to nam informację o zachowaniach wszystkich kierowców, których trasa przebiega obok myjni $x$, czyli takich, dla których $a_i \leqslant x \leqslant b_i$. Jeśli $c_i \geqslant s$, to taki klient umyje samochód w myjni $x$ (lub innej z taką samą ceną), zaś pozostałych nie będzie stać na skorzystanie z usług żadnej myjni. Dla kierowców nieprzejeżdżających obok myjni $x$ odcinek $[a_i, b_i]$ musi być całkowicie zawarty w przedziale $[1, x-1]$ bądź $[x+1, n]$. Sytuacje w tych przedziałach możemy rozpatrywać niezależnie, pamiętając jedynie o założeniu, że ceny we wszystkich

myjniach wynoszą przynajmniej $s$. Powyższa obserwacja pozwala sprowadzić nasz problem do badania mniejszych podproblemów, czyli do techniki zwanej programowaniem dynamicznym.

## Pierwsze podejście

Chcielibyśmy oznaczyć przez $t[a][b][s]$ maksymalny zysk, jaki możemy osiągnąć uwzględniając tylko klientów podróżujących na odcinku od myjni $a$ do myjni $b$ (w skrócie: na odcinku $[a, b]$), przy założeniu, że we wszystkich myjniach na tym odcinku ustalimy ceny nie mniejsze niż $s$. Jako że potencjalnie będziemy obliczać wszystkie możliwe wartości tablicy $t$, musimy najpierw uporać się z pewną subtelnością. Zakres cen w zadaniu to $[1, 500\,000]$ i uwzględnienie ich wszystkich prowadziłoby do ogromnej ilości obliczeń. Mało tego, tablica $t$ musiałaby mieć co najmniej $\binom{50}{2} \cdot 500\,000 \geqslant 6 \cdot 10^8$ pól, co przekroczyłoby dozwolony limit pamięci. Na szczęście możemy wykorzystać fakt, że liczba klientów jest ograniczona przez 4000.

**Lemat 1.** Wśród rozwiązań optymalnych istnieje takie, w którym zbiór cen pojawiających się w myjniach jest zawarty w zbiorze budżetów.

**Dowód:** Niech $B$ oznacza zbiór budżetów. Na początek zauważmy, że nie ma sensu ustalać cen wyższych niż $\max B$, toteż możemy ustalić pewne rozwiązanie optymalne, w którym ceny zawarte są w przedziale $[1, \max B]$.

Każdą cenę $s$ występującą w tym rozwiązaniu zastąpmy przez $s' = \min\{c \in B : c \geqslant s\}$. Otrzymujemy w ten sposób pewne nowe rozwiązanie. Klient o numerze $i$ wybierał poprzednio najtańszą myjnię z przedziału $[a_i, b_i]$, o ile wystarczał mu budżet $c_i$ bajtalarów. Nowa cena na tej myjni nie przekroczy $c_i$, zatem $i$-ty klient nie zrezygnuje z mycia samochodu. Z drugiej strony zapłaci on przynajmniej tyle samo, ponieważ cena na żadnej myjni nie została obniżona. Nowe rozwiązanie spełnia założenia lematu, a ponadto gwarantuje nie mniejszy zysk od wyjściowego, zatem również jest optymalne. ■

Niech $(s_j)$ oznacza posortowany rosnąco ciąg budżetów wszystkich klientów. Możemy teraz poprawić definicję tablicy $t$.

**Definicja 1.** Przez $t[a][b][j]$ oznaczamy maksymalny zysk, jaki możemy osiągnąć uwzględniając tylko klientów podróżujących na odcinku $[a, b]$, przy założeniu, że we wszystkich myjniach na tym odcinku ustalimy ceny nie mniejsze niż $s_j$. Dla wygody przyjmujemy $t[a][b][j] = 0$, jeśli $a > b$.

Wykorzystując obserwację ze wstępu, możemy zaproponować algorytm obliczania wartości $t[a][b][j]$. Na początek rozważmy przypadek, w którym wizyta w najtańszej myjni na odcinku $[a, b]$ kosztuje dokładnie $s_j$. Przeglądamy wszystkie pozycje $a \leqslant x \leqslant b$, w których może znajdować się ta myjnia, i maksymalizujemy wartość

$$s_j \cdot K(a, x, b, j) + t[a][x-1][j] + t[x+1][b][j], \tag{1}$$

gdzie $K(a, x, b, j)$ to liczba kierowców spełniających następujące warunki:

1. ich trasa jest zawarta w przedziale $[a, b]$ ($a \leqslant a_i$ oraz $b_i \leqslant b$),
2. przejeżdżają obok myjni $x$ ($a_i \leqslant x \leqslant b_i$),
3. są gotowi wydać $s_j$ bajtalarów na mycie samochodu ($c_i \geqslant s_j$).

Jeśli zaś cena w najtańszej myjni jest wyższa niż $s_j$, to $t[a][b][j] = t[a][b][j+1]$. Jesteśmy gotowi, aby zaprezentować pseudokod pierwszego rozwiązania. Wartości $K(a, x, b, j)$ możemy obliczać, przeglądając każdorazowo wszystkich klientów. Musimy jedynie uważać, aby uzupełniać tablicę $t$ w kolejności rosnących długości przedziałów (*len*) i malejących cen, aby móc odwołać się do wyników wcześniejszych obliczeń. Dla uproszczenia przyjmujemy, że ciąg $(s_j)$ jest długości $m$ (tzn. budżety klientów nie powtarzają się), oraz zakładamy, że tablica $t$ jest wyzerowana. Ze względu na przypisanie w linii 5, aby uniknąć rozpatrywania przypadku brzegowego, wygodnie dodatkowo ustalić $t[a][b][m+1] = 0$.

```
 1: begin
 2:   for len := 1 to n do
 3:     for a := 1 to n − len + 1 do begin
 4:       b := a + len − 1;
 5:       for j := m downto 1 do begin
 6:         t[a][b][j] := t[a][b][j + 1];   { drugi przypadek }
 7:         for x := a to b do begin   { pierwszy przypadek }
 8:           k := 0;
 9:           for i := 1 to m do   { obliczamy k = K(a, x, b, j) }
10:             if a ⩽ a_i ⩽ x and x ⩽ b_i ⩽ b and c_i ⩾ s_j then
11:               k := k + 1;
12:           t[a][b][j] := max(t[a][b][j], s_j · k + t[a][x − 1][j] + t[x + 1][b][j]);
13:         end
14:       end
15: end
```

Pełną implementację można znaleźć w plikach `myjs5.cpp` oraz `myjs6.pas`. Prostota powyższego pseudokodu pozwala na łatwe oszacowanie jego złożoności obliczeniowej: $O(n^3m^2)$. Takie rozwiązanie mogło otrzymać na zawodach 50 punktów.

## Usprawnienie

Zastanówmy się teraz, które fazy algorytmu można przyspieszyć. Na początek spróbujemy sprytniejszego podejścia do wyliczania $K(a, x, b, j)$ poprzez usunięcie z funkcji $K$ jej drugiego argumentu. Oznaczmy przez $K(a, b, j)$ liczbę tych klientów, dla których przedział mijanych myjni zawiera się w $[a, b]$ oraz ich budżet jest nie mniejszy niż $s_j$. Takich klientów można podzielić na trzy grupy: przejeżdżających obok myjni $x$, podróżujących tylko na lewo od niej oraz tylko na prawo. Zapisanie powyższej obserwacji w postaci równania prowadzi do bardzo pomocnego wzoru:

$$K(a, x, b, j) = K(a, b, j) - K(a, x-1, j) - K(x+1, b, j). \tag{2}$$

Wprowadźmy tablicę $k$ w celu spamiętania wszystkich wartości $K(a, b, j)$. Dzięki temu będziemy mogli obliczać $K(a, x, b, j)$ w czasie stałym, co daje nadzieję na algorytm o złożoności obliczeniowej $O(n^3m)$. Niestety każdorazowe przeglądanie wszystkich klientów w celu obliczenia $k[a][b][j]$ wymaga w sumie $O(n^2m^2)$ operacji, co daje jedynie niewielką poprawę w stosunku do poprzedniego podejścia. Rozwiązania o takiej złożoności ($O(n^3m + n^2m^2)$) mogły liczyć na 80 punktów. Przykłady implementacji można znaleźć w plikach `myjs3.cpp` oraz `myjs4.pas`.

## Rozwiązanie wzorcowe

Jedyna rzecz, której nam brakuje, to efektywna metoda wypełniania tablicy $k$. Aby obliczyć $k[a][b][j]$ wystarczy znać $k[a][b][j+1]$ oraz rozmieszczenie klientów o budżecie $s_j$. Możemy przeglądać wszystkich klientów $i$ o budżecie $s_j$ i za każdym razem dodawać 1 do $k[a][b][j]$ dla wszystkich par $(a, b)$, takich że $a \leqslant a_i$ oraz $b_i \leqslant b$. W ten sposób wykonamy $O(n^2)$ operacji dla każdego kierowcy oraz stałą liczbę dodatkowych operacji dla każdego pola w tablicy $k$. Poniżej przedstawiamy pseudokod tej procedury. W dalszym ciągu zakładamy, że ciąg $(s_j)$ jest długości $m$, tablice są wyzerowane, a ponadto przyjmujemy, że klienci zostali posortowani według niemalejących budżetów.

```
1:  begin
2:    i := m;
3:    for j := m downto 1 do begin
4:      for a := 1 to n do
5:        for b := a to n do
6:          k[a][b][j] := k[a][b][j + 1];
7:      while i > 0 and c_i = s_j do begin
8:        for a := 1 to a_i do
9:          for b := b_i to n do
10:           k[a][b][j] := k[a][b][j] + 1;
11:       i := i − 1;
12:     end
13:   end
14: end
```

Tym razem obliczanie obu tablic $k$ oraz $t$ wymaga $O(n^3m)$ operacji, podczas gdy złożoność pamięciowa algorytmu wynosi $O(n^2m)$. Rozwiązanie wzorcowe zostało zaimplementowane w pliku `myj3.cpp`. W plikach `myj.cpp` oraz `myj1.pas` użyto nieco innej implementacji: zamiast zastanawiać się, w jakiej kolejności przeglądać tablicę $t$, można wykorzystać rekurencję ze spamiętywaniem i po prostu wyliczać kolejne pola tablicy $t$, kiedy będą potrzebne. Należy jednak uważać – błąd przy spamiętywaniu wyników może prowadzić do wykładniczego czasu działania.

## Odzyskiwanie cen

Pominęliśmy w omówieniu jedną istotną kwestię, jaką jest odtwarzanie cen w myjniach dla optymalnego rozwiązania. W tym celu dla każdego stanu $(a, b, j)$ należy zapamiętać nie tylko maksymalny zysk, ale także miejsce najtańszej myjni na przedziale $[a, b]$. Następnie możemy rekurencyjnie odtworzyć znalezione rozwiązanie optymalne na coraz mniejszych przedziałach. Nietrudno jednak o pomyłkę w implementacji tej pozornie prostej procedury i dlatego uczestnicy, którzy nie zaprogramowali drugiej części zadania poprawnie, mogli dalej liczyć na 60% punktów za każdy test.

Przypomnijmy, że dla każdej trójki $(a, b, j)$ obliczaliśmy maksymalny zysk, jaki można osiągnąć uwzględniając klientów podróżujących wewnątrz przedziału $[a, b]$, przy założeniu, że wszystkie ceny na tym przedziale wynoszą przynajmniej $s_j$. Niech $opt[a][b][j]$ równa się pozycji myjni $x \in [a, b]$, w której należy ustalić cenę $s_j$, aby zmaksymalizować zysk. Jeśli optymalnie jest ustalić cenę wyższą od $s_j$ (tj. $t[a][b][j] = t[a][b][j+1]$), to przyjmujemy $opt[a][b][j] = -1$. Opracowanie kończymy pseudokodem funkcji rekurencyjnej, która znajduje optymalny przydział cen, o ile tablica $opt$ została obliczona wcześniej (najwygodniej obliczyć ją jednocześnie z tablicą $t$).

```
1:  function Odtworz(a, b, j)
2:  begin
3:    if b < a then return;
4:    x := opt[a][b][j];
5:    if x = -1 then
6:      Odtworz(a, b, j + 1);
7:    else begin
8:      cena[x] := s_j;
9:      Odtworz(a, x - 1, j);
10:     Odtworz(x + 1, b, j);
11:   end
12: end
```

**Wojciech Nadara** — Treść zadania, Opracowanie
**Marek Sokołowski** — Program

**Dostępna pamięć: 256 MB.** *OI, etap III, dzień pierwszy, 16.04.2015*

# Tablice kierunkowe

*Po całym roku wytężonego programowania Bajtazar postanowił udać się na wakacje. Gdy jechał swoim samochodem na wyczekiwane wczasy, mijał wiele* **tablic kierunkowych**, *na których były napisane aktualne odległości (w kilometrach) do różnych miast w Bajtlandii. O ile dokładny dystans pomiędzy tablicą a miastem nie musi wyrażać się całkowitą liczbą kilometrów, o tyle na tablicy mogą się znaleźć tylko liczby całkowite. Dlatego podane na tablicach odległości są* **zaokrągleniami** *prawdziwych odległości* **w dół**.

*Po podróży Bajtazar stwierdził, że informacje umieszczone na mijanych tablicach wyglądały podejrzanie. Uważa, że nie wszystkie tablice zostały rozmieszczone przez ludzi kompetentnych i informacje na niektórych z nich były sprzeczne. Bohater chciałby się dowiedzieć, jak bardzo dane na nich umieszczone były nieprawdziwe. W tym celu postanowił znaleźć największy zbiór tablic, na których informacje nie są wzajemnie sprzeczne. Niestety jest to dla niego zbyt trudne zadanie i poprosił Ciebie o pomoc. Na szczęście Bajtazar ma bardzo dobrą pamięć, zatem pamięta wszystkie tablice, które minął. Nie patrzył on jednak na licznik kilometrów w samochodzie, zatem nie wie, kiedy je zobaczył. Nawet kolejność ich napotykania nie jest dla niego jasna.*

*Zakładamy, że Bajtlandia jest prostą, a miasta są na tyle małe, że możemy je utożsamić z punktami na tej prostej. Przyjmujemy także, że w trakcie swojej podróży Bajtazar* **nie minął żadnego z miast**. *Zbiór tablic jest niesprzeczny, jeśli da się dobrać współrzędne tablic i miast tak, aby zaokrąglone odległości na tablicach były zgodne z prawdą. Oczywiście ani miasta, ani tablice nie muszą znajdować się w punktach o współrzędnych całkowitych. Żadne dwa miasta ani żadne dwie tablice nie mogą znajdować się w tych samych punktach. Co ciekawe, Bajtazar wie, że bajtlandzcy drogowcy nie są całkowicie niekompetentni (sam kiedyś nadzorował budowę drogi, którą jechał). Jest pewien, że istnieje zbiór co najmniej 20% tablic, na których widnieją niesprzeczne informacje.*

## Wejście

*W pierwszym wierszu standardowego wejścia znajdują się dwie liczby całkowite $n$ i $m$ ($1 \leqslant n \leqslant 1000$, $1 \leqslant m \leqslant 200$) oddzielone pojedynczym odstępem, oznaczające odpowiednio liczbę tablic napotkanych przez Bajtazara oraz liczbę miast w Bajtlandii. Każdy z kolejnych $n$ wierszy zawiera opis jednej tablicy; w $i$-tym z tych wierszy znajduje się ciąg $m$ liczb całkowitych $d_{i,1}, d_{i,2}, \ldots, d_{i,m}$ ($1 \leqslant d_{i,j} \leqslant 10^6$) pooddzielanych pojedynczymi odstępami, gdzie $d_{i,j}$ oznacza odległość do miasta o numerze $j$ (w kilometrach) na $i$-tej tablicy, zaokrągloną w dół do najbliższej liczby całkowitej.*

*W testach wartych 60% punktów zachodzą dodatkowe warunki $n \leqslant 500$, $m \leqslant 50$. W podzbiorze tych testów wartym 40% punktów zachodzi warunek $n \leqslant 100$, a w podzbiorze wartym 20% punktów zachodzi $n \leqslant 15$.*

## Wyjście

*W pierwszym wierszu standardowego wyjścia powinna się znaleźć jedna liczba całkowita t oznaczająca największą możliwą liczbę tablic, które przedstawiały niesprzeczne informacje. W drugim wierszu powinno się znaleźć t liczb całkowitych oznaczających numery tych tablic. Powinny one zostać podane w kolejności, w której mógł je napotykać Bajtazar. Jeżeli jest wiele możliwych rozwiązań, Twój program powinien wypisać którekolwiek.*

## Przykład

| *Dla danych wejściowych:* | *poprawnym wynikiem jest:* |
|---|---|
| `3 2`<br>`2 2`<br>`2 3`<br>`3 2` | `2`<br>`2 1` |

**Wyjaśnienie do przykładu:** *Jeżeli druga tablica będzie stała w punkcie $x = 0$, a pierwsza w punkcie $x = \frac{1}{2}$, pierwsze miasto będzie się mieściło w punkcie $x = 2\frac{1}{2}$, a drugie w punkcie $x = 3$, to odległości podane na tablicach o numerach 1 i 2 będą zaokrągleniami w dół prawdziwych odległości miast od tablic. Dla tablic o numerach 1 i 3 również istnieje poprawne rozmieszczenie.*

*Jest jasne, że druga i trzecia tablica są ze sobą sprzeczne, zatem nie istnieje rozmieszczenie tablic i miast, w którym wszystkie trzy tablice podawałyby prawidłowe informacje.*

**Testy „ocen":**

`1ocen`: *$n = 5$, $m = 1$; tablice kierunkowe wskazują różne zaokrąglone odległości do jedynego miasta;*

`2ocen`: *$n = 5$, $m = 2$; każde dwie tablice kierunkowe są ze sobą sprzeczne; w odpowiedzi należy podać dowolną z nich;*

`3ocen`: *$n = 200$, $m = 199$; na wszystkich tablicach widnieją niesprzeczne informacje – przykładowo, można umieścić $i$-tą tablicę w punkcie $\frac{i}{n}$, a $j$-te miasto w punkcie $10^6 + \frac{j}{n}$.*

# Rozwiązanie

## Zależności pomiędzy tabliczkami

Na początek zastanówmy się, kiedy zbiór tabliczek jest niesprzeczny. Dwa zbiory tabliczek nazwijmy *równoważnymi*, jeżeli oba są sprzeczne albo oba są niesprzeczne. Dla ustalenia uwagi załóżmy, że Bajtazar jechał samochodem z lewej na prawo. Zauważmy, że jeżeli wszystkie odległości na pewnej tabliczce zmniejszymy (zwiększymy) o 1, to otrzymamy równoważny zbiór tabliczek, ponieważ potencjalne umiejscowienie tabliczek i miast w drugim z układów można utrzymać, przesuwając tę tabliczkę o 1 w prawo w pierwszym z układów (i na odwrót). Podobnie rzecz się ma, gdybyśmy

chcieli zmniejszyć (zwiększyć) odległości do konkretnego miasta o 1 na wszystkich tabliczkach. Poprawne rozmieszczenie w drugim z układów otrzymamy, przesuwając dane miasto o 1 w lewo (prawo) w pierwszym z układów. W ten sposób na niektórych tabliczkach mogą chwilowo pojawić się liczby ujemne – nie jest to jednak problemem, o ile zadbamy, by na końcu tego procesu wszystkie znów były nieujemne.

Odległość do miasta $j$ na tabliczce $i$ oznaczmy zgodnie z treścią zadania przez $d_{i,j}$. Następnie wykonajmy operację $d_{i,j} := d_{i,j} - d_{1,j}$, otrzymując układ równoważny (od odległości do każdego miasta odjęliśmy tyle samo). Dzięki temu pierwsza tabliczka składa się z samych zer. Następnie dla każdej tabliczki $i$ odejmijmy od odległości na niej napisanych $\min_j\{d_{i,j}\}$, ponownie otrzymując układ tabliczek równoważnych. Wtedy na każdej tabliczce będą widniały nieujemne liczby całkowite i na każdej będzie co najmniej jedno 0, a ponadto na pierwszej tabliczce wciąż będą same zera. Otrzymany układ odległości na tabliczce o numerze $i$ nazwijmy *znormalizowaną tabliczką o numerze $i$ względem tabliczki* 1. Analogicznie można wprowadzić pojęcie normalizacji względem dowolnej innej tabliczki.

Bez straty ogólności możemy umieścić pierwszą tabliczkę w punkcie $x = 0$. Wtedy wszystkie miasta muszą leżeć na odcinku $[0, 1)$, gdyż ich odległości od tej tabliczki po zaokrągleniu w dół są równe 0. Co więcej, wszystkie tabliczki mogą znaleźć się na odcinku $(-1, 0]$, gdyż dla każdej z nich odległość od co najmniej jednego z miast zaokrąglona w dół jest równa 0. Jeżeli wobec tego na którejkolwiek z tabliczek widnieje liczba co najmniej 2, to układ tabliczek jest sprzeczny. Zatem jeśli układ jest niesprzeczny, na wszystkich tabliczkach muszą być umieszczone wyłącznie liczby 0 i 1.

Zauważmy teraz, że w żadnym niesprzecznym układzie tabliczek nie może zachodzić $d_{i_1,j_1} > d_{i_2,j_1}$ oraz $d_{i_1,j_2} < d_{i_2,j_2}$ (patrz przykład w treści zadania). Pozwala nam to wprowadzić liniowy porządek na tabliczkach, który będzie oznaczał porządek, w którym znormalizowane tabliczki będą umieszczone na odcinku $(-1, 0]$. Innymi słowy, dla dowolnych dwóch tabliczek o indeksach $i_1$ oraz $i_2$, dla każdego miasta $j$ zachodzi albo $d_{i_1,j} \leqslant d_{i_2,j}$, albo $d_{i_2,j} \leqslant d_{i_1,j}$. W pierwszym z tych przypadków wiemy, że tabliczka o indeksie $i_2$ stoi wcześniej na osi liczbowej, a w drugim przypadku wcześniej stoi tabliczka $i_1$. Jeżeli tabliczki mają dokładnie takie same odległości, to możemy je umieścić w dowolnej kolejności. Okazuje się, że jeżeli w naszym przypadku, do którego sprowadziliśmy początkowe pytanie, ani nie zachodzi $d_{i_1,j_1} > d_{i_2,j_1}$ oraz jednocześnie $d_{i_1,j_2} < d_{i_2,j_2}$ dla odpowiednich indeksów, ani po znormalizowaniu na tabliczkach nie ma liczb większych od 1, to jest to już warunek wystarczający do tego, aby oryginalny zbiór tabliczek był niesprzeczny. Fakt, że na tabliczkach da się wprowadzić opisany porządek, oznacza tyle, że jeżeli posortujemy tabliczki niemalejąco względem sumy odległości na nich napisanych, to zbiór miast, do których odległość na pewnej tabliczce wynosi 1, zawiera się w analogicznym zbiorze miast dla następnej tabliczki (odpowiada to kolejności tabliczek od prawej do lewej na odcinku $(-1, 0]$). Jeżeli wszystkie te warunki są spełnione, to możemy w kolejności sumy liczb na tabliczkach stawiać je w odstępach $\frac{1}{n}$ na odcinku $(-1, 0]$. Łatwo zauważyć, że dla każdego miasta da się je postawić na dokładnie jednym odcinku długości $\frac{1}{n}$ wewnątrz odcinka $[0, 1)$, tak aby wszystkie odległości do tego miasta napisane na tabliczkach były prawdziwe.

Przykładowo, jeśli mamy $m = 4$ miasta i $n = 5$ pięć tabliczek, które po znormalizowaniu wyglądają następująco:

$$0\,0\,0\,0,\;\; 0\,1\,0\,0,\;\; 0\,1\,0\,0,\;\; 0\,1\,1\,0,\;\; 1\,1\,1\,0,$$

to tabliczki postawimy kolejno w punktach 0, $-\frac{1}{5}$, $-\frac{2}{5}$, $-\frac{3}{5}$ i $-\frac{4}{5}$. Miasto 2 możemy teraz umieścić gdziekolwiek na odcinku $(\frac{4}{5}, 1)$, miasto 3 na odcinku $(\frac{2}{5}, \frac{3}{5})$, miasto 1 na odcinku $(\frac{1}{5}, \frac{2}{5})$, a miasto 4 na odcinku $(0, \frac{1}{5})$.

W ten sposób możemy stworzyć rozwiązanie wykładnicze sprawdzające dla każdego podzbioru tabliczek, czy jest on niesprzeczny. Otrzymamy rozwiązanie działające w czasie $O(2^n n^2 m)$; przykładową implementację można znaleźć w pliku `tabs1.cpp`.

## Rozwiązanie brutalne, aczkolwiek rokujące

Poczynione do tej pory obserwacje pozwalają nam już przymierzyć się do stworzenia algorytmu działającego w czasie wielomianowym. Ustalmy $i$-tą tabliczkę i dokonajmy normalizacji pozostałych względem niej. W takim zbiorze tabliczek będziemy szukać jak największego zbioru tabliczek spełniającego warunki opisane w poprzednim akapicie, tzn. takiego ciągu tabliczek, że liczby na nich napisane to wyłącznie 0 i 1 oraz dla każdych dwóch tabliczek zbiór pozycji, na których występują jedynki na wcześniejszej z nich, zawiera się w zbiorze pozycji, na których występują jedynki na późniejszej z nich. Tabliczki, na których występują liczby różne od 0 i 1, możemy całkowicie zignorować.

Stwórzmy zatem graf, w którym krawędź od tabliczki $a$ do tabliczki $b$ istnieje wtedy i tylko wtedy, gdy zbiór pozycji z odległością 1 na tabliczce $a$ zawiera się w zbiorze pozycji z odległością 1 na tabliczce $b$ – taką uporządkowaną parę tabliczek $(a, b)$ nazwijmy dopuszczalną. Zauważmy, że postawiony w zadaniu problem sprowadza się do znalezienia najdłuższej ścieżki w DAG-u (czyli w acyklicznym grafie skierowanym), co jesteśmy w stanie zrobić w czasie $O(n^2)$ za pomocą programowania dynamicznego w kolejności wierzchołków zgodnej z porządkiem topologicznym. To daje nam pierwszy wielomianowy algorytm na rozwiązanie tego zadania (powtarzamy to dla normalizacji względem każdej możliwej tabliczki). Krawędzie w tym grafie wyznaczamy, sprawdzając dla każdej pary, czy jest dopuszczalna.

Porządek topologiczny, w którym powinniśmy przetwarzać wierzchołki, jest równy porządkowi wyznaczonemu przez sumy liczb na tabliczkach. Jeżeli dwie tabliczki mają identyczną sumę odległości, to w porządku kolejność między nimi możemy ustalić dowolnie.

W takim razie po wyznaczeniu rzeczonego grafu i znalezieniu w nim długości najdłuższej ścieżki (i powtórzeniu tego dla każdej tabliczki jako tabliczki, względem której normalizujemy) już wiemy, jak duży zbiór tabliczek stanowi odpowiedź. Z tego samego programowania dynamicznego jesteśmy także w stanie w łatwy sposób odzyskać samą najdłuższą ścieżkę, czyli najlepszy zbiór tabliczek. Pozostaje jeszcze problem, jak odtworzyć oryginalną kolejność, w której Bajtazar napotykał tabliczki. Jednak aby ją odzyskać, wystarczy jedynie posortować tabliczki nierosnąco względem sumy odległości (oryginalnych!) na nich napisanych.

Nasuwa się jeszcze techniczny problem: jak szybko sprawdzić, czy dana para tabliczek $(a, b)$ jest dopuszczalna. Można jawnie przeiterować się po wszystkich pozycjach i sprawdzić, czy nie istnieje taka pozycja, na której na tabliczce $a$ widnieje 1, a na tabliczce $b$ widnieje 0, i wtedy zajmie nam to $O(m)$ czasu dla ustalonej pary. Zauważmy, że informacja zawarta na jednej tabliczce przedstawia się jako $m \leqslant 200$ bitów, a tyle możemy zawrzeć w czterech zmiennych typu całkowitego 64-bitowego, np. `long long`

w C++. Po zamienieniu ciągów bitów na odpowiadające im liczby, sprawdzenie, czy jeden ciąg bitów zawiera się w drugim, to po prostu wykonanie pewnych operacji bitowych na liczbach je reprezentujących, co wykonuje się w czasie stałym. Zatem jeżeli zamieniliśmy odpowiednio wcześniej ciągi $m$ bitów na odpowiadające im czwórki zmiennych typu `long long`, to sprawdzenie, czy dana para tabliczek jest dopuszczalna, odbywa się za pomocą 4 operacji bitowych. W ogólności jest to złożoność $O(\frac{m}{B})$, gdzie $B$ jest długością słowa maszynowego w systemie, na którym pracujemy. W naszym przypadku mamy $B = 64$. Implementacja tej części w języku C++ znacznie się upraszcza, jeżeli zamiast trzymania ciągu bitów w kilku zmiennych typu `long long` użyjemy struktury `bitset`. Pozwala ona wykonywać w istotnie łatwiejszy sposób wiele operacji na ciągach bitów długości $m$ w czasie $O(\frac{m}{B})$. Czytelników, którzy nie znają tej struktury, zdecydowanie zachęcamy do zapoznania się z nią oraz jej interfejsem.

Tak oto otrzymaliśmy algorytm rozwiązujący zadanie w czasie $O(n^2m+n^3+\frac{n^3m}{B})$. Istotnie, dla każdej z tabliczek, których jest $n$, najpierw musimy znormalizować wszystkie tabliczki względem niej, co zajmuje $O(nm)$ czasu. Potem stosujemy programowanie dynamiczne szukające najdłuższej ścieżki w DAG-u (w którym musimy sprawdzać istnienie potencjalnie $n^2$ krawędzi), co zajmuje $O(n^2+\frac{n^2m}{B})$ czasu. Stąd pojedyncza faza zajmuje $O(nm+n^2+\frac{n^2m}{B})$ czasu, czyli $n$ faz zajmuje $O(n^2m+n^3+\frac{n^3m}{B})$ czasu. Złożoność pamięciowa wynosi $O(nm)$. Takie rozwiązanie w zupełności wystarcza do rozwiązania testów wartych 60% punktów (implementacje: `tabs4.cpp` – z typem `long long`, `tabs5.cpp` – z typem `bitset`). Przy nieuważnej implementacji bez masek bitowych rozwiązanie działa w czasie $O(n^3m)$ i zdobywało na zawodach ok. 40% punktów (implementacje: `tabs2.cpp`, `tabs3.cpp`).

## Rozwiązanie wzorcowe

Zauważmy ciekawą własność przedstawionego rozwiązania. Mianowicie optymalną odpowiedź znajdziemy w każdej fazie, w której normalizujemy wszystkie tabliczki względem **dowolnej** tabliczki należącej do optymalnego rozwiązania (dlaczego?). To podsuwa myśl, że jeżeli rozwiązanie jest duże, to powinniśmy prędko je znaleźć, co pozwala zastosować algorytm randomizowany! Możemy mianowicie w każdym kroku normalizować względem losowo wybranej tabliczki. Jeżeli trafimy na tabliczkę należącą do rozwiązania, to normalizując względem niej, znajdziemy rozwiązanie. Jeżeli rozwiązanie będzie się składało z $\geqslant \frac{n}{5}$ tabliczek, to prawdopodobieństwo, że nie znajdziemy go w ciągu 50 faz, wynosi $(\frac{4}{5})^{50} \approx 1{,}5 \cdot 10^{-5}$, co jest w pełni satysfakcjonujące. Takie rozwiązanie będzie działało w czasie $O(f \cdot (nm + n^2 + \frac{n^2m}{B}))$, gdzie $f$ jest liczbą faz, co dla $f = 50$, $n \leqslant 1000$, $m \leqslant 200$ powinno być wystarczająco szybkie i udzielać błędnej odpowiedzi w pomijalnie małej liczbie przypadków. Wystarcza ono do rozwiązania właściwej wersji zadania.

Implementacje takiego rozwiązania można znaleźć w plikach `tab.cpp`, `tab2.c` i `tab3.pas`.

**Jacek Tomasiewicz**
Treść zadania, Opracowanie

**Aleksander Łukasiewicz**
Program

**Dostępna pamięć: 128 MB.**

*OI, etap III, dzień pierwszy, 16.04.2015*

# Wilcze doły

*Król Bajtocji, Bajtazar III Zuchwały, planuje najazd na zamek wroga. Zamek jest z trzech stron otoczony niemożliwą do sforsowania fosą, więc Bajtazarowi pozostaje przypuścić atak na czwartą ścianę zamku. Sprawa nie jest jednak taka prosta, gdyż królewscy zwiadowcy donieśli o tym, że wzdłuż tej ściany wróg wykopał głębokie wilcze doły. Bajtazar chciałby zaatakować jak najdłuższy spójny fragment tej ściany. W tym celu będzie musiał zrównać z ziemią niektóre doły. Król postanowił, że część z nich przysypie piachem, a część przykryje Wielką Dechą.*

*Wzdłuż ściany wykopanych jest $n$ dołów. Król Bajtazar posiada $p$ worków z piachem. Do przysypania $i$-tego dołu potrzebne jest $w_i$ takich worków. Ponadto, Wielka Decha pozwala na przykrycie $d$ sąsiednich dołów.*

*Pomóż Bajtazarowi znaleźć długość najdłuższego fragmentu ściany, który będzie mógł zaatakować, jeśli optymalnie wykorzysta worki z piachem i Wielką Dechę. Innymi słowy, oblicz, ile maksymalnie kolejnych dołów może zostać zrównanych z ziemią.*

## Wejście

*Pierwszy wiersz standardowego wejścia zawiera trzy liczy całkowite $n$, $p$ oraz $d$ ($1 \leqslant d \leqslant n \leqslant 2\,000\,000$, $0 \leqslant p \leqslant 10^{16}$) pooddzielane pojedynczymi odstępami, oznaczające odpowiednio liczbę dołów, liczbę worków z piachem oraz długość Wielkiej Dechy.*

*Kolejny wiersz opisuje doły i zawiera ciąg $n$ liczb całkowitych $w_1, w_2, \ldots, w_n$ ($1 \leqslant w_i \leqslant 10^9$) pooddzielanych pojedynczymi odstępami; $w_i$ oznacza liczbę worków potrzebnych do przysypania $i$-tego dołu.*

*W testach wartych łącznie 30% punktów zachodzi dodatkowy warunek $n \leqslant 3000$.*

## Wyjście

*Pierwszy i jedyny wiersz standardowego wyjścia powinien zawierać jedną liczbę całkowitą, równą długości najdłuższego spójnego fragmentu ściany, na który Bajtazar może przypuścić atak.*

## Przykład

| *Dla danych wejściowych:* | *poprawnym wynikiem jest:* |
|---|---|
| `9 7 2`<br>`3 4 1 9 4 1 7 1 3` | `5` |

**Wyjaśnienie do przykładu**: *Bajtazar może przysypać doły o numerach 2, 3 i 6 (zużywając do tego celu 6 spośród 7 posiadanych worków z piachem) oraz przykryć Wielką Dechą doły 4 i 5. W ten sposób król zrówna z ziemią pięć kolejnych dołów (o numerach od 2 do 6).*

**Testy „ocen”:**

`1ocen`: *$n = 100$, $p = 49$, $d = 50$, wszystkie $w_i = 1$; można zrównać z ziemią wszystkie doły poza jednym;*

`2ocen`: *$n = 500\,000$, $p = 1$, $d = 1000$, doły o parzystych numerach potrzebują dwóch worków z piachem, zaś doły o nieparzystych numerach potrzebują jednego worka.*

# Rozwiązanie

W zadaniu mamy dany ciąg liczb całkowitych $w_1, \ldots, w_n$. Musimy znaleźć najdłuższy fragment ciągu $(w_i)$, taki że suma elementów tego fragmentu, z pominięciem pewnego spójnego kawałka fragmentu o długości $d$ (przykrytego Wielką Dechą), nie przekracza liczby dostępnych worków z piachem $p$. Fragment (niekoniecznie najdłuższy) spełniający ten warunek, nazwiemy *poprawnym*.

## Rozwiązanie siłowe $O(n^3)$

Każdy spójny fragment ciągu możemy rozpatrzeć osobno poprzez sprawdzenie, czy jest on poprawny. Sprawdzenie poprawności fragmentu możemy łatwo wykonać w czasie liniowym: wystarczy przykładać początek deski w każdym możliwym miejscu, jednocześnie aktualizując sumę elementów poza deską. Przy przesuwaniu deski o jedną pozycję w prawo, sumę elementów poza deską można aktualizować w czasie stałym. Jako że wszystkich spójnych fragmentów jest $O(n^2)$, a pojedyncze sprawdzenie zajmuje czas liniowy względem długości fragmentu, złożonością czasową tego rozwiązania jest $O(n^3)$.

Rozwiązanie to zaimplementowane jest w pliku `wils1.cpp`. Za poprawne zaprogramowanie takiego rozwiązania na zawodach można było uzyskać około 20% punktów.

## Rozwiązanie wolne $O(n^2 \log n)$

Zauważmy, że jeżeli istnieje poprawny fragment długości $s$, to istnieją też krótsze poprawne fragmenty o długościach $s-1$, $s-2$, $\ldots$, $d$. Natomiast jeśli nie istnieje poprawny fragment o długości $s$, to nie istnieją też poprawne fragmenty o długościach większych niż $s$. Dzięki tej obserwacji, długość najdłuższego fragmentu może zostać wyszukana binarnie. W ten sposób zredukujemy nasz problem do logarytmicznej liczby pytań czy istnieje poprawny fragment o ustalonej długości.

Wszystkich fragmentów o ustalonej długości $s$ jest $O(n)$. Jeśli każdy z nich sprawdzimy w czasie liniowym względem długości rozpatrywanego fragmentu, to otrzymamy całkowitą złożoność czasową $O(n^2 \log n)$.

Implementacja takiego rozwiązania znajduje się w pliku `wils2.cpp`. Rozwiązanie tego typu otrzymywało na zawodach około 30% punktów.

## Rozwiązanie szybkie $O(n \log^2 n)$

Podobnie jak w rozwiązaniu wolnym, najdłuższy poprawny fragment będziemy wyszukiwać binarnie. Następnie przejrzymy wszystkie fragmenty o zadanej długości $s$. Przypomnijmy, że poprawny fragment to taki, którego suma elementów z pominięciem pewnego spójnego kawałka o długości $d$, nie przekracza liczby $p$. Opłaca się zatem, żeby pominięty kawałek miał jak największą sumę elementów.

Aby móc efektywnie obliczać sumy elementów w dowolnym fragmencie, w pierwszym kroku wyznaczymy ciąg sum częściowych ciągu $(w_i)$. $i$-tą sumę częściową (dla $1 \leqslant i \leqslant n$) definiujemy jako $a_i = w_1 + w_2 + \ldots + w_i$, jednocześnie przyjmując $a_0 = 0$. Zauważmy, że $a_i = a_{i-1} + w_i$, więc ciąg $a_1, \ldots, a_n$ można obliczyć w czasie $O(n)$. Wartości $a_i$ pozwalają obliczyć sumę elementów w dowolnym fragmencie $w_i, \ldots, w_j$ w czasie stałym ze wzoru $a_j - a_{i-1}$.

Pozostaje pokazać, jak znajdować kawałek długości $d$ o maksymalnej sumie. W tym celu utwórzmy ciąg liczb reprezentujących sumy elementów w kolejnych kawałkach o długości $d$. Dokładniej, niech $x_i = w_i + w_{i+1} + \ldots + w_{i+d-1} = a_{i+d-1} - a_{i-1}$, gdzie $1 \leqslant i \leqslant n - d + 1$. Zauważmy, że optymalny kawałek o długości $d$ zawarty we fragmencie $w_i, \ldots, w_j$ (gdzie $j - i + 1 \geqslant d$) ma sumę $\max(x_i, x_{i+1}, \ldots, x_{j-d+1})$.

Poniższy algorytm sprawdza czy istnieje poprawny fragment o długości $s \geqslant d$:

```
1: for i := 1 to n − s + 1 do begin
2:    j := i + s − 1; { wyznaczamy koniec przedziału }
3:    suma := a_j − a_{i−1}; { liczymy sumę całego przedziału }
4:    if suma − max(x_i, x_{i+1}, ..., x_{j−d+1}) ⩽ p then
5:       return true;
6: end
7: return false;
```

Znajdowanie maksymalnego (lub minimalnego) elementu w zadanym spójnym fragmencie ciągu jest dosyć częstym i standardowym problemem, znanym pod nazwą *RMQ* (ang. Range Minimum Query). Istnieje kilka klasycznych struktur danych rozwiązujących ten problem. Drzewo przedziałowe (opisane na przykład w opracowaniu zadania *Tetris 3D* z I etapu XIII Olimpiady Informatycznej [13]) pozwala znajdować największą wartość we fragmencie w czasie $O(\log n)$. Przy użyciu pamięci rzędu $O(n \log n)$, można także zastosować strukturę podobną do słownika podsłów bazowych (opisanego np. w [24]), pozwalającą odpowiadać na zapytania w czasie stałym. Ta metoda była jednak trudna do wykorzystania na zawodach właśnie ze względu na duży narzut pamięciowy. Znana jest także optymalna struktura danych dla problemu RMQ, która po przetworzeniu ciągu w czasie $O(n)$, odpowiada na zapytania w czasie $O(1)$. Jest ona jednak dość trudna w implementacji i dlatego jest rzadko stosowana w praktyce.

Z tych względów, do zaimplementowania tego rozwiązania najlepiej było użyć drzewa przedziałowego. Wtedy całe rozwiązanie działa w czasie $O(n \log^2 n)$. Takie rozwiązanie otrzymywało na zawodach około 80% punktów i zostało zaimplementowane w pliku `wils5.cpp`.

## Rozwiązanie prawie optymalne $O(n \log n)$

Aby otrzymać prostsze i bardziej efektywne rozwiązanie naszego zadania, można zauważyć, że występujące w nim zapytania o maksymalną wartość w przedziale są bardzo szczególnej postaci – przedział, o który pytamy, „pełznie" przez tablicę. Wystarczy nam zatem struktura danych podobna do kolejki, udostępniająca operacje wstawiania elementu na koniec, usunięcia elementu z początku oraz odczytywania maksimum. Jeśli umielibyśmy wykonywać takie operacje w zamortyzowanym czasie stałym, sprawdzenie pojedynczej długości w wyszukiwaniu binarnym moglibyśmy zrealizować w czasie liniowym.

Szczęśliwie, taka struktura danych wystąpiła już w rozwiązaniach zadań olimpijskich: w zadaniu *Temperatura* z II etapu XVIII Olimpiady Informatycznej [18] oraz w zadaniu *Piloci* z III etapu XVII Olimpiady Informatycznej [17].

Całkowita złożoność czasowa tego rozwiązania wynosi zatem $O(n \log n)$. Jego implementacja znajduje się w pliku `wils11.cpp`. Za poprawny program tego typu można było uzyskać około 90% punktów.

## Rozwiązanie wzorcowe $O(n)$

Aby otrzymać algorytm optymalny, zmodyfikujemy poprzednie rozwiązanie poprzez pozbycie się wyszukiwania binarnego. Zauważmy, że jeśli fragment $w_i, \ldots, w_j$ jest poprawny, to nieznacznie krótszy fragment $w_{i+1}, \ldots, w_j$ (gdzie $j - i \geqslant d$), również jest poprawny. Dzięki tej obserwacji możemy użyć tzw. metody gąsienicy – będziemy wydłużali z prawej strony fragment będący kandydatem na optymalne rozwiązanie, dopóki będzie on poprawny. Gdy takie rozszerzenie nie będzie możliwe, przesuniemy początek fragmentu-kandydata o jedną pozycję w prawo. W ten sposób dla każdej możliwej pozycji $i$ początku fragmentu znajdziemy najdalszą pozycję $j$ taką, że fragment $w_i, \ldots, w_j$ jest poprawny. W szczególności, tym sposobem na pewno nie pominiemy żadnego z optymalnych fragmentów.

```
1:  wynik := d
2:  j := d;
3:  for i := 1 to n − d + 1 do begin
4:    j := max(j, i + d − 1);
5:    { niezmiennik: w_i, ..., w_j jest poprawny }
6:    while j+1 ⩽ n and (a_{j+1} − a_{i−1}) − max(x_i, x_{i+1}, ..., x_{j−d+2}) ⩽ p do begin
7:      j := j + 1;
8:    end
9:    wynik := max(wynik, j − i + 1);
10: end
11: return wynik;
```

Zauważmy, że w powyższym algorytmie wartość zmiennej $j$ nie maleje, a przy każdym obrocie pętli **while** jest zwiększana. Dodatkowo $j \leqslant n$, więc wewnętrzna pętla wykona $O(n)$ obrotów.

Tak jak poprzednio, aby obliczać wartości $\max(x_i, x_{i+1}, \ldots, x_{j-d+2})$, potrzebujemy struktury danych, która wyznacza maksymalną wartość w spójnym fragmencie ciągu $(x_i)$. Jeżeli użyjemy drzewa przedziałowego, uzyskamy rozwiązanie działające w czasie $O(n \log n)$. Jeśli natomiast skorzystamy z dwustronnej kolejki utrzymującej maksimum z przechowywanych wartości, uzyskamy optymalny algorytm liniowy.

Implementacje rozwiązań z użyciem kolejki znajdują się w plikach `wil.cpp` oraz `wil1.cpp`, a rozwiązanie używające drzewa przedziałowego znajduje się w pliku `wil3.cpp`. Poprawne zaprogramowanie dowolnego z tych rozwiązań było nagradzane maksymalną liczbą punktów.

## Testy

Testy były podzielone na 9 grup. Każda z nich zawierała testy następujących typów:

- całkowicie losowe,
- większość dołów głębokich plus oaza o płytkich dołach,
- doły tworzące lejek,
- poprzeplatane płytkie i głębokie doły.

**Marek Sokołowski**
Treść zadania, Opracowanie

**Tomasz Syposz**
Program

**Dostępna pamięć: 8 MB.** *OI, etap III, dzień drugi, 17.04.2015*

# Kolekcjoner Bajtemonów

*Bajtazar bardzo lubi kolekcjonować karty z Bajtemonami. Na każdej karcie w jego talii narysowany jest jeden Bajtemon wraz z numerem katalogowym, będącym liczbą całkowitą z przedziału $[1, 2 \cdot 10^9]$. Bajtazar nie zgromadził jeszcze wszystkich Bajtemonów. Jego kolekcja jest dość osobliwa: każdy Bajtemon, który się w niej znajduje, występuje na kilku kartach, co więcej, każdy na takiej samej liczbie kart.*

*Pewnego dnia Bajtazar zorientował się, że ktoś podkradł mu kilka (jedną lub więcej) kart z jego kolekcji. Wie, że na brakujących kartach był ten sam Bajtemon i że na szczęście został mu przynajmniej jeden egzemplarz tej karty. Niestety, nasz bohater ma bardzo mały rozumek i zdążył już zapomnieć, jaki to był Bajtemon. Czy jesteś w stanie mu pomóc i przypomnieć, jakich kart mu brakuje? Twój program też musi pamiętać o ograniczeniach pamięci...*

*Napisz program komunikujący się z biblioteką służącą do przeglądania talii kart Bajtazara, który znajdzie numer katalogowy podkradzionego Bajtemona.*

## Komunikacja

*Aby użyć biblioteki, należy wpisać na początku programu:*

- **C/C++***:* `#include "ckollib.h"`
- **Pascal***:* `uses pkollib;`

*Biblioteka udostępnia następujące funkcje i procedury:*

- `karta` – *Daje w wyniku liczbę całkowitą z przedziału $[1, 2 \cdot 10^9]$ bądź 0. Wartość 0 oznacza koniec talii kart, zaś dodatnia liczba całkowita – numer katalogowy Bajtemona zapisany na kolejnej karcie. Po otrzymaniu wyniku funkcji 0, Twój program może wciąż wywoływać funkcję* `karta` *– odpowiada to kolejnemu przeglądaniu talii kart przez Bajtazara. Przy każdym przeglądaniu talii, numery katalogowe Bajtemonów podawane są w tej samej kolejności.*
  - **C/C++:** `int karta();`
  - **Pascal:** `function karta: LongInt;`
- `odpowiedz(wynik)` – *Odpowiada bibliotece, że* `wynik` *to numer katalogowy Bajtemona, który znajduje się na podkradzionych kartach. Wywołanie tej funkcji* **kończy działanie Twojego programu**.
  - **C/C++:** `void odpowiedz(int wynik);`
  - **Pascal:** `procedure odpowiedz(wynik: LongInt);`

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

## Ocenianie

*Twój program otrzyma za dany test pełną punktację, jeśli przejrzy całą talię nie więcej niż raz (tj. nie wywoła funkcji* `karta` *po tym, gdy pierwszy raz da ona w wyniku wartość 0).*

*Jeśli program rozpocznie przeglądanie talii po raz drugi, otrzyma tylko 40% punktów za dany test.*

*W przypadku, gdy program zacznie przeglądać talię po raz trzeci, biblioteka przerwie działanie programu i nie uzyskasz za ten test żadnych punktów.*

*Liczba kart w talii nie przekroczy 60 000 000. Ponadto, w testach o łącznej wartości 20% punktów liczba kart w talii nie przekroczy 200 000.*

## Przykładowy przebieg programu

| C/C++ | Pascal | Wynik | Wyjaśnienie |
|---|---|---|---|
| `k = karta();` | `k := karta;` | 13 | Numer katalogowy Bajtemona z pierwszej karty. |
| `k = karta();` | `k := karta;` | 13 | |
| `k = karta();` | `k := karta;` | 39 | |
| `k = karta();` | `k := karta;` | 26 | |
| `k = karta();` | `k := karta;` | 26 | |
| `k = karta();` | `k := karta;` | 0 | Koniec talii. |
| `k = karta();` | `k := karta;` | 13 | Nowy obieg talii; karty są w tej samej kolejności, co poprzednio. |
| `odpowiedz(39);` | `odpowiedz(39);` | | Udzielamy odpowiedzi. Program kończy swoje działanie. |

*Powyższy przebieg programu jest poprawny (dokonuje nie więcej niż dwóch przeglądnięć talii i daje poprawny wynik). Jednakże rozpoczyna on drugie przeglądanie talii, więc program otrzyma w tym przypadku 40% punktów. Aby uzyskać pełną punktację, należy dokonać tylko jednego przeglądnięcia talii.*

## Eksperymenty

*W katalogu* `dlazaw` *dostępna jest przykładowa biblioteka, która pozwoli Ci przetestować poprawność formalną rozwiązania. Biblioteka wczytuje opis talii kart ze standardowego wejścia w następującym formacie:*

- *w pierwszym wierszu dodatnia liczba całkowita $n$ – liczba kart w talii,*
- *w drugim wierszu $n$ liczb całkowitych z przedziału $[1, 2 \cdot 10^9]$ – numery Bajtemonów na kolejnych kartach talii.*

*Przykładowa biblioteka* **nie sprawdza**, *czy faktycznie w talii istnieje tylko jeden Bajtemon narysowany na mniejszej liczbie kart niż wszystkie pozostałe Bajtemony występujące w talii.*

*Przykładowe wejście dla biblioteki znajduje się w pliku* `kol0.in`. *Po wywołaniu procedury* `odpowiedz`, *biblioteka wypisuje na standardowe wyjście informację o udzielonej odpowiedzi, liczbie wywołań funkcji* `karta` *i liczbie rozpoczętych przebiegów.*

*W tym samym katalogu znajdują się przykładowe rozwiązania* `kol.c`*,* `kol.cpp` *i* `kol.pas` *korzystające z biblioteki. Rozwiązania te nie są poprawne i zawsze odpowiadają, że szukanym numerem Bajtemona jest numer umieszczony na ostatniej karcie w talii.*

*Do kompilacji rozwiązania wraz z biblioteką służą polecenia:*

- **C***:* `gcc -O2 -static ckollib.c kol.c -lm`
- **C++***:* `g++ -O2 -static ckollib.c kol.cpp -lm -std=gnu++0x`
- **Pascal***:* `ppc386 -O2 -XS -Xt kol.pas`

*Plik z rozwiązaniem i biblioteka powinny znajdować się w tym samym katalogu.*

# Rozwiązanie

W zadaniu mamy dany ciąg liczb. Wiadomo o nim, że każdy element występuje w nim dokładnie tyle samo razy – poza jedną z liczb $x$, której liczba wystąpień jest mniejsza. Naszym zadaniem jest znaleźć element $x$.

Inaczej niż w klasycznych zadaniach olimpijskich, tutaj nie możemy wczytywać danych ze standardowego wejścia. Jedynym sposobem na poznanie liczb jest odpytywanie funkcji z biblioteki dostarczonej w trakcie zawodów przez organizatorów. Na zawodników czekał jeszcze szereg kolejnych utrudnień – początkowo ani nie znamy długości ciągu (jego długość poznajemy dopiero, gdy dojdziemy do końca ciągu), ani nie wiemy, ile razy każda liczba w tym ciągu występuje. Dodatkowym problemem jest też niewielki rozmiar pamięci dostępnej dla programu. Tak więc szukamy rozwiązania działającego w pamięci stałej lub niemal stałej. Ostatecznie wolno nam przejrzeć cały ciąg tylko raz (lub dwa razy, jeśli celujemy w rozwiązanie uzyskujące częściową punktację), w kolejności określonej przez bibliotekę.

W dalszym rozumowaniu oznaczamy przez $n$ długość ciągu, przez $A$ największy element w ciągu, natomiast przez $k$ liczbę różnych elementów występujących w ciągu.

## Rozwiązania pamięciochłonne

Najbardziej brutalnym rozwiązaniem problemu może być siłowe wyznaczenie liczby wystąpień każdego elementu w ciągu. Jedną z metod osiągnięcia tego celu może być zapisanie całego ciągu do tablicy. Następnie elementy tablicy możemy łatwo posortować, a potem w czasie liniowym określić, który element występuje najrzadziej. Rozwiązanie to ma złożoność obliczeniową $\Theta(n \log n)$, jednakże złożoność pamięciowa wynosi $\Theta(n)$ i jest zdecydowanie niedopuszczalna przy limicie pamięci równym 8 MB. Implementacja tej metody znajduje się w plikach `kols1.cpp`, `kols3.pas`, `kols4.c` i dostawała na zawodach 20% punktów.

Można nieco usprawnić poprzednie rozwiązanie i zauważyć, że niepotrzebnie tracimy pamięć na przechowywanie elementów o tej samej wartości. Korzystając ze struktury umiejącej grupować wystąpienia identycznych elementów (np. struktury `map` z biblioteki STL w C++), możemy dla każdego dotychczas napotkanego elementu utrzymywać liczbę jego wystąpień. Otrzymujemy w ten sposób rozwiązanie działające w czasie $\Theta(n \log k)$ i pamięci $\Theta(k)$. Implementacja znajduje się w pliku `kols2.cpp` i na zawodach otrzymywała 30% punktów.

## Gdyby każdy element występował maksymalnie dwukrotnie

Poniżej przedstawiamy dość znany problem. Czytelnikowi polecamy zastanowić się nad jego rozwiązaniem przed przeczytaniem rozwiązania.

*Mamy bardzo długi ciąg liczb, o którym wiadomo, że każda liczba występuje w nim dwukrotnie, oprócz jednego elementu, który nie ma swojego „wspólnika". Zadanie polega na znalezieniu „samotnika" w czasie liniowym i pamięci stałej (zakładamy, że operacje na liczbach są wykonywane w czasie stałym).*

Istnieje bardzo proste rozwiązanie tej zagadki. Okazuje się bowiem, że poszukiwana liczba jest dokładnie alternatywą wykluczającą (tj. `xor`) wszystkich liczb w ciągu.

Operację tę najłatwiej zrozumieć, wypisując w rzędach rozwinięcia binarne rozważanych przez nas liczb. Wynik operacji ma ustawiony $i$-ty bit, gdy nieparzyście wiele `xor`-owanych liczb ma ten bit ustawiony. Poniższy przykład wyznacza `xor` liczb 14, 11, 14, 5 i 11:

$$\begin{array}{rcccc} 14_{(10)} = & 1 & 1 & 1 & 0 \\ 11_{(10)} = & 1 & 0 & 1 & 1 \\ 14_{(10)} = & 1 & 1 & 1 & 0 \\ 5_{(10)} = & 0 & 1 & 0 & 1 \\ 11_{(10)} = & 1 & 0 & 1 & 1 \\ \hline & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} \end{array} = \mathbf{5_{(10)}}.$$

Zauważmy, że jeśli jakaś liczba występuje w ciągu dwukrotnie, to nie wpływa na parzystość bitów w wyniku. Wobec tego na wynik wpływa tylko pojedyncza samotna liczba i to ona jest wynikiem.

Rozwiązanie naiwnie zakładające, że każdy element występuje w ciągu co najwyżej dwukrotnie, znajduje się w pliku `kolb9.cpp`. Nie dostaje żadnych punktów.

## W kierunku dobrego rozwiązania

Na szczęście jednak powyższe przemyślenia można wykorzystać w poprawnym algorytmie, choć wymagającym dwóch przejrzeń ciągu.

Niech $p$ oznacza liczbę wystąpień każdego elementu, którego żadna kopia nie została usunięta z ciągu. Liczbę $p$ możemy wyznaczyć w pierwszym przebiegu po ciągu. Bierzemy bowiem dwa najwcześniejsze różne elementy w ciągu i liczymy, ile razy występują one w ciągu. Oznaczmy te liczby wystąpień przez $p_1, p_2$. Wtedy:

- Jeśli nie znaleźliśmy dwóch różnych elementów w ciągu, oznacza to, że szukaną liczbą jest którykolwiek element ciągu. Jest to przypadek szczególny – pominięcie go narażało zawodników na utratę części punktów.
- Jeśli $p_1 = p_2$, to $p = p_1 = p_2$; wciąż nie wiemy jednak, który element występuje mniej niż $p$ razy.

- Jeśli $p_1 < p_2$, to pierwszy napotkany element w ciągu występuje mniej razy niż inne elementy; to on jest wynikiem. Analogicznie rozpatrujemy przypadek $p_1 > p_2$.

Skoro już wiemy, ile razy powinien wystąpić każdy element w ciągu, możemy wykonać operację podobną do powyższej. Zamiast jednak wyznaczać parzystość liczby wystąpień każdego bitu, możemy wyznaczyć resztę z dzielenia przez $p$ tej liczby. Dla przykładu, dla liczb $14, 11, 11, 5, 14, 14, 5, 11$ dowiadujemy się, że $p = 3$. Wtedy:

$$\begin{array}{rcccc} 14_{(10)} = & 1 & 1 & 1 & 0 \\ 11_{(10)} = & 1 & 0 & 1 & 1 \\ 11_{(10)} = & 1 & 0 & 1 & 1 \\ 5_{(10)} = & 0 & 1 & 0 & 1 \\ 14_{(10)} = & 1 & 1 & 1 & 0 \\ 14_{(10)} = & 1 & 1 & 1 & 0 \\ 5_{(10)} = & 0 & 1 & 0 & 1 \\ 11_{(10)} = & 1 & 0 & 1 & 1 \\ \hline & \mathbf{0} & \mathbf{2} & \mathbf{0} & \mathbf{2} \end{array} \quad \rightarrow \mathbf{5_{(10)}}.$$

Podobnie jak poprzednio, liczby występujące $p$-krotnie w ciągu nie wpływają na wynik. Wobec tego wynik wyznacza liczba występująca $m < p$ razy. Po wyznaczeniu reszty z dzielenia przez $p$ liczby wystąpień kolejnych bitów każdy bit wynikowej liczby wystąpi w wyniku dokładnie $m$ razy.

Podane rozwiązanie wykonuje dwa przebiegi po ciągu i udziela poprawnej odpowiedzi w czasie $\Theta(n \log A)$ i w pamięci $\Theta(\log A)$. Implementacja znajduje się w pliku `kol3.cpp` i otrzymuje 40% punktów.

## Rozwiązanie wzorcowe

Powyższą metodę można nieznacznie zmodyfikować w taki sposób, by wykorzystać tylko jeden przebieg po ciągu.

Zauważmy bowiem, iż podczas rozważania kolejnych elementów nie musimy dla każdego bitu wyniku od razu liczyć reszty z dzielenia przez $p$ liczby elementów ciągu z ustawionym tym bitem. Możemy tę resztę wyznaczyć dopiero po przejrzeniu całego ciągu. Wtedy równolegle możemy też otrzymać wartość $p$. Wynik obliczamy w taki sam sposób, jak poprzednio.

Rozwiązanie zostało zaimplementowane w pliku `kol2.cpp`. Działa w czasie $\Theta(n \log A)$ i pamięci $\Theta(\log A)$, dokonuje jednego przebiegu ciągu i otrzymuje maksymalną punktację.

## Jeszcze szybciej

Za pomocą jednego prostego zabiegu możemy przyspieszyć program. Do tej pory zliczaliśmy bowiem bity, czyli „przytwierdziliśmy się" do systemu pozycyjnego o podstawie 2. Nic jednak nie stoi na przeszkodzie, by tę podstawę zwiększyć do pewnego $d$. Dla ułatwienia rozważań obierzmy $d = 2^{16}$. Wtedy dla każdej pozycji cyfry w liczbie

zliczamy, ile razy każda cyfra wystąpiła na tej pozycji. Dla każdej wynikowej cyfry jej liczba wystąpień będzie liczbą niepodzielną przez $p$.

Ponieważ $d^2 > 2 \cdot 10^9$, to potrzebujemy w obecnej chwili tylko dwóch tablic. Mają one co prawda większy rozmiar ($d$), jednak każda kolejna napotkana liczba wymaga jedynie dwóch operacji na tablicach. Odzyskanie wyniku wymaga jedynie przejrzenia obu tablic.

Zauważmy, że nie możemy ustalić tutaj $d = 2 \cdot 10^9$, gdyż tablica rozmiaru $\Theta(d)$ nie miała prawa zmieścić się w tak małej pamięci.

Rozwiązanie jest zaimplementowane w plikach `kol.cpp`, `kol4.cpp`, `kol5.pas`. Oczywiście również dostaje maksymalną liczbę punktów.

## Rozwiązanie alternatywne

Można było też nieznacznie zmienić typ informacji wyliczanych w rozwiązaniu. Ustalmy $d \geqslant 1$. Możemy w łatwy sposób dla każdej reszty od 0 do $d-1$ wyznaczyć, ile liczb w ciągu ma tę resztę z dzielenia przez $d$. Wtedy ta reszta, której liczba wystąpień jest niepodzielna przez $p$, jest resztą z dzielenia przez $d$ szukanej liczby.

Zamiast jednak zapisywać liczbę w systemie pozycyjnym o podstawie $d$, możemy wybrać kilka różnych liczb $d_i$ i obliczyć resztę z dzielenia wyniku przez $d_i$. W ten sposób mamy wyznaczony wynik, jednak w dość nietypowej postaci – w formie reszt z dzielenia przez kilka wybranych przez nas liczb. Na szczęście umiemy na podstawie podanych liczb odzyskać wynik:

**Twierdzenie 1 (Chińskie twierdzenie o resztach).** *Mamy dane względnie pierwsze liczby naturalne* $1 \leqslant d_1, d_2, \ldots, d_q$ *oraz reszty* $0 \leqslant r_1 < d_1$, $0 \leqslant r_2 < d_2$, $\ldots$, $0 \leqslant r_q < d_q$. *Wtedy istnieje dokładnie jedna liczba całkowita* $x$, *że* $0 \leqslant x < d_1 d_2 \ldots d_q$ *oraz*

$$\begin{aligned} x \bmod d_1 &= r_1, \\ x \bmod d_2 &= r_2, \\ &\ldots, \\ x \bmod d_q &= r_q. \end{aligned}$$

Konstruktywny dowód tego twierdzenia można znaleźć w opracowaniu zadania *Permutacja* z XV Olimpiady Informatycznej [15].

Program `kol6.cpp` ustala $q = 2$ oraz liczby pierwsze $d_1 = 45\,007$ i $d_2 = 45\,011$. Ponieważ $d_1 d_2 > 2 \cdot 10^9$, to wynik jest wyznaczony poprawnie. Rozwiązanie otrzymuje maksymalną liczbę punktów.

## Testy

Ponieważ dane wejściowe mogły być bardzo duże, niemożliwym było wczytywanie całego ciągu z pliku wejściowego. Wobec tego potrzebna była metoda pozwalająca na generowanie kolejnych elementów losowo wyglądającego ciągu, który jednak będzie miał własności opisane w treści zadania. Zainteresowanych Czytelników odsyłamy do implementacji generatora testów `kolingen.cpp`.

**Tomasz Idziaszek**
Treść zadania, Opracowanie

**Kamil Dębowski**
Program

**Dostępna pamięć: 256 MB.** *OI, etap III, dzień drugi, 17.04.2015*

# Modernizacja autostrady

*W Bajtocji znajduje się $n$ miast połączonych gęstą siecią dróg. Niestety, większość z tych dróg jest kiepskiej jakości. W związku z tym w ostatnich latach zbudowano dodatkowo $n-1$ nowoczesnych autostrad. Autostradami można dojechać z każdego miasta do każdego innego, ale ten luksus nie jest za darmo – za przejechanie każdą autostradą trzeba uiścić osobną opłatę.*

*Obywatelom Bajtocji nie podobają się wysokie opłaty za przejazdy autostradami. Aby uspokoić opinię publiczną, minister transportu postanowił zmodernizować sieć autostrad. Tegoroczny budżet pozwoli jedynie na rozbiórkę jednej autostrady i zbudowanie jednej nowej (być może w tym samym miejscu, ale nowocześniejszej). Minister chce wybrać położenie starej i nowej autostrady tak, by nadal można było przejechać autostradami pomiędzy dowolną parą miast i żeby największa liczba autostrad na trasie pomiędzy dwoma miastami była tak mała, jak to tylko możliwe. Taki scenariusz nazywamy* **optymistycznym**.

*Z kolei minister skarbu chciałby, aby modernizacja autostrady przyczyniła się do podreperowania budżetu Bajtocji. Chciałby zatem, żeby po modernizacji nadal można było przejechać autostradami pomiędzy dowolną parą miast, ale żeby największa liczba autostrad na trasie między dwoma miastami była tak duża, jak to tylko możliwe. Taki scenariusz nazywamy* **pesymistycznym**.

*Pogłoski o planach obu ministrów przedostały się do prasy. Dziennikarz Bajtazar pisze na ten temat artykuł, w którym przedstawi najbardziej optymistyczny i najbardziej pesymistyczny scenariusz modernizacji. Pomóż mu i napisz program, który dostarczy mu konkretnych danych do artykułu.*

## Wejście

*W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita $n$ ($3 \leqslant n \leqslant 500\,000$) oznaczająca liczbę miast w Bajtocji. Miasta numerujemy liczbami od 1 do $n$. Dalej następuje $n-1$ wierszy opisujących autostrady. 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 autostrada pomiędzy miastami o numerach $a_i$ i $b_i$.*

*W testach wartych łącznie 30% punktów zachodzi dodatkowy warunek $n \leqslant 1000$.*

## Wyjście

*W pierwszym wierszu standardowego wyjścia należy wypisać pięć liczb całkowitych $k$, $x_1$, $y_1$, $x_2$ i $y_2$ opisujących optymistyczny scenariusz: liczba autostrad na najdłuższej trasie pomiędzy dwoma miastami to $k$, dokonana jest rozbiórka autostrady między miastami $x_1$ i $y_1$ oraz budowana jest autostrada między miastami $x_2$ i $y_2$. W drugim wierszu należy wypisać w takim samym formacie najbardziej pesymistyczny scenariusz. Miasta, między którymi autostradę budujemy lub rozbieramy, można podać w dowolnej kolejności. Jeśli istnieje więcej niż jedno rozwiązanie, Twój program powinien wypisać dowolne z nich.*

### Ocenianie

*Twój program musi wypisać dwa wiersze. Odpowiedź w każdym z nich będzie oceniana niezależnie. Jeśli Twój program poda poprawną odpowiedź tylko dla jednego ze scenariuszy, uzyskasz połowę punktów za test. W takim wypadku zawartość drugiego wiersza nie ma znaczenia.*

### Przykład

| *Dla danych wejściowych:* | *jednym z poprawnych wyników jest:* |
|---|---|
| `6` | `3 4 2 2 5` |
| `1 2` | `5 2 1 1 6` |
| `2 3` | |
| `2 4` | |
| `4 5` | |
| `6 5` | |

**Testy „ocen”:**

`1ocen`: $n = 5$, *gwiazda;*

`2ocen`: $n = 1000$, *ścieżka;*

`3ocen`: $n = 2^{18}$, *każde miasto o numerze* $i > 1$ *jest połączone autostradą z miastem* $\lfloor \frac{i}{2} \rfloor$.

## Rozwiązanie

Sieć autostrad łączących miasta w Bajtocji możemy przedstawić jako graf: jego wierzchołki odpowiadają miastom, a krawędzie – łączącym je autostradom. Graf jest spójny (bo autostradami można przejechać z dowolnego miasta do dowolnego innego) i posiada $n$ wierzchołków i $n - 1$ krawędzi, zatem jest *drzewem*. *Ścieżką* pomiędzy parą wierzchołków w drzewie nazwiemy listę krawędzi, którymi musimy przejść, aby dostać się z jednego z tych wierzchołków do drugiego. Liczbę krawędzi na liście nazywamy *długością* tej ścieżki. Nas będą interesować najdłuższe ścieżki w drzewie; każdą taką ścieżkę nazywamy *średnicą*.

Zadanie polega na usunięciu jednej krawędzi z drzewa i dodaniu jednej nowej tak, by graf pozostał drzewem i żeby po tej operacji miał jak najmniejszą (jak największą) średnicę.

### Optymalne łączenie poddrzew

Załóżmy, że wiemy, którą z krawędzi należy usunąć, i zastanawiamy się, które wierzchołki należy połączyć, by osiągnąć jeden z podanych celów. Jeśli usuniemy ustaloną krawędź, to drzewo rozpadnie się na dwa poddrzewa – nazwijmy je poddrzewami $T_A$ i $T_B$. Aby graf pozostał drzewem, musimy dodać krawędź, która łączy pewien wierzchołek z poddrzewa $T_A$ z pewnym wierzchołkiem z poddrzewa $T_B$.

Nietrudno się przekonać, że jeśli chcemy połączyć te dwa poddrzewa tak, by średnica nowo powstałego drzewa $T$ była jak *największa*, to najlepiej jest połączyć wierzchołki, które są końcami średnic w poddrzewach $T_A$ i $T_B$. Dla ustalenia uwagi, niech $d_A$ i $d_B$ będą długościami średnic w poddrzewach $T_A$ i $T_B$, natomiast $v_A$ i $v_B$ będą pewnymi wierzchołkami leżącymi na końcach tych średnic. Połączenie krawędzią wierzchołków $v_A$ i $v_B$ da nam drzewo zawierające ścieżkę $p_{\max}$ o długości

$$d_{\max} = d_A + 1 + d_B.$$

Pokażemy teraz, że ścieżka ta jest średnicą. Niech $p$ będzie dowolną średnicą drzewa $T$. Zauważmy, że średnica $p$ nie może w całości zawierać się ani w poddrzewie $T_A$, ani w poddrzewie $T_B$, bo miałaby wtedy długość co najwyżej $\max(d_A, d_B) < d_{\max}$. Musi zatem przechodzić nowo dodaną krawędzią; niech $p_A$ i $p_B$ oznaczają długości kawałków tej średnicy, które leżą odpowiednio w poddrzewach $T_A$ i $T_B$ (więc długość średnicy $p$ to $p_A + 1 + p_B$). Z definicji średnic mamy $p_A \leqslant d_A$ i $p_B \leqslant d_B$, zatem $p_A + 1 + p_B \leqslant d_{\max}$. To pokazuje, że ścieżka $p_{\max}$ jest również średnicą drzewa $T$. W podobny sposób można pokazać, że połączenie poddrzew $T_A$ i $T_B$ dowolną inną krawędzią nie da drzewa o średnicy większej niż $d_{\max}$.

Połączenie poddrzew $T_A$ i $T_B$ tak, by średnica nowo powstałego drzewa $T$ była jak *najmniejsza*, jest trochę trudniejsze. Średnica drzewa $T$ może albo znajdować się w całości w jednym z poddrzew $T_A$ lub $T_B$ (i wtedy ma długość $d_A$ lub $d_B$), albo przechodzić nowo dodaną krawędzią. Tę krawędź powinniśmy wybrać tak, by zminimalizować odległości jej końców do najdalszych wierzchołków poddrzew $T_A$ i $T_B$.

Zauważmy, że skoro poddrzewo $T_A$ ma średnicę o długości $d_A$, to odległość między dowolnym wierzchołkiem $u$ poddrzewa $T_A$ a tym końcem średnicy, który znajduje się dalej od wierzchołka $u$, wynosi co najmniej $\lceil d_A/2 \rceil$. Z kolei dla wierzchołka $u_A$, który jest środkiem średnicy (jeśli długość średnicy jest nieparzysta, to jako środek można wybrać dowolny z dwóch wierzchołków leżących na środkowej krawędzi), odległość do dowolnego innego wierzchołka poddrzewa $T_A$ wynosi co najwyżej $\lceil d_A/2 \rceil$.

Wynika z tego, że jeśli połączymy krawędzią wierzchołki $u_A$ i $u_B$, będące środkami średnic poddrzew $T_A$ i $T_B$, to w tak powstałym drzewie $T$ długość najdłuższej ścieżki przechodzącej nowo dodaną krawędzią będzie wynosić $\lceil d_A/2 \rceil + 1 + \lceil d_B/2 \rceil$, i co więcej, połączenie tych poddrzew dowolną inną krawędzią nie da krótszej ścieżki. Podsumowując, najmniejsza średnica, jaką da się uzyskać, ma długość

$$d_{\min} = \max(d_A,\ d_B,\ \lceil d_A/2 \rceil + 1 + \lceil d_B/2 \rceil).$$

## Rozwiązanie kwadratowe

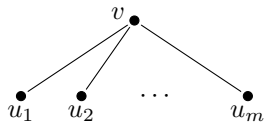
Rozpatrujemy niezależnie każdą z $n-1$ krawędzi drzewa jako kandydata do usunięcia. To wyznacza nam podział drzewa na dwa poddrzewa $T_A$ i $T_B$. Dla każdego z nich znajdujemy następujące wartości: długość średnicy ($d_A$, $d_B$), jeden z jej końców ($v_A$, $v_B$) oraz środek ($u_A$, $u_B$). Jako kandydata na scenariusz pesymistyczny rozpatrujemy dodanie krawędzi między wierzchołkami $v_A$ i $v_B$, co daje średnicę o długości $d_A + 1 + d_B$. Jako kandydata na scenariusz optymistyczny rozpatrujemy dodanie krawędzi między wierzchołkami $u_A$ i $u_B$, co daje średnicę o długości $\max(d_A, d_B, \lceil d_A/2 \rceil + 1 + \lceil d_B/2 \rceil)$.

Znajdowanie średnicy w poddrzewie $T_A$ (a potem niezależnie w poddrzewie $T_B$) wykonujemy standardowym algorytmem opartym o metodę programowania dynamicznego. Ukorzeniamy drzewo w dowolnym wierzchołku $v_0$. Następnie, zaczynając od liści i poruszając się w górę drzewa, dla każdego węzła $v$ wyznaczamy dwie wartości: $d[v]$ – średnicę poddrzewa zaczepionego w węźle $v$ oraz $h[v]$ – wysokość poddrzewa zaczepionego w $v$.

Jeśli $v$ jest liściem, to oczywiście $h[v] = d[v] = 0$. W ogólnym przypadku, gdy węzeł $v$ posiada $m$ 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:

$$h[v] = 1 + \max_i h[u_i],$$
$$d[v] = \max\big(\max_i d[u_i],\ h[v],\ \max_{i \neq j}(h[u_i] + 2 + h[u_j])\big).$$

Poprawność wzoru na $d[v]$ wynika z faktu, że średnica w poddrzewie $v$ może albo nie przechodzić przez węzeł $v$ (czyli być w całości zawarta w poddrzewie zaczepionym w jednym z synów węzła $v$), albo kończyć się w węźle $v$, albo przechodzić przez węzeł $v$.




Wyznaczenie wartości $h[v]$ i $d[v]$ możemy zaimplementować w czasie liniowym od liczby synów $m$. Tak więc wyznaczenie długości średnicy całego drzewa (czyli $d[v_0]$) możemy zaimplementować w czasie $O(n)$.

Aby wyznaczyć końce średnic, będziemy dla poddrzewa zaczepionego w każdym wierzchołku $v$ pamiętali nie tylko jego wysokość $h[v]$ i długość średnicy $d[v]$, lecz także dowolny z najgłębszych liści w tym poddrzewie $\ell[v]$ oraz parę węzłów $\alpha[v], \beta[v]$ będącą końcami tej średnicy. Uaktualnianie tych wartości nie jest trudne, przykładowo:

$$\alpha[v], \beta[v] = \begin{cases} \alpha[u_i], \beta[u_i] & \text{jeśli } d[v] = d[u_i] \text{ dla pewnego } i, \\ v, \ell[v] & \text{jeśli } d[v] = h[v], \\ \ell[u_i], \ell[u_j] & \text{jeśli } d[v] = h[u_i] + 2 + h[u_j] \text{ dla pewnych } i \neq j. \end{cases}$$

Mając końce średnicy całego drzewa $\alpha[v_0], \beta[v_0]$, bez kłopotu możemy wyznaczyć jej środek, przeszukując drzewo. Zatem wyznaczenie wartości $d_A$, $v_A$ i $u_A$ dla ustalonego poddrzewa $T_A$ zajmie czas $O(n)$.

Tak więc złożoność czasowa całego rozwiązania to $O(n^2)$. Implementacja tego typu rozwiązania znajduje się w pliku `mods1.cpp`.

## Rozwiązanie liniowe

Zauważmy, że powyższy algorytm wyznaczający średnicę w drzewie wyznacza również średnice niektórych poddrzew tego drzewa. Zmodyfikujemy go teraz tak, by wyznaczał średnice *wszystkich* poddrzew w sumarycznym czasie $O(n)$, dzięki czemu uzyskamy algorytm o złożoności liniowej.

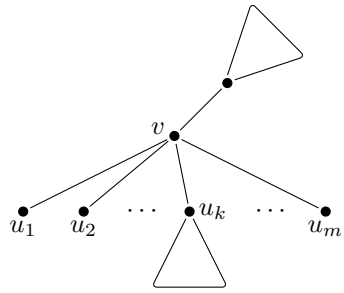
Ukorzeńmy nasze początkowe drzewo w pewnym wierzchołku $v_0$ i wykonajmy powyższy algorytm oparty na programowaniu dynamicznym. Rozpatrzmy dowolną krawędź drzewa jako kandydata do usunięcia; niech to będzie krawędź łącząca pewien węzeł $v$ z jego ojcem. Usunięcie tej krawędzi podzieli nam drzewo na poddrzewo $T_A$ (poddrzewo zaczepione w węźle $v$) oraz poddrzewo $T_B$ (reszta drzewa zawierająca ojca węzła $v$; nazwiemy ją *dopełnieniem* poddrzewa $v$). Zauważmy, że $d_A = d[v]$, pozostaje zatem obliczyć średnicę poddrzewa $T_B$. Zrobimy to znowu programowaniem dynamicznym. Tym razem jednak zaczniemy od korzenia i będziemy poruszać się w dół drzewa, licząc wartości: $D[v]$ – średnica dopełnienia poddrzewa $v$ oraz $H[v]$ – długość najdłuższej ścieżki w dopełnieniu poddrzewa $v$ zaczynającej się w ojcu $v$.

Ponieważ w przypadku korzenia $v_0$ dopełnienie poddrzewa $v_0$ nie istnieje (bo nie istnieje krawędź do ojca), możemy przyjąć $H[v_0] = D[v_0] = -\infty$. W ogólnym przypadku węzła $v$, dla którego obliczyliśmy już wartości $D[v]$ i $H[v]$, następująca rekurencja pozwala wyznaczyć wartości dla jego synów:

$$H[u_k] = \max\big(0,\ 1 + H[v],\ \max_{i\neq k}(1 + h[u_i])\big),$$

$$D[u_k] = \max\big(D[v],\ \max_{i\neq k} d[u_i],\ H[u_k],\ \max_{i\neq k}(H[v] + 2 + h[u_i]),\ \max_{i\neq j\neq k\neq i}(h[u_i] + 2 + h[u_j])\big).$$

Poprawność wzoru na $D[u_k]$ wynika z faktu, że średnica w dopełnieniu poddrzewa $u_k$ może albo nie przechodzić przez węzeł $v$ (i być w całości zawarta w dopełnieniu poddrzewa $v$ lub w całości zawarta w poddrzewie jednego z synów węzła $v$), albo kończyć się w węźle $v$, albo przechodzić przez węzeł $v$ (i przechodzić przez ojca $v$ lub nie).




Całość obliczeń dla węzła $v$ o $m$ synach można zaimplementować w czasie $O(m)$. Przykładowo, aby szybko wyznaczać składnik $\max_{i\neq j\neq k\neq i}(h[u_i]+2+h[u_j])$, wystarczy na początku w czasie $O(m)$ wyznaczyć *trzy* indeksy $i_1$, $i_2$, $i_3$, które odpowiadają największym trzem elementom $h[u_{i_1}]$, $h[u_{i_2}]$ oraz $h[u_{i_3}]$, a następnie dla kolejnych wartości $k$ w czasie stałym wybierać największe dwa ze zbioru $\{i_1, i_2, i_3\} \setminus \{k\}$.

Pozostaje wyznaczyć końce i środki średnic. O ile jednak potrzebowaliśmy znać długości średnic dla wszystkich poddrzew, to końców i środków potrzebujemy jedynie dla dwóch poddrzew z optymalnego podziału. Możemy więc je wyznaczyć, korzystając z algorytmu opisanego w poprzednim rozdziale.

W ten sposób otrzymujemy rozwiązanie wzorcowe, działające w czasie liniowym. Przykładową implementację można znaleźć w pliku `mod.cpp`.

**Karol Pokorski**
Treść zadania, Opracowanie

**Marcin Andrychowicz**
Program

**Dostępna pamięć: 128 MB.**

*OI, etap III, dzień drugi, 17.04.2015*

# Wycieczki

*Bajtazar odkrył piękno wycieczek rowerowych. Swój $k$-dniowy urlop zamierza spędzić w przepięknej Bajtocji. Każdego dnia chce wybrać się na wycieczkę rowerową biegnącą inną trasą. Chciałby stopniować poziom trudności wycieczek, zatem każda kolejna wycieczka nie powinna być krótsza od poprzedniej. A dokładniej: $i$-tego dnia Bajtazar chce zrobić wycieczkę $i$-tą najkrótszą możliwą trasą biegnącą przez miasta Bajtocji. Pomóż Bajtazarowi obliczyć długość ostatniej wycieczki, na którą wybierze się $k$-tego dnia urlopu.*

*W Bajtocji znajduje się $n$ miast, ponumerowanych od 1 do $n$. Miasta są połączone jednokierunkowymi drogami, a długość każdej drogi wynosi 1, 2 lub 3 kilometry. Drogi mogą prowadzić tunelami i estakadami. Rozważamy wycieczki zaczynające się i kończące w dowolnych miastach Bajtocji. Dopuszczamy też wycieczki prowadzące przez dane miasto lub drogę wielokrotnie.*

## Wejście

*Pierwszy wiersz standardowego wejścia zawiera trzy liczby całkowite $n$, $m$ i $k$ ($1 \leqslant n \leqslant 40$, $1 \leqslant m \leqslant 1000$, $1 \leqslant k \leqslant 10^{18}$) pooddzielane pojedynczymi odstępami, oznaczające liczbę miast, liczbę dróg i liczbę dni urlopu. W kolejnych $m$ wierszach znajdują się opisy dróg, po jednej w wierszu. Opis każdej drogi składa się z trzech liczb całkowitych $u$, $v$ i $c$ ($1 \leqslant u, v \leqslant n$, $u \neq v$, $1 \leqslant c \leqslant 3$) pooddzielanych pojedynczymi odstępami, oznaczających jednokierunkową drogę z miasta $u$ do miasta $v$ o długości $c$ kilometrów. Pomiędzy parą miast może być więcej niż jedna droga.*

*W testach wartych łącznie 75% punktów zachodzą dodatkowo warunki $n \leqslant 15$, $m \leqslant 200$, $k \leqslant 10^{12}$. Ponadto w podzbiorze tych testów wartym łącznie 50% punktów zachodzi dodatkowo dla każdej drogi $c = 1$. W końcu w podzbiorze tych testów wartym łącznie 25% punktów zachodzi jeszcze dodatkowo warunek $k \leqslant 1\,000\,000$.*

## Wyjście

*W pierwszym i jedynym wierszu standardowego wyjścia Twój program powinien wypisać długość $k$-tej najkrótszej wycieczki. Jeśli jest mniej niż $k$ różnych wycieczek (czyli Bajtazar będzie zmuszony zakończyć swój urlop wcześniej), należy wypisać $-1$.*

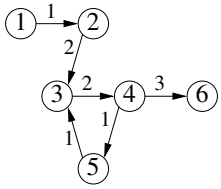
## Przykład

*Dla danych wejściowych:*

```
6 6 11
1 2 1
2 3 2
3 4 2
4 5 1
5 3 1
4 6 3
```

*poprawnym wynikiem jest:*

```
4
```




**Wyjaśnienie do przykładu:**
*Wycieczki długości 1:* $1 \to 2$, $5 \to 3$, $4 \to 5$.
*Wycieczki długości 2:* $2 \to 3$, $3 \to 4$, $4 \to 5 \to 3$.
*Wycieczki długości 3:* $4 \to 6$, $1 \to 2 \to 3$, $3 \to 4 \to 5$, $5 \to 3 \to 4$.
*Jedenastą najkrótszą wycieczką (długości 4) może być na przykład:* $5 \to 3 \to 4 \to 5$.

**Testy „ocen":**

`1ocen`: $n = 10$, $k = 46$; *drogi o losowych długościach tworzące ścieżkę; istnieje tylko 45 możliwych wycieczek, więc poprawną odpowiedzią jest* $-1$;

`2ocen`: $n = 15$, $k = 10^{12}$; *z każdego miasta do każdego innego jest droga o długości 3.*

# Rozwiązanie

W zadaniu dany jest niewielki skierowany graf $G$ zawierający $n$ wierzchołków ($n \leqslant 40$). W grafie mogą występować krawędzie wielokrotne. Waga każdej krawędzi należy do zbioru $\{1, 2, 3\}$. Należy wyznaczyć długość $k$-tej najkrótszej ścieżki w tym grafie.

## Rozwiązanie dla jednostkowych wag krawędzi

Spróbujmy najpierw rozwiązać łatwiejszą wersję zadania, w której wagi wszystkich krawędzi są równe 1.

Graf podany na wejściu można przedstawić w postaci macierzy sąsiedztwa $M$, w której wyraz $M_{u,v}$ będzie równy liczbie krawędzi prowadzących z wierzchołka $u$ do wierzchołka $v$.

### Mnożenie macierzy

Powiemy, że macierz $M$ jest wymiaru $p \times q$, jeżeli zawiera $p$ wierszy i $q$ kolumn. Reprezentowanie macierzy w pamięci komputera jest bardzo łatwe i sprowadza się do stworzenia tablicy dwuwymiarowej. Co ważniejsze jednak, do rozwiązania zadania będziemy używać własności macierzy jako pojęcia matematycznego z algebry liniowej, nie zaś jedynie jako reprezentacja informacji w pamięci komputera.

Jedną z podstawowych operacji dotyczących macierzy jest możliwość ich mnożenia. W wyniku pomnożenia macierzy $A$ wymiaru $p \times q$ i macierzy $B$ wymiaru $q \times r$ uzyskujemy macierz $A \cdot B$ wymiaru $p \times r$, w której każdy wyraz możemy obliczyć następującym wzorem:

$$(A \cdot B)_{u,v} = \sum_{w=1}^{q} A_{u,w} \cdot B_{w,v}. \tag{1}$$

Operacja mnożenia macierzy, w przeciwieństwie do mnożenia liczb, nie jest operacją przemienną, jest jednak operacją łączną. Dodatkowo warty odnotowania jest fakt, że macierze $A$ i $B$ nie muszą być kwadratowe, aby móc je pomnożyć, jednak powyższy wzór (i definicja operacji mnożenia macierzy) wymusza, żeby liczba kolumn macierzy $A$ była równa liczbie wierszy macierzy $B$. Do rozwiązania zadania będziemy używać jedynie macierzy kwadratowych, co dodatkowo ułatwia analizę problemu (oraz implementację rozwiązania).

Obliczenie iloczynu dwóch macierzy wymiaru $p \times p$ można wykonać bezpośrednio ze wzoru (1) w czasie $O(p^3)$, gdyż obliczenie każdego z $p^2$ wyrazów macierzy wynikowej zajmuje czas liniowy względem $p$.

Istnieją efektywniejsze metody mnożenia macierzy, na przykład algorytm Strassena ($O(p^{\log_2 7})$) lub algorytm Coppersmitha-Winograda ($O(p^{2,38})$). Zastosowanie tych algorytmów niekoniecznie jednak jest użyteczne na zawodach, ze względu na trudność implementacji oraz stosunkowo duży stały czynnik ukryty w notacji asymptotycznej.

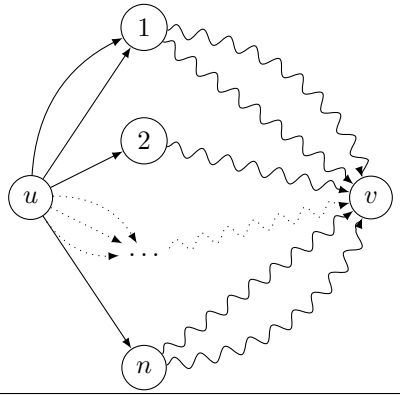
### Potęgowanie macierzy a ścieżki w grafie

Oprócz mnożenia macierzy można zdefiniować operację potęgowania macierzy $M$ (tym razem już tylko kwadratowej). Dla potęg całkowitych dodatnich będzie ona określona podobnie do mnożenia liczb: $M^1 = M$, $M^{t+1} = M \cdot M^t$.

Okazuje się, że operacja potęgowania macierzy ma bardzo silne powiązanie z wyznaczaniem liczby ścieżek w grafie. Dokładniej, wyraz w $u$-tym wierszu i $v$-tej kolumnie $t$-tej potęgi macierzy sąsiedztwa $M$ grafu $G$ jest równy liczbie różnych ścieżek $t$-krawędziowych z wierzchołka $u$ do wierzchołka $v$.

Fakt ten możemy udowodnić, korzystając z indukcji matematycznej: dla $t = 1$ każdy wyraz macierzy $M_{u,v}$ jest po prostu liczbą krawędzi (czyli ścieżek składających się z jednej krawędzi) pomiędzy $u$ i $v$. Dowolną ścieżkę $(t+1)$-krawędziową (dla $t \geqslant 1$) pomiędzy $u$ i $v$ możemy zawsze podzielić na dwie części: część pierwszą (nazwijmy ją *krótką*) jednokrawędziową z wierzchołka $u$ do pewnego wierzchołka pośredniego $w$ oraz część drugą (nazwijmy ją *długą*) $t$-krawędziową z wierzchołka $w$ do $v$ (rys. 1). Aby zliczyć wszystkie ścieżki $(t+1)$-krawędziowe pomiędzy wierzchołkami $u$ i $v$ wystarczy zatem zsumować liczbę takich ścieżek dla każdego wierzchołka pośredniego $w$. Dla ustalonego wierzchołka pośredniego $w$ dowolna krótka ścieżka z $u$ do $w$ (jest ich $M_{u,w}$) z dowolną długą ścieżką z $w$ do $v$ (na mocy założenia indukcyjnego jest ich $(M^t)_{w,v}$) tworzą poprawną $(t+1)$-krawędziową ścieżkę z $u$ do $v$. Zatem liczba tych ścieżek to:

$$\sum_{w=1}^{n} M_{u,w} \cdot (M^t)_{w,v} = (M \cdot M^t)_{u,v} = (M^{t+1})_{u,v}.$$




Rys. 1: Ilustracja dowodu kroku indukcyjnego: ścieżki $(t+1)$-krawędziowe z wierzchołka $u$ do wierzchołka $v$. Krótkie krawędzie reprezentują ścieżki jednokrawędziowe (część krótką), zaś krawędzie falowane reprezentują ścieżki $t$-krawędziowe (część długą).

Bezpośredni algorytm obliczania $t$-tej potęgi macierzy wymiaru $n \times n$ działa w czasie $O(n^3 \cdot t)$. Jednak dzięki temu, że operacja mnożenia macierzy jest łączna, można zastosować tę samą technikę co w szybkim potęgowaniu liczb: $M^{2t} = M^t \cdot M^t$ oraz $M^{2t+1} = M \cdot M^t \cdot M^t$. W ten sposób można zredukować wykładnik potęgi dwukrotnie, wykonując maksymalnie dwa mnożenia macierzy, co prowadzi do redukcji liczby mnożeń z liniowej do logarytmicznej względem $t$ i złożoności czasowej $O(n^3 \cdot \log t)$. Można na to też spojrzeć jak na wykonanie innego podziału ścieżki na część krótką i długą – wykonując bardziej zrównoważony podział, można sprowadzić większy problem (obliczenia potęgi $t$ macierzy $M$) do dwóch takich samych, ale sporo mniejszych problemów (obliczenia potęgi $\lfloor \frac{t}{2} \rfloor$ macierzy $M$).

### Zliczanie ścieżek długości co najwyżej $\ell$

Na podstawie rozumowania z potęgowaniem macierzy można wyznaczyć liczbę ścieżek długości równej $\ell$ (po prostu licząc $M^\ell$ i sumując wszystkie wyrazy otrzymanej macierzy).

Co jednak, jeśli chcielibyśmy wyznaczyć liczbę ścieżek o długości co najwyżej $\ell$? Można stworzyć graf $G'$ poprzez dodanie do grafu $G$ dodatkowego wierzchołka startowego $s$, z którego wychodzą pojedyncze krawędzie jednostkowej długości do wszystkich pozostałych wierzchołków z grafu $G$, a także pętla do wierzchołka $s$. Wówczas liczba ścieżek długości dokładnie $\ell+1$ z $s$ do wszystkich pozostałych wierzchołków (poza $s$) w grafie $G'$ jest równa liczbie ścieżek długości co najwyżej $\ell$ w oryginalnym grafie $G$. Faktycznie, ścieżka $u \rightsquigarrow v$ długości $t \leqslant \ell$ w grafie $G$ odpowiada ścieżce długości $\ell+1$ w grafie $G'$, w której najpierw $\ell - t$ razy przechodzimy pętlą przy wierzchołku $s$, potem krawędzią z $s$ do $u$ i na końcu ścieżką $u \rightsquigarrow v$.

### Właściwe rozwiązanie

Wróćmy do zadania wyznaczenia $k$-tej najkrótszej ścieżki w grafie $G$. Wiadomo, że jeśli rozwiązanie istnieje, to musi mieć długość co najwyżej $k$. Gdyby tak nie było, czyli

gdyby szukana ścieżka miała długość większą niż $k$, to rozpatrując wszystkie prefiksy tejże ścieżki, uzyskalibyśmy co najmniej $k$ ścieżek od niej krótszych – sprzeczność.

A zatem skoro wiadomo, że długość szukanej ścieżki (wykładnik potęgi macierzy $M$) musi być równa co najmniej 1 oraz co najwyżej $k$, rozwiązanie zadania nasuwa się samo: możemy zastosować wyszukiwanie binarne długości ścieżki. Należy znaleźć takie $\ell$, że ścieżek długości mniejszej niż $\ell$ będzie mniej niż $k$, zaś ścieżek długości co najwyżej $\ell$ będzie już co najmniej $k$. W ten sposób uzyskaliśmy rozwiązanie uproszczonej wersji zadania (dla wag jednostkowych krawędzi) działające w czasie $O(n^3 \log^2 k)$. Zgodnie z uwagą w treści zadania takie rozwiązanie warte było na zawodach 50% punktów. Implementację tego rozwiązania Czytelnik znajdzie w pliku `wycb1.cpp`.

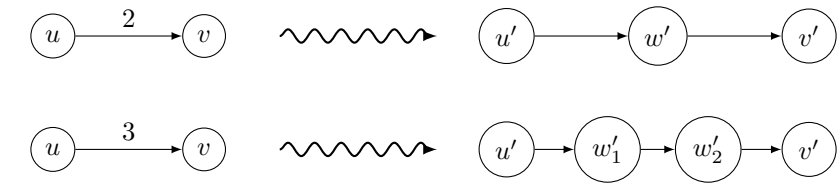
Rozwiązanie to można nieco poprawić. Zamiast wyszukiwania binarnego można najpierw obliczyć liczby ścieżek o długościach co najwyżej 1, 2, 4, 8, ... (kolejne potęgi dwójki), aż liczba ścieżek przekroczy $k$. W ten sposób obliczymy co najwyżej $\log k$ macierzy: $M$, $M^2$, $M^4$, $M^8$, ... w czasie $O(n^3 \log k)$, a także ustalimy pozycję najstarszego zapalonego bitu wyniku (długości szukanej ścieżki), czyli znajdziemy takie $M^{2^p}$, które daje co najwyżej $k$ ścieżek, że $M^{2^{p+1}}$ daje więcej niż $k$ ścieżek. Teraz można próbować ustalać kolejne bity wyniku (od lewej do prawej) poprzez domnażanie do ostatniej uzyskanej macierzy ($M^{2^p}$) wcześniej obliczonych macierzy dla coraz mniejszych wykładników ($M^{2^{p-1}}$, $M^{2^{p-2}}$, ...).

Zobrazujmy powyższy pomysł na krótkim przykładzie, w którym poszukiwany wynik wynosi $\ell = 11$. Najpierw, obliczając kolejno $M^1$, $M^2$, $M^4$, $M^8$, algorytm uzyskuje nie więcej niż $k$ ścieżek, zaś w $M^{16}$ jest już ich więcej niż $k$. To znaczy, że wynik jest pomiędzy 8 a 15 (ma cztery bity). Do macierzy $M^8$ można domnożyć uprzednio obliczoną $M^4$, co da w wyniku $M^{12}$, której suma liczby ścieżek będzie zbyt duża (większa niż $k$), więc wynik musi być pomiędzy 8 a 11 (drugi najstarszy bit jest zgaszony). Następnie do macierzy $M^8$ można domnożyć uprzednio obliczoną $M^2$, co da $M^{10}$ i mniej niż $k$ ścieżek, czyli trzeci bit wyniku jest równy 1. Na końcu do macierzy $M^{10}$ algorytm domnaża $M^1$ i sprawdza, że wynikiem jest $\ell = 11$.

Łącznie mnożeń macierzy w tej fazie będzie też $O(\log k)$, co powoduje, że złożoność całego rozwiązania redukuje się do $O(n^3 \log k)$.

## Rozwiązanie dla krawędzi o wagach $\{1, 2, 3\}$

W macierzy sąsiedztwa $M$ dla grafu $G$ zapamiętujemy jedynie liczbę krawędzi pomiędzy wierzchołkami. Na przechowanie wag krawędzi nie ma już miejsca. Teoretycznie można by rozważyć dodanie dodatkowych wierzchołków pomiędzy końcami krawędzi o wagach 2 i 3; patrz rys. 2.



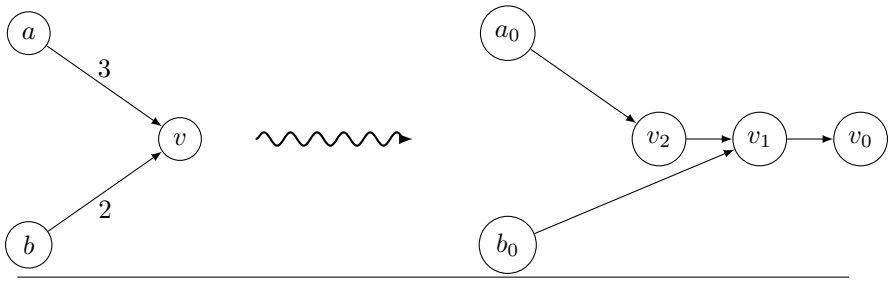

Rys. 2: Dodatkowe wierzchołki pomiędzy końcami krawędzi.

Niestety, taka metoda w pesymistycznym przypadku powoduje zwiększenie liczby wierzchołków o dwukrotność liczby krawędzi w oryginalnym grafie. Rozwiązanie oparte o ten pomysł jest więc zbyt wolne.

Wystarczy jednak zauważyć, że wierzchołki pośrednie dla różnych krawędzi można uwspólnić. Dokładniej: dla każdego wierzchołka $u$ można stworzyć jego trzy kopie: $u_2$, $u_1$ oraz $u_0$. Będą one określały kolejno:

- $u_2$: wierzchołek pośredni oznaczający, że do wierzchołka $u$ pozostały dwie krawędzie,
- $u_1$: wierzchołek pośredni oznaczający, że do wierzchołka $u$ pozostała jedna krawędź,
- $u_0$: oryginalny wierzchołek $u$ z grafu.

Taka definicja pozwala uwspólniać wierzchołki pośrednie i działa nawet wtedy, gdy wagi krawędzi są różne (rys. 3).




Rys. 3: Uwspólnienie wierzchołków pomiędzy końcami krawędzi o różnej długości.

Ostatecznie mamy gwarancję, że liczba wierzchołków w nowym grafie, w którym krawędzie są już jednostkowej długości, nie przekracza $3n$. Tak oto uzyskujemy rozwiązanie o złożoności $O(n^3 \log k)$, które na zawodach otrzymywało maksymalną punktację.

Aby uzyskać liczbę ścieżek długości co najwyżej $\ell$, postępujemy analogicznie jak w rozwiązaniu dla jednostkowych wag krawędzi, z tą tylko różnicą, że do wierzchołka startowego podłączone są jedynie oryginalne wierzchołki grafu (z indeksem 0) oraz zliczamy ścieżki kończące się tylko w tych wierzchołkach.

Implementację takiego rozwiązania można odnaleźć w plikach `wyc.cpp` oraz `wyc0.pas`.

# XXVII Międzynarodowa Olimpiada Informatyczna,

*Ałmaty, Kazachstan 2015*

**Krzysztof Diks, Lech Duraj**
Tłumaczenie

**Dostępna pamięć: 1500 MB, limit czasu: 2 s.** *IOI 2015, dzień pierwszy, 28.07.2015*

# Pamiątki

*Trwa ostatnia część ceremonii otwarcia IOI 2015. W czasie ceremonii każda reprezentacja powinna była otrzymać pudełko z prezentem – pamiątką od organizatorów. Niestety, prawie wszyscy wolontariusze zapatrzyli się na ceremonię, przez co zapomnieli o rozdaniu pudełek. Jedyną osobą, która pamięta o prezentach, jest Aman. Aman jest bardzo zaangażowanym wolontariuszem i chce, aby Olimpiada udała się jak najlepiej. Postanowił więc sam roznieść wszystkie pamiątki, i to możliwie najszybciej.*

*Ceremonia odbywa się w sali, której widownia ma kształt okręgu podzielonego na $L$ identycznych sekcji, ponumerowanych od 0 do $L-1$. Dla każdego $0 \leqslant i \leqslant L-2$, sekcja $i$ sąsiaduje z sekcją $i+1$, a dodatkowo sekcja $L-1$ sąsiaduje z sekcją 0. Na sali jest $N$ reprezentacji. Każda z nich siedzi w jednej spośród sekcji. Kilka reprezentacji może siedzieć w tej samej sekcji. Niektóre sekcje mogą być puste.*

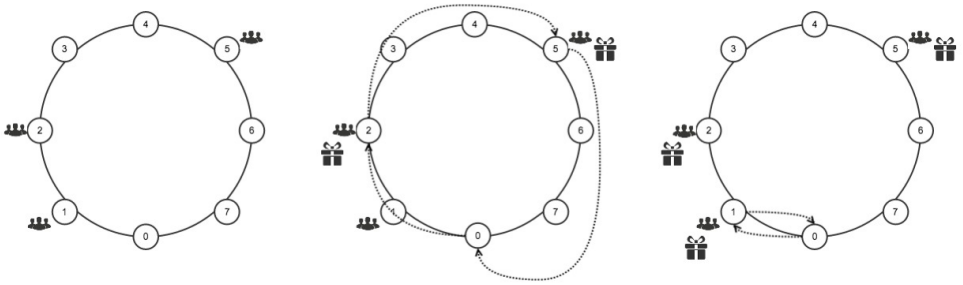
*Do rozdania jest $N$ identycznych pamiątek. Na początku Aman znajduje się w sekcji 0, w której znajdują się wszystkie pamiątki. Aman powinien doręczyć każdej reprezentacji jedną pamiątkę, a po zakończeniu zadania powrócić do sekcji 0 (zwróć uwagę, że w sekcji 0 również mogą znajdować się reprezentacje).*

*Aman może naraz nieść co najwyżej $K$ pamiątek. Musi je za każdym razem zabrać z sekcji 0, przy czym wzięcie pamiątek nie zajmuje mu czasu. Po zabraniu pamiątek Aman musi je nieść tak długo, aż wszystkie rozda reprezentacjom. Kiedy Aman niesie co najmniej jedną pamiątkę i znajduje się w sekcji, w której jest nieobdarowana jeszcze reprezentacja, może tej reprezentacji wręczyć jedną pamiątkę. To również dzieje się bez straty czasu. Jedyne, na co Aman potrzebuje czasu, to chodzenie pomiędzy sekcjami – przejście do sąsiedniej sekcji zawsze zajmuje mu dokładnie jedną sekundę. Aman może chodzić po okręgu w obie strony, a jego prędkość nie zależy od tego, ile niesie ze sobą pamiątek.*

*Oblicz, ile wynosi najmniejsza liczba sekund, jakiej Aman potrzebuje na doręczenie wszystkich pamiątek i powrót do punktu wyjścia.*

## Przykład

*W przykładzie dane są $N = 3$ reprezentacje, Aman może naraz unieść $K = 2$ pamiątki, zaś na widowni jest $L = 8$ sekcji. Reprezentacje siedzą w sekcjach 1, 2 i 5.*



*Jedno z możliwych optymalnych rozwiązań podane jest na rysunku powyżej. Aman bierze najpierw dwie pamiątki, wręcza jedną z nich reprezentacji w sekcji 2, a drugą reprezentacji w sekcji 5, po czym wraca do sekcji 0. Łącznie zajmuje mu to 8 sekund. Następnie Aman doręcza pamiątkę do sekcji 1 i wraca z powrotem do sekcji 0, co zajmuje mu 2 sekundy. Całkowity czas wynosi zatem 10 sekund.*

## Zadanie

*Dane są liczby $N$, $K$, $L$, a także miejsca, w których siedzą wszystkie reprezentacje. Oblicz, ile sekund potrzebuje Aman na doręczenie wszystkich pamiątek. W tym celu musisz zaimplementować funkcję* `delivery`*:*

- `delivery(N, K, L, positions)` – *funkcja ta zostanie wywołana przez program sprawdzający dokładnie raz.*
  - `N`*: liczba reprezentacji.*
  - `K`*: maksymalna liczba pamiątek, jaką może unieść Aman.*
  - `L`*: liczba sekcji na widowni.*
  - `positions`*: tablica długości* `N`*. Liczby* `positions[0]`*, ...,* `positions[N-1]` *to numery sekcji, które zajmują kolejne reprezentacje. Elementy tablicy* `positions` *są uporządkowane niemalejąco.*
  - *Funkcja powinna zwracać najmniejszą możliwą liczbę sekund, w której Aman może zrealizować swoje zadanie.*

## Podzadania

| **podzadanie** | **liczba punktów** | $N$ | $K$ | $L$ |
|---|---|---|---|---|
| *1* | *10* | $1 \leqslant N \leqslant 1000$ | $K = 1$ | $1 \leqslant L \leqslant 10^9$ |
| *2* | *10* | $1 \leqslant N \leqslant 1000$ | $K = N$ | $1 \leqslant L \leqslant 10^9$ |
| *3* | *15* | $1 \leqslant N \leqslant 10$ | $1 \leqslant K \leqslant N$ | $1 \leqslant L \leqslant 10^9$ |
| *4* | *15* | $1 \leqslant N \leqslant 1000$ | $1 \leqslant K \leqslant N$ | $1 \leqslant L \leqslant 10^9$ |
| *5* | *20* | $1 \leqslant N \leqslant 10^6$ | $1 \leqslant K \leqslant 3000$ | $1 \leqslant L \leqslant 10^9$ |
| *6* | *30* | $1 \leqslant N \leqslant 10^7$ | $1 \leqslant K \leqslant N$ | $1 \leqslant L \leqslant 10^9$ |

## Przykładowy program sprawdzający

*Przykładowy program sprawdzający czyta dane z wejścia w następującym formacie:*

- *wiersz 1:* `N K L`
- *wiersz 2:* `positions[0] ... positions[N-1]`

*Program wypisuje wartość zwróconą przez funkcję* `delivery`*.*

**Krzysztof Diks, Lech Duraj**
Tłumaczenie

**Dostępna pamięć: 1500 MB, limit czasu: 1 s.** *IOI 2015, dzień pierwszy, 28.07.2015*

# Waga szalkowa

*Amina ma sześć monet, ponumerowanych od 1 do 6. Wiadomo, że wszystkie monety mają różne masy (ciężary). Amina chce uporządkować monety rosnąco względem ich mas. W tym celu zbudowała wagę szalkową nowego typu.*

*Tradycyjna waga szalkowa ma dwie szalki. Korzystanie z takiej wagi polega na umieszczeniu po jednej monecie na każdej z szalek i wskazaniu cięższej z nich.*

*Nowa waga Aminy jest bardziej skomplikowana. Waga ma cztery szalki, oznaczone A, B, C, D, i pracuje w czterech trybach, w każdym odpowiadając na inne pytanie dotyczące monet położonych na szalkach. Użycie wagi polega na umieszczeniu po jednej monecie na każdej z szalek A, B i C. Dodatkowo, w trybie czwartym, należy umieścić także monetę na szalce D.*

*W każdym z trybów 1–4 dostajemy odpowiedź na pytanie dotyczące monet umieszczonych na szalkach:*

1. *Która z monet na szalkach A, B i C jest najcięższa?*
2. *Która z monet na szalkach A, B i C jest najlżejsza?*
3. *Która z monet na szalkach A, B i C jest medianą? (Medianą jest ta moneta wśród trzech, która nie jest ani najcięższa, ani najlżejsza).*
4. *Wśród monet na szalkach A, B i C rozważamy tylko te, które są cięższe od tej z szalki D. Jeśli są takie monety, to która z nich jest najlżejsza? W przeciwnym przypadku (gdy nie ma takich monet), która z monet na szalkach A, B i C jest najlżejsza?*

## Zadanie

*Napisz program, który uporządkuje sześć monet Aminy rosnąco względem ich mas. Program może korzystać z wagi Aminy do porównywania mas monet. Twój program będzie musiał rozwiązać wiele przypadków testowych, każdy odpowiadający nowemu zestawowi sześciu monet.*

*W Twoim programie powinny zostać zaimplementowane funkcje* `init` *oraz* `orderCoins`*. W każdym wykonaniu Twojego programu, program sprawdzający wywoła najpierw, dokładnie raz, funkcję* `init`*. Jako jej parametr podana zostanie liczba przypadków testowych, a dodatkowo będziesz mógł zainicjować dowolne zmienne. Następnie program sprawdzający będzie wywoływał funkcję* `orderCoins` *raz dla każdego przypadku testowego.*

- `init(T)`
  - `T`*: Liczba przypadków testowych, które program będzie musiał rozwiązać podczas tego przebiegu.* `T` *jest liczbą całkowitą z zakresu* $1, \ldots, 18$*.*
  - *Funkcja nie zwraca żadnej wartości.*
- `orderCoins()` *– funkcja wywoływana dokładnie raz dla każdego przypadku testowego.*
  - *Ta funkcja powinna wyznaczyć poprawne uporządkowanie monet Aminy, wywołując funkcje programu sprawdzającego* `getLightest()`*,* `getHeaviest()`*,* `getMedian()` *i/lub* `getNextLightest()`*.*

- *Gdy funkcja wyznaczy właściwe uporządkowanie, powinna je podać, wywołując funkcję programu sprawdzającego* `answer()`.
- *Po wywołaniu* `answer()`, *funkcja* `orderCoins()` *powinna zakończyć swoje działanie. Funkcja nie zwraca żadnej wartości.*

*W swoim programie możesz korzystać z następujących funkcji programu sprawdzającego:*

- `answer(W)` – *Twój program powinien użyć tej funkcji do podania obliczonej odpowiedzi.*
  - `W`*: Tablica o długości 6, zawierająca poprawne uporządkowanie monet. Elementy tablicy od* `W[0]` *do* `W[5]` *powinny zawierać numery monet (czyli numery od 1 do 6) w kolejności odpowiadającej monetom od najlżejszej do najcięższej.*
  - *Twój program powinien wywołać tę funkcję z* `orderCoins()`*, raz dla każdego przypadku testowego.*
  - *Ta funkcja nie zwraca żadnej wartości.*
- `getHeaviest(A, B, C)`, `getLightest(A, B, C)`, `getMedian(A, B, C)` – *te funkcje odpowiadają użyciom wagi Aminy odpowiednio w trybach 1, 2 i 3.*
  - `A`, `B`, `C`*: Monety, które umieszcza się odpowiednio na szalkach A, B i C.* `A`, `B` *i* `C` *powinny być różnymi liczbami całkowitymi z zakresu od 1 do 6 włącznie.*
  - *Każda z tych funkcji zwraca jedną z liczb* `A`, `B` *i* `C`*: numer właściwej monety. Dla przykładu,* `getHeaviest(A, B, C)` *zwraca numer najcięższej z tych trzech monet.*
- `getNextLightest(A, B, C, D)` – *ta funkcja odpowiada trybowi 4 wagi Aminy.*
  - `A`, `B`, `C`, `D`*: Monety, które umieszcza się odpowiednio na szalkach A, B, C i D.* `A`, `B`, `C` *i* `D` *powinny być różnymi liczbami całkowitymi z zakresu od 1 do 6 włącznie.*
  - *Funkcja zwraca jedną z liczb* `A`, `B` *i* `C`*: numer monety wskazanej przez wagę pracującą w trybie 4. Innymi słowy, zwracana moneta jest najlżejszą spośród tych monet na szalkach A, B i C, które są cięższe niż moneta na szalce D; lub, jeśli wszystkie te monety są lżejsze od monety na szalce D, zwracana jest najlżejsza z monet na szalkach A, B i C.*

## Podzadania

*To zadanie nie posiada żadnych podzadań. Zamiast tego Twoja punktacja zależy od liczby ważeń, które wykona program (łączna liczba wywołań funkcji* `getLightest()`, `getHeaviest()`, `getMedian()` *i/lub* `getNextLightest()`*).*

*Twój program będzie wykonywany wiele razy, za każdym razem dla wielu przypadków testowych. Niech r będzie liczbą wykonań Twojego programu (liczbą testów w zadaniu). Jeśli Twój program nie znajdzie poprawnej kolejności monet dla któregokolwiek przypadku testowego w jakimkolwiek wykonaniu programu, otrzymasz 0 punktów. W przeciwnym przypadku poszczególne wykonania programu są punktowane następująco.*

*Niech Q będzie najmniejszą liczbą ważeń, która umożliwia posortowanie dowolnego ciągu sześciu monet na wadze Aminy. Żeby uczynić zadanie bardziej interesującym, nie podajemy tutaj wartości Q.*

*Załóżmy, że największa liczba ważeń wykonanych przez Twój program dla wszystkich podanych przypadków testowych wynosi $Q + y$, dla pewnej liczby całkowitej $y$. Rozważmy teraz pojedyncze wykonanie programu. Niech największa liczba ważeń dla wszystkich $T$ przypadków testowych w tym wykonaniu wynosi $Q + x$, dla pewnej nieujemnej liczby całkowitej $x$. (Jeśli użyjesz mniej niż $Q$ ważeń dla każdego przypadku testowego, wówczas $x = 0$). Wówczas liczba punktów zdobyta za to wykonanie wyniesie $\frac{100}{r((x+y)/5+1)}$, zaokrąglone* **w dół** *do dwóch cyfr po przecinku.*

*W szczególności, jeśli Twój program wykona co najwyżej $Q$ ważeń dla każdego przypadku testowego w każdym wykonaniu programu, otrzymasz 100 punktów.*

## Przykład

*Załóżmy, że kolejność monet od najlżejszej do najcięższej jest następująca: 3 4 6 2 1 5.*

| **wywołania funkcji** | **wyniki** | **objaśnienie** |
|---|---|---|
| `getMedian(4, 5, 6)` | *6* | *Moneta 6 jest medianą wśród monet 4, 5 i 6.* |
| `getHeaviest(3, 1, 2)` | *1* | *Moneta 1 jest najcięższą wśród monet 1, 2 i 3.* |
| `getNextLightest(2, 3, 4, 5)` | *3* | *Monety 2, 3 i 4 są wszystkie lżejsze od monety 5, zatem zostanie zwrócona najlżejsza spośród nich (3).* |
| `getNextLightest(1, 6, 3, 4)` | *6* | *Monety 1 i 6 są obie cięższe niż moneta 4. Spośród monet 1 i 6, moneta 6 jest najlżejsza.* |
| `getHeaviest(3, 5, 6)` | *5* | *Moneta 5 jest najcięższą wśród monet 3, 5 i 6.* |
| `getMedian(1, 5, 6)` | *1* | *Moneta 1 jest medianą wśród monet 1, 5 i 6.* |
| `getMedian(2, 4, 6)` | *6* | *Moneta 6 jest medianą wśród monet 2, 4 i 6.* |
| `answer([3, 4, 6, 2, 1, 5])` | | *Program znalazł właściwą odpowiedź dla tego przypadku testowego.* |

## Przykładowy program sprawdzający

*Przykładowy program sprawdzający wczytuje dane w następującym formacie:*

- *wiersz 1:* `T` *– liczba przypadków testowych*
- *każdy z wierszy od 2 do* `T` *+ 1: ciąg 6 różnych liczb od 1 do 6: kolejność monet od najlżejszej do najcięższej.*

*Dla przykładu, dla dwóch przypadków testowych, w których monety są dane odpowiednio w kolejności 1 2 3 4 5 6 i 3 4 6 2 1 5, dane mają postać:*

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

*Przykładowy program sprawdzający wypisuje zawartość tablicy, która została przekazana jako parametr do funkcji* `answer()`.

**Krzysztof Diks, Lech Duraj**
Tłumaczenie

**Dostępna pamięć: 1500 MB, limit czasu: 4 s.** *IOI 2015, dzień pierwszy, 28.07.2015*

# Zespoły

*W klasie jest $N$ uczniów, ponumerowanych od 0 do $N-1$. Każdego dnia nauczyciel zadaje uczniom pewną liczbę projektów. Każdy z projektów musi zostać zrealizowany przez pewien zespół uczniów, tego samego dnia. Projekty mogą różnić się między sobą trudnością – o każdym projekcie wiadomo, jaka jest dokładna liczba uczniów w zespole, który powinien nad nim pracować.*

*Każdy uczeń ma własne zdanie co do wielkości zespołów, w jakich może się znaleźć. Ściślej mówiąc, uczeń numer $i$ może należeć tylko do zespołu, który ma co najmniej $A[i]$ i co najwyżej $B[i]$ uczniów. Każdego dnia ucznia można włączyć do co najwyżej jednego zespołu, przy czym niektórzy uczniowie mogą pozostać bez przydziału.*

*Nauczyciel wybrał już projekty na najbliższe $Q$ dni. Dla każdego z tych dni określ, czy możliwe jest podzielenie uczniów na zespoły tak, aby nad każdym projektem pracował jeden zespół.*

## Przykład

*Załóżmy, że jest $N = 4$ uczniów i $Q = 2$ dni pracy. Życzenia uczniów podane są w poniższej tabeli:*

| **uczeń** | *0* | *1* | *2* | *3* |
|---|---|---|---|---|
| *A* | *1* | *2* | *2* | *2* |
| *B* | *2* | *3* | *3* | *4* |

*Pierwszego dnia są $M = 2$ projekty, wymagające zespołów o wielkości odpowiednio $K[0] = 1$ oraz $K[1] = 3$. Takie dwa zespoły można zbudować, przydzielając ucznia 0 do jednoosobowego zespołu, a pozostałych trzech uczniów łącząc w kolejny zespół.*

*Drugiego dnia znowu są $M = 2$ projekty, lecz tym razem potrzebne są zespoły o rozmiarach $K[0] = 1$ oraz $K[1] = 1$. Takiego przydziału zrealizować nie można, ponieważ tylko jeden uczeń chce pracować w zespole o wielkości 1.*

## Zadanie

*Dane są opisy wszystkich uczniów: liczba $N$ oraz tablice $A$ i $B$, a także ciąg $Q$ zapytań, z których każde dotyczy jednego dnia. Zapytanie zawiera liczbę $M$ projektów do realizacji oraz ciąg $K$ o długości $M$ zawierający wielkości zespołów do sformowania. Dla każdego zapytania, Twój program musi odpowiedzieć, czy da się odpowiednio przydzielić uczniów do zespołów.*

*Musisz zaimplementować funkcje* `init` *oraz* `can`*:*

- `init(N, A, B)` – *Program sprawdzający wywoła tę funkcję dokładnie raz.*
  - `N`*: liczba uczniów.*
  - `A`*: tablica długości* `N`*:* `A[i]` *jest minimalną możliwą wielkością zespołu dla ucznia $i$.*

  - `B`*: tablica długości* `N`*:* `B[i]` *jest maksymalną możliwą wielkością zespołu dla ucznia* $i$*.*
  - *Funkcja ta nie powinna zwracać żadnej wartości.*
  - *Możesz założyć, że* $1 \leqslant A[i] \leqslant B[i] \leqslant N$ *dla* $i = 0, \ldots, N-1$*.*

- `can(M, K)` *– Po jednokrotnym wywołaniu* `init`*, program sprawdzający wywoła tę funkcję* $Q$ *razy, raz dla każdego dnia.*
  - `M`*: liczba projektów zadanych tego dnia.*
  - `K`*: tablica długości* `M` *zawierająca wymagane rozmiary zespołów dla kolejnych projektów.*
  - *Funkcja powinna zwracać 1, jeśli możliwe jest sformowanie odpowiednich zespołów, lub 0 w przeciwnym wypadku.*
  - *Możesz założyć, że* $1 \leqslant$ `M` $\leqslant N$ *oraz że dla każdego* $i = 0, \ldots,$ `M` $- 1$ *zachodzi* $1 \leqslant$ `K[i]` $\leqslant N$*. Zwróć uwagę, że suma wszystkich* `K[i]` *może przekroczyć* $N$*.*

## Podzadania

*Niech* $S$ *oznacza sumę wartości* `M` *we wszystkich wywołaniach* `can(M, K)`*.*

| **podzadanie** | **liczba punktów** | $N$ | $Q$ | **dodatkowe warunki** |
|---|---|---|---|---|
| *1* | *21* | $1 \leqslant N \leqslant 100$ | $1 \leqslant Q \leqslant 100$ | *brak* |
| *2* | *13* | $1 \leqslant N \leqslant 100\,000$ | $Q = 1$ | *brak* |
| *3* | *43* | $1 \leqslant N \leqslant 100\,000$ | $1 \leqslant Q \leqslant 100\,000$ | $S \leqslant 100\,000$ |
| *4* | *23* | $1 \leqslant N \leqslant 500\,000$ | $1 \leqslant Q \leqslant 200\,000$ | $S \leqslant 200\,000$ |

## Przykładowy program sprawdzający

*Przykładowy program sprawdzający czyta dane z wejścia w następującej postaci:*

- *wiersz 1:* `N`
- *wiersze 2, . . . ,* `N + 1:` `A[i] B[i]`
- *wiersz* `N + 2:` `Q`
- *wiersze* `N + 3, . . . , N + Q + 2:` `M K[0] K[1] ... K[M - 1]`

*Dla każdego zapytania program wypisuje wartość zwróconą przez funkcję* `can`*.*

**Krzysztof Diks, Lech Duraj**
Tłumaczenie

**Dostępna pamięć: 1500 MB, limit czasu: 1,5 s.** *IOI 2015, dzień drugi, 30.07.2015*

# Konie

*Mansur, wzorem swoich starożytnych przodków, jest zapalonym hodowcą koni. Posiada obecnie największe stado w całym Kazachstanie. Nie zawsze jednak tak było. Ongiś, $N$ lat temu, Mansur był jedynie* **dżygitem** *(kaz.* **młodym człowiekiem***) mającym tylko jednego konia. Marzył wtedy, aby zarobić dużo pieniędzy i stać się* **bajem** *(kaz.* **bogaczem***).*

*Ponumerujmy lata działalności Mansura liczbami $0, 1, \ldots, N-1$, od najdawniejszych do najnowszych. Podczas każdego roku stado mogło zwiększyć swoją liczebność, w zależności od pogody. Dla każdego roku $i$, Mansur pamięta współczynnik wzrostu – całkowitą dodatnią liczbę $X[i]$. Jeśli na początku roku $i$ stado liczyło $h$ koni, na koniec tego roku było już $h \cdot X[i]$ koni w stadzie.*

*Konie można było sprzedawać wyłącznie pod koniec roku. Dla każdego roku $i$, Mansur pamięta również dodatnią liczbę całkowitą $Y[i]$ – cenę, jaką można było otrzymać za sprzedaż jednego konia. Po każdym roku można było sprzedać dowolnie wiele spośród posiadanych koni, wszystkie po tej samej cenie $Y[i]$.*

*Mansur zastanawia się, ile najwięcej pieniędzy mógł był zarobić, gdyby wybrał najlepsze momenty do sprzedaży koni w ciągu tych $N$ lat. Masz obecnie zaszczyt gościć u Mansura w czasie* **toi** *(kaz.* **wakacji***), Mansur zadał więc to pytanie Tobie.*

*Dodatkowo, w miarę snucia opowieści, Mansur przypomina sobie nowe fakty. Dokona on zatem $M$ poprawek swojej historii. Każda z poprawek zmieni dokładnie jedną wartość $X[i]$ albo dokładnie jedną wartość $Y[i]$. Po każdej poprawce Mansur znowu zapyta Cię o największą sumę pieniędzy, jaką mógł zarobić. Poprawki Mansura kumulują się – każda z Twoich odpowiedzi musi brać pod uwagę wszystkie poprzednie poprawki. Zwróć też uwagę na fakt, że każde $X[i]$ lub $Y[i]$ może być zmieniane wielokrotnie.*

*Odpowiedzi na pytania Mansura mogą być wielkimi sumami pieniędzy. Aby uniknąć dużych liczb, podaj wszystkie odpowiedzi modulo $10^9 + 7$.*

## Przykład

*Załóżmy, że historia trwa $N = 3$ lata z następującymi parametrami:*

| **rok** | *0* | *1* | *2* |
|---|---|---|---|
| $X$ | *2* | *1* | *3* |
| $Y$ | *3* | *4* | *1* |

*Dla tych wartości Mansur zarobi najwięcej, jeśli sprzeda oba swoje konie pod koniec roku 1. Cała historia wygląda wtedy następująco:*

- *Na początku Mansur posiada 1 konia.*
- *Po roku 0 będzie miał $1 \cdot X[0] = 2$ konie.*
- *Po roku 1 będzie miał $2 \cdot X[1] = 2$ konie.*
- *Teraz może sprzedać oba konie za łączną sumę $2 \cdot Y[1] = 8$.*

*Teraz załóżmy, że jest $M = 1$ poprawka: zmiana $Y[1]$ na 2. Po poprawce dane są następujące:*

| **rok** | *0* | *1* | *2* |
|---|---|---|---|
| $X$ | *2* | *1* | *3* |
| $Y$ | *3* | *2* | *1* |

*W tym wypadku jednym z możliwych najlepszych rozwiązań jest sprzedaż jednego konia po roku 0, a trzech po roku 2. Cała historia wygląda wtedy następująco:*

- *Na początku Mansur posiada 1 konia.*
- *Po roku 0 będzie miał $1 \cdot X[0] = 2$ konie.*
- *Sprzedaje jednego z koni po cenie $Y[0] = 3$, pozostaje mu więc jeden koń.*
- *Po roku 1 będzie miał $1 \cdot X[1] = 1$ konia.*
- *Po roku 2 będzie miał $1 \cdot X[2] = 3$ konie.*
- *Teraz sprzedaje wszystkie konie za sumę $3 \cdot Y[2] = 3$. Zarobił łącznie $3 + 3 = 6$.*

## Zadanie

*Dane są liczba $N$, tablice $X$, $Y$ oraz lista poprawek. Oblicz maksymalny zysk ze sprzedaży koni Mansura przed wszystkimi poprawkami, a także po każdej poprawce. Wyniki podaj modulo $10^9 + 7$.*

*Musisz zaimplementować funkcje* `init`, `updateX` *i* `updateY`.

- `init(N, X, Y)` – *Program sprawdzający wywoła tę funkcję dokładnie raz.*
  - `N`*: liczba lat.*
  - `X`*: tablica długości* `N`*. Dla każdego $0 \leqslant i \leqslant$* `N` *$- 1$,* `X[i]` *to współczynnik wzrostu w roku $i$.*
  - `Y`*: tablica długości* `N`*. Dla każdego $0 \leqslant i \leqslant$* `N` *$- 1$,* `Y[i]` *to cena konia po roku $i$.*
  - *Wartości* `X` *oraz* `Y` *to liczby podane przez Mansura na początku (przed wszystkimi poprawkami).*
  - *Po zakończeniu działania funkcji* `init` *tablice* `X` *i* `Y` *pozostają zaalokowane i możesz dowolnie modyfikować ich zawartość.*
  - *Funkcja powinna zwrócić maksymalny zysk Mansura dla podanych początkowych wartości* `X` *oraz* `Y`*, modulo $10^9 + 7$.*
- `updateX(pos, val)`
  - `pos`*: liczba całkowita z zakresu $0, \ldots, N - 1$.*
  - `val`*: nowa wartość $X[$*`pos`*$]$.*
  - *Funkcja powinna zwracać maksymalny zysk Mansura po uwzględnieniu tej poprawki, modulo $10^9 + 7$.*

- `updateY(pos, val)`
  - `pos`*: liczba całkowita z zakresu* $0, \ldots, N-1$.
  - `val`*: nowa wartość* $Y[$`pos`$]$.
  - *Funkcja powinna zwracać maksymalny zysk Mansura po uwzględnieniu tej poprawki, modulo* $10^9 + 7$.

*Możesz założyć, że zarówno początkowe, jak i zaktualizowane później wartości* $X[i]$ *oraz* $Y[i]$ *są liczbami z zakresu od 1 do* $10^9$ *włącznie. Po jednokrotnym wywołaniu* `init`*, program sprawdzający wywoła funkcje* `updateX` *oraz* `updateY` *pewną liczbę razy. Łączna liczba wywołań* `updateX` *i* `updateY` *będzie wynosiła* $M$.

## Podzadania

| **podzadanie** | **liczba punktów** | $N$ | $M$ | **dodatkowe ograniczenia** |
|---|---|---|---|---|
| *1* | *17* | $1 \leqslant N \leqslant 10$ | $M = 0$ | $X[i], Y[i] \leqslant 10$, $X[0] \cdot X[1] \cdot \ldots \cdot X[N-1] \leqslant 1000$ |
| *2* | *17* | $1 \leqslant N \leqslant 1000$ | $0 \leqslant M \leqslant 1000$ | *brak* |
| *3* | *20* | $1 \leqslant N \leqslant 500\,000$ | $0 \leqslant M \leqslant 100\,000$ | $X[i] \geqslant 2$ *oraz* `val` $\geqslant 2$ *odpowiednio dla* `init` *i* `updateX` |
| *4* | *23* | $1 \leqslant N \leqslant 500\,000$ | $0 \leqslant M \leqslant 10\,000$ | *brak* |
| *5* | *23* | $1 \leqslant N \leqslant 500\,000$ | $0 \leqslant M \leqslant 100\,000$ | *brak* |

## Przykładowy program sprawdzający

*Przykładowy program sprawdzający czyta dane z pliku* `horses.in` *w następującej postaci:*

- *wiersz 1:* `N`
- *wiersz 2:* `X[0] ... X[N - 1]`
- *wiersz 3:* `Y[0] ... Y[N - 1]`
- *wiersz 4:* `M`
- *wiersze 5, ...,* `M + 4`*: trzy liczby* `type pos val` *(*`type` $= 1$ *dla* `updateX` *oraz* `type` $= 2$ *dla* `updateY`*).*

*Przykładowy program sprawdzający wypisuje wartość zwróconą przez funkcję* `init`*, a po niej wartości zwrócone przez wywołania funkcji* `updateX` *i* `updateY`.

**Krzysztof Diks, Lech Duraj**
Tłumaczenie

**Dostępna pamięć: 1500 MB, limit czasu: 1 s.** *IOI 2015, dzień drugi, 30.07.2015*

# Sortowanie

*Aizhan pragnie posortować ciąg $N$ liczb całkowitych $S[0], S[1], \ldots, S[N-1]$. Ciąg ten składa się z parami różnych liczb z zakresu od 0 do $N-1$. Aizhan chce posortować zadany ciąg rosnąco, zamieniając elementy ciągu parami. Jej przyjaciel Ermek ma także zamiar zamieniać parami niektóre elementy ciągu, ale niekoniecznie w sposób, który pomagałby Aizhan.*

*Ermek i Aizhan modyfikują ciąg w rundach. W każdej rundzie pierwszą zamianę wykonuje Ermek, a po nim zamiany dokonuje Aizhan. Osoba dokonująca zamiany wybiera dwa indeksy, po czym zamienia miejscami elementy o tych indeksach. Indeksy wybrane do zamiany niekoniecznie muszą być różne – jeśli oba są takie same, oznacza to, że w rzeczywistości element pozostaje na swoim miejscu.*

*Aizhan wie, że Ermekowi nie zależy na posortowaniu ciągu. Ponadto wie ona, jakich zamian Ermek ma zamiar dokonać. Ermek planuje wziąć udział w co najwyżej $M$ rundach zamian elementów ciągu. Rundy są ponumerowane od 0 do $M-1$. W $i$-tej rundzie, dla $i = 0, 1, \ldots, M-1$, Ermek wybiera do zamiany elementy o indeksach $X[i]$ oraz $Y[i]$.*

*Aizhan chce posortować ciąg $S$. Przed każdą rundą, jeśli Aizhan widzi, że ciąg jest już posortowany rosnąco, to przerywa cały proces sortowania. Mając dany ciąg $S$ oraz listę zamian, które chce wykonać Ermek, znajdź ciąg zamian dla Aizhan, który pozwoli jej posortować ciąg $S$. Dodatkowo, w niektórych podzadaniach musisz wyznaczyć najkrótszy taki ciąg. Możesz założyć, że zawsze możliwe jest posortowanie ciągu $S$ w $M$ lub mniej rundach.*

*Zauważ, że jeśli Aizhan widzi, że ciąg $S$ jest posortowany po zamianie Ermeka, może do zamiany wskazać elementy o tym samym indeksie (np. 0 i 0). W wyniku takiej zamiany ciąg $S$ pozostaje posortowany po całej rundzie, tak więc Aizhan osiąga swój cel. Zauważ także, że jeżeli początkowy ciąg $S$ jest posortowany, to minimalna liczba rund potrzebnych do posortowania go wynosi 0.*

## Przykład 1

*Załóżmy, że:*

- *Początkowy ciąg to $S = 4, 3, 2, 1, 0$.*
- *Ermek chce wykonać $M = 6$ zamian.*
- *Ciągi $X$ i $Y$ opisujące pary indeksów elementów zamienianych przez Ermeka to $X = 0, 1, 2, 3, 0, 1$ oraz $Y = 1, 2, 3, 4, 1, 2$. Innymi słowy, pary indeksów elementów, które Ermek zamierza zamienić, to kolejno: $(0, 1)$, $(1, 2)$, $(2, 3)$, $(3, 4)$, $(0, 1)$ i $(1, 2)$.*

*W tej sytuacji Aizhan może posortować $S$, otrzymując w trzech rundach kolejność $0, 1, 2, 3, 4$. Osiągnie to, wybierając pary indeksów $(0, 4)$, $(1, 3)$, a na końcu $(3, 4)$. Poniższa tabela pokazuje, jak będzie wyglądał ciąg po zamianach Ermeka i Aizhan:*

| runda | gracz | pary indeksów | ciąg |
|---|---|---|---|
| *początek* | | | $4, 3, 2, 1, 0$ |
| $0$ | *Ermek* | $(0, 1)$ | $3, 4, 2, 1, 0$ |
| $0$ | *Aizhan* | $(0, 4)$ | $0, 4, 2, 1, 3$ |
| $1$ | *Ermek* | $(1, 2)$ | $0, 2, 4, 1, 3$ |
| $1$ | *Aizhan* | $(1, 3)$ | $0, 1, 4, 2, 3$ |
| $2$ | *Ermek* | $(2, 3)$ | $0, 1, 2, 4, 3$ |
| $2$ | *Aizhan* | $(3, 4)$ | $0, 1, 2, 3, 4$ |

## Przykład 2

*Załóżmy, że:*

- *Początkowy ciąg to* $S = 3, 0, 4, 2, 1$.
- *Ermek chce wykonać* $M = 5$ *zamian.*
- *Pary indeksów elementów, które Ermek zamierza zamienić ze sobą, to kolejno:* $(1, 1)$, $(4, 0)$, $(2, 3)$, $(1, 4)$ *i* $(0, 4)$.

*W tej sytuacji Aizhan może posortować* $S$ *w trzech rundach, wybierając na przykład pary indeksów* $(1, 4)$, $(4, 2)$ *oraz* $(2, 2)$. *Poniższa tabela pokazuje, jak będzie wyglądał ciąg po zamianach Ermeka i Aizhan:*

| runda | gracz | pary indeksów | ciąg |
|---|---|---|---|
| *początek* | | | $3, 0, 4, 2, 1$ |
| $0$ | *Ermek* | $(1, 1)$ | $3, 0, 4, 2, 1$ |
| $0$ | *Aizhan* | $(1, 4)$ | $3, 1, 4, 2, 0$ |
| $1$ | *Ermek* | $(4, 0)$ | $0, 1, 4, 2, 3$ |
| $1$ | *Aizhan* | $(4, 2)$ | $0, 1, 3, 2, 4$ |
| $2$ | *Ermek* | $(2, 3)$ | $0, 1, 2, 3, 4$ |
| $2$ | *Aizhan* | $(2, 2)$ | $0, 1, 2, 3, 4$ |

## Zadanie

*Dane są: ciąg* $S$, *liczba* $M$ *oraz listy indeksów* $X$ *i* $Y$. *Znajdź kolejność zamian, których Aizhan może użyć do posortowania ciągu* $S$. *W podzadaniach 5 i 6 Twój ciąg zamian dodatkowo musi być najkrótszy z możliwych.*

*Musisz zaimplementować funkcję* `findSwapPairs`*:*

- `findSwapPairs(N, S, M, X, Y, P, Q)` – *Funkcja ta zostanie wywołana przez program sprawdzający dokładnie raz.*
  - `N`*: długość ciągu* $S$.
  - `S`*: tablica zawierająca początkowy ciąg* $S$.
  - `M`*: liczba zamian, które chce wykonać Ermek.*
  - `X`, `Y`*: tablice liczb całkowitych długości* `M`. *Dla* $0 \leqslant i \leqslant$ `M` $- 1$, *Ermek w rundzie* $i$ *zamieni elementy ciągu* $S$ *o indeksach* `X[i]` *oraz* `Y[i]`.

- `P`, `Q`: *tablice liczb całkowitych. Użyj tych tablic, aby podać jeden z możliwych ciągów zamian, których może użyć Aizhan do posortowania ciągu $S$. Niech $R$ będzie długością ciągu zamian znalezionego przez Twój program. Dla każdego $i$ między $0$ a $R-1$ włącznie, indeksy zamienione przez Aizhan powinny zostać zapisane w* `P[i]` *oraz* `Q[i]`.

  *Możesz założyć, że tablice* `P` *i* `Q` *zostały już zaalokowane i każda z nich ma długość* `M`.
- *Funkcja ta powinna zwrócić wartość $R$ (zdefiniowaną powyżej).*

## Podzadania

| **podzadanie** | **liczba punktów** | **max** $N$ | $M$ | **dodatkowe ograniczenia na** $X$, $Y$ | **wymagane** $R$ |
|---|---|---|---|---|---|
| *1* | *8* | *5* | $M = N^2$ | $X[i] = Y[i] = 0$ *dla wszystkich* $i$ | $R \leqslant M$ |
| *2* | *12* | *100* | $M = 30N$ | $X[i] = Y[i] = 0$ *dla wszystkich* $i$ | $R \leqslant M$ |
| *3* | *16* | *100* | $M = 30N$ | $X[i] = 0, Y[i] = 1$ *dla wszystkich* $i$ | $R \leqslant M$ |
| *4* | *18* | *500* | $M = 30N$ | *brak* | $R \leqslant M$ |
| *5* | *20* | *2000* | $M = 3N$ | *brak* | *najmniejsze możliwe* |
| *6* | *26* | *200 000* | $M = 3N$ | *brak* | *najmniejsze możliwe* |

*Możesz założyć, że istnieje rozwiązanie, które wymaga $M$ lub mniej rund. We wszystkich podzadaniach $N \geqslant 1$.*

## Przykładowy program sprawdzający

*Przykładowy program sprawdzający czyta dane z pliku* `sorting.in` *w następującej postaci:*

- *wiersz 1:* `N`
- *wiersz 2:* `S[0] ... S[N - 1]`
- *wiersz 3:* `M`
- *wiersze 4, ..., `M + 3`:* `X[i] Y[i]`

*Program wypisuje na wyjście kolejno:*

- *wiersz 1: wartość zwróconą przez funkcję* `findSwapPairs`
- *wiersz $2 + i$ dla $0 \leqslant i < R$:* `P[i] Q[i]`

**Krzysztof Diks, Lech Duraj**
Tłumaczenie

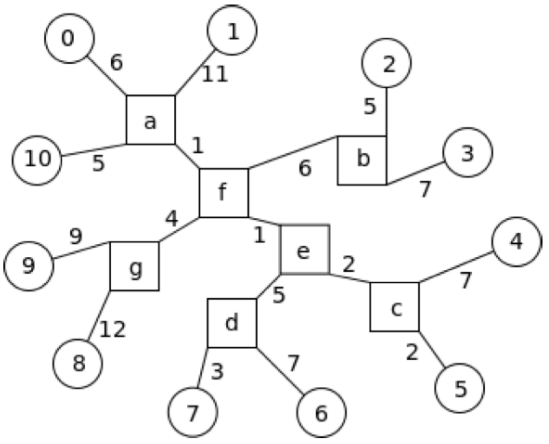
**Dostępna pamięć: 1500 MB, limit czasu: 1 s.** *IOI 2015, dzień drugi, 30.07.2015*

# Osady

*W Kazachstanie jest $N$ osad ponumerowanych od 0 do $N-1$. W tym kraju jest też pewna, nieznana liczba miast. Osady i miasta nazywamy wspólnie* **miejscowościami**.

*Wszystkie miejscowości w Kazachstanie są połączone jedną siecią dwukierunkowych dróg. Każda droga łączy dwie różne miejscowości. Każda para miejscowości jest bezpośrednio połączona co najwyżej jedną drogą. Wiadomo, że każda osada jest połączona bezpośrednio z dokładnie jedną inną miejscowością, a każde miasto jest połączone bezpośrednio z co najmniej trzema miejscowościami. O sieci dróg wiemy ponadto, że dla każdej pary miejscowości $a$ i $b$ istnieje dokładnie jeden sposób przejazdu pomiędzy nimi, jeśli tylko każda z dróg jest użyta co najwyżej raz.*

*Rysunek poniżej przedstawia sieć złożoną z 11 osad i 7 miast. Osady są oznaczone kółkami, a ich etykiety są liczbami, natomiast miasta są oznaczone kwadratami, a ich etykiety są literami.*




*Każda droga ma dodatnią, całkowitoliczbową długość. Odległość pomiędzy dwiema miejscowościami jest równa sumie długości dróg na trasie pomiędzy nimi.*

*Dla każdego miasta $C$, niech $r(C)$ będzie odległością do najbardziej odległej od $C$ osady. Miasto $C$ nazywamy* **centrum**, *jeśli odległość $r(C)$ jest najmniejsza spośród wszystkich miast. Odległość między centrum i najbardziej odległą osadą oznaczamy przez $R$. Tak więc $R$ jest najmniejszą wartością spośród wszystkich $r(C)$.*

*W powyższym przykładzie najbardziej odległą osadą od miasta $a$ jest osada 8, a odległość między nimi wynosi $r(a) = 1 + 4 + 12 = 17$. Dla miasta $g$ także mamy $r(g) = 17$. (Jedną z najbardziej odległych od $g$ osad jest osada 6). Jedynym centrum w tym przykładzie jest miasto $f$, dla którego $r(f) = 16$. Zatem w tym przykładzie $R$ wynosi 16.*

*Usunięcie centrum powoduje podział sieci na kilka spójnych części. Centrum nazwiemy* **zrównoważonym**, *jeśli każda z tych części zawiera co najwyżej $\lfloor N/2 \rfloor$ osad (zaznaczmy przy tym, że liczymy tylko osady, nie miasta). Przypomnijmy, że $\lfloor x \rfloor$ oznacza największą liczbę całkowitą nie większą niż $x$.*

*W naszym przykładzie centrum jest miasto $f$. Po usunięciu miasta $f$, sieć zostaje rozbita na cztery spójne części. Części te zawierają następujące zbiory osad: $\{0, 1, 10\}$, $\{2, 3\}$, $\{4, 5, 6, 7\}$ i $\{8, 9\}$. Żadna z części nie zawiera więcej niż $\lfloor 11/2 \rfloor = 5$ osad, zatem miasto $f$ jest zrównoważonym centrum.*

## Zadanie

*Jedyną początkową informacją o sieci miejscowości i dróg jest liczba $N$ osad. Nie znamy liczby miast. Nie wiemy także nic o strukturze połączeń drogowych. Nowe informacje możemy zdobywać jedynie, zadając pytania o odległości pomiędzy parami osad.*

*Twoim zadaniem jest:*

- *We wszystkich podzadaniach: obliczyć odległość $R$.*
- *W podzadaniach od 3 od 6: stwierdzić, czy w sieci znajduje się zrównoważone centrum.*

*Musisz zaimplementować funkcję* `hubDistance`*. Program sprawdzający rozpatrzy w jednym przebiegu wiele przypadków testowych. Liczba przypadków testowych dla jednego przebiegu nie przekracza 40. Dla każdego przypadku testowego program sprawdzający wywoła dokładnie raz Twoją funkcję* `hubDistance`*. Upewnij się, że Twoja funkcja inicjuje wszystkie niezbędne zmienne za każdym razem, gdy jest wywoływana.*

- `hubDistance(N, sub)`
  - `N`*: liczba osad.*
  - `sub`*: numer podzadania (zobacz wyjaśnienie w podrozdziale Podzadania).*
  - *Jeśli* `sub` *jest równe 1 lub 2, funkcja może zwrócić $R$ lub $-R$.*
  - *Jeśli* `sub` *jest większe od 2, to jeśli istnieje zrównoważone centrum, funkcja powinna zwrócić $R$, a w przeciwnym przypadku powinna zwrócić $-R$.*

*Twoja funkcja* `hubDistance` *może otrzymywać informacje o sieci dróg, wywołując funkcję programu sprawdzającego* `getDistance(i, j)`*. Ta funkcja zwraca odległość pomiędzy osadami* `i` *oraz* `j`*. Jeśli* `i` *oraz* `j` *są takie same, funkcja zwraca 0. Funkcja zwraca 0, także gdy argumenty są nieprawidłowe.*

## Podzadania

*W każdym przypadku testowym:*

- *$N$ jest liczbą pomiędzy 6 i 110 włącznie.*
- *Odległość pomiędzy każdą parą osad wynosi od 1 do 1 000 000 włącznie.*

*Twój program może wykonać ograniczoną liczbę zapytań. Ograniczenie zależy od podzadania i jest podane w tabeli poniżej. Jeśli Twój program przekroczy dozwoloną liczbę zapytań, to jego wykonywanie zostanie przerwane i wówczas przyjmuje się, że program dał złą odpowiedź.*

| **podzadanie** | **liczba punktów** | **liczba zapytań** | **znajduje zrównoważone centrum?** | **dodatkowe warunki** |
|---|---|---|---|---|
| *1* | *13* | $\frac{n(n-1)}{2}$ | *NIE* | *brak* |
| *2* | *12* | $\lceil \frac{7n}{2} \rceil$ | *NIE* | *brak* |
| *3* | *13* | $\frac{n(n-1)}{2}$ | *TAK* | *brak* |
| *4* | *10* | $\lceil \frac{7n}{2} \rceil$ | *TAK* | *z każdego miasta wychodzą* **dokładnie** *trzy drogi* |
| *5* | *13* | $5n$ | *TAK* | *brak* |
| *6* | *39* | $\lceil \frac{7n}{2} \rceil$ | *TAK* | *brak* |

*Przypomnijmy, że $\lceil x \rceil$ oznacza najmniejszą liczbę całkowitą większą od lub równą $x$.*

## Przykładowy program sprawdzający

*Zauważ, że numer podzadania jest częścią danych wejściowych. Przykładowy program sprawdzający zachowuje się różnie w zależności od numeru podzadania.*

*Przykładowy program sprawdzający czyta dane z pliku* `towns.in` *podane w następującym formacie:*

- *wiersz 1: Numer podzadania i liczba przypadków testowych.*
- *wiersz 2: $N_1$, liczba osad w pierwszym przypadku testowym.*
- *następne $N_1$ wierszy: $j$-ta liczba ($1 \leqslant j \leqslant N_1$) w $i$-tym z tych wierszy ($1 \leqslant i \leqslant N_1$) jest odległością pomiędzy osadami $i-1$ oraz $j-1$.*
- *Dalej następuje opis kolejnych przypadków testowych, w takim samym formacie jak w pierwszym przypadku testowym.*

*Dla każdego przypadku testowego, przykładowy program sprawdzający wypisuje wartość zwracaną przez* `hubDistance` *oraz w oddzielnym wierszu liczbę zapytań.*

*Plik z danymi dla przykładu powyżej ma postać:*

```
1 1
11
0 17 18 20 17 12 20 16 23 20 11
17 0 23 25 22 17 25 21 28 25 16
18 23 0 12 21 16 24 20 27 24 17
20 25 12 0 23 18 26 22 29 26 19
17 22 21 23 0 9 21 17 26 23 16
12 17 16 18 9 0 16 12 21 18 11
20 25 24 26 21 16 0 10 29 26 19
16 21 20 22 17 12 10 0 25 22 15
23 28 27 29 26 21 29 25 0 21 22
20 25 24 26 23 18 26 22 21 0 19
11 16 17 19 16 11 19 15 22 19 0
```

*Ten format różni się od zwyczajowego specyfikowania listy dróg. Zauważ, że wolno Ci modyfikować przykładowy program sprawdzający, tak żeby używał innego formatu danych.*

# XXI Bałtycka Olimpiada Informatyczna,

*Warszawa-Józefów, 2015*

**Lech Duraj, Bartosz Szreder**
Tłumaczenie

**Dostępna pamięć: 256 MB.** *BOI 2015, dzień próbny, 29.04.2015*

# Ciąg

*Ciąg $n$ liczb całkowitych $a_1, a_2, \ldots, a_n$ można uporządkować na wiele różnych sposobów. Dla każdego możliwego porządku liczb w ciągu można określić jego* **współczynnik bałaganu**, *który jest sumą wartości bezwzględnych różnic między każdą parą kolejnych elementów. Tak więc dla ciągu $(a_1, a_2, \ldots, a_n)$ współczynnik bałaganu jest równy $|a_1 - a_2| + |a_2 - a_3| + \ldots + |a_{n-1} - a_n|$. Twoim zadaniem jest obliczenie, dla podanego ciągu składającego się z $n$ różnych liczb, jaki maksymalny współczynnik bałaganu można otrzymać przestawiając jego elementy, a także ile jest sposobów ustawienia elementów, które dadzą taki współczynnik.*

## Wejście

*Pierwszy wiersz wejścia zawiera pojedynczą liczbę całkowitą $n$ ($2 \leqslant n \leqslant 27$) – długość ciągu. W drugim wierszu podanych jest $n$ parami różnych liczb $a_1, a_2, \ldots, a_n$ ($1 \leqslant a_i \leqslant 100\,000$).*

## Wyjście

*Twój program powinien wypisać na wyjście dwie liczby, każdą w osobnym wierszu. Pierwsza liczba to największy możliwy do osiągnięcia współczynnik bałaganu, druga – liczba ustawień wyrazów ciągu, dla których osiągana jest ta wartość współczynnika bałaganu.*

## Przykład

*Dla danych wejściowych:*

```
3
1 7 4
```

*poprawnym wynikiem jest:*

```
9
4
```

**Wyjaśnienie do przykładu:** *W poniższej tabeli podane jest wszystkie 6 ustawień wyrazów ciągu 1 7 4. Cztery spośród nich dają maksymalną wartość współczynnika bałaganu, równą 9.*

| *Ustawienie* | *Współczynnik bałaganu* |
|---|---|
| *1 7 4* | *9* |
| *1 4 7* | *6* |
| *7 1 4* | *9* |
| *7 4 1* | *6* |
| *4 1 7* | *9* |
| *4 7 1* | *9* |

## Ocenianie

| Podzadanie | Ograniczenia | Punkty |
|---|---|---|
| *1* | $n \leqslant 10$ | *20* |
| *2* | *oceniana jest tylko pierwsza z wypisanych na wyjście liczb* | *30* |
| *3* | *n jest liczbą parzystą* | *25* |
| *4* | *n jest liczbą nieparzystą* | *25* |

**Lech Duraj, Bartosz Szreder**
Tłumaczenie

**Dostępna pamięć: 512 MB.** *BOI 2015, dzień pierwszy, 30.04.2015*

# Edytor

*Bajtazar jest programistą pracującym nad nowym, rewolucyjnym edytorem tekstu. W jego edytorze będą dostępne dwa rodzaje operacji: pierwszy to zwykła* **edycja tekstu**, *zaś drugi to* **cofnięcie** *jednej z poprzednich operacji. Nowym pomysłem Bajtazara jest wprowadzenie operacji* **wielopoziomowego cofania**, *które działa w następujący sposób.*

*Edycja tekstu to operacja* **poziomu 0**. *Operacja* **cofnięcia poziomu** *i polega na znalezieniu oraz cofnięciu ostatnio wykonanej, nie wycofanej operacji o poziomie $i - 1$ albo niższym. W szczególności, cofnięcie poziomu 1 wycofuje ostatnią operację edycji tekstu, zaś cofnięcie poziomu 2 może cofnąć zarówno edycje, jak i cofnięcia poziomu 1 (ale nie cofnięcia wyższych poziomów).*

*Opiszmy to bardziej formalnie. Każda z już wykonanych operacji może być w jednym z dwóch stanów:* `aktywna` *albo* `wycofana`. *Niech $X$ będzie jedną z operacji. Zaraz po jej wykonaniu jest ona* `aktywna`. *Jeśli $X$ jest operacją cofnięcia poziomu $i$, znajdujemy ostatnią operację* `aktywną` *poziomu co najwyżej $i - 1$ (oznaczmy ją $X_1$) i zmieniamy jej stan na* `wycofaną`. *Jeśli $X_1$ sama była operacją cofnięcia i spowodowała wycofanie innej operacji $X_2$, musimy wtedy przywrócić $X_2$ do stanu* `aktywnego`. *Dalej postępujemy według tej samej reguły – jeśli operacja $X_j$ jest operacją cofnięcia i wpłynęła na stan jednej z poprzednich operacji $X_{j+1}$, zmieniamy również stan operacji $X_{j+1}$, oczywiście uwzględniając dalsze skutki tego faktu. Cały ten ciąg czynności kończy się, kiedy osiągniemy operację edycji tekstu.*

*Dla uproszczenia, całą zawartość tekstu w edytorze będziemy reprezentować przez jedną liczbę całkowitą $s$, zwaną* **stanem edytora**, *na początku równą 0. Dla każdej operacji edycji znany jest stan, do którego doprowadza ona edytor. Stan edytora zależy wyłącznie od ostatniej operacji edycji będącej w stanie* `aktywnym`. *Pomóż Bajtazarowi i napisz program, który śledzi stan edytora.*

*Przeanalizujmy następujący przykład. Poniższa tabela zawiera kilka operacji przeprowadzonych przez Bajtazara oraz stan edytora po każdej z nich. Symbol $E_s$ oznacza operację edycji tekstu zmieniającą stan edytora na $s$, zaś $U_i$ to operacja cofnięcia poziomu $i$.*

| *Operacja* | | $E_1$ | $E_2$ | $E_5$ | $U_1$ | $U_1$ | $U_3$ | $E_4$ | $U_2$ | $U_1$ | $U_1$ | $E_1$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Stan edytora* | *0* | *1* | *2* | *5* | *2* | *1* | *2* | *4* | *2* | *1* | *0* | *1* |

*Na początku Bajtazar wykonał trzy operacje edycji, zmieniające stan edytora najpierw z 0 na 1, potem na 2, w końcu na 5. Potem wykonał dwie operacje cofnięcia poziomu 1 – pierwsza cofnęła operację $E_5$, zaś druga cofnęła $E_2$ – zmieniając ich stan na* `wycofane`. *W ten sposób stan edytora powrócił do 1. Kolejną operacją było cofnięcie poziomu 3, które wpłynęło na operację $U_1$ (przez co stała się* `wycofana`*), tym samym przywracając operację $E_2$ (czyniąc ją na powrót* `aktywną`*). Stan edytora przez to znowu zmienił się na 2. Operacja $U_2$ cofnęła operację $E_4$, operacja $U_1$ cofnęła (wcześniej przywróconą) operację $E_2$, przedostatnia operacja ($U_1$) cofnęła operację $E_1$, zaś ostatnia operacja to $E_1$, ustalająca stan edytora na 1.*

## Wejście

*Pierwszy wiersz wejścia zawiera liczbę całkowitą dodatnią $n$, będącą liczbą operacji wykonanych przez Bajtazara. Kolejnych $n$ wierszy zawiera opisy operacji, po jednym w wierszu. Opis składa się z pojedynczej liczby całkowitej $a_i$ ($-n \leqslant a_i \leqslant n$, $a_i \neq 0$). Jeśli $a_i > 0$, to operacja ta jest edycją tekstu, która zmienia stan edytora na $a_i$. Jeśli $a_i < 0$, to jest to operacja cofnięcia poziomu $-a_i$.*

*Możesz założyć, że dla każdej operacji cofnięcia będzie istniała wcześniejsza operacja niższego poziomu w stanie* `aktywnym`*, która będzie mogła zostać cofnięta.*

## Wyjście

*Twój program powinien wypisać $n$ wierszy. $i$-ty wiersz wyjścia powinien zawierać jedną liczbę całkowitą – stan edytora po wykonaniu pierwszych $i$ operacji podanych na wejściu.*

## Przykład

| *Dla danych wejściowych:* | *poprawnym wynikiem jest:* |
|---|---|
| `11` | `1` |
| `1` | `2` |
| `2` | `5` |
| `5` | `2` |
| `-1` | `1` |
| `-1` | `2` |
| `-3` | `4` |
| `4` | `2` |
| `-2` | `1` |
| `-1` | `0` |
| `-1` | `1` |
| `1` | |

## Ocenianie

| Podzadanie | Ograniczenia | Punkty |
|---|---|---|
| *1* | $n \leqslant 5000$ | *20* |
| *2* | $n \leqslant 300\,000$, *jedynie operacje* $E_i$ *oraz* $U_1$ | *15* |
| *3* | $n \leqslant 300\,000$, *oceniana jest wyłącznie ostatnia wypisana liczba (uwaga: pozostałe* $n-1$ *liczb wciąż musi się zawierać między 0 a* $n$*)* | *28* |
| *4* | $n \leqslant 300\,000$ | *37* |

**Lech Duraj, Bartosz Szreder**
Tłumaczenie

**Dostępna pamięć: 256 MB.** *BOI 2015, dzień pierwszy, 30.04.2015*

# Gra w kręgle

*Bajtazar jest miłośnikiem gry w kręgle, a także statystyki. Swego czasu spisywał on wyniki gier w kręgle. Niestety, niektóre zapisy rozmazały się i w związku z tym są teraz nieczytelne. Bajtazar poprosił Cię o napisanie programu znajdującego liczbę rozgrywek zgodnych z jego notatkami.*

## Reguły gry w kręgle

*Gra w kręgle składa się z* $n$ **rund***:* $n-1$ **zwykłych** *rund i jednej rundy* **końcowej**. *W typowej grze* $n = 10$. *Na początku każdej rundy 10* **kręgli** *ustawianych jest pionowo na końcu toru. W czasie rundy gracz wykonuje dwa* **rzuty** *(oprócz rundy końcowej, kiedy może mieć trzy rzuty), w których za pomocą kuli próbuje przewrócić jak najwięcej kręgli na końcu toru. Każda runda zapisywana jest w postaci dwóch znaków (trzech w przypadku rundy końcowej).*

*Dla każdego rzutu gracz otrzymuje tyle* **punktów bazowych**, *ile przewrócił w tym rzucie kręgli. Liczba punktów bazowych w każdej rundzie jest sumą punktów bazowych ze wszystkich rzutów w tej rundzie. Jeśli graczowi uda się przewrócić wszystkie 10 kręgli w jednej zwykłej rundzie, to oprócz 10 punktów bazowych otrzymuje jeszcze* **punkty dodatkowe**.

*Reguły przyznawania punktów dodatkowych w rundzie zwykłej są następujące:*

- *Przewrócenie wszystkich 10 kręgli w pierwszym rzucie rundy określane jest jako* **strike**. *Runda zostaje wtedy zakończona, a punkty dodatkowe przyznaje się na podstawie wyników następnych dwóch rzutów: liczba przyznanych punktów dodatkowych równa jest liczbie uzyskanych punktów bazowych w tych dwóch rzutach. Strike jest oznaczany jako* `X`*.*

- *Przewrócenie wszystkich 10 kręgli w dwóch rzutach określane jest jako* **spare**. *Punkty dodatkowe przyznaje się na podstawie wyniku następnego rzutu: liczba przyznanych punktów dodatkowych równa jest liczbie uzyskanych punktów bazowych w tym rzucie. Spare jest oznaczany jako* `A/`*, gdzie* $A$ *jest cyfrą opisującą liczbę kręgli przewróconych w pierwszym rzucie tej rundy.*

- *Jeśli w obu rzutach przewrócono łącznie 9 albo mniej kręgli, gracz otrzymuje tylko punkty bazowe. Taki wynik oznaczany jest jako* $AB$*, gdzie* $A$ *jest cyfrą opisującą liczbę kręgli przewróconych w pierwszym rzucie tej rundy,* $B$ *jest cyfrą opisującą liczbę kręgli przewróconych w drugim rzucie (*$A + B < 10$*).*

*Punkty dodatkowe dodawane są do wyniku rundy, w której rzucono strike albo spare, a nie do wyniku późniejszych rund, w których dodatkowe punkty zostały rzeczywiście zdobyte.*

*Reguły rozgrywania rundy końcowej:*

- *Początkowo gracz ma do dyspozycji dwa rzuty. Jeśli przewróci 9 albo mniej kręgli w tych rzutach, runda dobiega końca. W przeciwnym przypadku (udało się rzucić strike albo spare) gracz otrzymuje trzeci rzut w tej rundzie. Za każdym razem, gdy graczowi uda się*

*przewrócić wszystkie kręgle w którymkolwiek z tych trzech rzutów, kręgle są ponownie ustawiane do pozycji początkowej przed następnym rzutem. Wynik ostatniej rundy jest łączną liczbą kręgli przewróconych w tych rzutach i wszystkie te punkty są uznawane za punkty bazowe.*

- *Jest zatem siedem możliwości, w jakie ostatnia runda może być zapisana (A i B oznaczają cyfry):*

| *Oznaczenie* | *Opis* | *Punkty* |
|---|---|---|
| `xxx` | *potrójny strike* | *30* |
| `xx`$A$ | *podwójny strike i rzut przewracający A kręgli* | $20 + A$ |
| `x`$A$`/` | *strike i spare, w którym pierwszy rzut przewrócił A kręgli* | *20* |
| `x`$AB$ | *strike i dwa rzuty, przewracające kolejno A i B kręgli* ($A + B < 10$) | $10 + A + B$ |
| $A$`/x` | *spare przewracający A kręgli w pierwszym rzucie, a następnie strike* | *20* |
| $A$`/`$B$ | *spare przewracający A kręgli w pierwszym rzucie i ostatni rzut przewracający B kręgli* | $10 + B$ |
| $AB$`-` | *dwa rzuty, przewracające kolejno A i B kręgli* ($A + B < 10$) | $A + B$ |

*Każda rozgrywka opisywana jest za pomocą ciągu* $2n + 1$ *znaków. Na końcu rozgrywki obliczana jest całkowita liczba zdobytych punktów w każdej z rund. Na przykład całkowita liczba zdobytych punktów opisanych ciągiem* `08x-7/2/x-x-23441/0/x` *obliczana jest następująco:*

| *Runda* | *Opis* | *Pkt. bazowe* | *Pkt. dodatkowe* | *Pkt. w rundzie* | *Suma pkt.* |
|---|---|---|---|---|---|
| *1* | `08` | *0 + 8* | — | *8* | *8* |
| *2* | `x-` | *10* | *7 + 3* | *20* | *28* |
| *3* | `7/` | *7 + 3* | *2* | *12* | *40* |
| *4* | `2/` | *2 + 8* | *10* | *20* | *60* |
| *5* | `x-` | *10* | *10 + 2* | *22* | *82* |
| *6* | `x-` | *10* | *2 + 3* | *15* | *97* |
| *7* | `23` | *2 + 3* | — | *5* | *102* |
| *8* | `44` | *4 + 4* | — | *8* | *110* |
| *9* | `1/` | *1 + 9* | *0* | *10* | *120* |
| *Końcowa* | `0/x` | *0 + 10 + 10* | — | *20* | *140* |

## Wejście

*Pierwszy wiersz danych wejściowych zawiera pojedynczą liczbę* $q$ ($1 \leqslant q \leqslant 25$) *– liczbę przypadków testowych. Następne* $3q$ *wierszy zawiera opis przypadków testowych. Każdy przypadek testowy opisany jest w trzech wierszach.*

*Pierwszy wiersz opisu przypadku testowego zawiera pojedynczą liczbę* $n$ ($2 \leqslant n \leqslant 10$) *– liczbę rund. Drugi wiersz zawiera ciąg* $2n + 1$ *znaków reprezentujących opis gry w notatkach Bajtazara. Rozmyte znaki są zastąpione symbolem* `?`*. Trzeci wiersz zawiera* $n$ *liczb oddzielonych pojedynczymi odstępami – sumaryczne liczby punktów po każdej rundzie. Liczby te są albo czytelne w całości, albo zupełnie zamazane. Nieczytelne wyniki zastąpione są wartością* `-1`*.*

## Wyjście

*Twój program powinien wypisać q wierszy – po jednym wierszu na przypadek testowy, odpowiednio do kolejności ich występowania w danych wejściowych.*

*Dla każdego przypadku testowego Twój program powinien wypisać jedną wartość całkowitą – liczbę różnych możliwych rozgrywek odpowiadających opisowi rozgrywki z wejścia. Dwie gry są uznawane za różne wtedy i tylko wtedy, gdy różnią się przynajmniej jednym rzutem, tzn. zapisy ich rozgrywek różnią się na przynajmniej jednej z $2n+1$ pozycji. Możesz przyjąć, że dla każdego opisu rozgrywki z wejścia istnieje przynajmniej jedna rozgrywka zgodna z zasadami gry, której wyniki są zgodne z notatkami Bajtazara. Możesz także przyjąć, że wynik mieści się w 64-bitowej liczbie całkowitej ze znakiem.*

## Przykład

| *Dla danych wejściowych:* | *poprawnym wynikiem jest:* |
|---|---|
| `2`<br>`10`<br>`08x-7/2/x?x-23??1/???`<br>`8 -1 40 60 82 97 102 110 120 140`<br>`5`<br>`x-x-23?/00-`<br>`22 37 42 52 52` | `9`<br>`10` |

**Wyjaśnienie do przykładu:** *W rundzie 5 pierwszego przypadku testowego po znaku* `x` *jedynym możliwym znakiem jest* `-`*. W rundzie 8 gracz zdobył łącznie 8 punktów, zatem jest dokładnie 9 możliwości, jak taka wartość mogła zostać otrzymana: $0+8$, $1+7$, ..., $8+0$. W rundzie 9 nie zostały przyznane żadne punkty dodatkowe, zatem w pierwszym rzucie rundy końcowej gracz nie zdobył żadnych punktów. Żeby zdobyć 20 punktów w ostatnich dwóch rzutach, jedyną możliwością był spare, po którym nastąpił strike jako ostatni rzut rundy końcowej. Zatem jest dokładnie 9 różnych gier zgodnych z wejściowym ciągiem notatek.*

*W drugim przypadku testowym każda cyfra od* `0` *do* `9` *jest zgodna z notatkami Bajtazara.*

**Dodatkowe testy przykładowe:** *W systemie zawodów znajdują się dodatkowe testy zawierające liczne przypadki testowe, w których zachodzi $n = 2$.*

## Ocenianie

| **Podzadanie** | **Ograniczenia (w każdym przypadku testowym)** | **Punkty** |
|---|---|---|
| *1* | *maksymalnie 6 znaków* `?` *w ciągu wejściowym* | *16* |
| *2* | *wynik nie przekracza $10^9$* | *17* |
| *3* | *z notatkami Bajtazara nie jest zgodna żadna gra, której opis zawiera symbol* `x` *lub* `/` | *26* |
| *4* | *ciąg wejściowy kończy się* `00-` *(gracz zdobył 0 punktów w rundzie końcowej) oraz wyniki ostatnich $\min(3, n)$ rund są zapisane jako* `-1` | *23* |
| *5* | *brak dodatkowych ograniczeń na dane wejściowe* | *18* |

**Lech Duraj, Bartosz Szreder**
Tłumaczenie

**Dostępna pamięć: 256 MB.** *BOI 2015, dzień pierwszy, 30.04.2015*

# Sieć

*Władze Bajtocji uznały, że najwyższy czas, aby ich mały kraj został podłączony do Internetu. Dzięki temu Bajtocjanie będą mogli wreszcie startować w zawodach programistycznych oraz oglądać filmiki ze słodkimi kotkami. Najpierw zbudowano sieć szkieletową, łączącą wszystkie n komputerów w Bajtocji. Sieć ta składa się z bezpośrednich połączeń między parami komputerów, wybranymi tak, aby między każdą parą bajtockich komputerów istniała bezpośrednia lub pośrednia komunikacja.*

*Jako że Bajtocja nie jest zbyt bogatym krajem, sieć została zbudowana oszczędnie, w strukturze* **drzewa** *(tzn. istnieje dokładnie $n-1$ bezpośrednich połączeń pomiędzy komputerami). Później okazało się, że takie rozwiązanie ma zasadniczą wadę – wystarczy, aby zepsuło się chociaż jedno połączenie, a sieć Bajtocji rozpadnie się na części, które nie będą mogły się ze sobą komunikować.*

*Aby poprawić niezawodność sieci, podjęto decyzję o rozbudowaniu jej tak, aby była odporna na zepsucie się jednego połączenia. Twoje zadanie jest następujące: mając daną listę bezpośrednich połączeń między komputerami (których jest $n-1$), znajdź minimalną liczbę nowych połączeń, które trzeba stworzyć, aby zerwanie jednego połączenia nie przerwało komunikacji między żadną parą komputerów.*

## Wejście

*Pierwszy wiersz wejścia zawiera liczbę całkowitą $n$ ($n \geqslant 3$) – liczbę komputerów w Bajtocji. Dla uproszczenia, komputery numerujemy kolejnymi liczbami całkowitymi od 1 do $n$. Każdy z $n-1$ następnych wierszy zawiera dwie liczby całkowite $a$ i $b$ ($1 \leqslant a, b \leqslant n$, $a \neq b$) opisujące bezpośrednie połączenie między komputerami o numerach $a$ i $b$.*

## Wyjście

*W pierwszym wierszu wyjścia Twój program powinien wypisać liczbę całkowitą $k$ – minimalną liczbę połączeń, które trzeba dodać do sieci. W kolejnych $k$ wierszach powinny znaleźć się po dwie liczby całkowite $a$, $b$ ($1 \leqslant a, b \leqslant n$, $a \neq b$) oznaczające numery komputerów, które należy połączyć. Połączenia mogą zostać wypisane w dowolnej kolejności. Jeśli jest więcej niż jedno rozwiązanie, Twój program może wypisać dowolne z nich.*

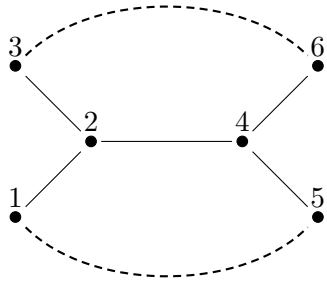
## Przykłady

*Dla danych wejściowych:*

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

*jednym z poprawnych wyników jest:*

```
2
1 5
3 6
```

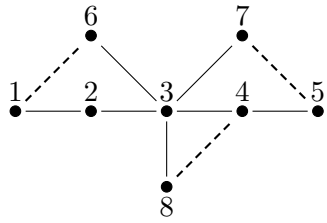



*a dla danych wejściowych:*

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

*jednym z poprawnych wyników jest:*

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




## Ocenianie

| Podzadanie | Ograniczenia | Punkty |
|---|---|---|
| *1* | $n \leqslant 10$ | *18* |
| *2* | $n \leqslant 2000$ | *45* |
| *3* | $n \leqslant 500\,000$ | *37* |

**Lech Duraj, Bartosz Szreder**
Tłumaczenie

**Dostępna pamięć: 256 MB.** *BOI 2015, dzień drugi, 1.05.2015*

# Haker

*Haker Bajtazar zakwalifikował się do finałów tegorocznej edycji Międzynarodowej Olimpiady Hakerskiej. Jedna z konkurencji na Olimpiadzie polega na pojedynku hakera z administratorem systemu. W tej konkurencji danych jest n komputerów, ponumerowanych kolejnymi liczbami całkowitymi od 1 do n, połączonych w pierścień, tzn. połączone są komputery o numerach 1 i n oraz k i k + 1 (dla k = 1, . . . , n − 1).*

*Zawody przeprowadzane są w formie gry pomiędzy hakerem a administratorem:*

- *Bajtazar wykonuje pierwszy ruch. Potem ruch wykonują na zmianę administrator i Bajtazar.*
- *Bajtazar w pierwszym ruchu wybiera dowolny komputer i włamuje się do niego.*
- *Administrator, w swoim pierwszym ruchu, wybiera dowolny komputer (poza tym, który uległ włamaniu) i zabezpiecza go.*
- *W dalszych ruchach Bajtazar albo nie robi niczego, albo wybiera dowolny komputer, do którego jeszcze się nie włamał i który nie jest zabezpieczony, ale jest bezpośrednio przyłączony do dowolnego komputera, do którego wcześniej się włamał, a następnie włamuje się do tego komputera.*
- *W dalszych ruchach administrator albo nie robi niczego, albo wybiera dowolny komputer, który nie uległ włamaniu i który nie jest zabezpieczony, ale jest bezpośrednio przyłączony do dowolnego już zabezpieczonego komputera, a następnie zabezpiecza ten komputer.*
- *Gra kończy się, gdy żaden z graczy nie wykona ruchu przez dwie kolejne rundy.*

*Na początku gry żaden komputer nie uległ włamaniu i żaden komputer nie jest zabezpieczony.*

*Każdy komputer k ma określoną liczbą całkowitą $v_k$ wartość danych, które przechowuje. Bajtazar otrzymuje liczbę punktów równą sumie wartości danych we wszystkich komputerach, do których się włamał. Bajtazar jest świetnym hakerem, ale jego pojęcie o algorytmice jest mgliste. Dlatego poprosił Cię o napisanie programu, który obliczy, jaka jest maksymalna liczba punktów, które może zdobyć w tej konkurencji, przy założeniu, że administrator gra optymalnie.*

## Wejście

*W pierwszym wierszu wejścia znajduje się jedna liczba całkowita n (n > 2), oznaczająca liczbę komputerów. W drugim wierszu znajduje się n liczb całkowitych $v_1$, $v_2$, . . . , $v_n$ ($1 \leqslant v_k \leqslant 2000$). Liczba $v_k$ oznacza wartość danych przechowywanych w komputerze o numerze k.*

## Wyjście

*W pierwszym i jedynym wierszu wyjścia wypisz jedną liczbę całkowitą, oznaczającą maksymalny wynik punktowy Bajtazara w grze przeciwko grającemu optymalnie administratorowi.*

## Przykłady

*Dla danych wejściowych:*
```
4
7 6 8 4
```
*poprawnym wynikiem jest:*
```
13
```
*a dla danych wejściowych:*
```
5
1 1 1 1 1
```
*poprawnym wynikiem jest:*
```
3
```

**Wyjaśnienie do przykładu:** *W pierwszym przykładzie Bajtazar powinien zacząć od włamania się do komputera 2, za co otrzyma 6 punktów. Administrator zabezpieczy komputer 3. Bajtazar wtedy włamie się do niezabezpieczonego komputera 1, otrzymując 7 punktów. Następnie administrator zabezpieczy komputer 4.*

## Ocenianie

| Podzadanie | Ograniczenia | Punkty |
|---|---|---|
| *1* | $n \leqslant 300$ | *20* |
| *2* | $n \leqslant 5000$ | *20* |
| *3* | $n \leqslant 500\,000$; *włamanie do komputera 1 jest optymalnym pierwszym ruchem dla Bajtazara* | *20* |
| *4* | $n \leqslant 500\,000$ | *40* |

**Lech Duraj, Bartosz Szreder**
Tłumaczenie

**Dostępna pamięć: 256 MB.** *BOI 2015, dzień drugi, 1.05.2015*

# Przeciąganie liny

*Przeciąganie liny to w Bajtocji bardzo popularny sport, którego zasady są niezwykle proste: dwie drużyny ciągną linę, każda w swoją stronę. Jesteś komisarzem na dorocznych charytatywnych zawodach w przeciąganiu liny. Twoim zadaniem jest podzielenie $2n$ zawodników na dwie $n$-osobowe drużyny w taki sposób, aby zawody były jak najciekawsze dla publiczności (rozgrywka jest najbardziej widowiskowa, gdy siła obu drużyn jest zbliżona).*

*Lina używana na zawodach ma $n$ możliwych miejsc chwytu po lewej stronie oraz $n$ po prawej stronie. Każdy z graczy (wszyscy bajtoccy przeciągacze liny są dość wybredni) zadeklarował dwa numery swoich ulubionych miejsc – jedno po lewej, drugie po prawej stronie. Znana jest również siła każdego z zawodników, wyrażona pewną liczbą całkowitą dodatnią.*

*Organizatorzy meczu zwrócili się do Ciebie z następującym pytaniem: mając daną liczbę naturalną $k$, czy możliwe jest stworzenie dwóch drużyn po $n$ zawodników tak, aby każdy zawodnik otrzymał jedno ze swoich ulubionych miejsc (oczywiście dwie osoby nie mogą zajmować tego samego miejsca), a sumaryczna siła drużyn nie różniła się o więcej niż $k$?*

## Wejście

*Pierwszy wiersz wejścia zawiera liczbę całkowitą dodatnią $n$, będącą liczbą miejsc po każdej stronie liny, oraz liczbę $k \leqslant 20n$ – maksymalną dozwoloną różnicę między siłą drużyn.*

*Zawodników, dla uproszczenia, będziemy numerować od 1 do $2n$. Każdy z kolejnych $2n$ wierszy wejścia zawiera informację o jednym z nich: w $i$-tym wierszu podane są trzy liczby całkowite dodatnie $l_i$, $r_i$ oraz $s_i$ ($1 \leqslant l_i, r_i \leqslant n$, $1 \leqslant s_i \leqslant 20$), które oznaczają (odpowiednio) ulubione miejsce $i$-tego zawodnika po lewej stronie liny, ulubione miejsce po prawej stronie liny oraz jego siłę.*

## Wyjście

*W pierwszym i jedynym wierszu wyjścia Twój program powinien wypisać pojedyncze słowo* `YES`*, jeśli możliwy jest podział graczy na drużyny zgodnie z warunkami zadania, lub słowo* `NO` *– w przeciwnym przypadku.*

## Przykłady

*W pierwszym przykładzie możemy po lewej stronie liny umieścić zawodników 1, 3, 6, 7 (całkowita siła takiej drużyny to $1 + 8 + 2 + 1 = 12$), zaś zawodników 2, 4, 5 i 8 po prawej stronie (łączna siła $2 + 2 + 5 + 2 = 11$). Różnica sił drużyn wynosi w takim przypadku 1.*

*Dla danych wejściowych:*

```
4 1
1 1 1
2 1 2
2 2 8
1 2 2
3 3 5
3 3 2
4 4 1
4 4 2
```

*poprawnym wynikiem jest:*

```
YES
```

*W drugim przykładzie można zauważyć, że obaj gracze o sile 4 muszą zawsze trafić do tej samej drużyny, nie da się więc osiągnąć różnicy mniejszej niż 6.*

*Dla danych wejściowych:*

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

*poprawnym wynikiem jest:*

```
NO
```

## Ocenianie

| **Podzadanie** | **Ograniczenia** | **Punkty** |
|---|---|---|
| *1* | $n \leqslant 10$ | *18* |
| *2* | $n \leqslant 2000$ | *30* |
| *3* | $n \leqslant 30\,000$, $s_i = 1$ | *23* |
| *4* | $n \leqslant 30\,000$ | *29* |

**Lech Duraj, Bartosz Szreder**
Tłumaczenie

**Dostępna pamięć: 256 MB.** *BOI 2015, dzień drugi, 1.05.2015*

# Ścieżki

*Bajtazar uwielbia życie na krawędzi: zamiast łatać dziury bezpieczeństwa swoich systemów, blokuje IP hakerów; wysyła rozwiązania zadań konkursowych bez testowania ich na przykładowych danych; a nade wszystko lubi, aby każdy plik na jego komputerze miał tak długą ścieżkę katalogową, na jaką pozwala system operacyjny (w systemie Linux, na przykład, limit długości ścieżki wynosi 4095 znaków).*

*Kiedy Bajtazar pracuje na komputerze należącym do kogoś innego, zazwyczaj nie wszystkie pliki spełniają ostatni z powyższych warunków. Wtedy zaczyna on tworzyć dowiązania symboliczne (***symlinki***) i odwoływać się do plików za ich pośrednictwem. Znając nazwy i położenie plików w systemie, sprawdź dla każdego z nich, czy Bajtazar może wygenerować jednego symlinka (o ustalonej wcześniej długości) takiego, żeby do tego pliku dało się odwołać poprzez ścieżkę o długości równej $k$.*

*Dla katalogu* `kat1`*, który zawiera katalog* `kat2`*, który zawiera katalog* `kat3` *itd. aż do katalogu* `katN`*, w którym znajduje się plik o nazwie* `plik`*,* **ścieżka bezwzględna** *do tego pliku to* `/kat1/kat2/.../katN/plik`*. Katalog główny systemu plików oznaczony jest przez* `/`*. Dla pliku zawartego w katalogu głównym jego ścieżka bezwzględna ma postać* `/plik`*.*

*Dowiązanie symboliczne (czyli symlink) to nazwany skrót do katalogu, który może być umieszczony w dowolnym katalogu w systemie plików.* **W tym zadaniu symlinki do plików są niedozwolone***. Używając symlinków, możemy wyznaczać* **alternatywne** *ścieżki plikowe. Na przykład, jeśli w katalogu* `/` *umieścimy symlink nazwany* `hello` *prowadzący do* `/`*, wtedy ścieżki* `/kat/plik`*,* `/hello/kat/plik` *i* `/hello/hello/kat/plik` *odnoszą się do tego samego pliku, ale mają różne długości.*

*Jeśli natomiast w katalogu* `/kat` *umieścimy symlink nazwany* `hi` *prowadzący do* `/`*, to ścieżki* `/kat/plik`*,* `/kat/hi/kat/plik` *i* `/kat/hi/kat/hi/kat/plik` *odnoszą się do tego samego pliku.*

*Symlinki mogą prowadzić zarówno w górę i w dół w hierarchii katalogów, jak i w bok (do innego poddrzewa systemu plików). W szczególności, symlink może prowadzić do katalogu, w którym sam się znajduje. Na potrzeby tego zadania, ścieżki zawierające odwołania do bieżącego katalogu poprzez* `./`*, ścieżki zawierające odwołania do katalogu nadrzędnego poprzez* `../` *oraz ścieżki zawierające* `//` *nie są dozwolone.*

## Wejście

*Pierwszy wiersz wejścia zawiera trzy dodatnie liczby całkowite: $n$ (liczba katalogów w systemie plików, nie licząc katalogu głównego), $m$ (liczba plików) oraz $k$ (żądana długość ścieżki katalogowej). Katalog główny reprezentowany jest liczbą 0, pozostałe katalogi numerowane są od 1 do $n$. Pliki są ponumerowane od 1 do $m$. Drugi wiersz wejścia zawiera liczbę $s$, oznaczającą długość nazwy symlinka, który generuje Bajtazar. Nie przejmujemy się samą nazwą tego symlinka – można założyć, że da się go nazwać tak, aby nie kolidował z żadną inną nazwą używaną w systemie plików.*

*W kolejnych $n$ wierszach wejścia znajdują się opisy katalogów (innych niż katalog główny) w systemie plików. W $i$-tym z tych wierszy znajdują się dwie liczby całkowite $p_i$ oraz $l_i$,*

*oznaczające (odpowiednio), że katalog o numerze $i$ znajduje się w katalogu o numerze $p_i$ oraz że nazwa tego katalogu składa się z $l_i$ znaków. Dla każdego $i$ zachodzi $p_i < i$.*

*W kolejnych $m$ wierszach znajdują się opisy plików. W $j$-tym z tych wierszy znajdują się dwie liczby całkowite $p_j$ oraz $l_j$, oznaczające (odpowiednio), że plik o numerze $j$ znajduje się w katalogu o numerze $p_j$ oraz że nazwa tego pliku składa się z $l_j$ znaków.*

*Wszystkie pliki i katalogi mają nazwy dodatniej długości, a ich ścieżki bezwzględne nie przekraczają $k$-znakowego limitu na długość ścieżki katalogowej w systemie plików.*

## Wyjście

*Twój program powinien wypisać $m$ wierszy, po jednym dla każdego pliku. W $j$-tym z tych wierszy należy wypisać* `YES`*, jeśli jest możliwe wygenerowanie symlinka o nazwie długości $s$ takiego, żeby do $j$-tego pliku można było utworzyć ścieżkę katalogową o długości dokładnie $k$, a w przeciwnym przypadku należy wypisać* `NO`*.*

## Przykład

| *Dla danych wejściowych:* | *poprawnym wynikiem jest:* |
|---|---|
| `2 4 22` | `YES` |
| `2` | `YES` |
| `0 1` | `YES` |
| `1 5` | `NO` |
| `2 13` | |
| `2 10` | |
| `1 4` | |
| `0 7` | |

**Wyjaśnienie do przykładu:** *Przyjmijmy, że nazwa symlinka to* `LL`*, nazwy katalogów to* `a` *i* `bbbbb`*, natomiast nazwy plików to* `ccccccccccccc`*,* `ddddddddddd`*,* `eeee` *i* `fffffff`*. Katalog główny zawiera katalog* `a` *i plik* `fffffff`*. Katalog* `a` *zawiera katalog* `bbbbb` *i plik* `eeee`*. Katalog* `bbbbb` *zawiera pliki* `ccccccccccccc` *i* `ddddddddddd`*.*

```
/
|-- a
|   |-- bbbbb
|   |   |-- ccccccccccccc
|   |   +-- ddddddddddd
|   +-- eeee
+-- fffffff
```

*W pierwszym przypadku ścieżka bezwzględna* `/a/bbbbb/ccccccccccccc` *jest już maksymalnej długości, więc nie potrzebujemy symlinka. W drugim przypadku możemy wygenerować dowiązanie* `/a/LL -> /a`*, co daje możliwą ścieżkę* `/a/LL/bbbbb/ddddddddddd`*. W trzecim przypadku symlink generujemy jako* `/a/LL -> /` *i odwołujemy się przez ścieżkę* `/a/LL/a/LL/a/LL/a/eeee`*. W czwartym przypadku nie da się tak utworzyć symlinka, żeby odwołanie do pliku* `fffffff` *było wymaganej długości.*

## Ocenianie

*We wszystkich podzadaniach zachodzi* $1 \leqslant k$, $s \leqslant 1\,000\,000$.

| **Podzadanie** | **Ograniczenia** | **Punkty** |
|---|---|---|
| *1* | $n, m \leqslant 500$ | *33* |
| *2* | $n, m \leqslant 3000$*; dla każdego pliku, dla którego odpowiedź brzmi* `YES`*, da się utworzyć symlink w taki sposób, że wystarczy nim przejść co najwyżej raz* | *33* |
| *3* | $n, m \leqslant 3000$ | *34* |

# XXII Olimpiada Informatyczna Europy Środkowej,

*Brno, Czechy, 2015*

**Kamil Dębowski, Marek Sommer**
Tłumaczenie

*CEOI 2015, dzień pierwszy, 1.07.2015*

# Mistrzostwa w Calvinballu

*Poza CEOI-em oraz mistrzostwami w hokeju na lodzie, Czechy organizują w tym roku także mistrzostwa w Calvinballu. Nie będziemy zagłębiać się w ledwo istniejące zasady tej gry, a skupimy się na sposobie wybierania podziału na drużyny.*

*W pojedynczej rozgrywce Calvinballu bierze udział $n$ graczy, podzielonych na dowolną liczbę niepustych drużyn. Gracze mają różne imiona. Do zapisu podziału używa się następującej konwencji. Najpierw w każdej drużynie osoba z alfabetycznie najmniejszym imieniem jest wybierana na kapitana. Następnie drużyny są sortowane alfabetycznie po imionach kapitanów i są potem numerowane od 1. Na koniec wypisujemy graczy w kolejności alfabetycznej i piszemy przy każdym numer drużyny, do której należy.*

*Dla przykładu: są trzy drużyny; w jednej grają Calvin, Hobbes i Susie, w jednej Tom i Jerry, a w jednej tylko Batman. Wtedy zapis podziału wygląda następująco:*

| | |
|---|---|
| *Batman* | *1* |
| *Calvin* | *2* |
| *Hobbes* | *2* |
| *Jerry* | *3* |
| *Susie* | *2* |
| *Tom* | *3* |

*Każdego dnia w mistrzostwach biorą udział ci sami gracze, ale za każdym razem ustalany jest inny podział na drużyny. Skoro gracze się nie zmieniają, to możemy dla uproszczenia pominąć ich imiona i zapisywać podział tylko jako ciąg numerów drużyn graczy w kolejności alfabetycznej (*`1 2 2 3 2 3`* w powyższym przykładzie). Mistrzostwa kończą się, gdy zostaną wypróbowane wszystkie możliwe podziały na drużyny.*

*Jako że jest dużo możliwych podziałów na drużyny, osobne wybieranie drużyn na każdy dzień wprowadzałoby niepotrzebny zamęt. W tym roku Międzynarodowa Komisja Dezorganizacji Calvinballu zdecydowała, że podział na drużyny odbywać się będzie zgodnie z leksykograficzną kolejnością ciągów reprezentujących te podziały. Zatem w powyższym przykładzie pierwszego dnia wszyscy gracze będą w tej samej drużynie (ciąg* `1 1 1 1 1 1`*), drugiego dnia wszyscy zagrają przeciwko Tomowi (ciąg* `1 1 1 1 1 2`*), ..., i ostatniego dnia każdy będzie sam w drużynie (ciąg* `1 2 3 4 5 6`*).*

*Dla danego zapisu podziału drużyn określ, którego dnia mistrzostw będzie miał miejsce taki podział. Wynik wypisz modulo 1 000 007.*

*Powyżej imiona graczy są wypisane tylko dla lepszego zobrazowania przykładu. Nie mają one znaczenia w tym zadaniu.*

## Wejście

*Opis podziału na drużyny należy wczytać ze standardowego wejścia. Pierwszy wiersz wejścia zawiera dodatnią liczbę całkowitą $n$ ($1 \leqslant n \leqslant 10\,000$). Drugi wiersz wejścia zawiera oddzielone pojedynczymi odstępami $n$ dodatnich liczb całkowitych, określających podział na drużyny opisany w zadaniu.*

## Wyjście

*Na standardowe wyjście wypisz jedną liczbę całkowitą – numer dnia mistrzostw, w którym użyty będzie podany podział na drużyny, modulo 1 000 007. Pierwszy dzień mistrzostw ma numer 1.*

## Przykład

*Dla danych wejściowych:*

```
3
1 2 2
```

*poprawnym wynikiem jest:*

```
4
```

**Wyjaśnienie do przykładu:** *Możliwe podziały na drużyny w 3-osobowym turnieju to* `1 1 1`, `1 1 2`, `1 2 1`, `1 2 2` *oraz* `1 2 3`.

## Ocenianie

*Jest 10 grup testów, każda warta 10 punktów. Limit na n w każdej grupie testów jest podany niżej.*

| *Grupa* | *1–3* | *4–5* | *6–7* | *8–10* |
|---|---|---|---|---|
| *Limit na n* | *14* | *100* | *1000* | *10 000* |

*Dodatkowo, 4. i 8. grupa testowa zawierają tylko jeden test, a na jego wejściu podany jest ciąg postaci* `1 2 3 ... n` *(opisuje on sytuację z ostatniego dnia mistrzostw, gdy każdy gra sam w swojej drużynie).*

**Kamil Dębowski, Marek Sommer**
Tłumaczenie

*CEOI 2015, dzień pierwszy, 1.07.2015*

# Potiomkinowski cykl

*Książę Potiomkin jest znany ze swoich fałszywych wiosek, tworzonych by zaimponować cesarzowej Katarzynie II.*

*Książę zorganizował dla cesarzowej przejażdżkę po takich wioskach. Wybrał pewną cykliczną trasę przebiegającą po swoich ziemiach, a w wygodnych lokalizacjach podstawiał grupę aktorów, którzy rozstawiali sztuczną wioskę i udawali szczęśliwych mieszkańców. Gdy cesarzowa opuszczała taką wioskę, aktorzy zwijali manatki, wyprzedzali ją i budowali wioskę w następnej lokalizacji.*

*Oczywiście, wybór odpowiedniej trasy nie jest prosty. Cesarzowa może czasem zboczyć na chwilę z trasy i gdyby wróciła w ten sposób do lokalizacji wcześniej odwiedzonej, to zobaczyłaby, że wioska, którą przed chwilą tam widziała, już nie istnieje – wtedy wszystko by się wydało. Dodatkowo, aby cesarzowa nie była zawiedziona, trasa powinna być odpowiednio długa i przebiegać przez przynajmniej cztery lokalizacje.*

*Masz daną mapę ziemi Potiomkina, zawierającą listę dwukierunkowych dróg łączących wybrane lokalizacje (drogi mogą przebiegać mostami i tunelami i są zbudowane tak, że nie przecinają się poza podanymi lokalizacjami). Znajdź trasę spełniającą warunki zdefiniowane przez Potiomkina, tzn. ciąg $s_1, \ldots, s_m$ lokalizacji taki, że:*

- $m \geqslant 4$,
- *wszystkie lokalizacje są parami różne (to znaczy $s_i \neq s_j$ dla $i \neq j$),*
- *lokalizacja $s_i$ jest połączona bezpośrednią drogą z lokalizacją $s_{i+1}$ dla każdego $i = 1, \ldots, m-1$, oraz lokalizacja $s_m$ jest połączona bezpośrednią drogą z lokalizacją $s_1$,*
- *nie ma żadnych innych bezpośrednich dróg pomiędzy odwiedzanymi lokalizacjami (tzn. jeśli dwie lokalizacje $s_i$ i $s_j$ są połączone bezpośrednią drogą, to sąsiadują ze sobą na cyklu).*

## Wejście

*W pierwszym wierszu standardowego wejścia znajdują się dwie liczby całkowite $N$ i $R$ ($0 \leqslant N \leqslant 1\,000$, $0 \leqslant R \leqslant 100\,000$), oznaczające odpowiednio liczbę lokalizacji i liczbę bezpośrednich dróg. W $i$-tym z następnych $R$ wierszy znajdują się dwie liczby całkowite $a_i$ i $b_i$ ($1 \leqslant a_i, b_i \leqslant N$, $a_i \neq b_i$), oznaczające, że lokalizacje $a_i$ i $b_i$ są połączone drogą. Każde dwie lokalizacje są połączone co najwyżej jedną drogą.*

## Wyjście

*Wypisz na standardowe wyjście ciąg $s_1, \ldots, s_m$ parami różnych liczb całkowitych oddzielonych pojedynczymi odstępami, oznaczający trasę określoną w treści zadania. Jeśli istnieje wiele poprawnych tras, możesz wypisać dowolną z nich. Jeśli żadna taka trasa nie istnieje, wypisz* „no”.

## Przykłady

*Dla danych wejściowych:*

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

*jednym z poprawnych wyników jest:*

```
2 3 4 5
```

*a dla danych wejściowych:*

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

*poprawnym wynikiem jest:*

```
no
```

## Ocenianie

*Jest 10 grup testów, a każda z nich warta jest 10 punktów. Limity na $N$ i $R$ w poszczególnych grupach są następujące:*

| *Grupa* | *1–3* | *4–5* | *6–7* | *8–10* |
|---|---|---|---|---|
| *Limit na $N$* | *10* | *100* | *300* | *1000* |
| *Limit na $R$* | *45* | *1000* | *20 000* | *100 000* |

**Kamil Dębowski, Marek Sommer**
Tłumaczenie

*CEOI 2015, dzień pierwszy, 1.07.2015*

# Rury

*W Rurlandii Lomikel jest guru rur. Zarządza rurami, odpływami, ściekami i nawet tunelami metra. Mieszkańcy oddają mu cześć przy świętych źródłach, rozmieszczonych po całej krainie i połączonych siecią ceremonialnych rur.*

*W Święta Wielki Hydraulik (najważniejszy spośród kapłanów Lomikela) odprawia skomplikowane rytuały, polegające na pompowaniu wody pomiędzy źródłami. Gdy Lomikel się rozgniewa, zdarza mu się zniszczyć pewną rurę, przez co Hydraulik musi użyć innych rur, aby woda mogła przepłynąć pomiędzy wybranymi źródłami. Nie zawsze jest to możliwe – jeśli pewne rury zostaną zniszczone, to nie będzie się dało poprowadzić wody inną drogą. Nazwijmy takie rury* **krytycznymi**. *Hydraulik musi bardziej na nie uważać. Na poniższym rysunku do przykładu pogrubiono wszystkie krytyczne rury.*

*Masz dany opis sieci źródeł i rur w Rurlandii. Twoim zadaniem jest znaleźć wszystkie krytyczne rury. Trudnością jest to, że sieć jest ogromna, a masz do dyspozycji mało pamięci.* **Limit pamięci w tym zadaniu jest równy jedynie 16 MB.**

## Wejście

*W pierwszym wierszu wejścia znajdują się dwie liczby całkowite $N$ i $M$ oddzielone pojedynczym odstępem. $N$ oznacza liczbę źródeł ($1 \leqslant N \leqslant 100\,000$), a $M$ oznacza liczbę rur ($1 \leqslant M \leqslant 6\,000\,000$).*

*W kolejnych $M$ wierszach znajdują się opisy poszczególnych rur. Każdy taki opis składa się z dwóch liczb całkowitych $u$ i $v$ ($1 \leqslant u, v \leqslant N$), oznaczających źródła połączone przy pomocy rury. Dwa źródła mogą być połączone wieloma rurami, ale jedna rura zawsze musi łączyć różne źródła.*

*Technikalia: Jest możliwe wielokrotne wczytywanie wejścia (seek), ale nie jest to konieczne, by rozwiązać zadanie. Weź pod uwagę to, że wczytywanie wejścia jest wolne, więc nie należy tego nadużywać.*

## Wyjście

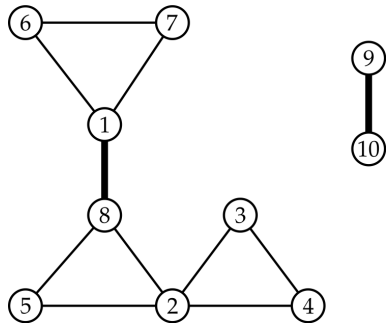
*Na standardowe wyjście wypisz wszystkie krytyczne rury. Opis jednej rury powinien zajmować jeden wiersz i składać się z dwóch liczb całkowitych oddzielonych pojedynczym odstępem: numerów źródeł, które ta rura łączy.*

*Krytyczne rury mogą być wypisane w dowolnej kolejności. Kolejność źródeł w opisie jednej rury też nie ma znaczenia.*

## Przykład

*Dla danych wejściowych:*

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




*jednym z poprawnych wyników jest:*

```
1 8
9 10
```

## Ocenianie

*Jest 10 grup testów, a każda z nich warta jest 10 punktów. Limity dla poszczególnych grup są następujące:*

| *Grupa* | *1* | *2* | *3* | *4* | *5* |
|---|---|---|---|---|---|
| *Limit na N* | *100* | *5000* | *4000* | *10 000* | *30 000* |
| *Limit na M* | *200* | *15 000* | *600 000* | *1 200 000* | *1 500 000* |

| *Grupa* | *6* | *7* | *8* | *9* | *10* |
|---|---|---|---|---|---|
| *Limit na N* | *70 000* | *80 000* | *100 000* | *100 000* | *100 000* |
| *Limit na M* | *2 000 000* | *3 000 000* | *4 000 000* | *5 000 000* | *6 000 000* |

**Kamil Dębowski, Marek Sommer**
Tłumaczenie

*CEOI 2015, dzień drugi, 3.07.2015*

# Mistrzostwa w Calvinballu raz jeszcze

*Jak zapewne pamiętasz, każdego roku w Czechach są organizowane Mistrzostwa w Calvinballu. W rozgrywce Calvinballu bierze udział $n$ graczy z różnymi imionami. Gracze są podzieleni na dowolną liczbę niepustych drużyn. Niektórzy gracze jednak nie lubią się nawzajem. Jest to relacja symetryczna: jeśli $A$ nie lubi $B$, to $B$ nie lubi $A$.*

*Międzynarodowa Komisja Dezorganizacji Calvinballu zdecydowała, by w ostatniej chwili zmienić sposób wyboru podziału na drużyny. W jednej drużynie nie mogą znaleźć się dwie osoby, które się nie lubią. Dodatkowo, liczba drużyn musi być tak mała, jak to tylko możliwe.*

*Dla przykładu niech Calvin, Hobbes, Susie, Tom, Jerry i Batman biorą udział w rozgrywce. Batman nie lubi się z każdą inną osobą, a Tom nie lubi się z Jerrym oraz z Hobbesem. Jest możliwe, by przeprowadzić rozgrywkę z trzema drużynami (na przykład Batman sam, Tom z Susie oraz w trzeciej drużynie Calvin, Hobbes i Jerry). Nie jest możliwy podział na dwie drużyny, gdyż Batman, Tom i Jerry nie lubią się wzajemnie, więc każdy musi być w innej drużynie. Nie jest dopuszczalny podział na cztery drużyny, gdyż istnieje możliwość podziału na mniejszą liczbę drużyn (trzy).*

*Na podstawie listy par osób, które się nie lubią, podaj przykład możliwego podziału graczy na drużyny (dowolny z możliwych, jeśli jest ich wiele).*

## Wejście

*W tym zadaniu pliki wejściowe znajdziesz na swoim komputerze, a do oceny wysłać musisz pliki wyjściowe. W katalogu* `/mo/problems/again` *znajdziesz 10 plików nazwanych* `input_000.txt`, *...,* `input_009.txt`. *Każdy plik ma następujący format:*

*Pierwszy wiersz zawiera dwie nieujemne liczby całkowite $n$ i $m$, oznaczające odpowiednio liczbę graczy i liczbę różnych par graczy, którzy się nie lubią. Gracze są ponumerowani od 1 do $n$. W następnych $m$ wierszach znajduje się opis nielubiących się par graczy; $i$-ty z nich zawiera dwie różne liczby całkowite $a_i$ i $b_i$ ($1 \leqslant a_i, b_i \leqslant n$), oznaczające, że gracze $a_i$ i $b_i$ nie lubią się nawzajem.*

## Wyjście

*Dla pliku wejściowego* `input_00`$k$`.txt` *(gdzie $k = 0, \ldots, 9$) przygotuj plik wyjściowy* `output_00`$k$`.txt` *o następującym formacie: Pierwszy wiersz powinien zawierać nieujemną liczbę $t$, oznaczającą liczbę drużyn. Następne $t$ wierszy powinno opisywać przydział graczy do drużyn; $i$-ty z nich powinien zawierać listę numerów graczy w $i$-tej drużynie. Drużyny oraz graczy w drużynach możesz wypisać w dowolnej kolejności.*

*Pliki wyjściowe należy wysyłać przez stronę konkursu. W przypadku wysyłania plików wyjściowych tylko do niektórych testów (nie wszystkich), pozostałe zostaną skopiowane z poprzedniej próby (o ile tam były). Jest też możliwość wysyłania plików wyjściowych pojedynczo.*

## Przykład

*Dla danych wejściowych:*

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

*jednym z możliwych wyników jest:*

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

**Wyjaśnienie do przykładu:** *Przykładowe wejście odpowiada sytuacji opisanej w treści zadania, z graczami ponumerowanymi następująco:*

| *Gracz* | *Calvin* | *Hobbes* | *Susie* | *Tom* | *Jerry* | *Batman* |
|---|---|---|---|---|---|---|
| *Numer* | *1* | *2* | *3* | *4* | *5* | *6* |

## Ocenianie

*10 punktów będzie przyznane za każdy z 10 plików wejściowych, do którego zostanie wysłany poprawny plik wyjściowy.*

**Kamil Dębowski, Marek Sommer**
**Tłumaczenie**

*CEOI 2015, dzień drugi, 3.07.2015*

# Mistrzostwa w Hokeju na Lodzie

*W tym roku Mistrzostwa Świata w Hokeju na Lodzie odbywają się w Czechach. Bobek przybył do Pragi i chciałby zobaczyć niektóre mecze. Bobek nie ma żadnych ograniczeń czasowych ani preferencji co do oglądanych drużyn. Gdyby miał wystarczająco dużo pieniędzy, byłby w stanie zobaczyć wszystkie mecze. Niestety, ma on ograniczoną liczbę koron czeskich i tylko tyle może wydać na bilety. Znając ceny biletów na każdy mecz, znajdź liczbę sposobów wybrania takiego zbioru meczów, by nie przekroczyć dostępnego budżetu. Dwa sposoby uznajemy za różne, jeśli istnieje mecz, który jest wybrany do oglądania tylko w jednym z tych sposobów.*

## Wejście

*Opis sytuacji Bobka znajduje się na standardowym wejściu. Pierwszy wiersz wejścia zawiera dwie dodatnie liczby całkowite $N$ i $M$ ($1 \leqslant N \leqslant 40$, $1 \leqslant M \leqslant 10^{18}$), oznaczające odpowiednio liczbę meczów i liczbę koron czeskich, które Bobek może wydać. Drugi wiersz zawiera $N$ oddzielonych pojedynczymi odstępami dodatnich liczb całkowitych, nieprzekraczających $10^{16}$, oznaczających koszty meczów podane w koronach czeskich.*

## Wyjście

*Wypisz jeden wiersz zawierający liczbę sposobów, na jakie Bobek może wybrać mecze do oglądania. Zauważ, że ze względu na limit na $N$, liczba ta nigdy nie przekroczy $2^{40}$.*

## Przykład

| *Dla danych wejściowych:* | *poprawnym wynikiem jest:* |
|---|---|
| `5 1000`<br>`100 1500 500 500 1000` | `8` |

**Wyjaśnienie do przykładu:** *Osiem możliwych sposobów to: (1) niezobaczenie żadnych meczów; (2) mecz kosztujący 100; (3) pierwszy mecz kosztujący 500; (4) drugi mecz kosztujący 500; (5) mecz kosztujący 100 i pierwszy mecz kosztujący 500; (6) mecz kosztujący 100 i drugi mecz kosztujący 500; (7) oba mecze kosztujące 500; (8) mecz kosztujący 1000.*

## Ocenianie

*Jest 10 grup testów, każda warta 10 punktów. Górne limity na $N$ i $M$ w każdej grupie są podane niżej.*

| *Grupa* | *1–2* | *3–4* | *5–7* | *8–10* |
|---|---|---|---|---|
| *Limit na $N$* | *10* | *20* | *40* | *40* |
| *Limit na $M$* | $10^6$ | $10^{18}$ | $10^6$ | $10^{18}$ |

**Kamil Dębowski, Marek Sommer**
Tłumaczenie

*CEOI 2015, dzień drugi, 3.07.2015*

# Nuclearia

*Dawno temu mieszkańcy Nuclearii zbudowali wiele elektrowni jądrowych. Przez lata wszystkie działały dobrze, aż pewnego razu Nuclearię nawiedziło trzęsienie ziemi. Katastrofa spowodowała eksplozję elektrowni jądrowych i promieniowanie zaczęło rozprzestrzeniać się po kraju. Gdy udało się okiełznać żywioł i promieniowanie, Ministerstwo Środowiska zaczęło szacować straty. Twoim zadaniem jest napisać program, który będzie odpowiadał na zapytania o napromieniowanie pewnych regionów kraju.*

### Jak rozprzestrzenia się promieniowanie

*Nuclearię można przedstawić jako prostokąt składający się z $W \times H$ pól. Każda elektrownia jądrowa zajmuje jedno pole i ma dwa parametry będące liczbami całkowitymi: $a$ i $b$. Wartość $a$ jest natężeniem promieniowania na polu, na którym znajduje się elektrownia, a wartość $b$ określa, jak szybko promieniowanie zmniejsza się w miarę oddalania się od elektrowni.*

*Dokładniej, natężenie promieniowania, które dotrze do pola $C = (x_C, y_C)$ od elektrowni na polu $P = (x_P, y_P)$, jest równe $max\,(0, a - b \cdot d(P,C))$, gdzie $d(P,C)$ jest odległością między polami $P$ i $C$ zdefiniowaną tak: $d(P,C) = max\,(|x_P - x_C|, |y_P - y_C|)$.*

*Całkowite promieniowanie na jednym polu jest równe* **sumie** *promieniowań pochodzących od poszczególnych elektrowni jądrowych.*

*Dla przykładu, rozważmy elektrownię z $a = 7$ i $b = 3$. Jej wybuch spowoduje promieniowanie wielkości 7 na polu, na którym się ona się znajduje, promieniowanie wielkości 4 na ośmiu sąsiednich polach i promieniowanie wielkości 1 na szesnastu polach odległych o 2. Gdyby ta elektrownia była położona na granicy Nuclearii lub jedno pole od granicy, to wybuch wpłynąłby również na pewne pola poza Nuclearią. Eksplozję, której promieniowanie rozprzestrzenia się poza granice Nuclearii, nazwijmy* **eksplozją graniczną**. *(W zadaniu nie jest istotne, czy eksplozja jest graniczna, czy nie. Ta definicja przyda się po prostu w sekcji „Ocenianie").*

### Zapytania

*Ministerstwo Środowiska za każdym razem pyta o średnie napromieniowanie pola w danym prostokątnym regionie. W Ministerstwie panuje wielka dezorganizacja, więc nie zakładaj niczego na temat zadanych regionów – mogą się powtarzać, pokrywać, zawierać...*

## Wejście

*Opis Nuclearii należy wczytać ze standardowego wejścia. W pierwszym wierszu znajdują się dwie dodatnie liczby całkowite $W$ i $H$ ($W \cdot H \leqslant 2\,500\,000$), oddzielone pojedynczym odstępem i oznaczające odpowiednio szerokość i wysokość Nuclearii. W drugim wierszu znajduje się liczba całkowita $N$, oznaczająca liczbę elektrowni jądrowych, które wybuchły ($1 \leqslant N \leqslant 200\,000$). W kolejnych $N$ wierszach znajdują się po cztery liczby całkowite $x_i$, $y_i$, $a_i$, $b_i$ ($1 \leqslant x_i \leqslant W$, $1 \leqslant y_i \leqslant H$, $1 \leqslant a_i, b_i \leqslant 10^9$), oznaczające, że na polu $(x_i, y_i)$ wybuchła elektrownia o parametrach $a_i$, $b_i$. Na każdym polu znajduje się co najwyżej jedna elektrownia.*

*W następnym wierszu znajduje się jedna liczba całkowita $Q$, oznaczająca liczbę zapytań Ministerstwa ($1 \leqslant Q \leqslant 200\,000$). W kolejnych $Q$ wierszach znajdują się po cztery liczby całkowite $x_{1j}$, $y_{1j}$, $x_{2j}$, $y_{2j}$ ($1 \leqslant x_{1j} \leqslant x_{2j} \leqslant W$, $1 \leqslant y_{1j} \leqslant y_{2j} \leqslant H$), oznaczające zapytanie o region będący prostokątem, którego lewy górny róg jest wyznaczony przez pole $(x_{1j}, y_{1j})$, a prawy dolny róg przez pole $(x_{2j}, y_{2j})$.*

*Możesz założyć, że całkowite promieniowanie w Nuclearii jest mniejsze niż $2^{63}$.*

## Wyjście

*Dla każdego zapytania wypisz jeden wiersz zawierający średnie napromieniowanie pola w zadanym regionie, zaokrąglone do najbliższej liczby całkowitej (połówki zaokrąglane są w górę).*

## Przykład

| *Dla danych wejściowych:* | *poprawnym wynikiem jest:* |
|---|---|
| `4 3` | `4` |
| `2` | `4` |
| `1 1 7 3` | `2` |
| `3 2 4 2` | `2` |
| `4` | |
| `1 2 2 3` | |
| `1 1 4 3` | |
| `4 2 4 2` | |
| `1 3 4 3` | |

**Wyjaśnienie do przykładu:** *Natężenie promieniowania w Nuclearii po dwóch eksplozjach jest następujące:*

| | | | |
|---|---|---|---|
| *7* | *6* | *3* | *2* |
| *4* | *6* | *5* | *2* |
| *1* | *3* | *3* | *2* |

*Pierwsza eksplozja jest eksplozją graniczną, a druga nie. Jeśli chodzi o zapytania:*

1. *Całkowite promieniowanie w kwadracie 2 na 2 jest równe 14, więc średnie promieniowanie jest równe $\frac{14}{4} = 3{,}5$ (zaokrąglone do 4).*
2. *Całkowite promieniowanie w Nuclearii jest równe 44, więc średnie promieniowanie jest równe $\frac{44}{12} \approx 3{,}67$ (zaokrąglone do 4).*
3. *Średnie promieniowanie na pojedynczym polu jest równe po prostu promieniowaniu na tym polu.*
4. *Średnie promieniowanie w ostatnim wierszu jest równe $\frac{9}{4} = 2{,}25$ (zaokrąglone do 2).*

## Ocenianie

*Jest 14 grup testów. Grupy o nieparzystych numerach zawierają jedynie elektrownie, dla których a jest wielokrotnością b. Dodatkowe warunki w grupach są następujące:*

| *Grupa* | *Dodatkowe warunki* | *Punkty* |
|---|---|---|
| *1* | $H = 1$, $N \cdot W \leqslant 10^8$, $Q \cdot W \leqslant 10^8$ | *3* |
| *2* | $H = 1$, $N \cdot W \leqslant 10^8$, $Q \cdot W \leqslant 10^8$ | *2* |
| *3* | $N \cdot W \cdot H \leqslant 10^8$, $Q \cdot W \cdot H \leqslant 10^8$ | *3* |
| *4* | $N \cdot W \cdot H \leqslant 10^8$, $Q \cdot W \cdot H \leqslant 10^8$ | *2* |
| *5* | $H = 1$, $N \cdot W \leqslant 10^8$ | *6* |
| *6* | $H = 1$, $N \cdot W \leqslant 10^8$ | *4* |
| *7* | $N \cdot W \cdot H \leqslant 10^8$ | *6* |
| *8* | $N \cdot W \cdot H \leqslant 10^8$ | *4* |
| *9* | $H = 1$ | *15* |
| *10* | $H = 1$ | *10* |
| *11* | *brak eksplozji granicznych* | *15* |
| *12* | *brak eksplozji granicznych* | *10* |
| *13* | *brak* | *12* |
| *14* | *brak* | *8* |

# Bibliografia


[1] *I Olimpiada Informatyczna 1993/1994*. Warszawa–Wrocław, 1994.

[2] *II Olimpiada Informatyczna 1994/1995*. Warszawa–Wrocław, 1995.

[3] *III Olimpiada Informatyczna 1995/1996*. Warszawa–Wrocław, 1996.

[4] *IV Olimpiada Informatyczna 1996/1997*. Warszawa, 1997.

[5] *V Olimpiada Informatyczna 1997/1998*. Warszawa, 1998.

[6] *VI Olimpiada Informatyczna 1998/1999*. Warszawa, 1999.

[7] *VII Olimpiada Informatyczna 1999/2000*. Warszawa, 2000.

[8] *VIII Olimpiada Informatyczna 2000/2001*. Warszawa, 2001.

[9] *IX Olimpiada Informatyczna 2001/2002*. Warszawa, 2002.

[10] *X Olimpiada Informatyczna 2002/2003*. Warszawa, 2003.

[11] *XI Olimpiada Informatyczna 2003/2004*. Warszawa, 2004.

[12] *XII Olimpiada Informatyczna 2004/2005*. Warszawa, 2005.

[13] *XIII Olimpiada Informatyczna 2005/2006*. Warszawa, 2006.

[14] *XIV Olimpiada Informatyczna 2006/2007*. Warszawa, 2007.

[15] *XV Olimpiada Informatyczna 2007/2008*. Warszawa, 2008.

[16] *XVI Olimpiada Informatyczna 2008/2009*. Warszawa, 2009.

[17] *XVII Olimpiada Informatyczna 2009/2010*. Warszawa, 2010.

[18] *XVIII Olimpiada Informatyczna 2010/2011*. Warszawa, 2011.

[19] *XIX Olimpiada Informatyczna 2011/2012*. Warszawa, 2012.

[20] *XX Olimpiada Informatyczna 2012/2013*. Warszawa, 2013.

[21] *XXI Olimpiada Informatyczna 2013/2014*. Warszawa, 2015.

[22] A. V. Aho, J. E. Hopcroft, J. D. Ullman. *Projektowanie i analiza algorytmów komputerowych*. PWN, Warszawa, 1983.

[23] L. Banachowski, A. Kreczmar. *Elementy analizy algorytmów.* WNT, Warszawa, 1982.

[24] L. Banachowski, K. Diks, W. Rytter. *Algorytmy i struktury danych.* WNT, Warszawa, 1996.

[25] J. Bentley. *Perełki oprogramowania.* WNT, Warszawa, 1992.

[26] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Wprowadzenie do algorytmów.* WNT, Warszawa, 2004.

[27] M. de Berg, M. van Kreveld, M. Overmars. *Geometria obliczeniowa. Algorytmy i zastosowania.* WNT, Warszawa, 2007.

[28] R. L. Graham, D. E. Knuth, O. Patashnik. *Matematyka konkretna.* PWN, Warszawa, 1996.

[29] D. Harel. *Algorytmika. Rzecz o istocie informatyki.* WNT, Warszawa, 1992.

[30] D. E. Knuth. *Sztuka programowania.* WNT, Warszawa, 2002.

[31] W. Lipski. *Kombinatoryka dla programistów.* WNT, Warszawa, 2004.

[32] E. M. Reingold, J. Nievergelt, N. Deo. *Algorytmy kombinatoryczne.* WNT, Warszawa, 1985.

[33] K. A. Ross, C. R. B. Wright. *Matematyka dyskretna.* PWN, Warszawa, 1996.

[34] R. Sedgewick. *Algorytmy w C++. Grafy.* RM, 2003.

[35] S. S. Skiena, M. A. Revilla. *Wyzwania programistyczne.* WSiP, Warszawa, 2004.

[36] P. Stańczyk. *Algorytmika praktyczna. Nie tylko dla mistrzów.* PWN, Warszawa, 2009.

[37] M. M. Sysło. *Algorytmy.* WSiP, Warszawa, 1998.

[38] M. M. Sysło. *Piramidy, szyszki i inne konstrukcje algorytmiczne.* WSiP, Warszawa, 1998.

[39] J. Tomasiewicz. *Zaprzyjaźnij się z algorytmami. Przewodnik dla początkujących i średniozaawansowanych.* PWN, Warszawa, 2016.

[40] R. J. Wilson. *Wprowadzenie do teorii grafów.* PWN, Warszawa, 1998.

[41] N. Wirth. *Algorytmy + struktury danych = programy.* WNT, Warszawa, 1999.

[42] *W poszukiwaniu wyzwań. Wybór zadań z konkursów programistycznych Uniwersytetu Warszawskiego.* Warszawa, 2012.

[43] *W poszukiwaniu wyzwań 2. Zadania z Akademickich Mistrzostw Polski w Programowaniu Zespołowym 2011-2014.* Warszawa, 2015.

Niniejsza publikacja stanowi wyczerpujące źródło informacji o zawodach XXII Olimpiady Informatycznej, przeprowadzonych w roku szkolnym 2014/2015. 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 XXVII Międzynarodowej Olimpiady Informatycznej, XXI Bałtyckiej Olimpiady Informatycznej i XXII Olimpiady Informatycznej Krajów Europy Środkowej.

*XXII Olimpiada Informatyczna* to pozycja polecana szczególnie uczniom przygotowującym się do udziału w zawodach następnych Olimpiad oraz nauczycielom informatyki, którzy znajdą w niej interesujące i przystępnie sformułowane problemy algorytmiczne wraz z rozwiązaniami. Książka może być wykorzystywana także jako pomoc do zajęć z algorytmiki na studiach informatycznych.

Sponsorzy Olimpiady:

**ISBN 978-83-64292-02-6**